AlloViz

Release 0.1

Francho Nerín-Fonz

Sep 04, 2023

CONTENTS

A Python package to interactively compute, analyze and visualize protein allosteric communication (residue interaction) networks and delta-networks.

AlloViz binds together some newly written modules with 8 Python packages that provide different ways of calculating residue interactions: GetContacts, correlationplus, dynetan, PyInteraph2, pytraj, MD-TASK, MDEntropy and CARDS.

For the same topology and molecular dynamics (MD) trajectory, the network can be constructed based on residue contacts, correlation of atom movement or dihedrals, or interaction energies, depending on the package selected. Moreover, for example for movement correlation, the movement tracked can be that of the whole residue, its center of mass, its alpha-C or its beta-C; and it can be calculated as the Pearson's correlation coefficient, Mutual Information (MI) or Linear MI (LMI). See all the *options*.

The resulting network can be analyzed with edge centrality metrics algorithms provided by the Python package NetworkX, and they can be visualized in an interactive Python Notebook (i.e., Jupyter) using nglview.

AlloViz can also be use through a *GUI*.

CHAPTER

ONE

INSTALL

The repository must be cloned along with all the submodules using the `--recursive` flag. Additional flags are recommended for speed:

```
git clone --recursive --shallow-submodules -j 9 https://github.com/frannerin/AlloViz
```

It is recommended to create a **virtual environment** with Miniconda or similars using the `conda-forge` channel (a fast dependency solver is recommended for speed: libamba solver for Miniconda or the Mamba version of Conda):

```
conda create -n AlloViz -c conda-forge --solver libmamba --file AlloViz/conda_
↪environment.txt
conda activate AlloViz
```

Finally, AlloViz is installed into the environment with `pip install ./AlloViz`.

Although not recommended, the virtual environment can also be created with **pip**:

```
python -m venv AlloViz/env
source AlloViz/env/bin/activate
pip install -r AlloViz/pip_requirements.txt
pip install ./AlloViz
```

> Python <3.10 is recommended (i.e., 3.9.16). `pytraj` and the construction of delta-networks won't be available in a pip environment, as AmberTools and pymol-open-source are needed (respectively) for that, and they aren't distributed through PyPi. Other additional dependencies might also need to be installed by hand.

CHAPTER

TWO

QUICKSTART

Check the *tutorial notebooks* or the *quickstart*.

CHAPTER

THREE

CITE

CHAPTER

FOUR

LICENSE

4.1 Network construction methods

Residue information extracted from trajectories	Atom/angle tracked	Package	Correlation measurement	Name in AlloViz
Atoms' displacements	Whole residue	dynetan	MI (Mutual Information)	dynetan
	Carbon	pytraj	Pearson's	pytraj_CB
	Carbon			pytraj_CA
		MD-TASK	Pearson's	MDTASK
		correlationplus	Pearson's	correlationplus_CA_Pear
			LMI (Linear MI)	correlationplus_CA_LMI
Dihedral angles	Phi Psi		Pearson's	correlationplus_Phi, correlationplus_Psi
	All backbone dihedrals (Phi and psi)			correlationplus_Backbone_Dihs
	Chi[1-4]	AlloViz	MI	AlloViz_Phi, AlloViz_Psi, AlloViz_Backbone_Dihs
	All side-chain dihedrals			AlloViz_Chi[1-4], AlloViz_Sidechain_Dihs
	All dihedrals			AlloViz_Dihs
		CARDS	MI, Pure-disorder MI, Disorder-mediated MI, Holistic MI	CARDS_[MI, Pure_disorder_MI, Disorder_mediated_MI, Holistic_MI]_[Phi, Psi, Backbone_Dihs]
				CARDS_[MI, Pure_disorder_MI, Disorder_mediated_MI, Holistic_MI]_Chi[1

4.2 Tutorials

4.2.1 Quickstart

[1]:
```
import AlloViz
```

System setup

The Protein class constructor processes the input data to prepare it for the downstream calculations, analyses and visualization.

A structure and Molecular Dynamics simulation trajectories from GPCRmd database of the GPCR Beta-2 adrenergic receptor (in complex with agonist epinephrine, GPCRmd ID 117) are included with the notebooks in AlloViz. Multiple trajectories can be passed in a list, e.g.: `["data/117/traj_1.xtc", "data/117/traj_2.xtc", "data/117/traj_3.xtc"]`

[2]:
```
system = AlloViz.Protein(pdb="data/117/protein.pdb",
                         trajs="data/117/traj_1.xtc",
                         path="data/117")
```

```
/home/frann/miniconda3/envs/alloviznew/lib/python3.9/site-packages/MDAnalysis/
↪coordinates/PDB.py:431: UserWarning: 1 A^3 CRYST1 record, this is usually a␣
↪placeholder. Unit cell dimensions will be set to None.
  warnings.warn("1 A^3 CRYST1 record,"
```

Network calculation

AlloViz brings together 8 different Python packages, previously peer-reviewed and published, which allow the construction of allosteric communication networks using different descriptors (plus a network construction method we implemented ourselves). All the available options can be found in this table (the last column has the names to be used in the calculate method below).

We are going to use `pytraj_CA` as network construction method, which measures the correlation of the residues' positions along the trajectory using Pearson's linear correlation coefficient. Multiple networks can be calculated at the same time supplying a list of names, e.g. `["dynetan", "GetContacts"]`

[3]:
```
system.calculate(pkgs="pytraj_CA")
```

```
pytraj_CA
adding raw data of pytraj_CA for data/117/protein.pdb:  ['data/117/data/pytraj_CA/raw/1.
↪pq']
Please, make sure to correctly cite the package used to compute the network: pytraj␣
↪(https://github.com/Amber-MD/pytraj#citation)
```

[3]:
```
<AlloViz.Wrappers.pytraj_w.pytraj_CA at 0x7f3125deda30>
```

The calculated raw data is both saved as a file for reconstitution later and added as an attribute of the Protein object:

[4]:
```
system.pytraj_CA.raw
```

[4]:
```
                    weight
GLN:26  GLU:27    0.924889
        ARG:28    0.816165
        ASP:29    0.729000
        GLU:30    0.779299
        VAL:31    0.828841
...                    ...
LEU:339 CYS:341   0.897811
        LEU:342   0.753879
LEU:340 CYS:341   0.938434
        LEU:342   0.849064
CYS:341 LEU:342   0.903325

[45753 rows x 1 columns]
```

Network filtering

AlloViz allows to filter the calculated networks according to different criteria (the different options are each of the functions defined in the Filtering module).

We are going to filter the dynetan network using the Spatially_distant filter, which filters out residue pairs of the network with a CA-CA distance below a threshold (default: 10 angstroms) to focus on long-range residue pairs. Multiple filterings can be performed supplying a list of strings to the filter method.

[5]:
```
system.filter(filterings="Spatially_distant")
# the same as doing: system.pyraj_CA.filter(filterings="Spatially_distant")
```

[5]:
```
<AlloViz.AlloViz.Filtering.Filtering at 0x7f31947652b0>
```

The results are stored as a new attribute of the `dynetan` results, with the filtered network stored as NetworkX's Graph object for later analysis:

[6]:
```
system.pytraj_CA.Spatially_distant.graphs
```

[6]:
```
{'weight': <networkx.classes.graph.Graph at 0x7f3194765df0>}
```

Network analysis

AlloViz can analyze the filtered networks with the betweenness centrality and current-flow betweenness centrality metrics (for edges and/or for nodes) functions of NetworkX. Other NetworkX functions that return per-edge or per-residue values can be passed to the analyze method using the instructions in the documentation.

For example, we can analyze the betweenness centrality ("btw") of both the edges and the nodes of the filtered network. As before, we can run multiple analysis at the same time by supplying lists.

[7]:
```
system.analyze(elements=["edges", "nodes"], metrics="btw")
# the same as: system.dynetan.Spatially_distant.analyze(elements=["edges", "nodes"],␣
↪metrics="btw")
```

```
adding analyzed edges <AlloViz.Wrappers.pytraj_w.pytraj_CA object at 0x7f3125deda30>␣
↪Spatially_distant data of for data/117/protein.pdb
adding analyzed nodes <AlloViz.Wrappers.pytraj_w.pytraj_CA object at 0x7f3125deda30>␣
↪Spatially_distant data of for data/117/protein.pdb
```

The results are stored as DataFrames as new attributes of the `Spatially_distant` filtering results, one for each element (nodes and edges) of the network.

[8]:
```
system.pytraj_CA.Spatially_distant.nodes
```

[8]:
```
             btw
GLN:26   0.010209
VAL:31   0.007019
TRP:32   0.002662
VAL:33   0.001760
VAL:34   0.007129
...           ...
LEU:342  0.000154
GLU:27   0.012629
ARG:28   0.002816
ASP:29   0.000154
GLU:30   0.002904

[303 rows x 1 columns]
```

Network visualization

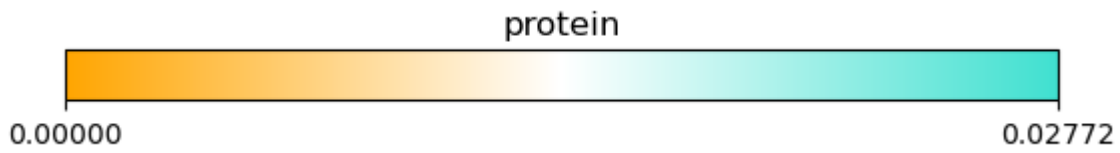
The analysis results of the edges or the nodes (and also both together) can be visualised on the protein structure with an interactive widget. The exact element that we wish to visualise can be specified to the view method, or we can exploit the custom edges' and nodes' DataFrames' view method.

For example, to view the nodes' betweenness centrality analysis results:

[9]:
```
system.pytraj_CA.Spatially_distant.nodes.view("btw")
# the same as: system.view(pkg="pytraj_CA", metric="btw", filtering="Spatially_distant",␣
↪element="nodes")
```

```
/home/frann/miniconda3/envs/alloviznew/lib/python3.9/site-packages/MDAnalysis/
↪coordinates/PDB.py:431: UserWarning: 1 A^3 CRYST1 record, this is usually a␣
↪placeholder. Unit cell dimensions will be set to None.
  warnings.warn("1 A^3 CRYST1 record,"
```

```
NGLWidget()
```



4.2.2 Delta network

A delta-network can be calculated by subtracting the edge weights of two networks that have "comparable" nodes (e.g., two networks of the same protein in different conformations, such as activation states). The delta-network reduces the noise in the data and provides a better way of comparing two systems, in opposition to a direct comparison of the networks' edge weights, and thus is able to capture more subtle differences.

[1]:
```
import AlloViz
```

Systems setup

We are going to use the structure and simulations of the class A GPCR Beta-2 adrenergic receptor (B2AR) in the active form bound to epinephrine (full agonist, GPCRmd ID 117) and in the inactive form bound to carazolol (inverse agonist, 160).

The GPCR argument can be a True boolean, in which case the structure will be processed as a GPCR and the corresponding generic numbering will be retrieved from GPCRdb. It can also be the ID of a GPCRmd database dynamics entry, to automatically retrieve the files from the database and process them.

[2]:
```
activeB2AR = AlloViz.Protein(pdb="data/117/protein.pdb",
                             trajs=["data/117/traj_1.xtc", "data/117/traj_2.xtc"],
                             path="data/117",
                             GPCR=True,
                             name="Active B2AR")
inactiveB2AR = AlloViz.Protein(pdb="data/160/protein.pdb",
                             trajs=["data/160/traj_1.xtc", "data/160/traj_2.xtc"],
                             path="data/160",
                             GPCR=True,
                             name="Inactive B2AR")
```

```
/home/frann/miniconda3/envs/alloviznew/lib/python3.9/site-packages/MDAnalysis/
↪coordinates/PDB.py:431: UserWarning: 1 A^3 CRYST1 record, this is usually a␣
↪placeholder. Unit cell dimensions will be set to None.
  warnings.warn("1 A^3 CRYST1 record,"
/home/frann/miniconda3/envs/alloviznew/lib/python3.9/site-packages/MDAnalysis/
↪coordinates/PDB.py:749: UserWarning: Unit cell dimensions not found. CRYST1 record set␣
↪to unitary values.
  warnings.warn("Unit cell dimensions not found. "
/home/frann/miniconda3/envs/alloviznew/lib/python3.9/site-packages/MDAnalysis/lib/util.
↪py:662: RuntimeWarning: Constructed NamedStream from a NamedStream
  warnings.warn("Constructed NamedStream from a NamedStream",
```

Network calculation

From all the available options, we are going to use correlationplus_CA_Pear as network construction method, which measures measures the correlation of the residues' alpha-carbons positions along the trajectory using the Pearson's correlation coefficient. We are going to pass 2 as the number of cores so that the network calculations of the two trajectory replicas run in parallel.

[3]:
```
activeB2AR.calculate(pkgs="correlationplus_CA_Pear", cores=2)
inactiveB2AR.calculate(pkgs="correlationplus_CA_Pear", cores=2)
```

```
correlationplus_CA_Pear
Please, make sure to correctly cite the package used to compute the network:␣
↪correlationplus (https://github.com/tekpinar/correlationplus#cite)
```

```
/home/frann/miniconda3/envs/alloviznew/lib/python3.9/site-packages/MDAnalysis/
↪coordinates/base.py:903: UserWarning: Reader has no dt information, set to 1.0 ps
  warnings.warn("Reader has no dt information, set to 1.0 ps")
```

```
adding raw data of correlationplus_CA_Pear for data/117/protein.pdb:  ['data/117/data/
↪correlationplus_CA_Pear/raw/1.pq', 'data/117/data/correlationplus_CA_Pear/raw/2.pq']
correlationplus_CA_Pear
```

```
/home/frann/miniconda3/envs/alloviznew/lib/python3.9/site-packages/MDAnalysis/
↪coordinates/PDB.py:431: UserWarning: 1 A^3 CRYST1 record, this is usually a␣
↪placeholder. Unit cell dimensions will be set to None.
  warnings.warn("1 A^3 CRYST1 record,"
```

```
Please, make sure to correctly cite the package used to compute the network:␣
↪correlationplus (https://github.com/tekpinar/correlationplus#cite)
```

```
/home/frann/miniconda3/envs/alloviznew/lib/python3.9/site-packages/MDAnalysis/
↪coordinates/PDB.py:431: UserWarning: 1 A^3 CRYST1 record, this is usually a␣
↪placeholder. Unit cell dimensions will be set to None.
  warnings.warn("1 A^3 CRYST1 record,"
/home/frann/miniconda3/envs/alloviznew/lib/python3.9/site-packages/MDAnalysis/
↪coordinates/base.py:903: UserWarning: Reader has no dt information, set to 1.0 ps
  warnings.warn("Reader has no dt information, set to 1.0 ps")
```

```
adding raw data of correlationplus_CA_Pear for data/160/protein.pdb:  ['data/160/data/
↪correlationplus_CA_Pear/raw/1.pq', 'data/160/data/correlationplus_CA_Pear/raw/2.pq']
```

[3]:
```
<AlloViz.Wrappers.correlationplus_w.correlationplus_CA_Pear at 0x7fd426da4280>
```

Network filtering

From all the available filters, we are going to choose "No_Sequence_Neighbors", which filters out residue pairs that are too close to each other in the protein sequence (less than 5 positions away in the sequence).

[4]:
```
activeB2AR.filter(filterings="No_Sequence_Neighbors")
inactiveB2AR.filter(filterings="No_Sequence_Neighbors")
```

[4]:
```
<AlloViz.AlloViz.Filtering.Filtering at 0x7fd426da4d60>
```

Network analysis

And finally, we are going to analyze the edges using the betweenness centrality metric ("btw").

[5]:
```
activeB2AR.analyze(elements="edges", metrics="btw")
inactiveB2AR.analyze(elements="edges", metrics="btw")
```

```
adding analyzed edges <AlloViz.Wrappers.correlationplus_w.correlationplus_CA_Pear object␣
↪at 0x7fd427582f70> No_Sequence_Neighbors data of for data/117/protein.pdb
adding analyzed edges <AlloViz.Wrappers.correlationplus_w.correlationplus_CA_Pear object␣
↪at 0x7fd426da4280> No_Sequence_Neighbors data of for data/160/protein.pdb
```

Delta-network construction

The Delta constructor performs the delta-network calculations automatically when supplied with two `Protein` objects.

[6]:
```
delta = AlloViz.Delta(activeB2AR, inactiveB2AR)
```

To find the corresponding nodes (and thus edges) to be subtracted between the two structures, open-source PyMOL is used to retrieve a sequence alignment from a structural superposition:

[7]:
```
print(delta._aln)
```

```
Alignment with 2 rows and 305 columns
QERDEVWVVGMGIVMSLIVLAIVFGNVLVITAIAKFERLQTVTN...LCL prot1
QERDEVWVVGMGIVMSLIVLAIVFGNVLVITAIAKFERLQTVTN...LR- prot2
```

[8]:
```
delta.view(pkg="correlationplus_CA_Pear", metric="btw", filtering="No_Sequence_Neighbors
↪", element="edges")
```

```
/home/frann/miniconda3/envs/alloviznew/lib/python3.9/site-packages/MDAnalysis/
↪coordinates/PDB.py:431: UserWarning: 1 A^3 CRYST1 record, this is usually a␣
↪placeholder. Unit cell dimensions will be set to None.
  warnings.warn("1 A^3 CRYST1 record,"
```

```
NGLWidget()
```

The global structure and communication of the two compared systems is expected to be alike (except for just a few conformational changes due to the perturbation), and thus most residue pair interactions will have similar weights. The calculation of the delta-network reduces this noise and helps to highlight the subtle differences between the two networks.

Each of the network's edge weights of one structure are subtracted from the corresponding edge weights of the first one. This connotes that, the more positive a value is in the resulting delta-network, the more important the edge is in the first structure (i.e., the edge has a significant weight in the first structure's network and, if its magnitude is preserved in the delta-network, it means that it was not as significant in the other structure's network and the subtraction did not quite affect it). Similarly, the more negative a value is in the delta-network, the more important the edge is in the other structure.

4.2.3 Use of NetworkX algorithms

[1]:
```
import AlloViz
```

System setup

We are going to use the structure and simulations of GPCR Beta-2 adrenergic receptor (B2AR) in the inactive form bound to carazolol (inverse agonist, 160). The network will be built with `PyInteraph2_Contacts`, which measures the distance-based residue contacts along the trajectory, and will be filtered with the `Spatially_distant` filter to leave out residue pairs too close to each other, with a minimum distance of 20 Angstroms.

[2]:
```
system = AlloViz.Protein(pdb="data/160/protein.pdb",
                         trajs="data/160/traj_1.xtc",
                         path="data/160")
```

```
/home/frann/miniconda3/envs/alloviznew/lib/python3.9/site-packages/MDAnalysis/
↪coordinates/PDB.py:431: UserWarning: 1 A^3 CRYST1 record, this is usually a␣
↪placeholder. Unit cell dimensions will be set to None.
  warnings.warn("1 A^3 CRYST1 record,"
```

[3]:
```
system.calculate(pkgs="PyInteraph2_Contacts")
```

```
PyInteraph2_Contacts
adding raw data of PyInteraph2_Contacts for data/160/protein.pdb:  ['data/160/data/
↪PyInteraph2_Contacts/raw/1.pq']
Please, make sure to correctly cite the package used to compute the network: PyInteraph2␣
↪(https://github.com/ELELAB/pyinteraph2)
```

[3]:
```
<AlloViz.Wrappers.PyInteraph2_w.PyInteraph2_Contacts at 0x7fbee7b8c8b0>
```

[4]:
```
system.filter(filterings="Spatially_distant", Interresidue_distance=20)
# the same as doing: system.PyInteraph2_Contacts.filter(filterings="Spatially_distant",␣
↪Interresidue_distance=20)
```

[4]:
```
<AlloViz.AlloViz.Filtering.Filtering at 0x7fbee8625a60>
```

Filtered networks are stored as NetworkX Graph objects and can be passed to NetworkX analysis functions.

[5]:
```
system.PyInteraph2_Contacts.Spatially_distant.graphs
```

[5]:
```
{'weight': <networkx.classes.graph.Graph at 0x7fbf740448e0>}
```

[6]:
```
graph = system.PyInteraph2_Contacts.Spatially_distant.graphs["weight"]
```

NetworkX

To use NetworkX's functions, we must specify that the edge weights are stored with the name "weight". For example, if we want to use NetworkX's shortest paths analysis:

[7]:
```
from networkx.algorithms.shortest_paths.generic import shortest_path
```

[8]:
```
paths = shortest_path(graph, weight="weight")
```

For example, to view all shortest paths of residue `ALA:134`:

[9]:
```
paths["ALA:134"]
```

[9]:
```
{'ALA:134': ['ALA:134'],
 'GLY:257': ['ALA:134', 'GLY:257'],
 'PRO:138': ['ALA:134', 'GLY:257', 'PRO:138'],
 'PHE:139': ['ALA:134', 'GLY:257', 'PHE:139'],
 'TYR:141': ['ALA:134', 'GLY:257', 'TYR:141'],
 'GLN:142': ['ALA:134', 'GLY:257', 'GLN:142'],
 'SER:143': ['ALA:134', 'GLY:257', 'SER:143'],
 'GLN:229': ['ALA:134', 'GLY:257', 'GLN:229'],
 'LEU:258': ['ALA:134', 'GLY:257', 'PHE:139', 'LEU:258'],
 'ARG:259': ['ALA:134', 'GLY:257', 'GLN:142', 'ARG:259'],
 'GLY:238': ['ALA:134', 'GLY:257', 'PRO:138', 'GLY:238'],
 'GLU:225': ['ALA:134', 'GLY:257', 'GLN:142', 'ARG:259', 'GLU:225'],
 'VAL:242': ['ALA:134', 'GLY:257', 'GLN:142', 'ARG:259', 'VAL:242'],
 'GLN:224': ['ALA:134', 'GLY:257', 'GLN:142', 'ARG:259', 'VAL:242', 'GLN:224'],
 'ARG:228': ['ALA:134', 'GLY:257', 'GLN:142', 'ARG:259', 'VAL:242', 'ARG:228'],
 'GLN:231': ['ALA:134', 'GLY:257', 'GLN:142', 'ARG:259', 'VAL:242', 'GLN:231'],
 'HIS:269': ['ALA:134', 'GLY:257', 'GLN:142', 'ARG:259', 'VAL:242', 'HIS:269'],
 'PHE:240': ['ALA:134',
  'GLY:257',
  'GLN:142',
  'ARG:259',
  'VAL:242',
  'ARG:228',
  'PHE:240'],
 'HIS:241': ['ALA:134',
  'GLY:257',
  'GLN:142',
  'ARG:259',
  'VAL:242',
  'ARG:228',
  'HIS:241'],
 'LEU:266': ['ALA:134',
  'GLY:257',
  'GLN:142',
  'ARG:259',
  'VAL:242',
  'ARG:228',
```

(continues on next page)

(continued from previous page)

```
 'PHE:240',
 'LEU:266'],
'LYS:227': ['ALA:134',
 'GLY:257',
 'GLN:142',
 'ARG:259',
 'VAL:242',
 'ARG:228',
 'HIS:241',
 'LYS:227']}
```

Alternative NetworkX's centrality analyses and visualization

AlloViz's `analyze` method allows to use any NetworkX analysis that returns a list of node values or edge values (e.g., node or edge centrality analyses), and its posterior visualization with the `view` method.

The absolute Python import of the analysis function with a custom short name for it must be passed as a dictionary to the `analyze` method. The default node_dict and edge_dict (with "btw" and "cfb") are already part of the Analysis module:

[10]:
```
AlloViz.AlloViz.Analysis.nodes_dict
```

[10]:
```
{'btw': 'networkx.algorithms.centrality.betweenness_centrality',
 'cfb': 'networkx.algorithms.centrality.current_flow_betweenness_centrality'}
```

Therefore, a new centrality analysis function for the nodes (e.g., load centrality) can be used with:

[11]:
```
system.analyze(elements="nodes", metrics="load", nodes_dict={"load": "networkx.
↪algorithms.centrality.load_centrality"})
```

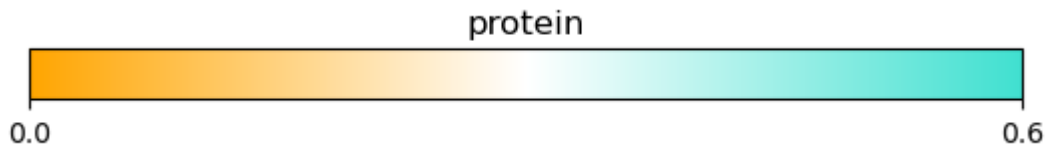
```
adding analyzed nodes <AlloViz.Wrappers.PyInteraph2_w.PyInteraph2_Contacts object at␣
↪0x7fbee7b8c8b0> Spatially_distant data of for data/160/protein.pdb
```

The load centrality of a node is the fraction of all shortest paths that pass through that node. The results can be visualized as a usual:

[12]:
```
system.view("PyInteraph2_Contacts", "load", "Spatially_distant", "nodes")
```

```
/home/frann/miniconda3/envs/alloviznew/lib/python3.9/site-packages/MDAnalysis/
↪coordinates/PDB.py:431: UserWarning: 1 A^3 CRYST1 record, this is usually a␣
↪placeholder. Unit cell dimensions will be set to None.
  warnings.warn("1 A^3 CRYST1 record,"
```

```
NGLWidget()
```



4.2.4 GUI

AlloViz can be run with a Graphical User Interface (GUI) that uses VMD (Visual Molecular Dynamics) for the visualization of the networks and analyses on the protein structure. VMD must be installed.

Note: The `json` library for VMD must be installed as well. It is already included in the alpha version of VMD 1.9.4.

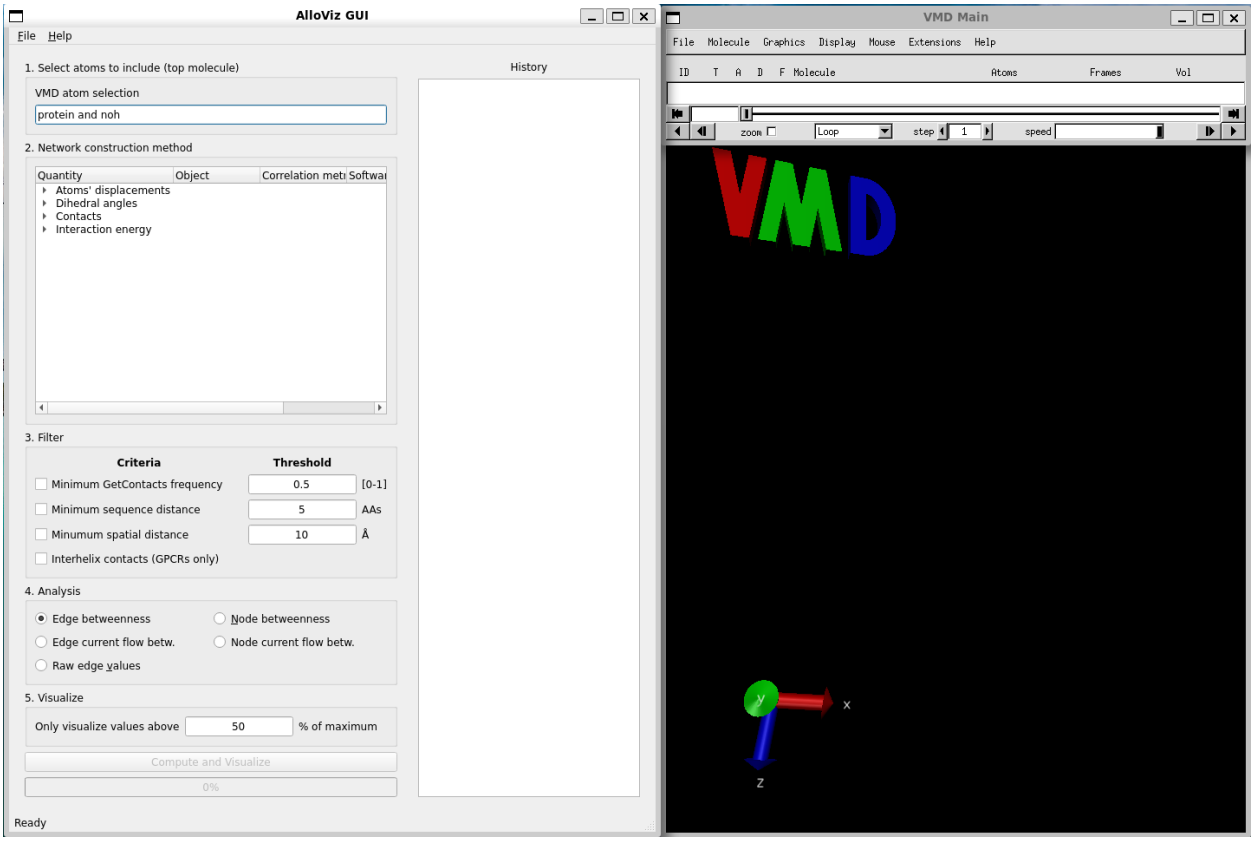
Alternatively, both stable VMD 1.9.3 and the `json` library can be installed in a conda environment: `conda install -c conda-forge --solver libmamba vmd tcllib`

`sqlite` 3.41 or higher might need to be installed in case of an `sqlite3`-related error when trying to launch the GUI.

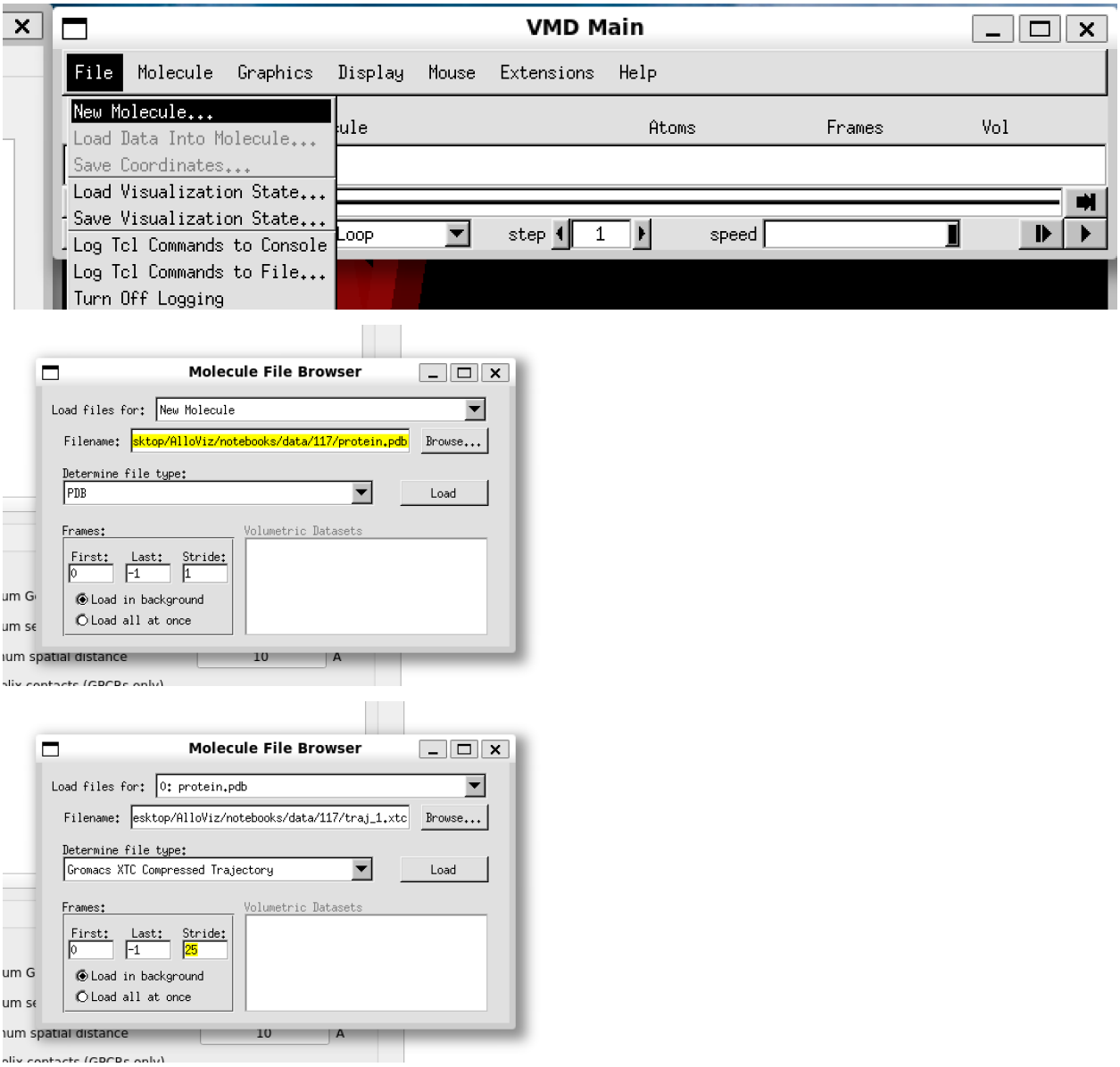
To launch the GUI, run the following command on an environment with AlloViz and VMD installed:

```
vmd -e gui/run_vmd.tcl
```

This is how the GUI looks like:



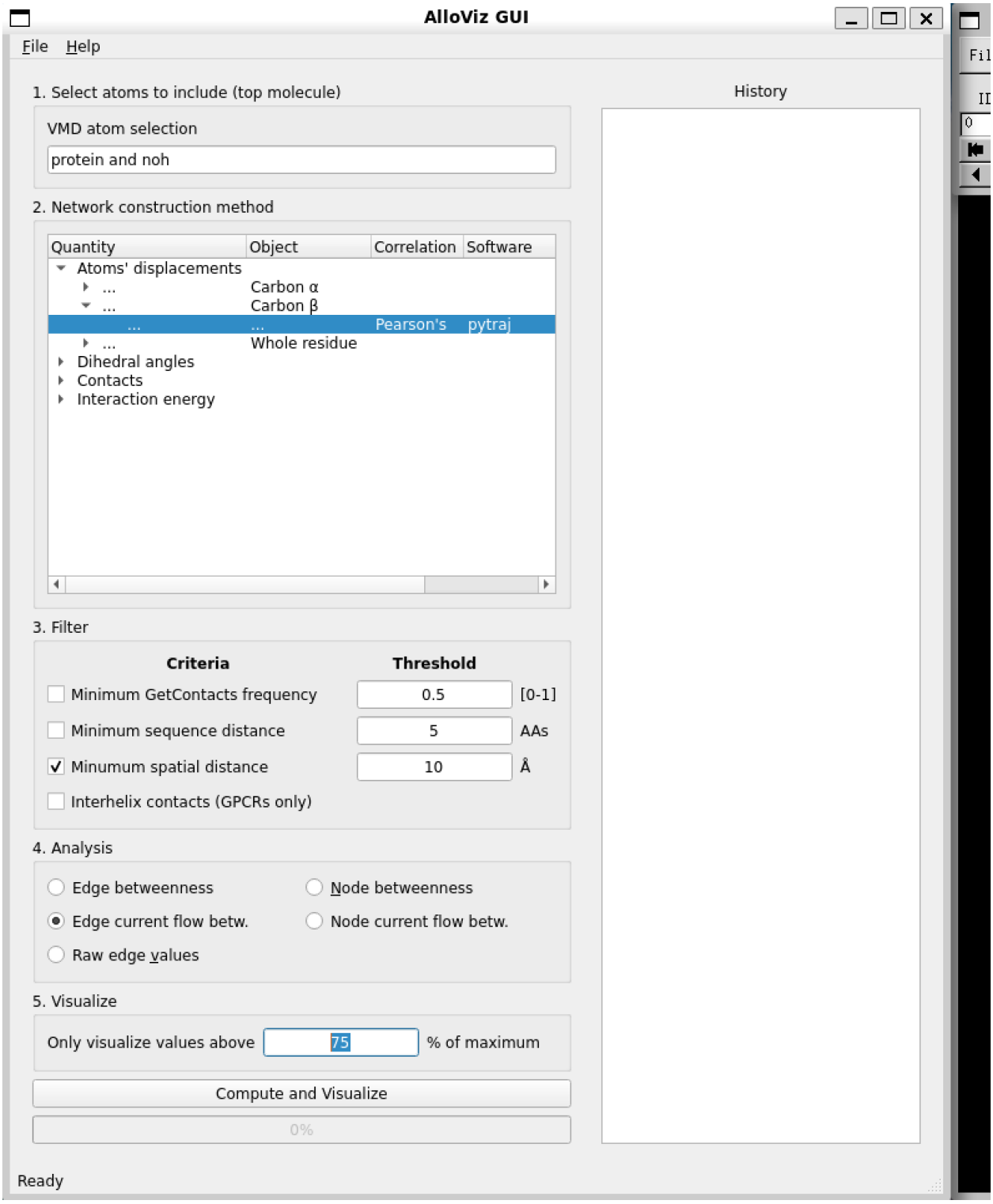
We can load one of the structures and trajectories that come with AlloViz's tutorial notebooks into VMD as usual. They are located in *AlloViz/tutorials/data/117* and they correspond to the GPCR Beta-2 adrenergic receptor in complex with agonist epinephrine (GPCRmd ID 117). We can use a stride of 25 frames for the trajectory to speed up the loading process.



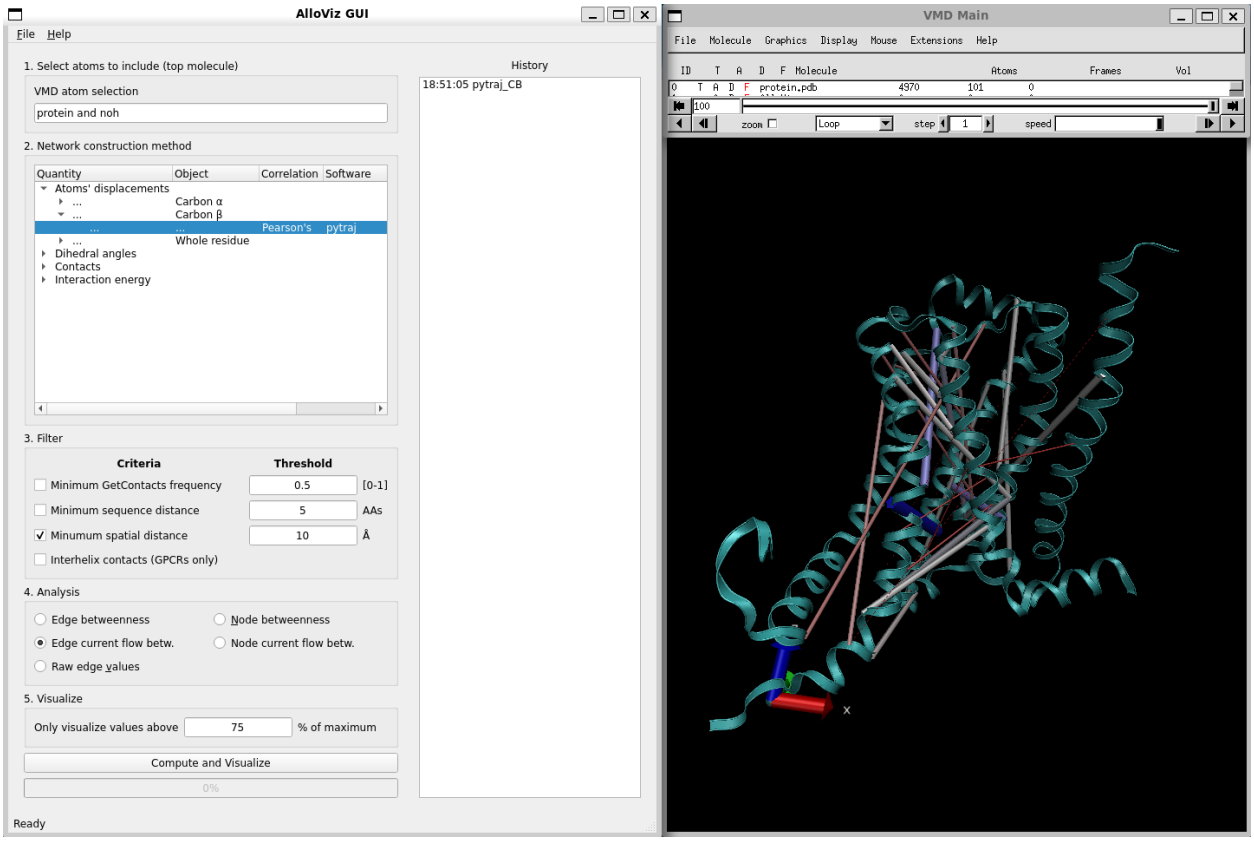
In the AlloViz GUI window we can choose the specific options we want to build, analyze and represent our network of choice:

1. The first step is to specify the atom selection to be taken into account for analysis. The trajectory provided with the notebooks is already processed and doesn't contain waters, ions or a ligand so we simply use the default selection of the whole protein.
2. Then, we choose the type of network to compute out of all the *options* AlloViz offers. In this case, we choose to measure the Pearson's correlation of the residues' beta-carbons using the package *pytraj*.
3. We choose to filter the computed network by only keeping residue pairs that are further apart than 10 Ångstroms in the structure.
4. Finally, we pick to analyze the networks' edges using the current-flow betweenness centrality metric.

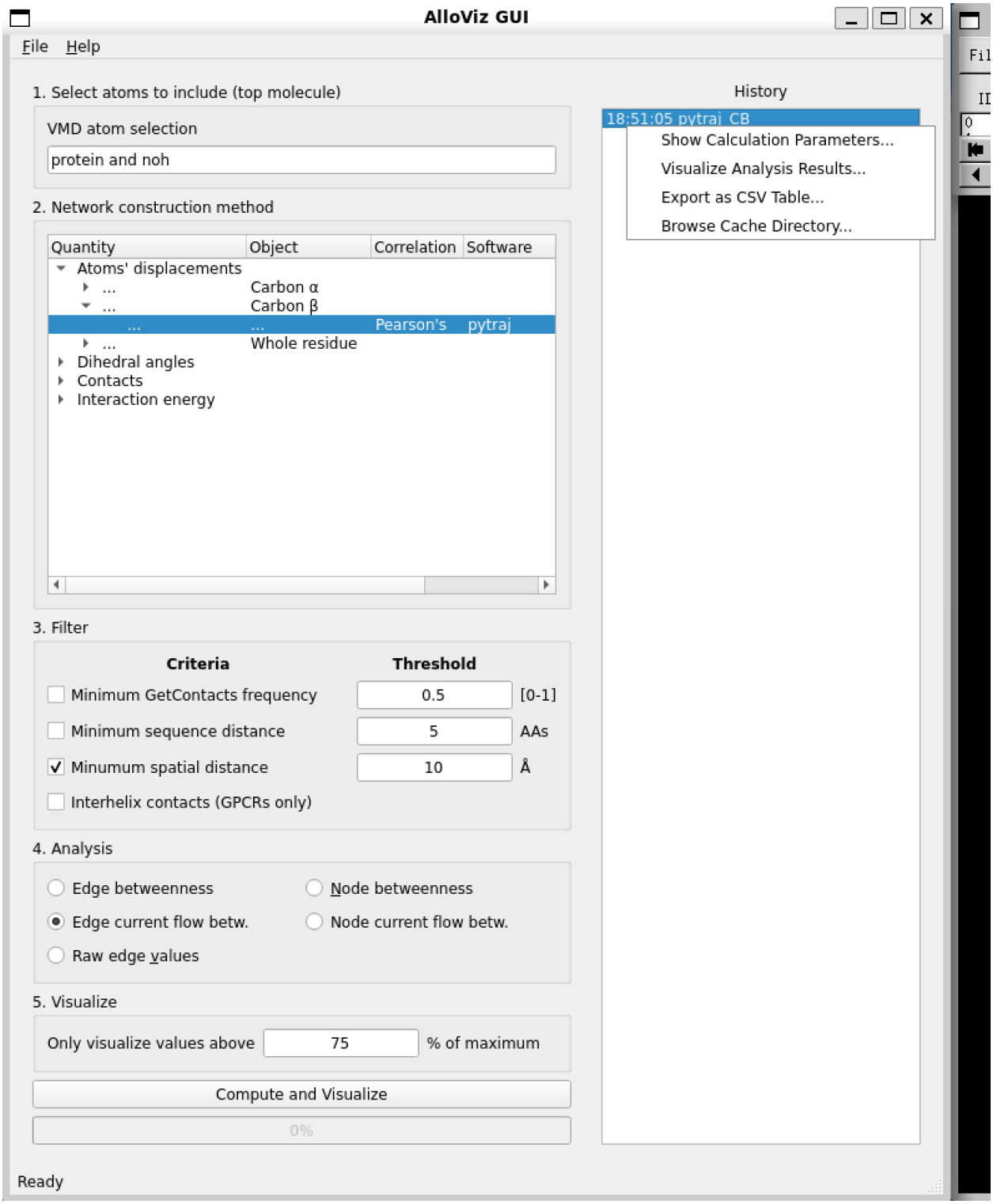
5. And choose to represent on the structure in the VMD visualizer the top edges with a value above 75% of the maximum value.



After clicking "Compute and Visualize", the selected options are represented on the protein structure on the VMD visualization window (we have changed the representation from the default to Cartoon):



Moreover, the calculated network and all the following analyses we might perform are logged in the right panel of the AlloViz VMD window, and we can get back to a previous analysis or even save the data in *.csv* format by right-clicking on the analysis and selecting the desired option:



Quickstart	Basic workflow of AlloViz
Delta network	Calculation of the delta-network of two proteins
Use of NetworkX algorithms	Use of other NetworkX analyses
Graphical User Interface	Tutorial of the AlloViz GUI with VMD

4.3 API reference

The main class *Protein* processes the structure and trajectory(ies) files, and allows to calculate, analyze and visualize the resulting networks.

The *Delta* class allows to calculate the delta-network(s) between two Proteins.

Classes

AlloViz.Protein([pdb, trajs, GPCR, name, ...])	AlloViz main class.
AlloViz.Delta(refstate, state2[, pymol_aln])	AlloViz class for calculating a delta-network.

4.3.1 AlloViz.Protein

class AlloViz.**Protein**(*pdb=''*, *trajs=[]*, *GPCR=False*, *name=None*, *path=None*, *protein_sel=None*, ***kwargs*)

Bases: object

AlloViz main class.

Objects of the class process the input structure and trajectory files and allow to *calculate()* the different networks, which are stored in classes of the *AlloViz.Wrappers* module and can be filtered, analyzed and visualized with subsequent methods.

Parameters

pdb
[str] Filename of the PDB structure to read, process and use.

trajs
[str or list] Filename(s) of the MD trajectory (or trajectories) to read and use. File format must be recognized by MDAnalysis (e.g., xtc).

GPCR
[bool or int, default: False] Use *True* if the structure is a GPCR, or pass the ID of a GPCRmd database dynamics entry, without specifying the *pdb* nor the *trajs* parameters, to automatically retrieve the files from the database and process them. They will be downloaded to *path*, which if left undefined will default to the GPCRmd ID.

name
[str, default: "protein"] Name string to be used, e.g., for the title of the colorbar shown when representing a network with nglviewer.

path
[str, optional] Path to store results in. It can exist already or not, and a new folder inside it called *data* will be created to store computation results and other files. If unspecified, it defaults to "." or the GPCRmd ID in case it is used.

protein_sel
[str, default: AlloViz.Protein._protein_sel] MDAnalysis atom selection string to select the protein structure from the Universe (e.g., in case simulations in biological conditions are used, to avoid selecting extra chains, water molecules, ions…). It defaults to "(same segid as protein) and (not segid LIG) and (not chainid L)" and it can be extended using, e.g., AlloViz.Protein._protein_sel + *" and {customselection}"*.

Other Parameters

psf
[str, optional] Optional kwarg of the filename of the .psf file corresponding to the pdb used. It is required alongside the *parameters* file to use the gRINN network construction method.

parameters
[str, optional] Optional kwarg of the filename of the MD simulation force-field parameters file in NAMD format. It is required alongside the *psf* file to use the gRINN network construction method.

special_res
[dict, optional] Optional kwarg of a dictionary containing a mapping of special residue 3/4-letter code(s) present in the structure to the corresponding standard 1-letter code(s).

Raises

FileNotFoundError
If any of the files passed in the *pdb* and *traj* (and *psf* and *parameters* if provided) parameters cannot be accessed.

See also:

AlloViz.Delta
Class for calculation of the delta-network(s) between two Protein objects.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> print(opioidGPCR.u)
<Universe with 88651 atoms>
```

Attributes

pdb
trajs
GPCR
protein
[Universe] Universe of the processed pdb with only the selected *protein_sel* atoms.

u
[Universe] Universe of the processed pdb and trajectory files with only the *protein_sel* atoms.

Methods

analyze([pkgs, filterings, elements, ...])	Analyzed filtered network
calculate([pkgs, cores])	Calculate rwa edge weights of allosteric networks.
filter([pkgs, filterings, ...])	Filter network edges
view(pkg, metric[, filtering, element, num, ...])	Visualize the analyzed networks on the protein structure.

AlloViz.Protein.analyze

Protein.**analyze**(*pkgs='all'*, *filterings='all'*, *elements='edges'*, *metrics='all'*, *cores=1*, *nodes_dict={'btw': 'networkx.algorithms.centrality.betweenness_centrality', 'cfb': 'networkx.algorithms.centrality.current_flow_betweenness_centrality'}*, *edges_dict={'btw': 'networkx.algorithms.centrality.edge_betweenness_centrality', 'cfb': 'networkx.algorithms.centrality.edge_current_flow_betweenness_centrality'}*, ***kwargs*)

Analyzed filtered network

Analyze the selected (un)filtered networks with the passed elements-metrics. It calls *AlloViz.AlloViz.Analysis.analyze()* and results are stored in instances of classes from the *AlloViz.AlloViz.Elements* module, which extend the pandas.DataFrame class.

Parameters

pkgs
[str or list, default: "all"] Package(s)/Network construction method(s) for which to analyze their raw edge weights, which must be already calculated and their data saved as instance attribute. In this case, "all" sends the computation for all available methods that are already calculated and saved as instance attributes.

filterings
[str or list, default: "all"] Filtering scheme(s) for which to perform the analyses, which must exist already for the selected packages. "all" sends the computation for all available schemes that are already saved.

elements
[str or list, {"edges", "nodes"}] Network elements for which to perform the analysis.

metrics
[str or list, default: "all"] Network metrics to compute, which must be keys in the *nodes_dict* or *edges_dict* dictionaries. Default is "all" and it sends the computation for all the metrics defined in the corresponding dictionary of the selected elements in *element*.

cores
[int, default: 1] Number of cores to use for parallelization with a *multiprocess* Pool. Default value only uses 1 core with a custom *AlloViz.utils.dummypool* that performs computations synchronously.

Other Parameters

nodes_dict, edges_dict
[dict] Optional kwarg(s) of the dictionary(ies) that maps network metrics custom names (e.g., betweenness centrality, "btw") with their corresponding NetworkX function (e.g., "networkx.algorithms.centrality.betweenness_centrality"). Functions strings must be written as if they were absolute imports, and must return a dictionary of edges or nodes, depending on the element dictionary in which they are. The keys of the dictionaries will be used to name the columns of the analyzed data that the functions produce. Defaults are *nodes_dict* and *edges_dict*.

****kwargs**
Other optional keyword arguments that will be passed to the NetworkX analysis function(s) that is(are) used on the method call in case they need extra parameters. All keyword arguments will be passed to all analysis function calls, so if the function doesn't accept the arguments there will be an error. *weight* parameter is already specified by AlloViz.

See also:

AlloViz.AlloViz.Analysis
Module with analysis functions.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.filter
Class method to filter the network(s) raw edge weights with different criteria.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.filter("dynetan", "GetContacts_edges")
>>> opioidGPCR.analyze("dynetan", "GetContacts_edges", "nodes", "btw")
<AlloViz.AlloViz.Elements.Nodes at 0x7f892c3c0fa0>
```

AlloViz.Protein.calculate

Protein.**calculate**(*pkgs='all'*, *cores=1*, ***kwargs*)

Calculate rwa edge weights of allosteric networks.

Send the computation of the raw edge weights for the selected network construction methods.

Parameters

pkgs
[str or list, default: "all"] Package(s)/Network construction method(s) for which to send raw edge weight computation. "all" sends the computation for all available methods within AlloViz (check *AlloViz.AlloViz.utils.pkgs*).

cores
[int, default: 1] Number of cores to use for parallelization with a *multiprocess* Pool. Default value only uses 1 core with a custom *AlloViz.utils.dummypool* that performs computations synchronously.

Other Parameters

taskcpus
[int, optional] Optional kwarg to specify the amount of cores that parallelizable network construction methods can use (i.e., AlloViz's method, getcontacts, dynetan, PyInteraph, MDEntropy and gRINN).

stride
[int, optional] Optional kwarg to specify the striding to be done on the trajectory(ies) for computationally-expensive network construction methods: i.e., dynetan and AlloViz's own method. Default is no striding.

namd
[str, optional] Optional kwarg pointing to the namd2 executable location; if the *namd* command is accessible through the CLI it is automatically retrieved with the *distutils* package.

chis
[int, optional] Optional kwarg to specify the number of side-chain chi dihedral angles (up to 5) to combine when sending the calculation of a child of the Combined_Dihs Wrappers' base class that includes chi dihedrals in its calculation.

MDEntropy_method
[str, optional, {"knn", "grassberger", "chaowangjost"}] Optional kwarg to specify the method to calculate the entropy of the variables for Mutual Information estimation when using one of the MDEntropy network construction methods (default: "grassberger").

See also:

AlloViz.Wrappers.Base.Base
Base class to launch and store calculation results.

AlloViz.Protein.filter
Class method to filter the network raw edge weights with different criteria.

AlloViz.Protein.analyze
Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Notes

Calculation results are stored as new instance attributes with the same name as the selected packages/network construction methods. If the object is created providing more than one trajectory file, the average and standard error of the weights between the replicas are also calculated.

Examples

If we have a 6-core computer and want to use 2 cores to compute the values for each of the three trajectories of the protein (e.g., GPCRmd stores three replicas of each structure):

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate("dynetan", cores=6, taskcpus=2)
>>> print(opioidGPCR.dynetan.raw.shape)
(41041, 5)
```

AlloViz.Protein.filter

Protein.**filter**(*pkgs='all'*, *filterings='all'*, ***, *GetContacts_threshold=0*, *Sequence_Neighbor_distance=5*, *Interresidue_distance=10*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

pkgs
[str or list, default: "all"] Package(s)/Network construction method(s) for which to analyze their raw edge weights, which must be already calculated and their data saved as instance attribute. In this case, "all" sends the computation for all available methodsthat are already calculated and saved as instance attributes.

filterings
[str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists. A list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.filter("dynetan", ["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Protein.view

Protein.**view**(*pkg*, *metric*, *filtering='All'*, *element='edges'*, *num: int = 20*, *colors: list = ['orange', 'turquoise']*, *nv=None*)

Visualize the analyzed networks on the protein structure.

Visualize the network corresponding to the selected package/network construction method, filtering scheme, network element and network metric or metrics, all of which must be previously calculated and analyzed. The number of (each of the) elements to show and the color scale can be specified. It calls *AlloViz.Elements.Element.view()*.

Parameters

pkg
[str] Package/Network construction method for which to show the network.

metric
[str] Network metric for which to show the network.

filtering
[str] Filtering scheme for which to show the network.

element
[str or list, {"edges", "nodes"}] Network element or elements to show on the protein structure representing the chosen package, filtering scheme and metric.

num
[int, default: 20] Number of (each of the) network elements to show on the structure.

colors
[list, default: ["orange", "turquoise"]] List of two colors to assign to the minimum and maximum values of the network to be represented, respectively. Middle value is assigned "white" and it will be the mean of the network values or 0 if the network has both negative and positive values.

nv
[nglview.NGLWidget, optional] A structure representation into which the shapes representing the chosen network elements will be added.

Returns

nglview.NGLWidget

See also:

AlloViz.AlloViz.Elements
Module with classes for storage and visualization of filtered and analyzed networks.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.filter
Class method to filter the network(s) raw edge weights with different criteria.

AlloViz.Protein.analyze
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate("dynetan", cores=6, taskcpus=2)
>>> opioidGPCR.filter("dynetan", "GetContacts_edges")
>>> opioidGPCR.analyze("dynetan", "GetContacts_edges", "nodes", "btw")
>>> opioidGPCR.view("dynetan", "btw", "GetContacts_edges", element=["edges",
↪"nodes"])
NGLWidget()
```

4.3.2 AlloViz.Delta

class AlloViz.**Delta**(*refstate*, *state2*, *pymol_aln='super'*)

Bases: `object`

AlloViz class for calculating a delta-network.

Used to calculate the delta-network between two *Protein* objects, using all the available combinations of packages/network construction methods, filterings, and elements-metrics that they have in common. A structural alignment of the structures is performed with PyMOL with the selected method to find the corresponding residues between the two structures for subtraction of edge weights.

Parameters

refstate
[*AlloViz.Protein*] Object of the *AlloViz.Protein* class to use as reference (values of the other *AlloViz.Protein* object will be subtracted from this one's values).

state2
[*AlloViz.Protein*] Object of the *AlloViz.Protein* class to compare with the reference one to build the delta-network (this one's values will be subtracted from the reference one's values).

pymol_aln
[{"super", "align", "cealign"}, default: "super"] PyMOL's method to apply for the structural alignment of the two structures, used to retrieve the corresponding residues between the two for subtraction of edge weights. It is performed by `AlloViz.Delta._make_struct_aln()`.

See also:

AlloViz.Protein
AlloViz main class for calculation of protein allosteric communication networks.

Notes

For PyMOL's structural alignment, *align* starts from a sequence alignment and is optimal to align structures with identical sequences, but is deeply affected by sequence differences. *super* performs a sequence-independent dynamic programming structural alignment and it works well for structures with low sequence identity. *cealign* uses the Combinatorial Extension (CE) algorithm and it is preferred for structures with little to no sequence similarity (twilight zone). The methods are described in PyMOL's documentation.

Examples

```
>>> activeB2AR = AlloViz.Protein(GPCR=117, name="Active B2AR")
>>> inactiveB2AR = AlloViz.Protein(GPCR=160, name="Inactive B2AR")
>>> for protein in [activeB2AR, inactiveB2AR]:
>>>     protein.calculate("pytraj_CA")
>>>     protein.analyze(metrics="btw")
>>> delta = AlloViz.Delta(activeB2AR, inactiveB2AR)
>>> print(delta.pytraj_CA.Whole.edges.df.shape)
(40690, 5)
```

Attributes

refstate
state2

Methods

view(pkg, metric[, filtering, element, num, ...])	Visualize the analyzed delta-networks on the protein structure.

AlloViz.Delta.view

Delta.**view**(*pkg*, *metric*, *filtering='All'*, *element='edges'*, *num=20*, *colors=['orange', 'turquoise']*, *nv=None*)

Visualize the analyzed delta-networks on the protein structure.

Visualize the delta-network corresponding to the selected package/network construction method, filtering scheme, network element and network metric or metrics, all of which must be already present in the delta-network object by having been previously calculated and analyzed in the respective Protein objects, and then correctly processed during the Delta object creation. Structure of the reference protein is used for displaying by default. The number of (each of the) elements to show and the color used for each Protein can be specified (first color will be used for negative values and thus for the second Protein, while the second color will be used for positive values and thus the reference Protein).

Parameters

pkg
[str, default: "all"] Package/Network construction method for which to show the delta-network.

metric
[str, default: "all"] Network metric for which to show the delta-network.

filtering
[str] Filtering scheme for which to show the delta-network.

element
[str or list, {"edges", "nodes"}] Delta-network element or elements to show on the protein structure representing the chosen package, filtering scheme and metric.

num
[int, default: 20] Number of (each of the) delta-network elements to show on the structure.

colors
[list, default: ["orange", "turquoise"]] List of two colors to assign to the minimum and maximum values of the delta-network to be represented, respectively. Abstractly, the first color will be used for negative values and thus for elements that have higher value in the second Protein, while the second color will be used for positive values, and thus for the reference Protein. Dela-network values are interpolated setting 0 as the middle value, which is assigned "white".

nv
[nglview.NGLWidget, optional] A structure representation into which the shapes representing the chosen delta-network elements will be added.

Returns

nglview.NGLWidget

Examples

```
>>> activeB2AR = AlloViz.Protein(GPCR=117, name="Active B2AR")
>>> inactiveB2AR = AlloViz.Protein(GPCR=160, name="Inactive B2AR")
>>> for protein in [activeB2AR, inactiveB2AR]:
>>>     protein.calculate("pytraj_CA")
>>>     protein.analyze(metrics="btw")
>>> delta = AlloViz.Delta(activeB2AR, inactiveB2AR)
>>> delta.view("pytraj_CA", "btw")
NGLWidget()
```

4.4 Complete API

AlloViz	AlloViz package

4.4.1 AlloViz

AlloViz package

The main classes are imported into the namespace: *Protein* and *Delta*.

AlloViz.AlloViz contains the rest of modules used by *Protein* and *Delta* to process information and send calculations and analyses and produce representations.

AlloViz.Wrappers contains the classes that wrap the code of the packages/network construction methods available in AlloViz, to send their calculations through the AlloViz code without needing to execute them independently, and process their results to store them in AlloViz objects.

Modules

AlloViz.AlloViz	Main modules used by AlloViz's to process/compute information
AlloViz.Wrappers	Wrappers of AlloViz's network construction methods/packages

AlloViz.AlloViz

Main modules used by AlloViz's to process/compute information

Contains the rest of modules used by *Protein* and *Delta* to process information and send calculations and analyses and produce representations.

Modules

AlloViz.AlloViz.Analysis	Module with functions to analyze filtered networks
AlloViz.AlloViz.Classes	Main AlloViz classes: *Protein* and *Delta*
AlloViz.AlloViz.Elements	Module with classes to store and represent analyzed network elements
AlloViz.AlloViz.Filtering	Module with classes to filter and store networks
AlloViz.AlloViz.info	Module containing variables and elements with AlloViz information
AlloViz.AlloViz.trajutils	Module containing functions for trajectory processing
AlloViz.AlloViz.utils	Module containing helper functions and variables used by many other modules

AlloViz.AlloViz.Analysis

Module with functions to analyze filtered networks

Main function *analyze()* manages the analysis of filtered networks. It calls *single_analysis()* for the analysis of single element-metric combinations and uses the NetworkX's functions defined in *nodes_dict* and *edges_dict*.

Module Attributes

nodes_dict	Dictionary that maps nodes network metrics custom names (e.g., betweenness centrality, "btw") with their corresponding NetworkX function (e.g., "networkx.algorithms.centrality.betweenness_centrality").
edges_dict	Dictionary that maps edges network metrics custom names (e.g., betweenness centrality, "btw") with their corresponding NetworkX function (e.g., "networkx.algorithms.centrality.edge_betweenness_centrality").

AlloViz.AlloViz.Analysis.nodes_dict

AlloViz.AlloViz.Analysis.**nodes_dict = {'btw': 'networkx.algorithms.centrality.betweenness_centrality', 'cfb': 'networkx.algorithms.centrality.current_flow_betweenness_centrality'}**

Dictionary that maps nodes network metrics custom names (e.g., betweenness centrality, "btw") with their corresponding NetworkX function (e.g., "networkx.algorithms.centrality.betweenness_centrality").

AlloViz.AlloViz.Analysis.edges_dict

AlloViz.AlloViz.Analysis.**edges_dict = {'btw': 'networkx.algorithms.centrality.edge_betweenness_centrality', 'cfb': 'networkx.algorithms.centrality.edge_current_flow_betweenness_centrality'}**

Dictionary that maps edges network metrics custom names (e.g., betweenness centrality, "btw") with their corresponding NetworkX function (e.g., "networkx.algorithms.centrality.edge_betweenness_centrality").

Functions

add_data(pqs, elem, data, filtered)	
analyze(filtered, elements, metrics, ...)	Analyze the filtered network
analyze_graph(args)	Analyze a graph/column from raw filtered data with an element-metric
single_analysis(graphs, metricf, metric, ...)	Analyze raw data with a single element-metric
wait_analyze(pqs)	

AlloViz.AlloViz.Analysis.add_data

AlloViz.AlloViz.Analysis.**add_data**(*pqs*, *elem*, *data*, *filtered*)

AlloViz.AlloViz.Analysis.analyze

AlloViz.AlloViz.Analysis.**analyze**(*filtered*, *elements*, *metrics*, *nodes_dict*, *edges_dict*, ***kwargs*)

Analyze the filtered network

Send the analyses of the passed filtered network for the specified combinations of elements-metrics. Each combination is analyzed independently with *single_analysis()* using NetworkX' functions and results are stored as new instances of classes from the *AlloViz.AlloViz.Elements* module, which extend the pandas.DataFrame class.

Parameters

filtered
[*Filtering* object] Filtered network object.

elements
[str or list, {"edges", "nodes"}] Network element for which to perform the analysis.

metrics
[str or list] Network metrics to compute, which must be keys in the *nodes_dict* or *edges_dict* dictionaries.

Other Parameters

nodes_dict, edges_dict
[dict] Optional kwarg(s) of the dictionary(ies) that maps network metrics custom names (e.g., betweenness centrality, "btw") with their corresponding NetworkX function (e.g., "networkx.algorithms.centrality.betweenness_centrality"). Functions strings must be written as if they were absolute imports, and must return a dictionary of edges or nodes, depending on the element dictionary in which they are. The keys of the dictionaries will be used to name the columns of the analyzed data that the functions produce. Defaults are `nodes_dict` and `edges_dict`.

****kwargs**
Other optional keyword arguments that will be passed to the NetworkX analysis function(s) that is(are) used on the method call in case they need extra parameters.

AlloViz.AlloViz.Analysis.analyze_graph

`AlloViz.AlloViz.Analysis.`**`analyze_graph`**`(`*`args`*`)`

Analyze a graph/column from raw filtered data with an element-metric

Analyze a stored, filtered raw data column with the passed combination of element-metric/NetworkX' analysis function and return the results.

Parameters

graph
[Graph object] Single column/graph object to analyze.

metricf
[str] NetworkX function to analyze data. It must be written as if it were an absolute import (e.g., "networkx.algorithms.centrality.betweenness_centrality").

colname
[str] Name of the analyzed column that it will have in the final DataFrame for saving.

AlloViz.AlloViz.Analysis.single_analysis

`AlloViz.AlloViz.Analysis.`**`single_analysis`**`(`*`graphs`*`, `*`metricf`*`, `*`metric`*`, `*`elem`*`, `*`pq`*`, `*`**kwargs`*`)`

Analyze raw data with a single element-metric

Analyze stored, filtered raw data with the passed combination of element-metric/NetworkX' analysis function and save the results.

Parameters

graphs
[dict of external:ref:*Graph <graph>* objects] Graphs to analyze.

metricf
[str] NetworkX function to analyze data. It must be written as if it were an absolute import (e.g., "networkx.algorithms.centrality.betweenness_centrality").

metric
[str] Network metric to compute, which must be a key in the *nodes_dict* or *edges_dict* dictionaries.

elem
[str or list, {"edges", "nodes"}] Network element for which the analysis is performed.

pq
[str] Name of the parquet (.pq) file in which to save the analysis results.

Other Parameters

****kwargs**
Other optional keyword arguments that will be passed to the NetworkX analysis function(s) that is(are) used on the method call in case they need extra parameters.

AlloViz.AlloViz.Analysis.wait_analyze

`AlloViz.AlloViz.Analysis.`**`wait_analyze`**`(`*`pqs`*`)`

AlloViz.AlloViz.Classes

Main AlloViz classes: *Protein* and *Delta*

The *Protein* class is AlloViz's main class, processing a structure and trajectory(ies) input files and allowing to calculate, analyze and visualize the allosteric communication networks with its associated methods.

The *Delta* class takes two *Protein* objects as input to calculate the delta-network to highlight the differences in the allosteric communication between two systems.

Notes

This module is not meant to be used directly, as the classes are imported in the namespace of AlloViz itself.

Classes

Delta(refstate, state2[, pymol_aln])	AlloViz class for calculating a delta-network.
Protein([pdb, trajs, GPCR, name, path, ...])	AlloViz main class.

AlloViz.AlloViz.Classes.Delta

class `AlloViz.AlloViz.Classes.`**`Delta`**`(`*`refstate`*, *`state2`*, *`pymol_aln='super'`*`)`

Bases: `object`

AlloViz class for calculating a delta-network.

Used to calculate the delta-network between two *Protein* objects, using all the available combinations of packages/network construction methods, filterings, and elements-metrics that they have in common. A structural alignment of the structures is performed with PyMOL with the selected method to find the corresponding residues between the two structures for subtraction of edge weights.

Parameters

refstate
[*AlloViz.Protein*] Object of the *AlloViz.Protein* class to use as reference (values of the other *AlloViz.Protein* object will be subtracted from this one's values).

state2
[*AlloViz.Protein*] Object of the *AlloViz.Protein* class to compare with the reference one to build the delta-network (this one's values will be subtracted from the reference one's values).

pymol_aln
[{"super", "align", "cealign"}, default: "super"] PyMOL's method to apply for the structural alignment of the two structures, used to retrieve the corresponding residues between the two for subtraction of edge weights. It is performed by `AlloViz.Delta._make_struct_aln()`.

See also:

AlloViz.Protein
AlloViz main class for calculation of protein allosteric communication networks.

Notes

For PyMOL's structural alignment, *align* starts from a sequence alignment and is optimal to align structures with identical sequences, but is deeply affected by sequence differences. *super* performs a sequence-independent dynamic programming structural alignment and it works well for structures with low sequence identity. *cealign* uses the Combinatorial Extension (CE) algorithm and it is preferred for structures with little to no sequence similarity (twilight zone). The methods are described in PyMOL's documentation.

Examples

```
>>> activeB2AR = AlloViz.Protein(GPCR=117, name="Active B2AR")
>>> inactiveB2AR = AlloViz.Protein(GPCR=160, name="Inactive B2AR")
>>> for protein in [activeB2AR, inactiveB2AR]:
>>>     protein.calculate("pytraj_CA")
>>>     protein.analyze(metrics="btw")
>>> delta = AlloViz.Delta(activeB2AR, inactiveB2AR)
>>> print(delta.pytraj_CA.Whole.edges.df.shape)
(40690, 5)
```

Attributes

refstate
state2

Methods

view(pkg, metric[, filtering, element, num, ...])	Visualize the analyzed delta-networks on the protein structure.

AlloViz.AlloViz.Classes.Delta.view

Delta.**view**(*pkg*, *metric*, *filtering='All'*, *element='edges'*, *num=20*, *colors=['orange', 'turquoise']*, *nv=None*)

Visualize the analyzed delta-networks on the protein structure.

Visualize the delta-network corresponding to the selected package/network construction method, filtering scheme, network element and network metric or metrics, all of which must be already present in the delta-network object by having been previously calculated and analyzed in the respective Protein objects, and then correctly processed during the Delta object creation. Structure of the reference protein is used for displaying by default. The number of (each of the) elements to show and the color used for each Protein can be specified (first color will be used for negative values and thus for the second Protein, while the second color will be used for positive values and thus the reference Protein).

Parameters

pkg
[str, default: "all"] Package/Network construction method for which to show the delta-network.

metric
[str, default: "all"] Network metric for which to show the delta-network.

filtering
[str] Filtering scheme for which to show the delta-network.

element
[str or list, {"edges", "nodes"}] Delta-network element or elements to show on the protein structure representing the chosen package, filtering scheme and metric.

num
[int, default: 20] Number of (each of the) delta-network elements to show on the structure.

colors
[list, default: ["orange", "turquoise"]] List of two colors to assign to the minimum and maximum values of the delta-network to be represented, respectively. Abstractly, the first color will be used for negative values and thus for elements that have higher value in the second Protein, while the second color will be used for positive values, and thus for the reference Protein. Dela-network values are interpolated setting 0 as the middle value, which is assigned "white".

nv
[nglview.NGLWidget, optional] A structure representation into which the shapes representing the chosen delta-network elements will be added.

Returns

nglview.NGLWidget

Examples

```
>>> activeB2AR = AlloViz.Protein(GPCR=117, name="Active B2AR")
>>> inactiveB2AR = AlloViz.Protein(GPCR=160, name="Inactive B2AR")
>>> for protein in [activeB2AR, inactiveB2AR]:
>>>     protein.calculate("pytraj_CA")
>>>     protein.analyze(metrics="btw")
>>> delta = AlloViz.Delta(activeB2AR, inactiveB2AR)
>>> delta.view("pytraj_CA", "btw")
NGLWidget()
```

AlloViz.AlloViz.Classes.Protein

class AlloViz.AlloViz.Classes.**Protein**(*pdb=''*, *trajs=[]*, *GPCR=False*, *name=None*, *path=None*, *protein_sel=None*, ***kwargs*)

Bases: `object`

AlloViz main class.

Objects of the class process the input structure and trajectory files and allow to *calculate()* the different networks, which are stored in classes of the *AlloViz.Wrappers* module and can be filtered, analyzed and visualized with subsequent methods.

Parameters

pdb
[str] Filename of the PDB structure to read, process and use.

trajs
[str or list] Filename(s) of the MD trajectory (or trajectories) to read and use. File format must be recognized by MDAnalysis (e.g., xtc).

GPCR
[bool or int, default: False] Use *True* if the structure is a GPCR, or pass the ID of a GPCRmd database dynamics entry, without specifying the *pdb* nor the *trajs* parameters, to automatically retrieve the files from the database and process them. They will be downloaded to *path*, which if left undefined will default to the GPCRmd ID.

name
[str, default: "protein"] Name string to be used, e.g., for the title of the colorbar shown when representing a network with nglviewer.

path
[str, optional] Path to store results in. It can exist already or not, and a new folder inside it called *data* will be created to store computation results and other files. If unspecified, it defaults to "." or the GPCRmd ID in case it is used.

protein_sel
[str, default: `AlloViz.Protein._protein_sel`] MDAnalysis atom selection string to select the protein structure from the Universe (e.g., in case simulations in biological conditions are used, to avoid selecting extra chains, water molecules, ions…). It defaults to "(same segid as protein) and (not segid LIG) and (not chainid L)" and it can be extended using, e.g., `AlloViz.Protein._protein_sel` + *" and {customselection}"*.

Other Parameters

psf
[str, optional] Optional kwarg of the filename of the .psf file corresponding to the pdb used. It is required alongside the *parameters* file to use the gRINN network construction method.

parameters
[str, optional] Optional kwarg of the filename of the MD simulation force-field parameters file in NAMD format. It is required alongside the *psf* file to use the gRINN network construction method.

special_res
[dict, optional] Optional kwarg of a dictionary containing a mapping of special residue 3/4-letter code(s) present in the structure to the corresponding standard 1-letter code(s).

Raises

FileNotFoundError
If any of the files passed in the *pdb* and *traj* (and *psf* and *parameters* if provided) parameters cannot be accessed.

See also:

AlloViz.Delta
Class for calculation of the delta-network(s) between two Protein objects.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> print(opioidGPCR.u)
<Universe with 88651 atoms>
```

Attributes

pdb
trajs
GPCR
protein
[Universe] Universe of the processed pdb with only the selected *protein_sel* atoms.

u
[Universe] Universe of the processed pdb and trajectory files with only the *protein_sel* atoms.

Methods

analyze([pkgs, filterings, elements, ...])	Analyzed filtered network
calculate([pkgs, cores])	Calculate rwa edge weights of allosteric networks.
filter([pkgs, filterings, ...])	Filter network edges
view(pkg, metric[, filtering, element, num, ...])	Visualize the analyzed networks on the protein structure.

AlloViz.AlloViz.Classes.Protein.analyze

Protein.**analyze**(*pkgs='all'*, *filterings='all'*, *elements='edges'*, *metrics='all'*, *cores=1*, *nodes_dict={'btw': 'networkx.algorithms.centrality.betweenness_centrality', 'cfb': 'networkx.algorithms.centrality.current_flow_betweenness_centrality'}*, *edges_dict={'btw': 'networkx.algorithms.centrality.edge_betweenness_centrality', 'cfb': 'networkx.algorithms.centrality.edge_current_flow_betweenness_centrality'}*, ***kwargs*)

Analyzed filtered network

Analyze the selected (un)filtered networks with the passed elements-metrics. It calls *AlloViz.AlloViz.Analysis.analyze()* and results are stored in instances of classes from the *AlloViz.AlloViz.Elements* module, which extend the pandas.DataFrame class.

Parameters

pkgs
[str or list, default: "all"] Package(s)/Network construction method(s) for which to analyze their raw edge weights, which must be already calculated and their data saved as instance attribute. In this case, "all" sends the computation for all available methods that are already calculated and saved as instance attributes.

filterings
[str or list, default: "all"] Filtering scheme(s) for which to perform the analyses, which must exist already for the selected packages. "all" sends the computation for all available schemes that are already saved.

elements
[str or list, {"edges", "nodes"}] Network elements for which to perform the analysis.

metrics
[str or list, default: "all"] Network metrics to compute, which must be keys in the *nodes_dict* or *edges_dict* dictionaries. Default is "all" and it sends the computation for all the metrics defined in the corresponding dictionary of the selected elements in *element*.

cores
[int, default: 1] Number of cores to use for parallelization with a *multiprocess* Pool. Default value only uses 1 core with a custom `AlloViz.utils.dummypool` that performs computations synchronously.

Other Parameters

nodes_dict, edges_dict
[dict] Optional kwarg(s) of the dictionary(ies) that maps network metrics custom names (e.g., betweenness centrality, "btw") with their corresponding NetworkX function (e.g., "networkx.algorithms.centrality.betweenness_centrality"). Functions strings must be written as if they were absolute imports, and must return a dictionary of edges or nodes, depending on the element dictionary in which they are. The keys of the dictionaries will be used to name the columns of the analyzed data that the functions produce. Defaults are *nodes_dict* and *edges_dict*.

****kwargs**
Other optional keyword arguments that will be passed to the NetworkX analysis function(s) that is(are) used on the method call in case they need extra parameters. All keyward arguments will be passed to all analysis function calls, so if the function doesn't accept the arguments there will be an error. *weight* parameter is already specified by AlloViz.

See also:

AlloViz.AlloViz.Analysis
Module with analysis functions.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.filter
Class method to filter the network(s) raw edge weights with different criteria.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.filter("dynetan", "GetContacts_edges")
>>> opioidGPCR.analyze("dynetan", "GetContacts_edges", "nodes", "btw")
<AlloViz.AlloViz.Elements.Nodes at 0x7f892c3c0fa0>
```

AlloViz.AlloViz.Classes.Protein.calculate

`Protein.`**`calculate`**(*pkgs='all'*, *cores=1*, ***kwargs*)

Calculate rwa edge weights of allosteric networks.

Send the computation of the raw edge weights for the selected network construction methods.

Parameters

pkgs
[str or list, default: "all"] Package(s)/Network construction method(s) for which to send raw edge weight computation. "all" sends the computation for all available methods within AlloViz (check *AlloViz.AlloViz.utils.pkgsl*).

cores
[int, default: 1] Number of cores to use for parallelization with a *multiprocess* Pool. Default value only uses 1 core with a custom `AlloViz.utils.dummypool` that performs computations synchronously.

Other Parameters

taskcpus
[int, optional] Optional kwarg to specify the amount of cores that parallelizable network construction methods can use (i.e., AlloViz's method, getcontacts, dynetan, PyInteraph, MDEntropy and gRINN).

stride
[int, optional] Optional kwarg to specify the striding to be done on the trajectory(ies) for computationally-expensive network construction methods: i.e., dynetan and AlloViz's own method. Default is no striding.

namd
[str, optional] Optional kwarg pointing to the namd2 executable location; if the *namd* command is accessible through the CLI it is automatically retrieved with the *distutils* package.

chis
[int, optional] Optional kwarg to specify the number of side-chain chi dihedral angles (up to 5) to combine when sending the calculation of a child of the Combined_Dihs Wrappers' base class that includes chi dihedrals in its calculation.

MDEntropy_method
[str, optional, {"knn", "grassberger", "chaowangjost"}] Optional kwarg to specify the method to calculate the entropy of the variables for Mutual Information estimation when using one of the MDEntropy network construction methods (default: "grassberger").

See also:

AlloViz.Wrappers.Base.Base
Base class to launch and store calculation results.

AlloViz.Protein.filter
Class method to filter the network raw edge weights with different criteria.

AlloViz.Protein.analyze
Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Notes

Calculation results are stored as new instance attributes with the same name as the selected packages/network construction methods. If the object is created providing more than one trajectory file, the average and standard error of the weights between the replicas are also calculated.

Examples

If we have a 6-core computer and want to use 2 cores to compute the values for each of the three trajectories of the protein (e.g., GPCRmd stores three replicas of each structure):

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate("dynetan", cores=6, taskcpus=2)
>>> print(opioidGPCR.dynetan.raw.shape)
(41041, 5)
```

AlloViz.AlloViz.Classes.Protein.filter

Protein.**filter**(*pkgs='all'*, *filterings='all'*, ***, *GetContacts_threshold=0*, *Sequence_Neighbor_distance=5*, *Interresidue_distance=10*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

pkgs
[str or list, default: "all"] Package(s)/Network construction method(s) for which to analyze their raw edge weights, which must be already calculated and their data saved as instance attribute. In this case, "all" sends the computation for all available methodsthat are already calculated and saved as instance attributes.

filterings
[str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists. A list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.filter("dynetan", ["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.AlloViz.Classes.Protein.view

Protein.**view**(*pkg*, *metric*, *filtering='All'*, *element='edges'*, *num: int = 20*, *colors: list = ['orange', 'turquoise']*, *nv=None*)

Visualize the analyzed networks on the protein structure.

Visualize the network corresponding to the selected package/network construction method, filtering scheme, network element and network metric or metrics, all of which must be previously calculated and analyzed. The number of (each of the) elements to show and the color scale can be specified. It calls `AlloViz.Elements.Element.view()`.

Parameters

pkg
[str] Package/Network construction method for which to show the network.

metric
[str] Network metric for which to show the network.

filtering
[str] Filtering scheme for which to show the network.

element
[str or list, {"edges", "nodes"}] Network element or elements to show on the protein structure representing the chosen package, filtering scheme and metric.

num
[int, default: 20] Number of (each of the) network elements to show on the structure.

colors
[list, default: ["orange", "turquoise"]] List of two colors to assign to the minimum and maximum values of the network to be represented, respectively. Middle value is assigned "white" and it will be the mean of the network values or 0 if the network has both negative and positive values.

nv
[`nglview.NGLWidget`, optional] A structure representation into which the shapes representing the chosen network elements will be added.

Returns

`nglview.NGLWidget`

See also:

AlloViz.AlloViz.Elements
Module with classes for storage and visualization of filtered and analyzed networks.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.filter
Class method to filter the network(s) raw edge weights with different criteria.

AlloViz.Protein.analyze
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate("dynetan", cores=6, taskcpus=2)
>>> opioidGPCR.filter("dynetan", "GetContacts_edges")
>>> opioidGPCR.analyze("dynetan", "GetContacts_edges", "nodes", "btw")
>>> opioidGPCR.view("dynetan", "btw", "GetContacts_edges", element=["edges",
↪"nodes"])
NGLWidget()
```

AlloViz.AlloViz.Elements

Module with classes to store and represent analyzed network elements

Parent base class *Element* extends `pandas.DataFrame` to store data and defines additional methods and private methods to represent the networks on the protein structure. Children classes differ on the type of network element (nodes or edges) displayed.

Classes

Edges(data, parent[, index, columns, dtype, ...])	Class for storage and viz of Edges
Element(data, parent[, index, columns, ...])	Base class for network storage and representation
Nodes(data, parent[, index, columns, dtype, ...])	Class for storage and viz of Nodes

AlloViz.AlloViz.Elements.Edges

class AlloViz.AlloViz.Elements.**Edges**(*data*, *parent*, *index=None*, *columns=None*, *dtype=None*, *copy=True*)

Bases: *Element*

Class for storage and viz of Edges

See this class' `_add_element()`

Attributes

T
The transpose of the DataFrame.

at
Access a single value for a row/column label pair.

attrs
Dictionary of global attributes of this dataset.

axes
Return a list representing the axes of the DataFrame.

columns
The column labels of the DataFrame.

```
>>> df = pd.DataFrame({'A': [1, 2], 'B': [3, 4]})
>>> df
     A  B
0    1  3
1    2  4
>>> df.columns
Index(['A', 'B'], dtype='object')
```

dtypes
Return the dtypes in the DataFrame.

empty
Indicator whether Series/DataFrame is empty.

flags
Get the properties associated with this pandas object.

iat
Access a single value for a row/column pair by integer position.

iloc
Purely integer-location based indexing for selection by position.

index
The index (row labels) of the DataFrame.

The index of a DataFrame is a series of labels that identify each row. The labels can be integers, strings, or any other hashable type. The index is used for label-based access and alignment, and can be accessed or modified using this attribute.

pandas.Index
The index labels of the DataFrame.

DataFrame.columns : The column labels of the DataFrame. DataFrame.to_numpy : Convert the DataFrame to a NumPy array.

```
>>> df = pd.DataFrame({'Name': ['Alice', 'Bob', 'Aritra'],
...                    'Age': [25, 30, 35],
...                    'Location': ['Seattle', 'New York', 'Kona']},
...                   index=([10, 20, 30]))
>>> df.index
Index([10, 20, 30], dtype='int64')
```

In this example, we create a DataFrame with 3 rows and 3 columns, including Name, Age, and Location information. We set the index labels to be the integers 10, 20, and 30. We then access the *index* attribute of the DataFrame, which returns an *Index* object containing the index labels.

```
>>> df.index = [100, 200, 300]
>>> df
    Name  Age Location
100  Alice   25  Seattle
200    Bob   30 New York
300  Aritra  35    Kona
```

In this example, we modify the index labels of the DataFrame by assigning a new list of labels to the *index* attribute. The DataFrame is then updated with the new labels, and the output shows the modified DataFrame.

loc
Access a group of rows and columns by label(s) or a boolean array.

ndim
Return an int representing the number of axes / array dimensions.

shape
Return a tuple representing the dimensionality of the DataFrame.

size
Return an int representing the number of elements in this object.

style
Returns a Styler object.

values
Return a Numpy representation of the DataFrame.

Methods

abs()	Return a Series/DataFrame with absolute numeric value of each element.
add(other[, axis, level, fill_value])	Get Addition of dataframe and other, element-wise (binary operator *add*).
add_prefix(prefix[, axis])	Prefix labels with string *prefix*.
add_suffix(suffix[, axis])	Suffix labels with string *suffix*.
agg([func, axis])	Aggregate using one or more operations over the specified axis.
aggregate([func, axis])	Aggregate using one or more operations over the specified axis.
align(other[, join, axis, level, copy, ...])	Align two objects on their axes with the specified join method.
all([axis, bool_only, skipna])	Return whether all elements are True, potentially over an axis.
any(*[, axis, bool_only, skipna])	Return whether any element is True, potentially over an axis.
apply(func[, axis, raw, result_type, args, ...])	Apply a function along an axis of the DataFrame.
applymap(func[, na_action])	Apply a function to a Dataframe elementwise.
asfreq(freq[, method, how, normalize, ...])	Convert time series to specified frequency.
asof(where[, subset])	Return the last row(s) without any NaNs before *where*.
assign(**kwargs)	Assign new columns to a DataFrame.
astype(dtype[, copy, errors])	Cast a pandas object to a specified dtype `dtype`.
at_time(time[, asof, axis])	Select values at particular time of day (e.g., 9:30AM).
backfill(*[, axis, inplace, limit, downcast])	Fill NA/NaN values by using the next valid observation to fill the gap.
between_time(start_time, end_time[, ...])	Select values between particular times of the day (e.g., 9:00-9:30 AM).
bfill(*[, axis, inplace, limit, downcast])	Fill NA/NaN values by using the next valid observation to fill the gap.
bool()	Return the bool of a single element Series or DataFrame.
boxplot([column, by, ax, fontsize, rot, ...])	Make a box plot from DataFrame columns.
clip([lower, upper, axis, inplace])	Trim values at input threshold(s).

continues on next page

Table 1 – continued from previous page

combine(other, func[, fill_value, overwrite])	Perform column-wise combine with another DataFrame.
combine_first(other)	Update null elements with value in the same location in *other*.
compare(other[, align_axis, keep_shape, ...])	Compare to another DataFrame and show the differences.
convert_dtypes([infer_objects, ...])	Convert columns to the best possible dtypes using dtypes supporting `pd.NA`.
copy([deep])	Make a copy of this object's indices and data.
corr([method, min_periods, numeric_only])	Compute pairwise correlation of columns, excluding NA/null values.
corrwith(other[, axis, drop, method, ...])	Compute pairwise correlation.
count([axis, numeric_only])	Count non-NA cells for each column or row.
cov([min_periods, ddof, numeric_only])	Compute pairwise covariance of columns, excluding NA/null values.
cummax([axis, skipna])	Return cumulative maximum over a DataFrame or Series axis.
cummin([axis, skipna])	Return cumulative minimum over a DataFrame or Series axis.
cumprod([axis, skipna])	Return cumulative product over a DataFrame or Series axis.
cumsum([axis, skipna])	Return cumulative sum over a DataFrame or Series axis.
describe([percentiles, include, exclude])	Generate descriptive statistics.
diff([periods, axis])	First discrete difference of element.
div(other[, axis, level, fill_value])	Get Floating division of dataframe and other, element-wise (binary operator *truediv*).
divide(other[, axis, level, fill_value])	Get Floating division of dataframe and other, element-wise (binary operator *truediv*).
dot(other)	Compute the matrix multiplication between the DataFrame and other.
drop([labels, axis, index, columns, level, ...])	Drop specified labels from rows or columns.
drop_duplicates([subset, keep, inplace, ...])	Return DataFrame with duplicate rows removed.
droplevel(level[, axis])	Return Series/DataFrame with requested index / column level(s) removed.
dropna(*[, axis, how, thresh, subset, ...])	Remove missing values.
duplicated([subset, keep])	Return boolean Series denoting duplicate rows.
eq(other[, axis, level])	Get Equal to of dataframe and other, element-wise (binary operator *eq*).
equals(other)	Test whether two objects contain the same elements.
eval(expr, *[, inplace])	Evaluate a string describing operations on DataFrame columns.
ewm([com, span, halflife, alpha, ...])	Provide exponentially weighted (EW) calculations.
expanding([min_periods, axis, method])	Provide expanding window calculations.
explode(column[, ignore_index])	Transform each element of a list-like to a row, replicating index values.
ffill(*[, axis, inplace, limit, downcast])	Fill NA/NaN values by propagating the last valid observation to next valid.
fillna([value, method, axis, inplace, ...])	Fill NA/NaN values using the specified method.
filter([items, like, regex, axis])	Subset the dataframe rows or columns according to the specified index labels.

continues on next page

Table 1 – continued from previous page

first(offset)	Select initial periods of time series data based on a date offset.
first_valid_index()	Return index for first non-NA value or None, if no non-NA value is found.
floordiv(other[, axis, level, fill_value])	Get Integer division of dataframe and other, element-wise (binary operator *floordiv*).
from_dict(data[, orient, dtype, columns])	Construct DataFrame from dict of array-like or dicts.
from_records(data[, index, exclude, ...])	Convert structured or record ndarray to DataFrame.
ge(other[, axis, level])	Get Greater than or equal to of dataframe and other, element-wise (binary operator *ge*).
get(key[, default])	Get item from object for given key (ex: DataFrame column).
groupby([by, axis, level, as_index, sort, ...])	Group DataFrame using a mapper or by a Series of columns.
gt(other[, axis, level])	Get Greater than of dataframe and other, element-wise (binary operator *gt*).
head([n])	Return the first *n* rows.
hist([column, by, grid, xlabelsize, xrot, ...])	Make a histogram of the DataFrame's columns.
idxmax([axis, skipna, numeric_only])	Return index of first occurrence of maximum over requested axis.
idxmin([axis, skipna, numeric_only])	Return index of first occurrence of minimum over requested axis.
infer_objects([copy])	Attempt to infer better dtypes for object columns.
info([verbose, buf, max_cols, memory_usage, ...])	Print a concise summary of a DataFrame.
insert(loc, column, value[, allow_duplicates])	Insert column into DataFrame at specified location.
interpolate([method, axis, limit, inplace, ...])	Fill NaN values using an interpolation method.
isetitem(loc, value)	Set the given value in the column with position *loc*.
isin(values)	Whether each element in the DataFrame is contained in values.
isna()	Detect missing values.
isnull()	DataFrame.isnull is an alias for DataFrame.isna.
items()	Iterate over (column name, Series) pairs.
iterrows()	Iterate over DataFrame rows as (index, Series) pairs.
itertuples([index, name])	Iterate over DataFrame rows as namedtuples.
join(other[, on, how, lsuffix, rsuffix, ...])	Join columns of another DataFrame.
keys()	Get the 'info axis' (see Indexing for more).
kurt([axis, skipna, numeric_only])	Return unbiased kurtosis over requested axis.
kurtosis([axis, skipna, numeric_only])	Return unbiased kurtosis over requested axis.
last(offset)	Select final periods of time series data based on a date offset.
last_valid_index()	Return index for last non-NA value or None, if no non-NA value is found.
le(other[, axis, level])	Get Less than or equal to of dataframe and other, element-wise (binary operator *le*).
lt(other[, axis, level])	Get Less than of dataframe and other, element-wise (binary operator *lt*).
map(func[, na_action])	Apply a function to a Dataframe elementwise.
mask(cond[, other, inplace, axis, level])	Replace values where the condition is True.
max([axis, skipna, numeric_only])	Return the maximum of the values over the requested axis.
mean([axis, skipna, numeric_only])	Return the mean of the values over the requested axis.

continues on next page

Table 1 – continued from previous page

median([axis, skipna, numeric_only])	Return the median of the values over the requested axis.
melt([id_vars, value_vars, var_name, ...])	Unpivot a DataFrame from wide to long format, optionally leaving identifiers set.
memory_usage([index, deep])	Return the memory usage of each column in bytes.
merge(right[, how, on, left_on, right_on, ...])	Merge DataFrame or named Series objects with a database-style join.
min([axis, skipna, numeric_only])	Return the minimum of the values over the requested axis.
mod(other[, axis, level, fill_value])	Get Modulo of dataframe and other, element-wise (binary operator *mod*).
mode([axis, numeric_only, dropna])	Get the mode(s) of each element along the selected axis.
mul(other[, axis, level, fill_value])	Get Multiplication of dataframe and other, element-wise (binary operator *mul*).
multiply(other[, axis, level, fill_value])	Get Multiplication of dataframe and other, element-wise (binary operator *mul*).
ne(other[, axis, level])	Get Not equal to of dataframe and other, element-wise (binary operator *ne*).
nlargest(n, columns[, keep])	Return the first *n* rows ordered by *columns* in descending order.
notna()	Detect existing (non-missing) values.
notnull()	DataFrame.notnull is an alias for DataFrame.notna.
nsmallest(n, columns[, keep])	Return the first *n* rows ordered by *columns* in ascending order.
nunique([axis, dropna])	Count number of distinct elements in specified axis.
pad(*[, axis, inplace, limit, downcast])	Fill NA/NaN values by propagating the last valid observation to next valid.
pct_change([periods, fill_method, limit, freq])	Fractional change between the current and a prior element.
pipe(func, *args, **kwargs)	Apply chainable functions that expect Series or DataFrames.
pivot(*, columns[, index, values])	Return reshaped DataFrame organized by given index / column values.
pivot_table([values, index, columns, ...])	Create a spreadsheet-style pivot table as a DataFrame.
plot	alias of PlotAccessor
pop(item)	Return item and drop from frame.
pow(other[, axis, level, fill_value])	Get Exponential power of dataframe and other, element-wise (binary operator *pow*).
prod([axis, skipna, numeric_only, min_count])	Return the product of the values over the requested axis.
product([axis, skipna, numeric_only, min_count])	Return the product of the values over the requested axis.
quantile([q, axis, numeric_only, ...])	Return values at the given quantile over requested axis.
query(expr, *[, inplace])	Query the columns of a DataFrame with a boolean expression.
radd(other[, axis, level, fill_value])	Get Addition of dataframe and other, element-wise (binary operator *radd*).
rank([axis, method, numeric_only, ...])	Compute numerical data ranks (1 through n) along axis.

continues on next page

Table 1 – continued from previous page

rdiv(other[, axis, level, fill_value])	Get Floating division of dataframe and other, element-wise (binary operator *rtruediv*).
reindex([labels, index, columns, axis, ...])	Conform DataFrame to new index with optional filling logic.
reindex_like(other[, method, copy, limit, ...])	Return an object with matching indices as other object.
rename([mapper, index, columns, axis, copy, ...])	Rename columns or index labels.
rename_axis([mapper, index, columns, axis, ...])	Set the name of the axis for the index or columns.
reorder_levels(order[, axis])	Rearrange index levels using input order.
replace([to_replace, value, inplace, limit, ...])	Replace values given in *to_replace* with *value*.
resample(rule[, axis, closed, label, ...])	Resample time-series data.
reset_index([level, drop, inplace, ...])	Reset the index, or a level of it.
rfloordiv(other[, axis, level, fill_value])	Get Integer division of dataframe and other, element-wise (binary operator *rfloordiv*).
rmod(other[, axis, level, fill_value])	Get Modulo of dataframe and other, element-wise (binary operator *rmod*).
rmul(other[, axis, level, fill_value])	Get Multiplication of dataframe and other, element-wise (binary operator *rmul*).
rolling(window[, min_periods, center, ...])	Provide rolling window calculations.
round([decimals])	Round a DataFrame to a variable number of decimal places.
rpow(other[, axis, level, fill_value])	Get Exponential power of dataframe and other, element-wise (binary operator *rpow*).
rsub(other[, axis, level, fill_value])	Get Subtraction of dataframe and other, element-wise (binary operator *rsub*).
rtruediv(other[, axis, level, fill_value])	Get Floating division of dataframe and other, element-wise (binary operator *rtruediv*).
sample([n, frac, replace, weights, ...])	Return a random sample of items from an axis of object.
select_dtypes([include, exclude])	Return a subset of the DataFrame's columns based on the column dtypes.
sem([axis, skipna, ddof, numeric_only])	Return unbiased standard error of the mean over requested axis.
set_axis(labels, *[, axis, copy])	Assign desired index to given axis.
set_flags(*[, copy, allows_duplicate_labels])	Return a new object with updated flags.
set_index(keys, *[, drop, append, inplace, ...])	Set the DataFrame index using existing columns.
shift([periods, freq, axis, fill_value, suffix])	Shift index by desired number of periods with an optional time *freq*.
skew([axis, skipna, numeric_only])	Return unbiased skew over requested axis.
sort_index(*[, axis, level, ascending, ...])	Sort object by labels (along an axis).
sort_values(by, *[, axis, ascending, ...])	Sort by the values along either axis.
sparse	alias of SparseFrameAccessor
squeeze([axis])	Squeeze 1 dimensional axis objects into scalars.
stack([level, dropna, sort, future_stack])	Stack the prescribed level(s) from columns to index.
std([axis, skipna, ddof, numeric_only])	Return sample standard deviation over requested axis.
sub(other[, axis, level, fill_value])	Get Subtraction of dataframe and other, element-wise (binary operator *sub*).
subtract(other[, axis, level, fill_value])	Get Subtraction of dataframe and other, element-wise (binary operator *sub*).
sum([axis, skipna, numeric_only, min_count])	Return the sum of the values over the requested axis.
swapaxes(axis1, axis2[, copy])	Interchange axes and swap values axes appropriately.
swaplevel([i, j, axis])	Swap levels i and j in a MultiIndex.

continues on next page

Table 1 – continued from previous page

tail([n])	Return the last *n* rows.
take(indices[, axis])	Return the elements in the given *positional* indices along an axis.
to_clipboard([excel, sep])	Copy object to the system clipboard.
to_csv([path_or_buf, sep, na_rep, ...])	Write object to a comma-separated values (csv) file.
to_dict([orient, into, index])	Convert the DataFrame to a dictionary.
to_excel(excel_writer[, sheet_name, na_rep, ...])	Write object to an Excel sheet.
to_feather(path, **kwargs)	Write a DataFrame to the binary Feather format.
to_gbq(destination_table[, project_id, ...])	Write a DataFrame to a Google BigQuery table.
to_hdf(path_or_buf, key[, mode, complevel, ...])	Write the contained data to an HDF5 file using HDFStore.
to_html([buf, columns, col_space, header, ...])	Render a DataFrame as an HTML table.
to_json([path_or_buf, orient, date_format, ...])	Convert the object to a JSON string.
to_latex([buf, columns, header, index, ...])	Render object to a LaTeX tabular, longtable, or nested table.
to_markdown([buf, mode, index, storage_options])	Print DataFrame in Markdown-friendly format.
to_numpy([dtype, copy, na_value])	Convert the DataFrame to a NumPy array.
to_orc([path, engine, index, engine_kwargs])	Write a DataFrame to the ORC format.
to_parquet([path, engine, compression, ...])	Write a DataFrame to the binary parquet format.
to_period([freq, axis, copy])	Convert DataFrame from DatetimeIndex to PeriodIndex.
to_pickle(path[, compression, protocol, ...])	Pickle (serialize) object to file.
to_records([index, column_dtypes, index_dtypes])	Convert DataFrame to a NumPy record array.
to_sql(name, con, *[, schema, if_exists, ...])	Write records stored in a DataFrame to a SQL database.
to_stata(path, *[, convert_dates, ...])	Export DataFrame object to Stata dta format.
to_string([buf, columns, col_space, header, ...])	Render a DataFrame to a console-friendly tabular output.
to_timestamp([freq, how, axis, copy])	Cast to DatetimeIndex of timestamps, at *beginning* of period.
to_xarray()	Return an xarray object from the pandas object.
to_xml([path_or_buffer, index, root_name, ...])	Render a DataFrame to an XML document.
transform(func[, axis])	Call `func` on self producing a DataFrame with the same axis shape as self.
transpose(*args[, copy])	Transpose index and columns.
truediv(other[, axis, level, fill_value])	Get Floating division of dataframe and other, element-wise (binary operator *truediv*).
truncate([before, after, axis, copy])	Truncate a Series or DataFrame before and after some index value.
tz_convert(tz[, axis, level, copy])	Convert tz-aware axis to target time zone.
tz_localize(tz[, axis, level, copy, ...])	Localize tz-naive index of a Series or DataFrame to target time zone.
unstack([level, fill_value, sort])	Pivot a level of the (necessarily hierarchical) index labels.
update(other[, join, overwrite, ...])	Modify in place using non-NA values from another DataFrame.
value_counts([subset, normalize, sort, ...])	Return a Series containing the frequency of each distinct row in the Dataframe.
var([axis, skipna, ddof, numeric_only])	Return unbiased variance over requested axis.
view(metric[, num, colors, nv])	Represent the selected metric in the structure
where(cond[, other, inplace, axis, level])	Replace values where the condition is False.

continues on next page

Table 1 – continued from previous page

xs(key[, axis, level, drop_level])	Return cross-section from the Series/DataFrame.

AlloViz.AlloViz.Elements.Edges.abs

Edges.**abs**() → None

Return a Series/DataFrame with absolute numeric value of each element.

This function only applies to elements that are all numeric.

Returns

abs
Series/DataFrame containing the absolute value of each element.

See also:

numpy.absolute
Calculate the absolute value element-wise.

Notes

For `complex` inputs, `1.2 + 1j`, the absolute value is $\sqrt{a^2 + b^2}$.

Examples

Absolute numeric values in a Series.

```
>>> s = pd.Series([-1.10, 2, -3.33, 4])
>>> s.abs()
0    1.10
1    2.00
2    3.33
3    4.00
dtype: float64
```

Absolute numeric values in a Series with complex numbers.

```
>>> s = pd.Series([1.2 + 1j])
>>> s.abs()
0    1.56205
dtype: float64
```

Absolute numeric values in a Series with a Timedelta element.

```
>>> s = pd.Series([pd.Timedelta('1 days')])
>>> s.abs()
0   1 days
dtype: timedelta64[ns]
```

Select rows with data closest to certain value using argsort (from StackOverflow).

```
>>> df = pd.DataFrame({
...     'a': [4, 5, 6, 7],
...     'b': [10, 20, 30, 40],
...     'c': [100, 50, -30, -50]
... })
>>> df
     a    b    c
0    4   10  100
1    5   20   50
2    6   30  -30
3    7   40  -50
>>> df.loc[(df.c - 43).abs().argsort()]
     a    b    c
1    5   20   50
0    4   10  100
2    6   30  -30
3    7   40  -50
```

AlloViz.AlloViz.Elements.Edges.add

Edges.**add**(*other*, *axis: Axis = 'columns'*, *level=None*, *fill_value=None*)

Get Addition of dataframe and other, element-wise (binary operator *add*).

Equivalent to `dataframe + other`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *radd*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *floordiv*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other
[scalar, sequence, Series, dict or DataFrame] Any single or multiple element data structure, or list-like object.

axis
[{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns. (1 or 'columns'). For Series input, axis to match Series index on.

level
[int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value
[float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame
Result of the arithmetic operation.

See also:

`DataFrame.add`
Add DataFrames.

DataFrame.sub
Subtract DataFrames.

DataFrame.mul
Multiply DataFrames.

DataFrame.div
Divide DataFrames (float division).

DataFrame.truediv
Divide DataFrames (float division).

DataFrame.floordiv
Divide DataFrames (integer division).

DataFrame.mod
Calculate modulo (remainder after division).

DataFrame.pow
Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
```

(continues on next page)

(continued from previous page)

```
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a dictionary by axis.

```
>>> df.mul({'angles': 0, 'degrees': 2})
            angles  degrees
circle           0      720
triangle         0      360
rectangle        0      720
```

```
>>> df.mul({'circle': 0, 'triangle': 2, 'rectangle': 3}, axis='index')
            angles  degrees
circle           0        0
triangle         6      360
rectangle       12     1080
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
```

(continues on next page)

(continued from previous page)

```
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

AlloViz.AlloViz.Elements.Edges.add_prefix

Edges.**add_prefix**(*prefix: str*, *axis: int | Literal['index', 'columns', 'rows'] | None = None*) → None

Prefix labels with string *prefix*.

For Series, the row labels are prefixed. For DataFrame, the column labels are prefixed.

Parameters

prefix
[str] The string to add before each label.

axis
[{0 or 'index', 1 or 'columns', None}, default None] Axis to add prefix on

New in version 2.0.0.

Returns

Series or DataFrame
New Series or DataFrame with updated labels.

See also:

`Series.add_suffix`
Suffix row labels with string *suffix*.

`DataFrame.add_suffix`
Suffix column labels with string *suffix*.

Examples

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s
0    1
1    2
2    3
3    4
dtype: int64
```

```
>>> s.add_prefix('item_')
item_0    1
item_1    2
item_2    3
item_3    4
dtype: int64
```

```
>>> df = pd.DataFrame({'A': [1, 2, 3, 4], 'B': [3, 4, 5, 6]})
>>> df
   A  B
0  1  3
1  2  4
2  3  5
3  4  6
```

```
>>> df.add_prefix('col_')
     col_A  col_B
0       1       3
1       2       4
2       3       5
3       4       6
```

AlloViz.AlloViz.Elements.Edges.add_suffix

Edges.**add_suffix**(*suffix: str*, *axis: int | Literal['index', 'columns', 'rows'] | None = None*) → None

Suffix labels with string *suffix*.

For Series, the row labels are suffixed. For DataFrame, the column labels are suffixed.

Parameters

suffix
[str] The string to add after each label.

axis
[{0 or 'index', 1 or 'columns', None}, default None] Axis to add suffix on

New in version 2.0.0.

Returns

Series or DataFrame
New Series or DataFrame with updated labels.

See also:

Series.add_prefix
Prefix row labels with string *prefix*.

DataFrame.add_prefix
Prefix column labels with string *prefix*.

Examples

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s
0    1
1    2
2    3
3    4
dtype: int64
```

```
>>> s.add_suffix('_item')
0_item    1
1_item    2
2_item    3
3_item    4
dtype: int64
```

```
>>> df = pd.DataFrame({'A': [1, 2, 3, 4], 'B': [3, 4, 5, 6]})
>>> df
   A  B
0  1  3
1  2  4
2  3  5
3  4  6
```

```
>>> df.add_suffix('_col')
     A_col  B_col
0       1       3
1       2       4
2       3       5
3       4       6
```

AlloViz.AlloViz.Elements.Edges.agg

Edges.**agg**(*func=None*, *axis: Axis = 0*, **args*, ***kwargs*)

Aggregate using one or more operations over the specified axis.

Parameters

func
[function, str, list or dict] Function to use for aggregating the data. If a function, must either work when passed a DataFrame or when passed to DataFrame.apply.

Accepted combinations are:

- function
- string function name
- list of functions and/or function names, e.g. `[np.sum, 'mean']`
- dict of axis labels -> functions, function names or list of such.

axis
[{0 or 'index', 1 or 'columns'}, default 0] If 0 or 'index': apply function to each column. If 1 or 'columns': apply function to each row.

***args**
Positional arguments to pass to *func*.

****kwargs**
Keyword arguments to pass to *func*.

Returns

scalar, Series or DataFrame
The return can be:

- scalar : when Series.agg is called with single function
- Series : when DataFrame.agg is called with a single function
- DataFrame : when DataFrame.agg is called with several functions

Return scalar, Series or DataFrame.

See also:

`DataFrame.apply`
Perform any type of operations.

`DataFrame.transform`
Perform transformation type operations.

`core.groupby.GroupBy`
Perform operations over groups.

core.resample.Resampler
Perform operations over resampled bins.

core.window.Rolling
Perform operations over rolling window.

core.window.Expanding
Perform operations over expanding window.

core.window.ExponentialMovingWindow
Perform operation over exponential weighted window.

Notes

The aggregation operations are always performed over an axis, either the index (default) or the column axis. This behavior is different from *numpy* aggregation functions (*mean*, *median*, *prod*, *sum*, *std*, *var*), where the default is to compute the aggregation of the flattened array, e.g., `numpy.mean(arr_2d)` as opposed to `numpy.mean(arr_2d, axis=0)`.

agg is an alias for *aggregate*. Use the alias.

Functions that mutate the passed object can produce unexpected behavior or errors and are not supported. See Mutating with User Defined Function (UDF) methods for more details.

A passed user-defined-function will be passed a Series for evaluation.

Examples

```
>>> df = pd.DataFrame([[1, 2, 3],
...                    [4, 5, 6],
...                    [7, 8, 9],
...                    [np.nan, np.nan, np.nan]],
...                   columns=['A', 'B', 'C'])
```

Aggregate these functions over the rows.

```
>>> df.agg(['sum', 'min'])
        A     B     C
sum  12.0  15.0  18.0
min   1.0   2.0   3.0
```

Different aggregations per column.

```
>>> df.agg({'A' : ['sum', 'min'], 'B' : ['min', 'max']})
        A    B
sum  12.0  NaN
min   1.0  2.0
max   NaN  8.0
```

Aggregate different functions over the columns and rename the index of the resulting DataFrame.

```
>>> df.agg(x=('A', 'max'), y=('B', 'min'), z=('C', 'mean'))
     A    B    C
x  7.0  NaN  NaN
y  NaN  2.0  NaN
z  NaN  NaN  6.0
```

Aggregate over the columns.

```
>>> df.agg("mean", axis="columns")
0    2.0
1    5.0
2    8.0
3    NaN
dtype: float64
```

AlloViz.AlloViz.Elements.Edges.aggregate

Edges.**aggregate**(*func=None*, *axis: Axis = 0*, **args*, ***kwargs*)

Aggregate using one or more operations over the specified axis.

Parameters

func
[function, str, list or dict] Function to use for aggregating the data. If a function, must either work when passed a DataFrame or when passed to DataFrame.apply.

Accepted combinations are:

- function
- string function name
- list of functions and/or function names, e.g. `[np.sum, 'mean']`
- dict of axis labels -> functions, function names or list of such.

axis
[{0 or 'index', 1 or 'columns'}, default 0] If 0 or 'index': apply function to each column. If 1 or 'columns': apply function to each row.

***args**
Positional arguments to pass to *func*.

****kwargs**
Keyword arguments to pass to *func*.

Returns

scalar, Series or DataFrame
The return can be:

- scalar : when Series.agg is called with single function
- Series : when DataFrame.agg is called with a single function
- DataFrame : when DataFrame.agg is called with several functions

Return scalar, Series or DataFrame.

See also:

`DataFrame.apply`
Perform any type of operations.

`DataFrame.transform`
Perform transformation type operations.

core.groupby.GroupBy
Perform operations over groups.

core.resample.Resampler
Perform operations over resampled bins.

core.window.Rolling
Perform operations over rolling window.

core.window.Expanding
Perform operations over expanding window.

core.window.ExponentialMovingWindow
Perform operation over exponential weighted window.

Notes

The aggregation operations are always performed over an axis, either the index (default) or the column axis. This behavior is different from *numpy* aggregation functions (*mean*, *median*, *prod*, *sum*, *std*, *var*), where the default is to compute the aggregation of the flattened array, e.g., `numpy.mean(arr_2d)` as opposed to `numpy.mean(arr_2d, axis=0)`.

agg is an alias for *aggregate*. Use the alias.

Functions that mutate the passed object can produce unexpected behavior or errors and are not supported. See Mutating with User Defined Function (UDF) methods for more details.

A passed user-defined-function will be passed a Series for evaluation.

Examples

```
>>> df = pd.DataFrame([[1, 2, 3],
...                    [4, 5, 6],
...                    [7, 8, 9],
...                    [np.nan, np.nan, np.nan]],
...                   columns=['A', 'B', 'C'])
```

Aggregate these functions over the rows.

```
>>> df.agg(['sum', 'min'])
        A     B     C
sum  12.0  15.0  18.0
min   1.0   2.0   3.0
```

Different aggregations per column.

```
>>> df.agg({'A' : ['sum', 'min'], 'B' : ['min', 'max']})
        A    B
sum  12.0  NaN
min   1.0  2.0
max   NaN  8.0
```

Aggregate different functions over the columns and rename the index of the resulting DataFrame.

```
>>> df.agg(x=('A', 'max'), y=('B', 'min'), z=('C', 'mean'))
     A    B    C
x  7.0  NaN  NaN
y  NaN  2.0  NaN
z  NaN  NaN  6.0
```

Aggregate over the columns.

```
>>> df.agg("mean", axis="columns")
0    2.0
1    5.0
2    8.0
3    NaN
dtype: float64
```

AlloViz.AlloViz.Elements.Edges.align

Edges.**align**(*other: NDFrameT*, *join: AlignJoin = 'outer'*, *axis: Axis | None = None*, *level: Level | None = None*, *copy: bool_t | None = None*, *fill_value: Hashable | None = None*, *method: FillnaOptions | None | lib.NoDefault = _NoDefault.no_default*, *limit: int | None | lib.NoDefault = _NoDefault.no_default*, *fill_axis: Axis | lib.NoDefault = _NoDefault.no_default*, *broadcast_axis: Axis | None | lib.NoDefault = _NoDefault.no_default*) → tuple[Self, NDFrameT]

Align two objects on their axes with the specified join method.

Join method is specified for each axis Index.

Parameters

other
[DataFrame or Series]

join
[{'outer', 'inner', 'left', 'right'}, default 'outer'] Type of alignment to be performed.

- left: use only keys from left frame, preserve key order.
- right: use only keys from right frame, preserve key order.
- outer: use union of keys from both frames, sort keys lexicographically.
- inner: use intersection of keys from both frames, preserve the order of the left keys.

axis
[allowed axis of the other object, default None] Align on index (0), columns (1), or both (None).

level
[int or level name, default None] Broadcast across a level, matching Index values on the passed MultiIndex level.

copy
[bool, default True] Always returns new objects. If copy=False and no reindexing is required then original objects are returned.

fill_value
[scalar, default np.nan] Value to use for missing values. Defaults to NaN, but can be any "compatible" value.

method
[{'backfill', 'bfill', 'pad', 'ffill', None}, default None] Method to use for filling holes in reindexed Series:

- pad / ffill: propagate last valid observation forward to next valid.
- backfill / bfill: use NEXT valid observation to fill gap.

Deprecated since version 2.1.

limit
[int, default None] If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled. Must be greater than 0 if not None.

Deprecated since version 2.1.

fill_axis
[{0 or 'index'} for Series, {0 or 'index', 1 or 'columns'} for DataFrame, default 0] Filling axis, method and limit.

Deprecated since version 2.1.

broadcast_axis
[{0 or 'index'} for Series, {0 or 'index', 1 or 'columns'} for DataFrame, default None] Broadcast values along this axis, if aligning two objects of different dimensions.

Deprecated since version 2.1.

Returns

tuple of (Series/DataFrame, type of other)
Aligned objects.

Examples

```
>>> df = pd.DataFrame(
...     [[1, 2, 3, 4], [6, 7, 8, 9]], columns=["D", "B", "E", "A"], index=[1, 2]
... )
>>> other = pd.DataFrame(
...     [[10, 20, 30, 40], [60, 70, 80, 90], [600, 700, 800, 900]],
...     columns=["A", "B", "C", "D"],
...     index=[2, 3, 4],
... )
>>> df
   D  B  E  A
1  1  2  3  4
2  6  7  8  9
>>> other
    A    B    C    D
2   10   20   30   40
3   60   70   80   90
4  600  700  800  900
```

Align on columns:

```
>>> left, right = df.align(other, join="outer", axis=1)
>>> left
   A  B   C  D  E
1  4  2 NaN  1  3
2  9  7 NaN  6  8
>>> right
    A    B    C    D   E
2   10   20   30   40 NaN
3   60   70   80   90 NaN
4  600  700  800  900 NaN
```

We can also align on the index:

```
>>> left, right = df.align(other, join="outer", axis=0)
>>> left
    D    B    E    A
1  1.0  2.0  3.0  4.0
2  6.0  7.0  8.0  9.0
3  NaN  NaN  NaN  NaN
4  NaN  NaN  NaN  NaN
>>> right
    A      B      C      D
1   NaN    NaN    NaN    NaN
2  10.0   20.0   30.0   40.0
3  60.0   70.0   80.0   90.0
4  600.0  700.0  800.0  900.0
```

Finally, the default *axis=None* will align on both index and columns:

```
>>> left, right = df.align(other, join="outer", axis=None)
>>> left
     A    B   C    D    E
1  4.0  2.0 NaN  1.0  3.0
2  9.0  7.0 NaN  6.0  8.0
3  NaN  NaN NaN  NaN  NaN
4  NaN  NaN NaN  NaN  NaN
>>> right
       A      B      C      D   E
1    NaN    NaN    NaN    NaN NaN
2   10.0   20.0   30.0   40.0 NaN
3   60.0   70.0   80.0   90.0 NaN
4  600.0  700.0  800.0  900.0 NaN
```

AlloViz.AlloViz.Elements.Edges.all

`Edges.all`(*axis: Axis = 0*, *bool_only: bool = False*, *skipna: bool = True*, ***kwargs*) → Series | bool

Return whether all elements are True, potentially over an axis.

Returns True unless there at least one element within a series or along a Dataframe axis that is False or equivalent (e.g. zero or empty).

Parameters

axis
[{0 or 'index', 1 or 'columns', None}, default 0] Indicate which axis or axes should be reduced. For *Series* this parameter is unused and defaults to 0.

- 0 / 'index' : reduce the index, return a Series whose index is the original column labels.
- 1 / 'columns' : reduce the columns, return a Series whose index is the original index.
- None : reduce all axes, return a scalar.

bool_only
[bool, default False] Include only boolean columns. Not implemented for Series.

skipna
[bool, default True] Exclude NA/null values. If the entire row/column is NA and skipna is True, then the result will be True, as for an empty row/column. If skipna is False, then NA are treated as True, because these are not equal to zero.

****kwargs**
[any, default None] Additional keywords have no effect but might be accepted for compatibility with NumPy.

Returns

Series or DataFrame
If level is specified, then, DataFrame is returned; otherwise, Series is returned.

See also:

`Series.all`
Return True if all elements are True.

`DataFrame.any`
Return True if one (or more) elements are True.

Examples

Series

```
>>> pd.Series([True, True]).all()
True
>>> pd.Series([True, False]).all()
False
>>> pd.Series([], dtype="float64").all()
True
>>> pd.Series([np.nan]).all()
True
>>> pd.Series([np.nan]).all(skipna=False)
True
```

DataFrames

Create a dataframe from a dictionary.

```
>>> df = pd.DataFrame({'col1': [True, True], 'col2': [True, False]})
>>> df
   col1   col2
0  True   True
1  True  False
```

Default behaviour checks if values in each column all return True.

```
>>> df.all()
col1     True
col2    False
dtype: bool
```

Specify `axis='columns'` to check if values in each row all return True.

```
>>> df.all(axis='columns')
0     True
1    False
dtype: bool
```

Or `axis=None` for whether every value is True.

```
>>> df.all(axis=None)
False
```

AlloViz.AlloViz.Elements.Edges.any

Edges.**any**(*, *axis: Axis = 0*, *bool_only: bool = False*, *skipna: bool = True*, ***kwargs*) → Series | bool

Return whether any element is True, potentially over an axis.

Returns False unless there is at least one element within a series or along a Dataframe axis that is True or equivalent (e.g. non-zero or non-empty).

Parameters

axis
[{0 or 'index', 1 or 'columns', None}, default 0] Indicate which axis or axes should be reduced. For *Series* this parameter is unused and defaults to 0.

- 0 / 'index' : reduce the index, return a Series whose index is the original column labels.
- 1 / 'columns' : reduce the columns, return a Series whose index is the original index.
- None : reduce all axes, return a scalar.

bool_only
[bool, default False] Include only boolean columns. Not implemented for Series.

skipna
[bool, default True] Exclude NA/null values. If the entire row/column is NA and skipna is True, then the result will be False, as for an empty row/column. If skipna is False, then NA are treated as True, because these are not equal to zero.

****kwargs**
[any, default None] Additional keywords have no effect but might be accepted for compatibility with NumPy.

Returns

Series or DataFrame
If level is specified, then, DataFrame is returned; otherwise, Series is returned.

See also:

numpy.any
Numpy version of this method.

Series.any
Return whether any element is True.

Series.all
Return whether all elements are True.

DataFrame.any
Return whether any element is True over requested axis.

DataFrame.all
Return whether all elements are True over requested axis.

Examples

Series

For Series input, the output is a scalar indicating whether any element is True.

```
>>> pd.Series([False, False]).any()
False
>>> pd.Series([True, False]).any()
True
>>> pd.Series([], dtype="float64").any()
False
>>> pd.Series([np.nan]).any()
False
>>> pd.Series([np.nan]).any(skipna=False)
True
```

DataFrame

Whether each column contains at least one True element (the default).

```
>>> df = pd.DataFrame({"A": [1, 2], "B": [0, 2], "C": [0, 0]})
>>> df
   A  B  C
0  1  0  0
1  2  2  0
```

```
>>> df.any()
A     True
B     True
```

(continues on next page)

(continued from previous page)

```
C    False
dtype: bool
```

Aggregating over the columns.

```
>>> df = pd.DataFrame({"A": [True, False], "B": [1, 2]})
>>> df
       A  B
0   True  1
1  False  2
```

```
>>> df.any(axis='columns')
0    True
1    True
dtype: bool
```

```
>>> df = pd.DataFrame({"A": [True, False], "B": [1, 0]})
>>> df
       A  B
0   True  1
1  False  0
```

```
>>> df.any(axis='columns')
0    True
1    False
dtype: bool
```

Aggregating over the entire DataFrame with `axis=None`.

```
>>> df.any(axis=None)
True
```

any for an empty DataFrame is an empty Series.

```
>>> pd.DataFrame([]).any()
Series([], dtype: bool)
```

AlloViz.AlloViz.Elements.Edges.apply

`Edges.apply`(*func: AggFuncType*, *axis: Axis = 0*, *raw: bool = False*, *result_type: Literal['expand', 'reduce', 'broadcast'] | None = None*, *args=()*, *by_row: Literal[False, 'compat'] = 'compat'*, ***kwargs*)

Apply a function along an axis of the DataFrame.

Objects passed to the function are Series objects whose index is either the DataFrame's index (`axis=0`) or the DataFrame's columns (`axis=1`). By default (`result_type=None`), the final return type is inferred from the return type of the applied function. Otherwise, it depends on the *result_type* argument.

Parameters

func
[function] Function to apply to each column or row.

axis
[{0 or 'index', 1 or 'columns'}, default 0] Axis along which the function is applied:

- 0 or 'index': apply function to each column.
- 1 or 'columns': apply function to each row.

raw
[bool, default False] Determines if row or column is passed as a Series or ndarray object:

- `False` : passes each row or column as a Series to the function.
- `True` : the passed function will receive ndarray objects instead. If you are just applying a NumPy reduction function this will achieve much better performance.

result_type
[{'expand', 'reduce', 'broadcast', None}, default None] These only act when `axis=1` (columns):

- 'expand' : list-like results will be turned into columns.
- 'reduce' : returns a Series if possible rather than expanding list-like results. This is the opposite of 'expand'.
- 'broadcast' : results will be broadcast to the original shape of the DataFrame, the original index and columns will be retained.

The default behaviour (None) depends on the return value of the applied function: list-like results will be returned as a Series of those. However if the apply function returns a Series these are expanded to columns.

args
[tuple] Positional arguments to pass to *func* in addition to the array/series.

by_row
[False or "compat", default "compat"] Only has an effect when `func` is a listlike or dictlike of funcs and the func isn't a string. If "compat", will if possible first translate the func into pandas methods (e.g. `Series().apply(np.sum)` will be translated to `Series().sum()`). If that doesn't work, will try call to apply again with `by_row=True` and if that fails, will call apply again with `by_row=False` (backward compatible). If False, the funcs will be passed the whole Series at once.

New in version 2.1.0.

****kwargs**
Additional keyword arguments to pass as keywords arguments to *func*.

Returns

Series or DataFrame
Result of applying `func` along the given axis of the DataFrame.

See also:

`DataFrame.map`
For elementwise operations.

`DataFrame.aggregate`
Only perform aggregating type operations.

`DataFrame.transform`
Only perform transforming type operations.

Notes

Functions that mutate the passed object can produce unexpected behavior or errors and are not supported. See Mutating with User Defined Function (UDF) methods for more details.

Examples

```
>>> df = pd.DataFrame([[4, 9]] * 3, columns=['A', 'B'])
>>> df
   A  B
0  4  9
1  4  9
2  4  9
```

Using a numpy universal function (in this case the same as `np.sqrt(df)`):

```
>>> df.apply(np.sqrt)
     A    B
0  2.0  3.0
1  2.0  3.0
2  2.0  3.0
```

Using a reducing function on either axis

```
>>> df.apply(np.sum, axis=0)
A    12
B    27
dtype: int64
```

```
>>> df.apply(np.sum, axis=1)
0    13
1    13
2    13
dtype: int64
```

Returning a list-like will result in a Series

```
>>> df.apply(lambda x: [1, 2], axis=1)
0    [1, 2]
1    [1, 2]
2    [1, 2]
dtype: object
```

Passing `result_type='expand'` will expand list-like results to columns of a Dataframe

```
>>> df.apply(lambda x: [1, 2], axis=1, result_type='expand')
   0  1
0  1  2
1  1  2
2  1  2
```

Returning a Series inside the function is similar to passing `result_type='expand'`. The resulting column names will be the Series index.

```
>>> df.apply(lambda x: pd.Series([1, 2], index=['foo', 'bar']), axis=1)
   foo  bar
0    1    2
1    1    2
2    1    2
```

Passing `result_type='broadcast'` will ensure the same shape result, whether list-like or scalar is returned by the function, and broadcast it along the axis. The resulting column names will be the originals.

```
>>> df.apply(lambda x: [1, 2], axis=1, result_type='broadcast')
   A  B
0  1  2
1  1  2
2  1  2
```

AlloViz.AlloViz.Elements.Edges.applymap

Edges.**applymap**(*func: PythonFuncType*, *na_action: NaAction | None = None*, ***kwargs*) → DataFrame

Apply a function to a Dataframe elementwise.

Deprecated since version 2.1.0: DataFrame.applymap has been deprecated. Use DataFrame.map instead.

This method applies a function that accepts and returns a scalar to every element of a DataFrame.

Parameters

func
[callable] Python function, returns a single value from a single value.

na_action
[{None, 'ignore'}, default None] If 'ignore', propagate NaN values, without passing them to func.

****kwargs**
Additional keyword arguments to pass as keywords arguments to *func*.

Returns

DataFrame
Transformed DataFrame.

See also:

`DataFrame.apply`
Apply a function along input axis of DataFrame.

`DataFrame.map`
Apply a function along input axis of DataFrame.

`DataFrame.replace`
Replace values given in *to_replace* with *value*.

Examples

```
>>> df = pd.DataFrame([[1, 2.12], [3.356, 4.567]])
>>> df
       0      1
0  1.000  2.120
1  3.356  4.567
```

```
>>> df.map(lambda x: len(str(x)))
   0  1
0  3  4
1  5  5
```

AlloViz.AlloViz.Elements.Edges.asfreq

Edges.**asfreq**(*freq: Frequency*, *method: FillnaOptions | None = None*, *how: Literal['start', 'end'] | None = None*, *normalize: bool_t = False*, *fill_value: Hashable | None = None*) → Self

Convert time series to specified frequency.

Returns the original data conformed to a new index with the specified frequency.

If the index of this Series/DataFrame is a `PeriodIndex`, the new index is the result of transforming the original index with `PeriodIndex.asfreq` (so the original index will map one-to-one to the new index).

Otherwise, the new index will be equivalent to `pd.date_range(start, end, freq=freq)` where `start` and `end` are, respectively, the first and last entries in the original index (see `pandas.date_range()`). The values corresponding to any timesteps in the new index which were not present in the original index will be null (`NaN`), unless a method for filling such unknowns is provided (see the `method` parameter below).

The *`resample()`* method is more appropriate if an operation on each group of timesteps (such as an aggregate) is necessary to represent the data at the new frequency.

Parameters

freq
[DateOffset or str] Frequency DateOffset or string.

method
[{'backfill'/'bfill', 'pad'/'ffill'}, default None] Method to use for filling holes in reindexed Series (note this does not fill NaNs that already were present):

- 'pad' / 'ffill': propagate last valid observation forward to next valid
- 'backfill' / 'bfill': use NEXT valid observation to fill.

how
[{'start', 'end'}, default end] For PeriodIndex only (see PeriodIndex.asfreq).

normalize
[bool, default False] Whether to reset output index to midnight.

fill_value
[scalar, optional] Value to use for missing values, applied during upsampling (note this does not fill NaNs that already were present).

Returns

Series/DataFrame
Series/DataFrame object reindexed to the specified frequency.

See also:

reindex
Conform DataFrame to new index with optional filling logic.

Notes

To learn more about the frequency strings, please see this link.

Examples

Start by creating a series with 4 one minute timestamps.

```
>>> index = pd.date_range('1/1/2000', periods=4, freq='T')
>>> series = pd.Series([0.0, None, 2.0, 3.0], index=index)
>>> df = pd.DataFrame({'s': series})
>>> df
                     s
2000-01-01 00:00:00  0.0
2000-01-01 00:01:00  NaN
2000-01-01 00:02:00  2.0
2000-01-01 00:03:00  3.0
```

Upsample the series into 30 second bins.

```
>>> df.asfreq(freq='30S')
                     s
2000-01-01 00:00:00  0.0
2000-01-01 00:00:30  NaN
2000-01-01 00:01:00  NaN
2000-01-01 00:01:30  NaN
2000-01-01 00:02:00  2.0
2000-01-01 00:02:30  NaN
2000-01-01 00:03:00  3.0
```

Upsample again, providing a `fill value`.

```
>>> df.asfreq(freq='30S', fill_value=9.0)
                     s
2000-01-01 00:00:00  0.0
2000-01-01 00:00:30  9.0
2000-01-01 00:01:00  NaN
2000-01-01 00:01:30  9.0
2000-01-01 00:02:00  2.0
2000-01-01 00:02:30  9.0
2000-01-01 00:03:00  3.0
```

Upsample again, providing a `method`.

```
>>> df.asfreq(freq='30S', method='bfill')
                     s
2000-01-01 00:00:00  0.0
2000-01-01 00:00:30  NaN
2000-01-01 00:01:00  NaN
2000-01-01 00:01:30  2.0
2000-01-01 00:02:00  2.0
2000-01-01 00:02:30  3.0
2000-01-01 00:03:00  3.0
```

AlloViz.AlloViz.Elements.Edges.asof

Edges.**asof**(*where*, *subset=None*)

Return the last row(s) without any NaNs before *where*.

The last row (for each element in *where*, if list) without any NaN is taken. In case of a `DataFrame`, the last row without NaN considering only the subset of columns (if not *None*)

If there is no good value, NaN is returned for a Series or a Series of NaN values for a DataFrame

Parameters

where
[date or array-like of dates] Date(s) before which the last row(s) are returned.

subset
[str or array-like of str, default *None*] For DataFrame, if not *None*, only use these columns to check for NaNs.

Returns

scalar, Series, or DataFrame
The return can be:

- scalar : when *self* is a Series and *where* is a scalar
- Series: when *self* is a Series and *where* is an array-like, or when *self* is a DataFrame and *where* is a scalar
- DataFrame : when *self* is a DataFrame and *where* is an array-like

Return scalar, Series, or DataFrame.

See also:

merge_asof
Perform an asof merge. Similar to left join.

Notes

Dates are assumed to be sorted. Raises if this is not the case.

Examples

A Series and a scalar *where*.

```
>>> s = pd.Series([1, 2, np.nan, 4], index=[10, 20, 30, 40])
>>> s
10    1.0
20    2.0
30    NaN
40    4.0
dtype: float64
```

```
>>> s.asof(20)
2.0
```

For a sequence *where*, a Series is returned. The first value is NaN, because the first element of *where* is before the first index value.

```
>>> s.asof([5, 20])
5     NaN
20    2.0
dtype: float64
```

Missing values are not considered. The following is `2.0`, not NaN, even though NaN is at the index location for `30`.

```
>>> s.asof(30)
2.0
```

Take all columns into consideration

```
>>> df = pd.DataFrame({'a': [10., 20., 30., 40., 50.],
...                    'b': [None, None, None, None, 500]},
...                   index=pd.DatetimeIndex(['2018-02-27 09:01:00',
...                                           '2018-02-27 09:02:00',
...                                           '2018-02-27 09:03:00',
...                                           '2018-02-27 09:04:00',
...                                           '2018-02-27 09:05:00']))
>>> df.asof(pd.DatetimeIndex(['2018-02-27 09:03:30',
...                           '2018-02-27 09:04:30']))
                     a   b
2018-02-27 09:03:30 NaN NaN
2018-02-27 09:04:30 NaN NaN
```

Take a single column into consideration

```
>>> df.asof(pd.DatetimeIndex(['2018-02-27 09:03:30',
...                           '2018-02-27 09:04:30']),
...         subset=['a'])
```

(continues on next page)

(continued from previous page)

```
                        a   b
2018-02-27 09:03:30  30.0 NaN
2018-02-27 09:04:30  40.0 NaN
```

AlloViz.AlloViz.Elements.Edges.assign

Edges.**assign**(***kwargs*) → DataFrame

Assign new columns to a DataFrame.

Returns a new object with all original columns in addition to new ones. Existing columns that are re-assigned will be overwritten.

Parameters

****kwargs**
[dict of {str: callable or Series}] The column names are keywords. If the values are callable, they are computed on the DataFrame and assigned to the new columns. The callable must not change input DataFrame (though pandas doesn't check it). If the values are not callable, (e.g. a Series, scalar, or array), they are simply assigned.

Returns

DataFrame
A new DataFrame with the new columns in addition to all the existing columns.

Notes

Assigning multiple columns within the same `assign` is possible. Later items in '**kwargs' may refer to newly created or modified columns in 'df'; items are computed and assigned into 'df' in order.

Examples

```
>>> df = pd.DataFrame({'temp_c': [17.0, 25.0]},
...                   index=['Portland', 'Berkeley'])
>>> df
          temp_c
Portland    17.0
Berkeley    25.0
```

Where the value is a callable, evaluated on *df*:

```
>>> df.assign(temp_f=lambda x: x.temp_c * 9 / 5 + 32)
          temp_c  temp_f
Portland    17.0    62.6
Berkeley    25.0    77.0
```

Alternatively, the same behavior can be achieved by directly referencing an existing Series or sequence:

```
>>> df.assign(temp_f=df['temp_c'] * 9 / 5 + 32)
          temp_c  temp_f
Portland    17.0    62.6
Berkeley    25.0    77.0
```

You can create multiple columns within the same assign where one of the columns depends on another one defined within the same assign:

```
>>> df.assign(temp_f=lambda x: x['temp_c'] * 9 / 5 + 32,
...           temp_k=lambda x: (x['temp_f'] + 459.67) * 5 / 9)
           temp_c  temp_f  temp_k
Portland    17.0    62.6  290.15
Berkeley    25.0    77.0  298.15
```

AlloViz.AlloViz.Elements.Edges.astype

Edges.**astype**(*dtype*, *copy: bool | None = None*, *errors: Literal['ignore', 'raise'] = 'raise'*) → None

Cast a pandas object to a specified dtype `dtype`.

Parameters

dtype
[str, data type, Series or Mapping of column name -> data type] Use a str, numpy.dtype, pandas.ExtensionDtype or Python type to cast entire pandas object to the same type. Alternatively, use a mapping, e.g. {col: dtype, ...}, where col is a column label and dtype is a numpy.dtype or Python type to cast one or more of the DataFrame's columns to column-specific types.

copy
[bool, default True] Return a copy when `copy=True` (be very careful setting `copy=False` as changes to values then may propagate to other pandas objects).

errors
[{'raise', 'ignore'}, default 'raise'] Control raising of exceptions on invalid data for provided dtype.

- `raise` : allow exceptions to be raised
- `ignore` : suppress exceptions. On error return original object.

Returns

same type as caller

See also:

`to_datetime`
Convert argument to datetime.

`to_timedelta`
Convert argument to timedelta.

`to_numeric`
Convert argument to a numeric type.

`numpy.ndarray.astype`
Cast a numpy array to a specified type.

Notes

Changed in version 2.0.0: Using `astype` to convert from timezone-naive dtype to timezone-aware dtype will raise an exception. Use `Series.dt.tz_localize()` instead.

Examples

Create a DataFrame:

```
>>> d = {'col1': [1, 2], 'col2': [3, 4]}
>>> df = pd.DataFrame(data=d)
>>> df.dtypes
col1    int64
col2    int64
dtype: object
```

Cast all columns to int32:

```
>>> df.astype('int32').dtypes
col1    int32
col2    int32
dtype: object
```

Cast col1 to int32 using a dictionary:

```
>>> df.astype({'col1': 'int32'}).dtypes
col1    int32
col2    int64
dtype: object
```

Create a series:

```
>>> ser = pd.Series([1, 2], dtype='int32')
>>> ser
0    1
1    2
dtype: int32
>>> ser.astype('int64')
0    1
1    2
dtype: int64
```

Convert to categorical type:

```
>>> ser.astype('category')
0    1
1    2
dtype: category
Categories (2, int32): [1, 2]
```

Convert to ordered categorical type with custom ordering:

```
>>> from pandas.api.types import CategoricalDtype
>>> cat_dtype = CategoricalDtype(
```

(continues on next page)

(continued from previous page)

```
...     categories=[2, 1], ordered=True)
>>> ser.astype(cat_dtype)
0    1
1    2
dtype: category
Categories (2, int64): [2 < 1]
```

Create a series of dates:

```
>>> ser_date = pd.Series(pd.date_range('20200101', periods=3))
>>> ser_date
0   2020-01-01
1   2020-01-02
2   2020-01-03
dtype: datetime64[ns]
```

AlloViz.AlloViz.Elements.Edges.at_time

Edges.**at_time**(*time*, *asof: bool = False*, *axis: int | Literal['index', 'columns', 'rows'] | None = None*) → None

Select values at particular time of day (e.g., 9:30AM).

Parameters

time
[datetime.time or str] The values to select.

axis
[{0 or 'index', 1 or 'columns'}, default 0] For *Series* this parameter is unused and defaults to 0.

Returns

Series or DataFrame

Raises

TypeError
If the index is not a `DatetimeIndex`

See also:

between_time
Select values between particular times of the day.

first
Select initial periods of time series based on a date offset.

last
Select final periods of time series based on a date offset.

`DatetimeIndex.indexer_at_time`
Get just the index locations for values at particular time of the day.

Examples

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='12H')
>>> ts = pd.DataFrame({'A': [1, 2, 3, 4]}, index=i)
>>> ts
                     A
2018-04-09 00:00:00  1
2018-04-09 12:00:00  2
2018-04-10 00:00:00  3
2018-04-10 12:00:00  4
```

```
>>> ts.at_time('12:00')
                     A
2018-04-09 12:00:00  2
2018-04-10 12:00:00  4
```

AlloViz.AlloViz.Elements.Edges.backfill

Edges.**backfill**(*, *axis: None | Axis = None*, *inplace: bool_t = False*, *limit: None | int = None*, *downcast: dict | None | lib.NoDefault = _NoDefault.no_default*) → Self | None

Fill NA/NaN values by using the next valid observation to fill the gap.

Deprecated since version 2.0: Series/DataFrame.backfill is deprecated. Use Series/DataFrame.bfill instead.

Returns

Series/DataFrame or None
Object with missing values filled or None if `inplace=True`.

Examples

Please see examples for `DataFrame.bfill()` or `Series.bfill()`.

AlloViz.AlloViz.Elements.Edges.between_time

Edges.**between_time**(*start_time*, *end_time*, *inclusive: Literal['left', 'right'] | Literal['both', 'neither'] = 'both'*, *axis: int | Literal['index', 'columns', 'rows'] | None = None*) → None

Select values between particular times of the day (e.g., 9:00-9:30 AM).

By setting `start_time` to be later than `end_time`, you can get the times that are *not* between the two times.

Parameters

start_time
[datetime.time or str] Initial time as a time filter limit.

end_time
[datetime.time or str] End time as a time filter limit.

inclusive
[{“both”, “neither”, “left”, “right”}, default “both”] Include boundaries; whether to set each bound as closed or open.

axis
[{0 or 'index', 1 or 'columns'}, default 0] Determine range time on index or columns value. For *Series* this parameter is unused and defaults to 0.

Returns

Series or DataFrame
Data from the original object filtered to the specified dates range.

Raises

TypeError
If the index is not a `DatetimeIndex`

See also:

at_time
Select values at a particular time of the day.

first
Select initial periods of time series based on a date offset.

last
Select final periods of time series based on a date offset.

`DatetimeIndex.indexer_between_time`
Get just the index locations for values between particular times of the day.

Examples

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='1D20min')
>>> ts = pd.DataFrame({'A': [1, 2, 3, 4]}, index=i)
>>> ts
                     A
2018-04-09 00:00:00  1
2018-04-10 00:20:00  2
2018-04-11 00:40:00  3
2018-04-12 01:00:00  4
```

```
>>> ts.between_time('0:15', '0:45')
                     A
2018-04-10 00:20:00  2
2018-04-11 00:40:00  3
```

You get the times that are *not* between two times by setting `start_time` later than `end_time`:

```
>>> ts.between_time('0:45', '0:15')
                     A
2018-04-09 00:00:00  1
2018-04-12 01:00:00  4
```

AlloViz.AlloViz.Elements.Edges.bfill

Edges.**bfill**(*, *axis: None | Axis = None*, *inplace: bool_t = False*, *limit: None | int = None*, *downcast: dict | None | lib.NoDefault = _NoDefault.no_default*) → Self | None

Fill NA/NaN values by using the next valid observation to fill the gap.

Parameters

axis
[{0 or 'index'} for Series, {0 or 'index', 1 or 'columns'} for DataFrame] Axis along which to fill missing values. For *Series* this parameter is unused and defaults to 0.

inplace
[bool, default False] If True, fill in-place. Note: this will modify any other views on this object (e.g., a no-copy slice for a column in a DataFrame).

limit
[int, default None] If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled. Must be greater than 0 if not None.

downcast
[dict, default is None] A dict of item->dtype of what to downcast if possible, or the string 'infer' which will try to downcast to an appropriate equal type (e.g. float64 to int64 if possible).

Returns

Series/DataFrame or None
Object with missing values filled or None if `inplace=True`.

Examples

For Series:

```
>>> s = pd.Series([1, None, None, 2])
>>> s.bfill()
0    1.0
1    2.0
2    2.0
3    2.0
dtype: float64
>>> s.bfill(limit=1)
0    1.0
1    NaN
2    2.0
3    2.0
dtype: float64
```

With DataFrame:

```
>>> df = pd.DataFrame({'A': [1, None, None, 4], 'B': [None, 5, None, 7]})
>>> df
     A     B
```

(continues on next page)

(continued from previous page)

```
0   1.0   NaN
1   NaN   5.0
2   NaN   NaN
3   4.0   7.0
>>> df.bfill()
      A     B
0   1.0   5.0
1   4.0   5.0
2   4.0   7.0
3   4.0   7.0
>>> df.bfill(limit=1)
      A     B
0   1.0   5.0
1   NaN   5.0
2   4.0   7.0
3   4.0   7.0
```

AlloViz.AlloViz.Elements.Edges.bool

Edges.**bool**() → bool

Return the bool of a single element Series or DataFrame.

Deprecated since version 2.1.0: bool is deprecated and will be removed in future version of pandas

This must be a boolean scalar value, either True or False. It will raise a ValueError if the Series or DataFrame does not have exactly 1 element, or that element is not boolean (integer values 0 and 1 will also raise an exception).

Returns

bool
The value in the Series or DataFrame.

See also:

Series.astype
Change the data type of a Series, including to boolean.

DataFrame.astype
Change the data type of a DataFrame, including to boolean.

numpy.bool_
NumPy boolean data type, used by pandas for boolean values.

Examples

The method will only work for single element objects with a boolean value:

```
>>> pd.Series([True]).bool()
True
>>> pd.Series([False]).bool()
False
```

```
>>> pd.DataFrame({'col': [True]}).bool()
True
>>> pd.DataFrame({'col': [False]}).bool()
False
```

AlloViz.AlloViz.Elements.Edges.boxplot

Edges.**boxplot**(*column=None*, *by=None*, *ax=None*, *fontsize: None | int = None*, *rot: int = 0*, *grid: bool = True*, *figsize: tuple[float, float] | None = None*, *layout=None*, *return_type=None*, *backend=None*, ***kwargs*)

Make a box plot from DataFrame columns.

Make a box-and-whisker plot from DataFrame columns, optionally grouped by some other columns. A box plot is a method for graphically depicting groups of numerical data through their quartiles. The box extends from the Q1 to Q3 quartile values of the data, with a line at the median (Q2). The whiskers extend from the edges of box to show the range of the data. By default, they extend no more than *1.5 * IQR (IQR = Q3 - Q1)* from the edges of the box, ending at the farthest data point within that interval. Outliers are plotted as separate dots.

For further details see Wikipedia's entry for boxplot.

Parameters

column
[str or list of str, optional] Column name or list of names, or vector. Can be any valid input to `pandas.DataFrame.groupby()`.

by
[str or array-like, optional] Column in the DataFrame to `pandas.DataFrame.groupby()`. One box-plot will be done per value of columns in *by*.

ax
[object of class matplotlib.axes.Axes, optional] The matplotlib axes to be used by boxplot.

fontsize
[float or str] Tick label font size in points or as a string (e.g., *large*).

rot
[float, default 0] The rotation angle of labels (in degrees) with respect to the screen coordinate system.

grid
[bool, default True] Setting this to True will show the grid.

figsize
[A tuple (width, height) in inches] The size of the figure to create in matplotlib.

layout
[tuple (rows, columns), optional] For example, (3, 5) will display the subplots using 3 rows and 5 columns, starting from the top-left.

return_type
[{'axes', 'dict', 'both'} or None, default 'axes'] The kind of object to return. The default is `axes`.

- 'axes' returns the matplotlib axes the boxplot is drawn on.
- 'dict' returns a dictionary whose values are the matplotlib Lines of the boxplot.
- 'both' returns a namedtuple with the axes and dict.
- when grouping with `by`, a Series mapping columns to `return_type` is returned.

  If `return_type` is *None*, a NumPy array of axes with the same shape as `layout` is returned.

backend
[str, default None] Backend to use instead of the backend specified in the option `plotting.backend`. For instance, 'matplotlib'. Alternatively, to specify the `plotting.backend` for the whole session, set `pd.options.plotting.backend`.

****kwargs**
All other plotting keyword arguments to be passed to `matplotlib.pyplot.boxplot()`.

Returns

result
See Notes.

See also:

`pandas.Series.plot.hist`
Make a histogram.

`matplotlib.pyplot.boxplot`
Matplotlib equivalent plot.

Notes

The return type depends on the *return_type* parameter:

- 'axes' : object of class matplotlib.axes.Axes
- 'dict' : dict of matplotlib.lines.Line2D objects
- 'both' : a namedtuple with structure (ax, lines)

For data grouped with `by`, return a Series of the above or a numpy array:

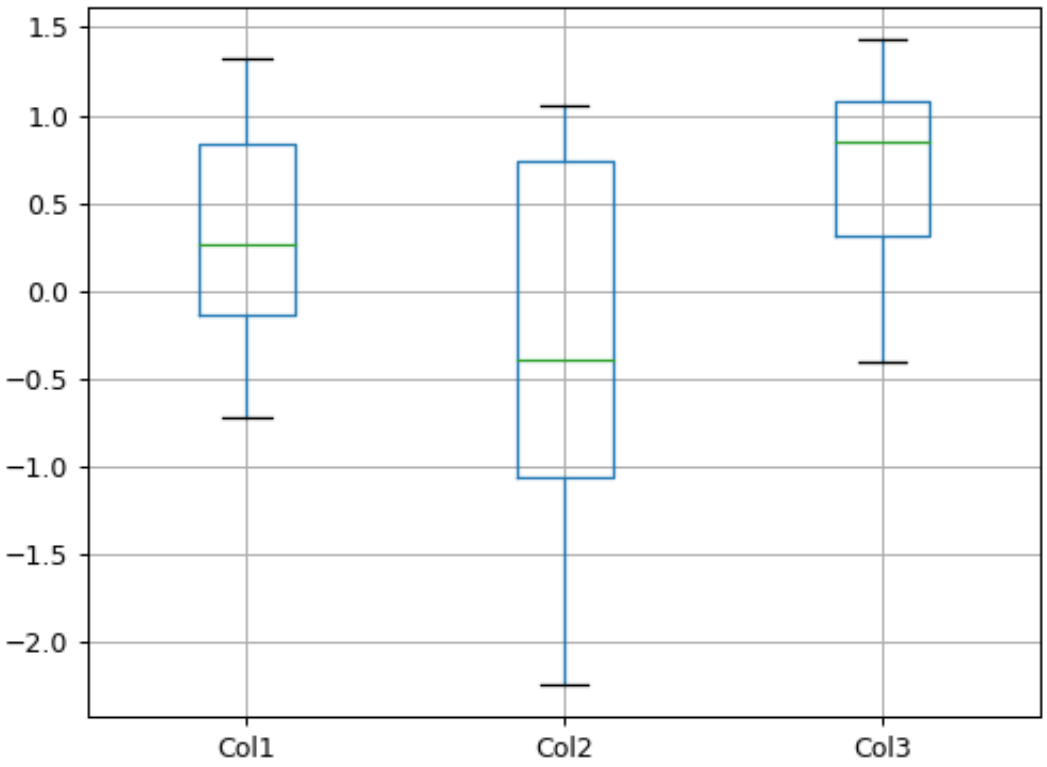
- `Series`
- `array` (for `return_type = None`)

Use `return_type='dict'` when you want to tweak the appearance of the lines after plotting. In this case a dict containing the Lines making up the boxes, caps, fliers, medians, and whiskers is returned.

Examples

Boxplots can be created for every column in the dataframe by `df.boxplot()` or indicating the columns to be used:

```
>>> np.random.seed(1234)
>>> df = pd.DataFrame(np.random.randn(10, 4),
...                   columns=['Col1', 'Col2', 'Col3', 'Col4'])
>>> boxplot = df.boxplot(column=['Col1', 'Col2', 'Col3'])
```



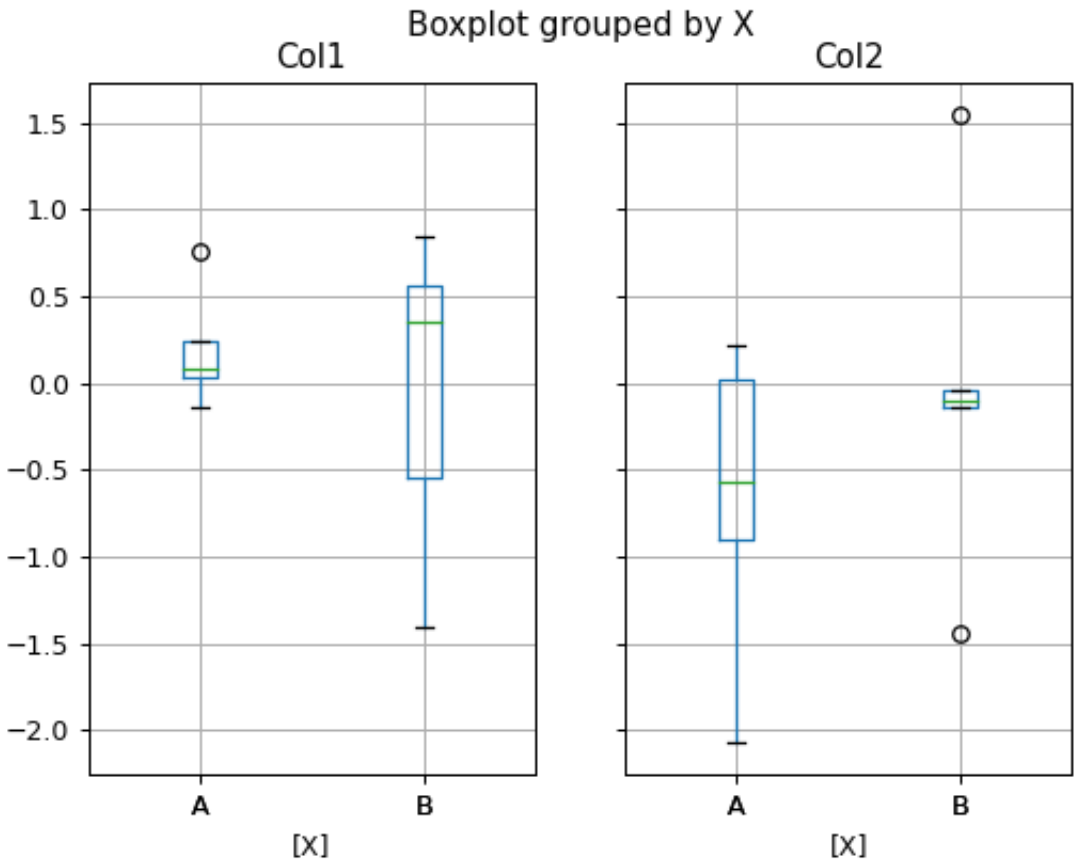
Boxplots of variables distributions grouped by the values of a third variable can be created using the option `by`. For instance:

```
>>> df = pd.DataFrame(np.random.randn(10, 2),
...                   columns=['Col1', 'Col2'])
>>> df['X'] = pd.Series(['A', 'A', 'A', 'A', 'A',
...                      'B', 'B', 'B', 'B', 'B'])
>>> boxplot = df.boxplot(by='X')
```

A list of strings (i.e. `['X', 'Y']`) can be passed to boxplot in order to group the data by combination of the variables in the x-axis:

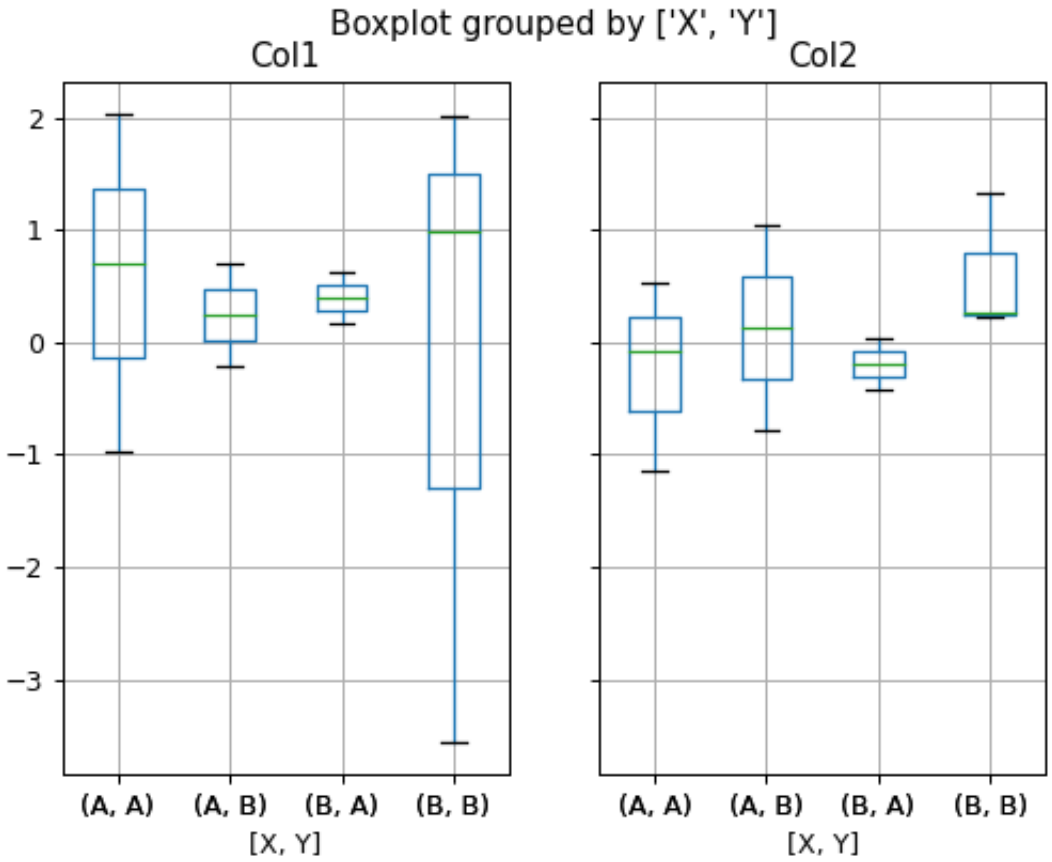
```
>>> df = pd.DataFrame(np.random.randn(10, 3),
...                   columns=['Col1', 'Col2', 'Col3'])
```

(continues on next page)


Boxplot grouped by X
Col1
Col2
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
A
B
[X]
A
B
[X]

(continued from previous page)

```
>>> df['X'] = pd.Series(['A', 'A', 'A', 'A', 'A',
...                      'B', 'B', 'B', 'B', 'B'])
>>> df['Y'] = pd.Series(['A', 'B', 'A', 'B', 'A',
...                      'B', 'A', 'B', 'A', 'B'])
>>> boxplot = df.boxplot(column=['Col1', 'Col2'], by=['X', 'Y'])
```



The layout of boxplot can be adjusted giving a tuple to `layout`:

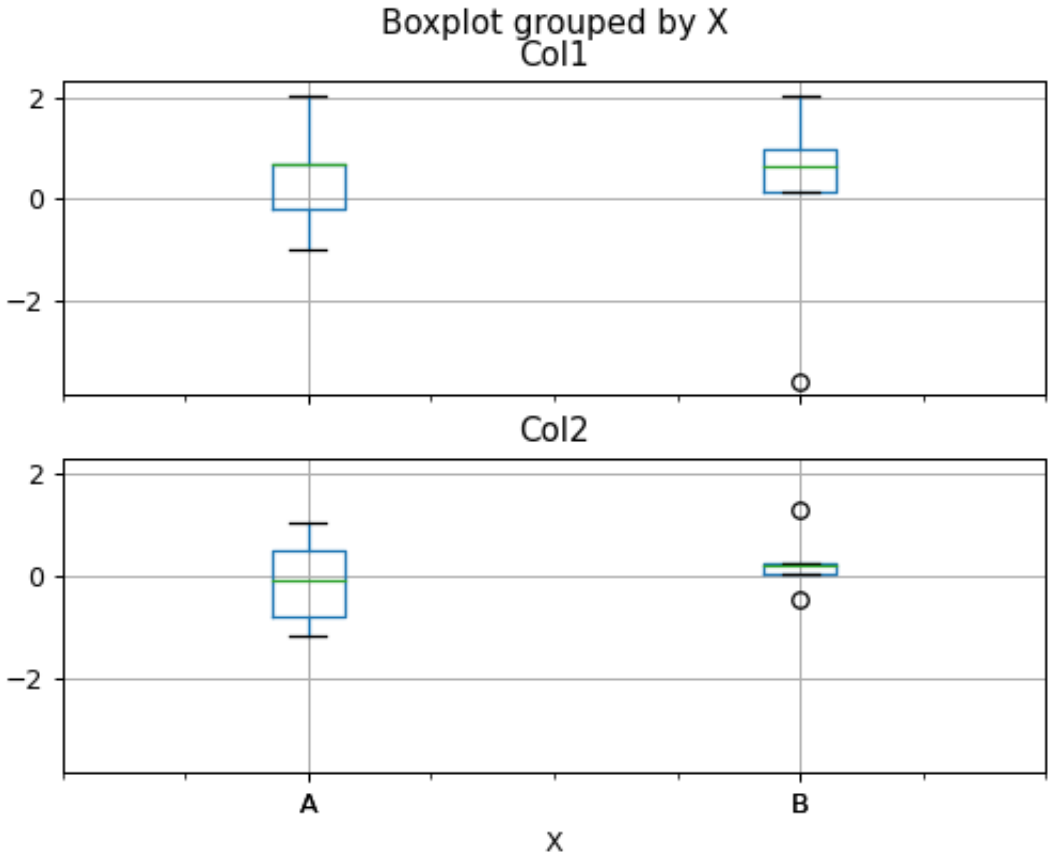
```
>>> boxplot = df.boxplot(column=['Col1', 'Col2'], by='X',
...                      layout=(2, 1))
```

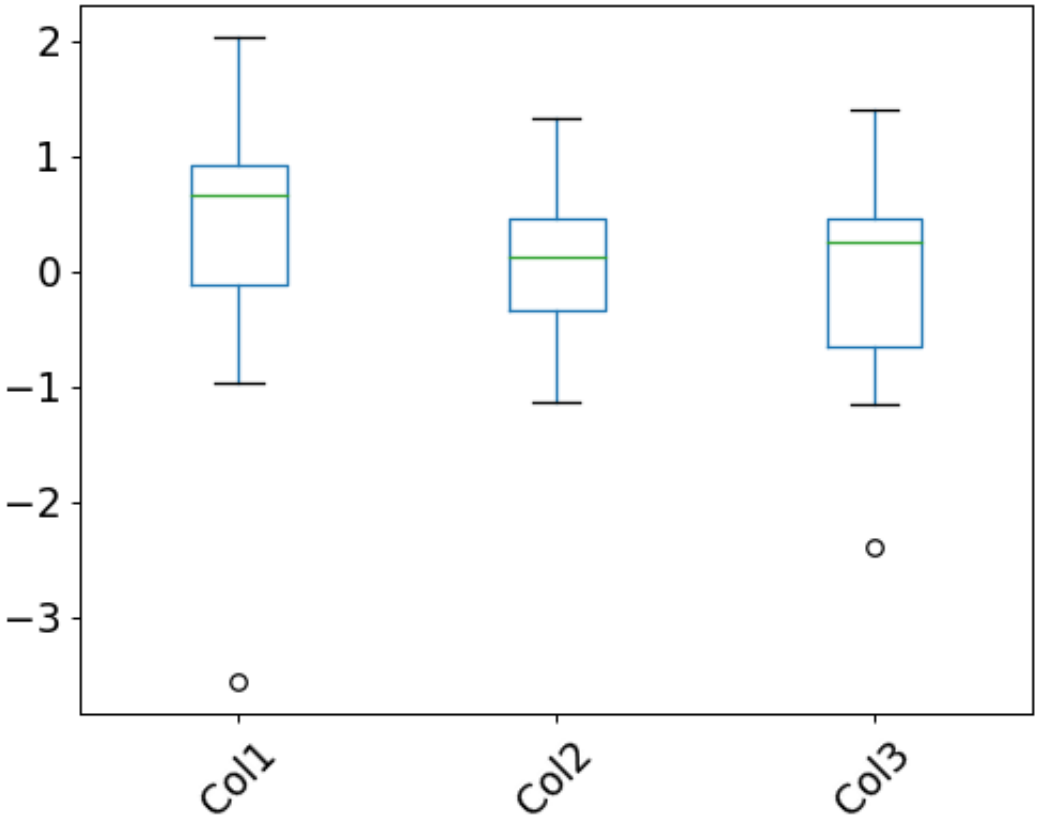
Additional formatting can be done to the boxplot, like suppressing the grid (`grid=False`), rotating the labels in the x-axis (i.e. `rot=45`) or changing the fontsize (i.e. `fontsize=15`):

```
>>> boxplot = df.boxplot(grid=False, rot=45, fontsize=15)
```

The parameter `return_type` can be used to select the type of element returned by *boxplot*. When `return_type='axes'` is selected, the matplotlib axes on which the boxplot is drawn are returned:

```
>>> boxplot = df.boxplot(column=['Col1', 'Col2'], return_type='axes')
>>> type(boxplot)
<class 'matplotlib.axes._axes.Axes'>
```


Boxplot grouped by X
Col1
2
0
−2
Col2
2
0
−2
A
B
X


2
1
0
−1
−2
−3
Col1
Col2
Col3

When grouping with `by`, a Series mapping columns to `return_type` is returned:

```
>>> boxplot = df.boxplot(column=['Col1', 'Col2'], by='X',
...                      return_type='axes')
>>> type(boxplot)
<class 'pandas.core.series.Series'>
```

If `return_type` is *None*, a NumPy array of axes with the same shape as `layout` is returned:

```
>>> boxplot = df.boxplot(column=['Col1', 'Col2'], by='X',
...                      return_type=None)
>>> type(boxplot)
<class 'numpy.ndarray'>
```

AlloViz.AlloViz.Elements.Edges.clip

`Edges.clip`(*lower=None*, *upper=None*, *, *axis: Axis | None = None*, *inplace: bool_t = False*, ***kwargs*) → Self | None

Trim values at input threshold(s).

Assigns values outside boundary to boundary values. Thresholds can be singular values or array like, and in the latter case the clipping is performed element-wise in the specified axis.

Parameters

lower
[float or array-like, default None] Minimum threshold value. All values below this threshold will be set to it. A missing threshold (e.g *NA*) will not clip the value.

upper
[float or array-like, default None] Maximum threshold value. All values above this threshold will be set to it. A missing threshold (e.g *NA*) will not clip the value.

axis
[{{0 or 'index', 1 or 'columns', None}}, default None] Align object with lower and upper along the given axis. For *Series* this parameter is unused and defaults to *None*.

inplace
[bool, default False] Whether to perform the operation in place on the data.

***args, **kwargs**
Additional keywords have no effect but might be accepted for compatibility with numpy.

Returns

Series or DataFrame or None
Same type as calling object with the values outside the clip boundaries replaced or None if `inplace=True`.

See also:

`Series.clip`
Trim values at input threshold in series.

`DataFrame.clip`
Trim values at input threshold in dataframe.

`numpy.clip`
Clip (limit) the values in an array.

Examples

```
>>> data = {'col_0': [9, -3, 0, -1, 5], 'col_1': [-2, -7, 6, 8, -5]}
>>> df = pd.DataFrame(data)
>>> df
   col_0  col_1
0      9     -2
1     -3     -7
2      0      6
3     -1      8
4      5     -5
```

Clips per column using lower and upper thresholds:

```
>>> df.clip(-4, 6)
   col_0  col_1
0      6     -2
1     -3     -4
2      0      6
3     -1      6
4      5     -4
```

Clips using specific lower and upper thresholds per column element:

```
>>> t = pd.Series([2, -4, -1, 6, 3])
>>> t
0    2
1   -4
2   -1
3    6
4    3
dtype: int64
```

```
>>> df.clip(t, t + 4, axis=0)
   col_0  col_1
0      6      2
1     -3     -4
2      0      3
3      6      8
4      5      3
```

Clips using specific lower threshold per column element, with missing values:

```
>>> t = pd.Series([2, -4, np.nan, 6, 3])
>>> t
0    2.0
1   -4.0
2    NaN
3    6.0
4    3.0
dtype: float64
```

```
>>> df.clip(t, axis=0)
col_0  col_1
0      9      2
1     -3     -4
2      0      6
3      6      8
4      5      3
```

AlloViz.AlloViz.Elements.Edges.combine

Edges.**combine**(*other: DataFrame*, *func: Callable[[Series, Series], Series | Hashable]*, *fill_value=None*, *overwrite: bool = True*) → DataFrame

Perform column-wise combine with another DataFrame.

Combines a DataFrame with *other* DataFrame using *func* to element-wise combine columns. The row and column indexes of the resulting DataFrame will be the union of the two.

Parameters

other
[DataFrame] The DataFrame to merge column-wise.

func
[function] Function that takes two series as inputs and return a Series or a scalar. Used to merge the two dataframes column by columns.

fill_value
[scalar value, default None] The value to fill NaNs with prior to passing any column to the merge func.

overwrite
[bool, default True] If True, columns in *self* that do not exist in *other* will be overwritten with NaNs.

Returns

DataFrame
Combination of the provided DataFrames.

See also:

DataFrame.combine_first
Combine two DataFrame objects and default to non-null values in frame calling the method.

Examples

Combine using a simple function that chooses the smaller column.

```
>>> df1 = pd.DataFrame({'A': [0, 0], 'B': [4, 4]})
>>> df2 = pd.DataFrame({'A': [1, 1], 'B': [3, 3]})
>>> take_smaller = lambda s1, s2: s1 if s1.sum() < s2.sum() else s2
>>> df1.combine(df2, take_smaller)
   A  B
0  0  3
1  0  3
```

Example using a true element-wise combine function.

```
>>> df1 = pd.DataFrame({'A': [5, 0], 'B': [2, 4]})
>>> df2 = pd.DataFrame({'A': [1, 1], 'B': [3, 3]})
>>> df1.combine(df2, np.minimum)
   A  B
0  1  2
1  0  3
```

Using *fill_value* fills Nones prior to passing the column to the merge function.

```
>>> df1 = pd.DataFrame({'A': [0, 0], 'B': [None, 4]})
>>> df2 = pd.DataFrame({'A': [1, 1], 'B': [3, 3]})
>>> df1.combine(df2, take_smaller, fill_value=-5)
   A    B
0  0 -5.0
1  0  4.0
```

However, if the same element in both dataframes is None, that None is preserved

```
>>> df1 = pd.DataFrame({'A': [0, 0], 'B': [None, 4]})
>>> df2 = pd.DataFrame({'A': [1, 1], 'B': [None, 3]})
>>> df1.combine(df2, take_smaller, fill_value=-5)
    A    B
0  0 -5.0
1  0  3.0
```

Example that demonstrates the use of *overwrite* and behavior when the axis differ between the dataframes.

```
>>> df1 = pd.DataFrame({'A': [0, 0], 'B': [4, 4]})
>>> df2 = pd.DataFrame({'B': [3, 3], 'C': [-10, 1], }, index=[1, 2])
>>> df1.combine(df2, take_smaller)
     A    B     C
0  NaN  NaN   NaN
1  NaN  3.0 -10.0
2  NaN  3.0   1.0
```

```
>>> df1.combine(df2, take_smaller, overwrite=False)
     A    B     C
0  0.0  NaN   NaN
1  0.0  3.0 -10.0
2  NaN  3.0   1.0
```

Demonstrating the preference of the passed in dataframe.

```
>>> df2 = pd.DataFrame({'B': [3, 3], 'C': [1, 1], }, index=[1, 2])
>>> df2.combine(df1, take_smaller)
   A    B   C
0  0.0  NaN NaN
1  0.0  3.0 NaN
2  NaN  3.0 NaN
```

```
>>> df2.combine(df1, take_smaller, overwrite=False)
     A    B   C
```

(continues on next page)

(continued from previous page)

```
0  0.0  NaN NaN
1  0.0  3.0 1.0
2  NaN  3.0 1.0
```

AlloViz.AlloViz.Elements.Edges.combine_first

Edges.**combine_first**(*other: DataFrame*) → DataFrame

Update null elements with value in the same location in *other*.

Combine two DataFrame objects by filling null values in one DataFrame with non-null values from other DataFrame. The row and column indexes of the resulting DataFrame will be the union of the two. The resulting dataframe contains the 'first' dataframe values and overrides the second one values where both first.loc[index, col] and second.loc[index, col] are not missing values, upon calling first.combine_first(second).

Parameters

other
[DataFrame] Provided DataFrame to use to fill null values.

Returns

DataFrame
The result of combining the provided DataFrame with the other object.

See also:

DataFrame.combine
Perform series-wise operation on two DataFrames using a given function.

Examples

```
>>> df1 = pd.DataFrame({'A': [None, 0], 'B': [None, 4]})
>>> df2 = pd.DataFrame({'A': [1, 1], 'B': [3, 3]})
>>> df1.combine_first(df2)
     A    B
0  1.0  3.0
1  0.0  4.0
```

Null values still persist if the location of that null value does not exist in *other*

```
>>> df1 = pd.DataFrame({'A': [None, 0], 'B': [4, None]})
>>> df2 = pd.DataFrame({'B': [3, 3], 'C': [1, 1]}, index=[1, 2])
>>> df1.combine_first(df2)
     A    B    C
0  NaN  4.0  NaN
1  0.0  3.0  1.0
2  NaN  3.0  1.0
```

AlloViz.AlloViz.Elements.Edges.compare

Edges.**compare**(*other: DataFrame*, *align_axis: Axis = 1*, *keep_shape: bool = False*, *keep_equal: bool = False*, *result_names: Suffixes = ('self', 'other')*) → DataFrame

Compare to another DataFrame and show the differences.

Parameters

other
[DataFrame] Object to compare with.

align_axis
[{0 or 'index', 1 or 'columns'}, default 1] Determine which axis to align the comparison on.

- **0, or 'index'**
  [Resulting differences are stacked vertically] with rows drawn alternately from self and other.
- **1, or 'columns'**
  [Resulting differences are aligned horizontally] with columns drawn alternately from self and other.

keep_shape
[bool, default False] If true, all rows and columns are kept. Otherwise, only the ones with different values are kept.

keep_equal
[bool, default False] If true, the result keeps values that are equal. Otherwise, equal values are shown as NaNs.

result_names
[tuple, default ('self', 'other')] Set the dataframes names in the comparison.

New in version 1.5.0.

Returns

DataFrame
DataFrame that shows the differences stacked side by side.

The resulting index will be a MultiIndex with 'self' and 'other' stacked alternately at the inner level.

Raises

ValueError
When the two DataFrames don't have identical labels or shape.

See also:

`Series.compare`
Compare with another Series and show differences.

`DataFrame.equals`
Test whether two objects contain the same elements.

Notes

Matching NaNs will not appear as a difference.

Can only compare identically-labeled (i.e. same shape, identical row and column labels) DataFrames

Examples

```
>>> df = pd.DataFrame(
...     {
...         "col1": ["a", "a", "b", "b", "a"],
...         "col2": [1.0, 2.0, 3.0, np.nan, 5.0],
...         "col3": [1.0, 2.0, 3.0, 4.0, 5.0]
...     },
...     columns=["col1", "col2", "col3"],
... )
>>> df
  col1  col2  col3
0    a   1.0   1.0
1    a   2.0   2.0
2    b   3.0   3.0
3    b   NaN   4.0
4    a   5.0   5.0
```

```
>>> df2 = df.copy()
>>> df2.loc[0, 'col1'] = 'c'
>>> df2.loc[2, 'col3'] = 4.0
>>> df2
  col1  col2  col3
0    c   1.0   1.0
1    a   2.0   2.0
2    b   3.0   4.0
3    b   NaN   4.0
4    a   5.0   5.0
```

Align the differences on columns

```
>>> df.compare(df2)
  col1       col3
  self other self other
0    a     c  NaN   NaN
2  NaN   NaN  3.0   4.0
```

Assign result_names

```
>>> df.compare(df2, result_names=("left", "right"))
  col1       col3
  left right left right
0    a     c  NaN   NaN
2  NaN   NaN  3.0   4.0
```

Stack the differences on rows

```
>>> df.compare(df2, align_axis=0)
        col1  col3
0 self     a   NaN
  other    c   NaN
2 self   NaN   3.0
  other  NaN   4.0
```

Keep the equal values

```
>>> df.compare(df2, keep_equal=True)
  col1       col3
  self other self other
0    a     c  1.0   1.0
2    b     b  3.0   4.0
```

Keep all original rows and columns

```
>>> df.compare(df2, keep_shape=True)
  col1       col2       col3
  self other self other self other
0    a     c  NaN   NaN  NaN   NaN
1  NaN   NaN  NaN   NaN  NaN   NaN
2  NaN   NaN  NaN   NaN  3.0   4.0
3  NaN   NaN  NaN   NaN  NaN   NaN
4  NaN   NaN  NaN   NaN  NaN   NaN
```

Keep all original rows and columns and also all original values

```
>>> df.compare(df2, keep_shape=True, keep_equal=True)
  col1       col2       col3
  self other self other self other
0    a     c  1.0   1.0  1.0   1.0
1    a     a  2.0   2.0  2.0   2.0
2    b     b  3.0   3.0  3.0   4.0
3    b     b  NaN   NaN  4.0   4.0
4    a     a  5.0   5.0  5.0   5.0
```

AlloViz.AlloViz.Elements.Edges.convert_dtypes

Edges.**convert_dtypes**(*infer_objects: bool = True*, *convert_string: bool = True*, *convert_integer: bool = True*, *convert_boolean: bool = True*, *convert_floating: bool = True*, *dtype_backend: Literal['pyarrow', 'numpy_nullable'] = 'numpy_nullable'*) → None

Convert columns to the best possible dtypes using dtypes supporting `pd.NA`.

Parameters

infer_objects
[bool, default True] Whether object dtypes should be converted to the best possible types.

convert_string
[bool, default True] Whether object dtypes should be converted to `StringDtype()`.

convert_integer
[bool, default True] Whether, if possible, conversion can be done to integer extension types.

convert_boolean
[bool, defaults True] Whether object dtypes should be converted to `BooleanDtypes()`.

convert_floating
[bool, defaults True] Whether, if possible, conversion can be done to floating extension types. If *convert_integer* is also True, preference will be give to integer dtypes if the floats can be faithfully casted to integers.

New in version 1.2.0.

dtype_backend
[{'numpy_nullable', 'pyarrow'}, default 'numpy_nullable'] Back-end data type applied to the resultant `DataFrame` (still experimental). Behaviour is as follows:

- `"numpy_nullable"`: returns nullable-dtype-backed `DataFrame` (default).
- `"pyarrow"`: returns pyarrow-backed nullable `ArrowDtype` DataFrame.

New in version 2.0.

Returns

Series or DataFrame
Copy of input object with new dtype.

See also:

infer_objects
Infer dtypes of objects.

`to_datetime`
Convert argument to datetime.

`to_timedelta`
Convert argument to timedelta.

`to_numeric`
Convert argument to a numeric type.

Notes

By default, `convert_dtypes` will attempt to convert a Series (or each Series in a DataFrame) to dtypes that support `pd.NA`. By using the options `convert_string`, `convert_integer`, `convert_boolean` and `convert_floating`, it is possible to turn off individual conversions to `StringDtype`, the integer extension types, `BooleanDtype` or floating extension types, respectively.

For object-dtyped columns, if `infer_objects` is `True`, use the inference rules as during normal Series/DataFrame construction. Then, if possible, convert to `StringDtype`, `BooleanDtype` or an appropriate integer or floating extension type, otherwise leave as `object`.

If the dtype is integer, convert to an appropriate integer extension type.

If the dtype is numeric, and consists of all integers, convert to an appropriate integer extension type. Otherwise, convert to an appropriate floating extension type.

Changed in version 1.2: Starting with pandas 1.2, this method also converts float columns to the nullable floating extension type.

In the future, as new dtypes are added that support `pd.NA`, the results of this method will change to support those new dtypes.

Examples

```
>>> df = pd.DataFrame(
...     {
...         "a": pd.Series([1, 2, 3], dtype=np.dtype("int32")),
...         "b": pd.Series(["x", "y", "z"], dtype=np.dtype("O")),
...         "c": pd.Series([True, False, np.nan], dtype=np.dtype("O")),
...         "d": pd.Series(["h", "i", np.nan], dtype=np.dtype("O")),
...         "e": pd.Series([10, np.nan, 20], dtype=np.dtype("float")),
...         "f": pd.Series([np.nan, 100.5, 200], dtype=np.dtype("float")),
...     }
... )
```

Start with a DataFrame with default dtypes.

```
>>> df
   a  b      c    d     e      f
0  1  x   True    h  10.0    NaN
1  2  y  False    i   NaN  100.5
2  3  z    NaN  NaN  20.0  200.0
```

```
>>> df.dtypes
a      int32
b     object
c     object
d     object
e    float64
f    float64
dtype: object
```

Convert the DataFrame to use best possible dtypes.

```
>>> dfn = df.convert_dtypes()
>>> dfn
   a  b      c     d     e      f
0  1  x   True     h    10   <NA>
1  2  y  False     i  <NA>  100.5
2  3  z   <NA>  <NA>    20  200.0
```

```
>>> dfn.dtypes
a             Int32
b    string[python]
c           boolean
d    string[python]
e             Int64
f           Float64
dtype: object
```

Start with a Series of strings and missing data represented by `np.nan`.

```
>>> s = pd.Series(["a", "b", np.nan])
>>> s
0      a
1      b
2    NaN
dtype: object
```

Obtain a Series with dtype `StringDtype`.

```
>>> s.convert_dtypes()
0       a
1       b
2    <NA>
dtype: string
```

AlloViz.AlloViz.Elements.Edges.copy

`Edges.copy`(*deep: bool | None = True*) → None

Make a copy of this object's indices and data.

When `deep=True` (default), a new object will be created with a copy of the calling object's data and indices. Modifications to the data or indices of the copy will not be reflected in the original object (see notes below).

When `deep=False`, a new object will be created without copying the calling object's data or index (only references to the data and index are copied). Any changes to the data of the original will be reflected in the shallow copy (and vice versa).

Parameters

deep
[bool, default True] Make a deep copy, including a copy of the data and the indices. With `deep=False` neither the indices nor the data are copied.

Returns

Series or DataFrame
Object type matches caller.

Notes

When `deep=True`, data is copied but actual Python objects will not be copied recursively, only the reference to the object. This is in contrast to *copy.deepcopy* in the Standard Library, which recursively copies object data (see examples below).

While `Index` objects are copied when `deep=True`, the underlying numpy array is not copied for performance reasons. Since `Index` is immutable, the underlying data can be safely shared and a copy is not needed.

Since pandas is not thread safe, see the gotchas when copying in a threading environment.

When `copy_on_write` in pandas config is set to `True`, the `copy_on_write` config takes effect even when `deep=False`. This means that any changes to the copied data would make a new copy of the data upon write (and vice versa). Changes made to either the original or copied variable would not be reflected in the counterpart. See Copy_on_Write for more information.

Examples

```
>>> s = pd.Series([1, 2], index=["a", "b"])
>>> s
a    1
b    2
dtype: int64
```

```
>>> s_copy = s.copy()
>>> s_copy
a    1
b    2
dtype: int64
```

Shallow copy versus default (deep) copy:

```
>>> s = pd.Series([1, 2], index=["a", "b"])
>>> deep = s.copy()
>>> shallow = s.copy(deep=False)
```

Shallow copy shares data and index with original.

```
>>> s is shallow
False
>>> s.values is shallow.values and s.index is shallow.index
True
```

Deep copy has own copy of data and index.

```
>>> s is deep
False
>>> s.values is deep.values or s.index is deep.index
False
```

Updates to the data shared by shallow copy and original is reflected in both; deep copy remains unchanged.

```
>>> s.iloc[0] = 3
>>> shallow.iloc[1] = 4
>>> s
a    3
b    4
dtype: int64
>>> shallow
a    3
b    4
dtype: int64
>>> deep
a    1
b    2
dtype: int64
```

Note that when copying an object containing Python objects, a deep copy will copy the data, but will not do so recursively. Updating a nested data object will be reflected in the deep copy.

```
>>> s = pd.Series([[1, 2], [3, 4]])
>>> deep = s.copy()
>>> s[0][0] = 10
>>> s
0    [10, 2]
1     [3, 4]
dtype: object
>>> deep
0    [10, 2]
1     [3, 4]
dtype: object
```

** Copy-on-Write is set to true: **

```
>>> with pd.option_context("mode.copy_on_write", True):
...     s = pd.Series([1, 2], index=["a", "b"])
...     copy = s.copy(deep=False)
...     s.iloc[0] = 100
...     s
a    100
b      2
dtype: int64
>>> copy
a    1
b    2
dtype: int64
```

AlloViz.AlloViz.Elements.Edges.corr

Edges.**corr**(*method: CorrelationMethod = 'pearson'*, *min_periods: int = 1*, *numeric_only: bool = False*) → DataFrame

Compute pairwise correlation of columns, excluding NA/null values.

Parameters

method
[{'pearson', 'kendall', 'spearman'} or callable] Method of correlation:

- pearson : standard correlation coefficient
- kendall : Kendall Tau correlation coefficient
- spearman : Spearman rank correlation
- **callable: callable with input two 1d ndarrays**
  and returning a float. Note that the returned matrix from corr will have 1 along the diagonals and will be symmetric regardless of the callable's behavior.

min_periods
[int, optional] Minimum number of observations required per pair of columns to have a valid result. Currently only available for Pearson and Spearman correlation.

numeric_only
[bool, default False] Include only *float*, *int* or *boolean* data.

New in version 1.5.0.

Changed in version 2.0.0: The default value of `numeric_only` is now `False`.

Returns

DataFrame
Correlation matrix.

See also:

`DataFrame.corrwith`
Compute pairwise correlation with another DataFrame or Series.

`Series.corr`
Compute the correlation between two Series.

Notes

Pearson, Kendall and Spearman correlation are currently computed using pairwise complete observations.

- Pearson correlation coefficient
- Kendall rank correlation coefficient
- Spearman's rank correlation coefficient

Examples

```
>>> def histogram_intersection(a, b):
...     v = np.minimum(a, b).sum().round(decimals=1)
...     return v
>>> df = pd.DataFrame([(.2, .3), (.0, .6), (.6, .0), (.2, .1)],
...                   columns=['dogs', 'cats'])
>>> df.corr(method=histogram_intersection)
      dogs  cats
dogs   1.0   0.3
cats   0.3   1.0
```

```
>>> df = pd.DataFrame([(1, 1), (2, np.nan), (np.nan, 3), (4, 4)],
...                   columns=['dogs', 'cats'])
>>> df.corr(min_periods=3)
      dogs  cats
dogs   1.0   NaN
cats   NaN   1.0
```

AlloViz.AlloViz.Elements.Edges.corrwith

`Edges.corrwith`(*other: DataFrame | Series*, *axis: Axis = 0*, *drop: bool = False*, *method: CorrelationMethod = 'pearson'*, *numeric_only: bool = False*) → Series

Compute pairwise correlation.

Pairwise correlation is computed between rows or columns of DataFrame with rows or columns of Series or DataFrame. DataFrames are first aligned along both axes before computing the correlations.

Parameters

other
[DataFrame, Series] Object with which to compute correlations.

axis
[{0 or 'index', 1 or 'columns'}, default 0] The axis to use. 0 or 'index' to compute row-wise, 1 or 'columns' for column-wise.

drop
[bool, default False] Drop missing indices from result.

method
[{'pearson', 'kendall', 'spearman'} or callable] Method of correlation:

- pearson : standard correlation coefficient
- kendall : Kendall Tau correlation coefficient
- spearman : Spearman rank correlation
- **callable: callable with input two 1d ndarrays**
  and returning a float.

numeric_only
[bool, default False] Include only *float*, *int* or *boolean* data.

New in version 1.5.0.

Changed in version 2.0.0: The default value of `numeric_only` is now `False`.

Returns

Series
Pairwise correlations.

See also:

`DataFrame.corr`
Compute pairwise correlation of columns.

Examples

```
>>> index = ["a", "b", "c", "d", "e"]
>>> columns = ["one", "two", "three", "four"]
>>> df1 = pd.DataFrame(np.arange(20).reshape(5, 4), index=index,␣
↪columns=columns)
>>> df2 = pd.DataFrame(np.arange(16).reshape(4, 4), index=index[:4],␣
↪columns=columns)
>>> df1.corrwith(df2)
one      1.0
two      1.0
three    1.0
four     1.0
dtype: float64
```

```
>>> df2.corrwith(df1, axis=1)
a    1.0
b    1.0
c    1.0
```

(continues on next page)

(continued from previous page)

```
d    1.0
e    NaN
dtype: float64
```

AlloViz.AlloViz.Elements.Edges.count

Edges.**count**(*axis: Axis = 0*, *numeric_only: bool = False*)

Count non-NA cells for each column or row.

The values *None*, *NaN*, *NaT*, `pandas.NA` are considered NA.

Parameters

axis
[{0 or 'index', 1 or 'columns'}, default 0] If 0 or 'index' counts are generated for each column. If 1 or 'columns' counts are generated for each row.

numeric_only
[bool, default False] Include only *float*, *int* or *boolean* data.

Returns

Series
For each column/row the number of non-NA/null entries.

See also:

Series.count
Number of non-NA elements in a Series.

DataFrame.value_counts
Count unique combinations of columns.

DataFrame.shape
Number of DataFrame rows and columns (including NA elements).

DataFrame.isna
Boolean same-sized DataFrame showing places of NA elements.

Examples

Constructing DataFrame from a dictionary:

```
>>> df = pd.DataFrame({"Person":
...                    ["John", "Myla", "Lewis", "John", "Myla"],
...                    "Age": [24., np.nan, 21., 33, 26],
...                    "Single": [False, True, True, True, False]})
>>> df
   Person   Age  Single
0    John  24.0   False
1    Myla   NaN    True
2   Lewis  21.0    True
3    John  33.0    True
4    Myla  26.0   False
```

Notice the uncounted NA values:

```
>>> df.count()
Person    5
Age       4
Single    5
dtype: int64
```

Counts for each **row**:

```
>>> df.count(axis='columns')
0    3
1    2
2    3
3    3
4    3
dtype: int64
```

AlloViz.AlloViz.Elements.Edges.cov

`Edges.cov`(*min_periods: None | int = None*, *ddof: int | None = 1*, *numeric_only: bool = False*) → DataFrame

Compute pairwise covariance of columns, excluding NA/null values.

Compute the pairwise covariance among the series of a DataFrame. The returned data frame is the covariance matrix of the columns of the DataFrame.

Both NA and null values are automatically excluded from the calculation. (See the note below about bias from missing values.) A threshold can be set for the minimum number of observations for each value created. Comparisons with observations below this threshold will be returned as `NaN`.

This method is generally used for the analysis of time series data to understand the relationship between different measures across time.

Parameters

min_periods
[int, optional] Minimum number of observations required per pair of columns to have a valid result.

ddof
[int, default 1] Delta degrees of freedom. The divisor used in calculations is `N - ddof`, where `N` represents the number of elements. This argument is applicable only when no `nan` is in the dataframe.

numeric_only
[bool, default False] Include only *float*, *int* or *boolean* data.

New in version 1.5.0.

Changed in version 2.0.0: The default value of `numeric_only` is now `False`.

Returns

DataFrame
The covariance matrix of the series of the DataFrame.

See also:

Series.cov
Compute covariance with another Series.

core.window.ewm.ExponentialMovingWindow.cov
Exponential weighted sample covariance.

core.window.expanding.Expanding.cov
Expanding sample covariance.

core.window.rolling.Rolling.cov
Rolling sample covariance.

Notes

Returns the covariance matrix of the DataFrame's time series. The covariance is normalized by N-ddof.

For DataFrames that have Series that are missing data (assuming that data is missing at random) the returned covariance matrix will be an unbiased estimate of the variance and covariance between the member Series.

However, for many applications this estimate may not be acceptable because the estimate covariance matrix is not guaranteed to be positive semi-definite. This could lead to estimate correlations having absolute values which are greater than one, and/or a non-invertible covariance matrix. See Estimation of covariance matrices for more details.

Examples

```
>>> df = pd.DataFrame([(1, 2), (0, 3), (2, 0), (1, 1)],
...                   columns=['dogs', 'cats'])
>>> df.cov()
          dogs      cats
dogs  0.666667 -1.000000
cats -1.000000  1.666667
```

```
>>> np.random.seed(42)
>>> df = pd.DataFrame(np.random.randn(1000, 5),
...                   columns=['a', 'b', 'c', 'd', 'e'])
>>> df.cov()
          a         b         c         d         e
a  0.998438 -0.020161  0.059277 -0.008943  0.014144
b -0.020161  1.059352 -0.008543 -0.024738  0.009826
c  0.059277 -0.008543  1.010670 -0.001486 -0.000271
d -0.008943 -0.024738 -0.001486  0.921297 -0.013692
e  0.014144  0.009826 -0.000271 -0.013692  0.977795
```

Minimum number of periods

This method also supports an optional `min_periods` keyword that specifies the required minimum number of non-NA observations for each column pair in order to have a valid result:

```
>>> np.random.seed(42)
>>> df = pd.DataFrame(np.random.randn(20, 3),
...                   columns=['a', 'b', 'c'])
>>> df.loc[df.index[:5], 'a'] = np.nan
```

(continues on next page)

(continued from previous page)

```
>>> df.loc[df.index[5:10], 'b'] = np.nan
>>> df.cov(min_periods=12)
          a         b         c
a  0.316741       NaN -0.150812
b       NaN  1.248003  0.191417
c -0.150812  0.191417  0.895202
```

AlloViz.AlloViz.Elements.Edges.cummax

Edges.**cummax**(*axis: Axis | None = None*, *skipna: bool = True*, **args*, ***kwargs*)

Return cumulative maximum over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative maximum.

Parameters

axis
[{0 or 'index', 1 or 'columns'}, default 0] The index or the name of the axis. 0 is equivalent to None or 'index'. For *Series* this parameter is unused and defaults to 0.

skipna
[bool, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

***args, **kwargs**
Additional keywords have no effect but might be accepted for compatibility with NumPy.

Returns

Series or DataFrame
Return cumulative maximum of Series or DataFrame.

See also:

core.window.expanding.Expanding.max
Similar functionality but ignores `NaN` values.

DataFrame.max
Return the maximum over DataFrame axis.

DataFrame.cummax
Return cumulative maximum over DataFrame axis.

DataFrame.cummin
Return cumulative minimum over DataFrame axis.

DataFrame.cumsum
Return cumulative sum over DataFrame axis.

DataFrame.cumprod
Return cumulative product over DataFrame axis.

Examples

Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cummax()
0    2.0
1    NaN
2    5.0
3    5.0
4    5.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cummax(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                   columns=list('AB'))
>>> df
     A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the maximum in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cummax()
     A    B
0  2.0  1.0
1  3.0  NaN
2  3.0  1.0
```

To iterate over columns and find the maximum in each row, use `axis=1`

```
>>> df.cummax(axis=1)
     A    B
0  2.0  2.0
1  3.0  NaN
2  1.0  1.0
```

AlloViz.AlloViz.Elements.Edges.cummin

Edges.**cummin**(*axis: Axis | None = None*, *skipna: bool = True*, **args*, ***kwargs*)

Return cumulative minimum over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative minimum.

Parameters

axis
[{0 or 'index', 1 or 'columns'}, default 0] The index or the name of the axis. 0 is equivalent to None or 'index'. For *Series* this parameter is unused and defaults to 0.

skipna
[bool, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

***args, **kwargs**
Additional keywords have no effect but might be accepted for compatibility with NumPy.

Returns

Series or DataFrame
Return cumulative minimum of Series or DataFrame.

See also:

core.window.expanding.Expanding.min
Similar functionality but ignores `NaN` values.

DataFrame.min
Return the minimum over DataFrame axis.

DataFrame.cummax
Return cumulative maximum over DataFrame axis.

DataFrame.cummin
Return cumulative minimum over DataFrame axis.

DataFrame.cumsum
Return cumulative sum over DataFrame axis.

DataFrame.cumprod
Return cumulative product over DataFrame axis.

Examples

Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cummin()
0    2.0
1    NaN
2    2.0
3   -1.0
4   -1.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cummin(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                   columns=list('AB'))
>>> df
     A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the minimum in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cummin()
     A    B
0  2.0  1.0
1  2.0  NaN
2  1.0  0.0
```

To iterate over columns and find the minimum in each row, use `axis=1`

```
>>> df.cummin(axis=1)
     A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

AlloViz.AlloViz.Elements.Edges.cumprod

Edges.**cumprod**(*axis: Axis | None = None*, *skipna: bool = True*, **args*, ***kwargs*)

Return cumulative product over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative product.

Parameters

axis
[{0 or 'index', 1 or 'columns'}, default 0] The index or the name of the axis. 0 is equivalent to None or 'index'. For *Series* this parameter is unused and defaults to 0.

skipna
[bool, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

***args, **kwargs**
Additional keywords have no effect but might be accepted for compatibility with NumPy.

Returns

Series or DataFrame
Return cumulative product of Series or DataFrame.

See also:

core.window.expanding.Expanding.prod
Similar functionality but ignores `NaN` values.

DataFrame.prod
Return the product over DataFrame axis.

DataFrame.cummax
Return cumulative maximum over DataFrame axis.

DataFrame.cummin
Return cumulative minimum over DataFrame axis.

DataFrame.cumsum
Return cumulative sum over DataFrame axis.

DataFrame.cumprod
Return cumulative product over DataFrame axis.

Examples

Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cumprod()
0     2.0
1     NaN
2    10.0
3   -10.0
4    -0.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cumprod(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                   columns=list('AB'))
>>> df
     A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the product in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cumprod()
     A    B
0  2.0  1.0
1  6.0  NaN
2  6.0  0.0
```

To iterate over columns and find the product in each row, use `axis=1`

```
>>> df.cumprod(axis=1)
     A    B
0  2.0  2.0
1  3.0  NaN
2  1.0  0.0
```

AlloViz.AlloViz.Elements.Edges.cumsum

Edges.**cumsum**(*axis: Axis | None = None*, *skipna: bool = True*, **args*, ***kwargs*)

Return cumulative sum over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative sum.

Parameters

axis
[{0 or 'index', 1 or 'columns'}, default 0] The index or the name of the axis. 0 is equivalent to None or 'index'. For *Series* this parameter is unused and defaults to 0.

skipna
[bool, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

***args, **kwargs**
Additional keywords have no effect but might be accepted for compatibility with NumPy.

Returns

Series or DataFrame
Return cumulative sum of Series or DataFrame.

See also:

core.window.expanding.Expanding.sum
Similar functionality but ignores `NaN` values.

DataFrame.sum
Return the sum over DataFrame axis.

DataFrame.cummax
Return cumulative maximum over DataFrame axis.

DataFrame.cummin
Return cumulative minimum over DataFrame axis.

DataFrame.cumsum
Return cumulative sum over DataFrame axis.

DataFrame.cumprod
Return cumulative product over DataFrame axis.

Examples

Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cumsum()
0    2.0
1    NaN
2    7.0
3    6.0
4    6.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cumsum(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                   columns=list('AB'))
>>> df
     A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the sum in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cumsum()
     A    B
0  2.0  1.0
1  5.0  NaN
2  6.0  1.0
```

To iterate over columns and find the sum in each row, use `axis=1`

```
>>> df.cumsum(axis=1)
     A    B
0  2.0  3.0
1  3.0  NaN
2  1.0  1.0
```

AlloViz.AlloViz.Elements.Edges.describe

`Edges.describe`(*percentiles=None*, *include=None*, *exclude=None*) → None

Generate descriptive statistics.

Descriptive statistics include those that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding `NaN` values.

Analyzes both numeric and object series, as well as `DataFrame` column sets of mixed data types. The output will vary depending on what is provided. Refer to the notes below for more detail.

Parameters

percentiles
[list-like of numbers, optional] The percentiles to include in the output. All should fall between 0 and 1. The default is `[.25, .5, .75]`, which returns the 25th, 50th, and 75th percentiles.

include
['all', list-like of dtypes or None (default), optional] A white list of data types to include in the result. Ignored for `Series`. Here are the options:

- 'all' : All columns of the input will be included in the output.
- A list-like of dtypes : Limits the results to the provided data types. To limit the result to numeric types submit `numpy.number`. To limit it instead to object columns submit the `numpy.object` data type. Strings can also be used in the style of `select_dtypes` (e.g. `df.describe(include=['O'])`). To select pandas categorical columns, use `'category'`
- None (default) : The result will include all numeric columns.

exclude
[list-like of dtypes or None (default), optional,] A black list of data types to omit from the result. Ignored for `Series`. Here are the options:

- A list-like of dtypes : Excludes the provided data types from the result. To exclude numeric types submit `numpy.number`. To exclude object columns submit the data type `numpy.object`. Strings can also be used in the style of `select_dtypes` (e.g. `df.describe(exclude=['O'])`). To exclude pandas categorical columns, use `'category'`
- None (default) : The result will exclude nothing.

Returns

Series or DataFrame
Summary statistics of the Series or Dataframe provided.

See also:

`DataFrame.count`
Count number of non-NA/null observations.

DataFrame.max
Maximum of the values in the object.

DataFrame.min
Minimum of the values in the object.

DataFrame.mean
Mean of the values.

DataFrame.std
Standard deviation of the observations.

DataFrame.select_dtypes
Subset of a DataFrame including/excluding columns based on their dtype.

Notes

For numeric data, the result's index will include `count`, `mean`, `std`, `min`, `max` as well as lower, `50` and upper percentiles. By default the lower percentile is `25` and the upper percentile is `75`. The `50` percentile is the same as the median.

For object data (e.g. strings or timestamps), the result's index will include `count`, `unique`, `top`, and `freq`. The `top` is the most common value. The `freq` is the most common value's frequency. Timestamps also include the `first` and `last` items.

If multiple object values have the highest count, then the `count` and `top` results will be arbitrarily chosen from among those with the highest count.

For mixed data types provided via a `DataFrame`, the default is to return only an analysis of numeric columns. If the dataframe consists only of object and categorical data without any numeric columns, the default is to return an analysis of both the object and categorical columns. If `include='all'` is provided as an option, the result will include a union of attributes of each type.

The *include* and *exclude* parameters can be used to limit which columns in a `DataFrame` are analyzed for the output. The parameters are ignored when analyzing a `Series`.

Examples

Describing a numeric `Series`.

```
>>> s = pd.Series([1, 2, 3])
>>> s.describe()
count    3.0
mean     2.0
std      1.0
min      1.0
25%      1.5
50%      2.0
75%      2.5
max      3.0
dtype: float64
```

Describing a categorical `Series`.

```
>>> s = pd.Series(['a', 'a', 'b', 'c'])
>>> s.describe()
```

(continues on next page)

(continued from previous page)

```
count     4
unique    3
top       a
freq      2
dtype: object
```

Describing a timestamp `Series`.

```
>>> s = pd.Series([
...     np.datetime64("2000-01-01"),
...     np.datetime64("2010-01-01"),
...     np.datetime64("2010-01-01")
... ])
>>> s.describe()
count                      3
mean     2006-09-01 08:00:00
min      2000-01-01 00:00:00
25%      2004-12-31 12:00:00
50%      2010-01-01 00:00:00
75%      2010-01-01 00:00:00
max      2010-01-01 00:00:00
dtype: object
```

Describing a `DataFrame`. By default only numeric fields are returned.

```
>>> df = pd.DataFrame({'categorical': pd.Categorical(['d','e','f']),
...                    'numeric': [1, 2, 3],
...                    'object': ['a', 'b', 'c']
...                    })
>>> df.describe()
       numeric
count      3.0
mean       2.0
std        1.0
min        1.0
25%        1.5
50%        2.0
75%        2.5
max        3.0
```

Describing all columns of a `DataFrame` regardless of data type.

```
>>> df.describe(include='all')
       categorical  numeric object
count            3      3.0      3
unique           3      NaN      3
top              f      NaN      a
freq             1      NaN      1
mean           NaN      2.0    NaN
std            NaN      1.0    NaN
min            NaN      1.0    NaN
25%            NaN      1.5    NaN
50%            NaN      2.0    NaN
```

(continues on next page)

(continued from previous page)

```
75%            NaN      2.5    NaN
max            NaN      3.0    NaN
```

Describing a column from a `DataFrame` by accessing it as an attribute.

```
>>> df.numeric.describe()
count    3.0
mean     2.0
std      1.0
min      1.0
25%      1.5
50%      2.0
75%      2.5
max      3.0
Name: numeric, dtype: float64
```

Including only numeric columns in a `DataFrame` description.

```
>>> df.describe(include=[np.number])
       numeric
count      3.0
mean       2.0
std        1.0
min        1.0
25%        1.5
50%        2.0
75%        2.5
max        3.0
```

Including only string columns in a `DataFrame` description.

```
>>> df.describe(include=[object])
       object
count       3
unique      3
top         a
freq        1
```

Including only categorical columns from a `DataFrame` description.

```
>>> df.describe(include=['category'])
       categorical
count            3
unique           3
top              d
freq             1
```

Excluding numeric columns from a `DataFrame` description.

```
>>> df.describe(exclude=[np.number])
       categorical object
count            3      3
unique           3      3
```

(continues on next page)

(continued from previous page)

```
top              f      a
freq             1      1
```

Excluding object columns from a `DataFrame` description.

```
>>> df.describe(exclude=[object])
       categorical  numeric
count            3      3.0
unique           3      NaN
top              f      NaN
freq             1      NaN
mean           NaN      2.0
std            NaN      1.0
min            NaN      1.0
25%            NaN      1.5
50%            NaN      2.0
75%            NaN      2.5
max            NaN      3.0
```

AlloViz.AlloViz.Elements.Edges.diff

`Edges.diff`(*periods: int = 1*, *axis: Axis = 0*) → DataFrame

First discrete difference of element.

Calculates the difference of a DataFrame element compared with another element in the DataFrame (default is element in previous row).

Parameters

periods
[int, default 1] Periods to shift for calculating difference, accepts negative values.

axis
[{0 or 'index', 1 or 'columns'}, default 0] Take difference over rows (0) or columns (1).

Returns

DataFrame
First differences of the Series.

See also:

`DataFrame.pct_change`
Percent change over given number of periods.

`DataFrame.shift`
Shift index by desired number of periods with an optional time freq.

`Series.diff`
First discrete difference of object.

Notes

For boolean dtypes, this uses `operator.xor()` rather than `operator.sub()`. The result is calculated according to current dtype in DataFrame, however dtype of the result is always float64.

Examples

Difference with previous row

```
>>> df = pd.DataFrame({'a': [1, 2, 3, 4, 5, 6],
...                    'b': [1, 1, 2, 3, 5, 8],
...                    'c': [1, 4, 9, 16, 25, 36]})
>>> df
   a  b   c
0  1  1   1
1  2  1   4
2  3  2   9
3  4  3  16
4  5  5  25
5  6  8  36
```

```
>>> df.diff()
     a    b     c
0  NaN  NaN   NaN
1  1.0  0.0   3.0
2  1.0  1.0   5.0
3  1.0  1.0   7.0
4  1.0  2.0   9.0
5  1.0  3.0  11.0
```

Difference with previous column

```
>>> df.diff(axis=1)
    a  b   c
0 NaN  0   0
1 NaN -1   3
2 NaN -1   7
3 NaN -1  13
4 NaN  0  20
5 NaN  2  28
```

Difference with 3rd previous row

```
>>> df.diff(periods=3)
     a    b     c
0  NaN  NaN   NaN
1  NaN  NaN   NaN
2  NaN  NaN   NaN
3  3.0  2.0  15.0
4  3.0  4.0  21.0
5  3.0  6.0  27.0
```

Difference with following row

```
>>> df.diff(periods=-1)
     a    b     c
0 -1.0  0.0  -3.0
1 -1.0 -1.0  -5.0
2 -1.0 -1.0  -7.0
3 -1.0 -2.0  -9.0
4 -1.0 -3.0 -11.0
5  NaN  NaN   NaN
```

Overflow in input dtype

```
>>> df = pd.DataFrame({'a': [1, 0]}, dtype=np.uint8)
>>> df.diff()
       a
0    NaN
1  255.0
```

AlloViz.AlloViz.Elements.Edges.div

Edges.**div**(*other*, *axis: Axis = 'columns'*, *level=None*, *fill_value=None*)

Get Floating division of dataframe and other, element-wise (binary operator *truediv*).

Equivalent to `dataframe / other`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *rtruediv*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *floordiv*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other
[scalar, sequence, Series, dict or DataFrame] Any single or multiple element data structure, or list-like object.

axis
[{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns. (1 or 'columns'). For Series input, axis to match Series index on.

level
[int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value
[float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame
Result of the arithmetic operation.

See also:

DataFrame.add
Add DataFrames.

DataFrame.sub
Subtract DataFrames.

DataFrame.mul
Multiply DataFrames.

DataFrame.div
Divide DataFrames (float division).

DataFrame.truediv
Divide DataFrames (float division).

DataFrame.floordiv
Divide DataFrames (integer division).

DataFrame.mod
Calculate modulo (remainder after division).

DataFrame.pow
Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
```

(continues on next page)

(continued from previous page)

```
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a dictionary by axis.

```
>>> df.mul({'angles': 0, 'degrees': 2})
            angles  degrees
circle           0      720
triangle         0      360
rectangle        0      720
```

```
>>> df.mul({'circle': 0, 'triangle': 2, 'rectangle': 3}, axis='index')
            angles  degrees
circle           0        0
triangle         6      360
rectangle       12     1080
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
```

(continues on next page)

(continued from previous page)

```
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

AlloViz.AlloViz.Elements.Edges.divide

Edges.**divide**(*other*, *axis: Axis = 'columns'*, *level=None*, *fill_value=None*)

Get Floating division of dataframe and other, element-wise (binary operator *truediv*).

Equivalent to `dataframe / other`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *rtruediv*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *floordiv*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other
[scalar, sequence, Series, dict or DataFrame] Any single or multiple element data structure, or list-like object.

axis
[{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns. (1 or 'columns'). For Series input, axis to match Series index on.

level
[int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value
[float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame
Result of the arithmetic operation.

See also:

`DataFrame.add`
Add DataFrames.

`DataFrame.sub`
Subtract DataFrames.

`DataFrame.mul`
Multiply DataFrames.

`DataFrame.div`
Divide DataFrames (float division).

`DataFrame.truediv`
Divide DataFrames (float division).

`DataFrame.floordiv`
Divide DataFrames (integer division).

`DataFrame.mod`
Calculate modulo (remainder after division).

`DataFrame.pow`
Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a dictionary by axis.

```
>>> df.mul({'angles': 0, 'degrees': 2})
           angles  degrees
circle          0      720
triangle        0      360
rectangle       0      720
```

```
>>> df.mul({'circle': 0, 'triangle': 2, 'rectangle': 3}, axis='index')
           angles  degrees
circle          0        0
triangle        6      360
rectangle      12     1080
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
```

(continues on next page)

(continued from previous page)

```
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

AlloViz.AlloViz.Elements.Edges.dot

Edges.**dot**(*other: AnyArrayLike | DataFrame*) → DataFrame | Series

Compute the matrix multiplication between the DataFrame and other.

This method computes the matrix product between the DataFrame and the values of an other Series, DataFrame or a numpy array.

It can also be called using `self @ other`.

Parameters

other
[Series, DataFrame or array-like] The other object to compute the matrix product with.

Returns

Series or DataFrame
If other is a Series, return the matrix product between self and other as a Series. If other is a DataFrame or a numpy.array, return the matrix product of self and other in a DataFrame of a np.array.

See also:

`Series.dot`
Similar method for Series.

Notes

The dimensions of DataFrame and other must be compatible in order to compute the matrix multiplication. In addition, the column names of DataFrame and the index of other must contain the same values, as they will be aligned prior to the multiplication.

The dot method for Series computes the inner product, instead of the matrix product here.

Examples

Here we multiply a DataFrame with a Series.

```
>>> df = pd.DataFrame([[0, 1, -2, -1], [1, 1, 1, 1]])
>>> s = pd.Series([1, 1, 2, 1])
>>> df.dot(s)
0    -4
1     5
dtype: int64
```

Here we multiply a DataFrame with another DataFrame.

```
>>> other = pd.DataFrame([[0, 1], [1, 2], [-1, -1], [2, 0]])
>>> df.dot(other)
    0   1
0   1   4
1   2   2
```

Note that the dot method give the same result as @

```
>>> df @ other
    0   1
0   1   4
1   2   2
```

The dot method works also if other is an np.array.

```
>>> arr = np.array([[0, 1], [1, 2], [-1, -1], [2, 0]])
>>> df.dot(arr)
    0   1
0   1   4
1   2   2
```

Note how shuffling of the objects does not change the result.

```
>>> s2 = s.reindex([1, 0, 2, 3])
>>> df.dot(s2)
0    -4
1     5
dtype: int64
```

AlloViz.AlloViz.Elements.Edges.drop

Edges.**drop**(*labels: IndexLabel | None = None*, ***, *axis: Axis = 0*, *index: IndexLabel | None = None*, *columns: IndexLabel | None = None*, *level: Level | None = None*, *inplace: bool = False*, *errors: IgnoreRaise = 'raise'*) → DataFrame | None

Drop specified labels from rows or columns.

Remove rows or columns by specifying label names and corresponding axis, or by directly specifying index or column names. When using a multi-index, labels on different levels can be removed by specifying the level. See the user guide for more information about the now unused levels.

Parameters

labels
[single label or list-like] Index or column labels to drop. A tuple will be used as a single label and not treated as a list-like.

axis
[{0 or 'index', 1 or 'columns'}, default 0] Whether to drop labels from the index (0 or 'index') or columns (1 or 'columns').

index
[single label or list-like] Alternative to specifying axis (`labels, axis=0` is equivalent to `index=labels`).

columns
[single label or list-like] Alternative to specifying axis (`labels, axis=1` is equivalent to `columns=labels`).

level
[int or level name, optional] For MultiIndex, level from which the labels will be removed.

inplace
[bool, default False] If False, return a copy. Otherwise, do operation in place and return None.

errors
[{'ignore', 'raise'}, default 'raise'] If 'ignore', suppress error and only existing labels are dropped.

Returns

DataFrame or None
Returns DataFrame or None DataFrame with the specified index or column labels removed or None if inplace=True.

Raises

KeyError
If any of the labels is not found in the selected axis.

See also:

`DataFrame.loc`
Label-location based indexer for selection by label.

`DataFrame.dropna`
Return DataFrame with labels on given axis omitted where (all or any) data are missing.

`DataFrame.drop_duplicates`
Return DataFrame with duplicate rows removed, optionally only considering certain columns.

`Series.drop`
Return Series with specified index labels removed.

Examples

```
>>> df = pd.DataFrame(np.arange(12).reshape(3, 4),
...                   columns=['A', 'B', 'C', 'D'])
>>> df
   A  B   C   D
0  0  1   2   3
1  4  5   6   7
2  8  9  10  11
```

Drop columns

```
>>> df.drop(['B', 'C'], axis=1)
   A   D
0  0   3
1  4   7
2  8  11
```

```
>>> df.drop(columns=['B', 'C'])
   A   D
0  0   3
1  4   7
2  8  11
```

Drop a row by index

```
>>> df.drop([0, 1])
   A  B   C   D
2  8  9  10  11
```

Drop columns and/or rows of MultiIndex DataFrame

```
>>> midx = pd.MultiIndex(levels=[['llama', 'cow', 'falcon'],
...                              ['speed', 'weight', 'length']],
...                      codes=[[0, 0, 0, 1, 1, 1, 2, 2, 2],
...                             [0, 1, 2, 0, 1, 2, 0, 1, 2]])
>>> df = pd.DataFrame(index=midx, columns=['big', 'small'],
...                   data=[[45, 30], [200, 100], [1.5, 1], [30, 20],
...                         [250, 150], [1.5, 0.8], [320, 250],
...                         [1, 0.8], [0.3, 0.2]])
>>> df
                big     small
llama   speed   45.0    30.0
        weight  200.0   100.0
        length  1.5     1.0
cow     speed   30.0    20.0
        weight  250.0   150.0
        length  1.5     0.8
falcon  speed   320.0   250.0
        weight  1.0     0.8
        length  0.3     0.2
```

Drop a specific index combination from the MultiIndex DataFrame, i.e., drop the combination `'falcon'` and `'weight'`, which deletes only the corresponding row

```
>>> df.drop(index=('falcon', 'weight'))
               big     small
llama   speed   45.0    30.0
        weight  200.0   100.0
        length  1.5     1.0
cow     speed   30.0    20.0
        weight  250.0   150.0
        length  1.5     0.8
falcon  speed   320.0   250.0
        length  0.3     0.2
```

```
>>> df.drop(index='cow', columns='small')
               big
llama   speed   45.0
        weight  200.0
        length  1.5
falcon  speed   320.0
        weight  1.0
        length  0.3
```

```
>>> df.drop(index='length', level=1)
               big     small
llama   speed   45.0    30.0
        weight  200.0   100.0
cow     speed   30.0    20.0
        weight  250.0   150.0
falcon  speed   320.0   250.0
        weight  1.0     0.8
```

AlloViz.AlloViz.Elements.Edges.drop_duplicates

Edges.**drop_duplicates**(*subset: Hashable | Sequence[Hashable] | None = None*, ***, *keep: DropKeep = 'first'*, *inplace: bool = False*, *ignore_index: bool = False*) → DataFrame | None

Return DataFrame with duplicate rows removed.

Considering certain columns is optional. Indexes, including time indexes are ignored.

Parameters

subset
: [column label or sequence of labels, optional] Only consider certain columns for identifying duplicates, by default use all of the columns.

keep
: [{'first', 'last', `False`}, default 'first'] Determines which duplicates (if any) to keep.

- 'first' : Drop duplicates except for the first occurrence.
- 'last' : Drop duplicates except for the last occurrence.
- `False` : Drop all duplicates.

inplace
: [bool, default `False`] Whether to modify the DataFrame rather than creating a new one.

ignore_index
[bool, default `False`] If `True`, the resulting axis will be labeled 0, 1, …, n - 1.

Returns

DataFrame or None
DataFrame with duplicates removed or None if `inplace=True`.

See also:

`DataFrame.value_counts`
Count unique combinations of columns.

Examples

Consider dataset containing ramen rating.

```
>>> df = pd.DataFrame({
...     'brand': ['Yum Yum', 'Yum Yum', 'Indomie', 'Indomie', 'Indomie'],
...     'style': ['cup', 'cup', 'cup', 'pack', 'pack'],
...     'rating': [4, 4, 3.5, 15, 5]
... })
>>> df
    brand style  rating
0  Yum Yum   cup     4.0
1  Yum Yum   cup     4.0
2  Indomie   cup     3.5
3  Indomie  pack    15.0
4  Indomie  pack     5.0
```

By default, it removes duplicate rows based on all columns.

```
>>> df.drop_duplicates()
    brand style  rating
0  Yum Yum   cup     4.0
2  Indomie   cup     3.5
3  Indomie  pack    15.0
4  Indomie  pack     5.0
```

To remove duplicates on specific column(s), use `subset`.

```
>>> df.drop_duplicates(subset=['brand'])
    brand style  rating
0  Yum Yum   cup     4.0
2  Indomie   cup     3.5
```

To remove duplicates and keep last occurrences, use `keep`.

```
>>> df.drop_duplicates(subset=['brand', 'style'], keep='last')
    brand style  rating
1  Yum Yum   cup     4.0
2  Indomie   cup     3.5
4  Indomie  pack     5.0
```

AlloViz.AlloViz.Elements.Edges.droplevel

Edges.**droplevel**(*level: Hashable | Sequence[Hashable]*, *axis: int | Literal['index', 'columns', 'rows'] = 0*) → None

Return Series/DataFrame with requested index / column level(s) removed.

Parameters

level
[int, str, or list-like] If a string is given, must be the name of a level If list-like, elements must be names or positional indexes of levels.

axis
[{0 or 'index', 1 or 'columns'}, default 0] Axis along which the level(s) is removed:

- 0 or 'index': remove level(s) in column.
- 1 or 'columns': remove level(s) in row.

For *Series* this parameter is unused and defaults to 0.

Returns

Series/DataFrame
Series/DataFrame with requested index / column level(s) removed.

Examples

```
>>> df = pd.DataFrame([
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12]
... ]).set_index([0, 1]).rename_axis(['a', 'b'])
```

```
>>> df.columns = pd.MultiIndex.from_tuples([
...     ('c', 'e'), ('d', 'f')
... ], names=['level_1', 'level_2'])
```

```
>>> df
level_1   c   d
level_2   e   f
a b
1 2       3   4
5 6       7   8
9 10     11  12
```

```
>>> df.droplevel('a')
level_1   c   d
level_2   e   f
b
2         3   4
6         7   8
10       11  12
```

```
>>> df.droplevel('level_2', axis=1)
level_1   c   d
a b
1 2       3   4
5 6       7   8
9 10     11  12
```

AlloViz.AlloViz.Elements.Edges.dropna

Edges.**dropna**(*, *axis: Axis = 0*, *how: AnyAll | lib.NoDefault = _NoDefault.no_default*, *thresh: int | lib.NoDefault = _NoDefault.no_default*, *subset: IndexLabel | None = None*, *inplace: bool = False*, *ignore_index: bool = False*) → DataFrame | None

Remove missing values.

See the User Guide for more on which values are considered missing, and how to work with missing data.

Parameters

axis
[{0 or 'index', 1 or 'columns'}, default 0] Determine if rows or columns which contain missing values are removed.

- 0, or 'index' : Drop rows which contain missing values.
- 1, or 'columns' : Drop columns which contain missing value.

Only a single axis is allowed.

how
[{'any', 'all'}, default 'any'] Determine if row or column is removed from DataFrame, when we have at least one NA or all NA.

- 'any' : If any NA values are present, drop that row or column.
- 'all' : If all values are NA, drop that row or column.

thresh
[int, optional] Require that many non-NA values. Cannot be combined with how.

subset
[column label or sequence of labels, optional] Labels along other axis to consider, e.g. if you are dropping rows these would be a list of columns to include.

inplace
[bool, default False] Whether to modify the DataFrame rather than creating a new one.

ignore_index
[bool, default `False`] If `True`, the resulting axis will be labeled 0, 1, …, n - 1.

New in version 2.0.0.

Returns

DataFrame or None
DataFrame with NA entries dropped from it or None if `inplace=True`.

See also:

`DataFrame.isna`
Indicate missing values.

DataFrame.notna
Indicate existing (non-missing) values.

DataFrame.fillna
Replace missing values.

Series.dropna
Drop missing values.

Index.dropna
Drop missing indices.

Examples

```
>>> df = pd.DataFrame({"name": ['Alfred', 'Batman', 'Catwoman'],
...                    "toy": [np.nan, 'Batmobile', 'Bullwhip'],
...                    "born": [pd.NaT, pd.Timestamp("1940-04-25"),
...                             pd.NaT]})
>>> df
       name        toy       born
0    Alfred        NaN        NaT
1    Batman  Batmobile 1940-04-25
2  Catwoman   Bullwhip        NaT
```

Drop the rows where at least one element is missing.

```
>>> df.dropna()
     name        toy       born
1  Batman  Batmobile 1940-04-25
```

Drop the columns where at least one element is missing.

```
>>> df.dropna(axis='columns')
       name
0    Alfred
1    Batman
2  Catwoman
```

Drop the rows where all elements are missing.

```
>>> df.dropna(how='all')
       name        toy       born
0    Alfred        NaN        NaT
1    Batman  Batmobile 1940-04-25
2  Catwoman   Bullwhip        NaT
```

Keep only the rows with at least 2 non-NA values.

```
>>> df.dropna(thresh=2)
       name        toy       born
1    Batman  Batmobile 1940-04-25
2  Catwoman   Bullwhip        NaT
```

Define in which columns to look for missing values.

```
>>> df.dropna(subset=['name', 'toy'])
       name        toy       born
1    Batman  Batmobile 1940-04-25
2  Catwoman   Bullwhip        NaT
```

AlloViz.AlloViz.Elements.Edges.duplicated

Edges.**duplicated**(*subset: Hashable | Sequence[Hashable] | None = None*, *keep: DropKeep = 'first'*) → Series

Return boolean Series denoting duplicate rows.

Considering certain columns is optional.

Parameters

subset
[column label or sequence of labels, optional] Only consider certain columns for identifying duplicates, by default use all of the columns.

keep
[{'first', 'last', False}, default 'first'] Determines which duplicates (if any) to mark.

- `first` : Mark duplicates as `True` except for the first occurrence.
- `last` : Mark duplicates as `True` except for the last occurrence.
- False : Mark all duplicates as `True`.

Returns

Series
Boolean series for each duplicated rows.

See also:

`Index.duplicated`
Equivalent method on index.

`Series.duplicated`
Equivalent method on Series.

`Series.drop_duplicates`
Remove duplicate values from Series.

`DataFrame.drop_duplicates`
Remove duplicate values from DataFrame.

Examples

Consider dataset containing ramen rating.

```
>>> df = pd.DataFrame({
...     'brand': ['Yum Yum', 'Yum Yum', 'Indomie', 'Indomie', 'Indomie'],
...     'style': ['cup', 'cup', 'cup', 'pack', 'pack'],
...     'rating': [4, 4, 3.5, 15, 5]
... })
>>> df
```

(continues on next page)

(continued from previous page)

```
    brand style  rating
0  Yum Yum   cup     4.0
1  Yum Yum   cup     4.0
2  Indomie   cup     3.5
3  Indomie  pack    15.0
4  Indomie  pack     5.0
```

By default, for each set of duplicated values, the first occurrence is set on False and all others on True.

```
>>> df.duplicated()
0    False
1     True
2    False
3    False
4    False
dtype: bool
```

By using 'last', the last occurrence of each set of duplicated values is set on False and all others on True.

```
>>> df.duplicated(keep='last')
0     True
1    False
2    False
3    False
4    False
dtype: bool
```

By setting `keep` on False, all duplicates are True.

```
>>> df.duplicated(keep=False)
0     True
1     True
2    False
3    False
4    False
dtype: bool
```

To find duplicates on specific column(s), use `subset`.

```
>>> df.duplicated(subset=['brand'])
0    False
1     True
2    False
3     True
4     True
dtype: bool
```

AlloViz.AlloViz.Elements.Edges.eq

Edges.**eq**(*other*, *axis: Axis = 'columns'*, *level=None*)

Get Equal to of dataframe and other, element-wise (binary operator *eq*).

Among flexible wrappers (*eq*, *ne*, *le*, *lt*, *ge*, *gt*) to comparison operators.

Equivalent to ==, *!*=, <=, <, >=, > with support to choose axis (rows or columns) and level for comparison.

Parameters

other
[scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.

axis
[{0 or 'index', 1 or 'columns'}, default 'columns'] Whether to compare by the index (0 or 'index') or columns (1 or 'columns').

level
[int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

Returns

DataFrame of bool
Result of the comparison.

See also:

DataFrame.eq
Compare DataFrames for equality elementwise.

DataFrame.ne
Compare DataFrames for inequality elementwise.

DataFrame.le
Compare DataFrames for less than inequality or equality elementwise.

DataFrame.lt
Compare DataFrames for strictly less than inequality elementwise.

DataFrame.ge
Compare DataFrames for greater than inequality or equality elementwise.

DataFrame.gt
Compare DataFrames for strictly greater than inequality elementwise.

Notes

Mismatched indices will be unioned together. *NaN* values are considered different (i.e. *NaN* != *NaN*).

Examples

```
>>> df = pd.DataFrame({'cost': [250, 150, 100],
...                    'revenue': [100, 250, 300]},
...                   index=['A', 'B', 'C'])
>>> df
   cost  revenue
A   250      100
B   150      250
C   100      300
```

Comparison with a scalar, using either the operator or method:

```
>>> df == 100
    cost  revenue
A  False     True
B  False    False
C   True    False
```

```
>>> df.eq(100)
    cost  revenue
A  False     True
B  False    False
C   True    False
```

When *other* is a `Series`, the columns of a DataFrame are aligned with the index of *other* and broadcast:

```
>>> df != pd.Series([100, 250], index=["cost", "revenue"])
    cost  revenue
A   True     True
B   True    False
C  False     True
```

Use the method to control the broadcast axis:

```
>>> df.ne(pd.Series([100, 300], index=["A", "D"]), axis='index')
   cost  revenue
A  True    False
B  True     True
C  True     True
D  True     True
```

When comparing to an arbitrary sequence, the number of columns must match the number elements in *other*:

```
>>> df == [250, 100]
    cost  revenue
A   True     True
B  False    False
C  False    False
```

Use the method to control the axis:

```
>>> df.eq([250, 250, 100], axis='index')
    cost  revenue
A   True    False
B  False     True
C   True    False
```

Compare to a DataFrame of different shape.

```
>>> other = pd.DataFrame({'revenue': [300, 250, 100, 150]},
...                      index=['A', 'B', 'C', 'D'])
>>> other
   revenue
A      300
B      250
C      100
D      150
```

```
>>> df.gt(other)
    cost  revenue
A  False    False
B  False    False
C  False     True
D  False    False
```

Compare to a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'cost': [250, 150, 100, 150, 300, 220],
...                              'revenue': [100, 250, 300, 200, 175, 225]},
...                             index=[['Q1', 'Q1', 'Q1', 'Q2', 'Q2', 'Q2'],
...                                    ['A', 'B', 'C', 'A', 'B', 'C']])
>>> df_multindex
      cost  revenue
Q1 A   250      100
   B   150      250
   C   100      300
Q2 A   150      200
   B   300      175
   C   220      225
```

```
>>> df.le(df_multindex, level=1)
       cost  revenue
Q1 A   True     True
   B   True     True
   C   True     True
Q2 A  False     True
   B   True    False
   C   True    False
```

AlloViz.AlloViz.Elements.Edges.equals

Edges.**equals**(*other: object*) → bool

Test whether two objects contain the same elements.

This function allows two Series or DataFrames to be compared against each other to see if they have the same shape and elements. NaNs in the same location are considered equal.

The row/column index do not need to have the same type, as long as the values are considered equal. Corresponding columns must be of the same dtype.

Parameters

other
[Series or DataFrame] The other Series or DataFrame to be compared with the first.

Returns

bool
True if all elements are the same in both objects, False otherwise.

See also:

Series.eq
Compare two Series objects of the same length and return a Series where each element is True if the element in each Series is equal, False otherwise.

DataFrame.eq
Compare two DataFrame objects of the same shape and return a DataFrame where each element is True if the respective element in each DataFrame is equal, False otherwise.

testing.assert_series_equal
Raises an AssertionError if left and right are not equal. Provides an easy interface to ignore inequality in dtypes, indexes and precision among others.

testing.assert_frame_equal
Like assert_series_equal, but targets DataFrames.

numpy.array_equal
Return True if two arrays have the same shape and elements, False otherwise.

Examples

```
>>> df = pd.DataFrame({1: [10], 2: [20]})
>>> df
    1   2
0  10  20
```

DataFrames df and exactly_equal have the same types and values for their elements and column labels, which will return True.

```
>>> exactly_equal = pd.DataFrame({1: [10], 2: [20]})
>>> exactly_equal
    1   2
0  10  20
>>> df.equals(exactly_equal)
True
```

DataFrames df and different_column_type have the same element types and values, but have different types for the column labels, which will still return True.

```
>>> different_column_type = pd.DataFrame({1.0: [10], 2.0: [20]})
>>> different_column_type
   1.0  2.0
0   10   20
>>> df.equals(different_column_type)
True
```

DataFrames df and different_data_type have different types for the same values for their elements, and will return False even though their column labels are the same values and types.

```
>>> different_data_type = pd.DataFrame({1: [10.0], 2: [20.0]})
>>> different_data_type
      1     2
0  10.0  20.0
>>> df.equals(different_data_type)
False
```

AlloViz.AlloViz.Elements.Edges.eval

`Edges.eval`(*expr: str*, *, *inplace: bool = False*, ***kwargs*) → Any | None

Evaluate a string describing operations on DataFrame columns.

Operates on columns only, not specific rows or elements. This allows *eval* to run arbitrary code, which can make you vulnerable to code injection if you pass user input to this function.

Parameters

expr
[str] The expression string to evaluate.

inplace
[bool, default False] If the expression contains an assignment, whether to perform the operation inplace and mutate the existing DataFrame. Otherwise, a new DataFrame is returned.

****kwargs**
See the documentation for *eval()* for complete details on the keyword arguments accepted by `query()`.

Returns

ndarray, scalar, pandas object, or None
The result of the evaluation or None if `inplace=True`.

See also:

`DataFrame.query`
Evaluates a boolean expression to query the columns of a frame.

`DataFrame.assign`
Can evaluate an expression or function to create new values for a column.

eval
Evaluate a Python expression as a string using various backends.

Notes

For more details see the API documentation for *eval()*. For detailed examples see enhancing performance with eval.

Examples

```
>>> df = pd.DataFrame({'A': range(1, 6), 'B': range(10, 0, -2)})
>>> df
   A   B
0  1  10
1  2   8
2  3   6
3  4   4
4  5   2
>>> df.eval('A + B')
0    11
1    10
2     9
3     8
4     7
dtype: int64
```

Assignment is allowed though by default the original DataFrame is not modified.

```
>>> df.eval('C = A + B')
   A   B   C
0  1  10  11
1  2   8  10
2  3   6   9
3  4   4   8
4  5   2   7
>>> df
   A   B
0  1  10
1  2   8
2  3   6
3  4   4
4  5   2
```

Multiple columns can be assigned to using multi-line expressions:

```
>>> df.eval(
...     '''
... C = A + B
... D = A - B
... '''
... )
   A   B   C  D
0  1  10  11 -9
1  2   8  10 -6
2  3   6   9 -3
```

(continues on next page)

(continued from previous page)

```
3  4   4   8  0
4  5   2   7  3
```

AlloViz.AlloViz.Elements.Edges.ewm

Edges.**ewm**(*com: float | None = None*, *span: float | None = None*, *halflife: float | TimedeltaConvertibleTypes | None = None*, *alpha: float | None = None*, *min_periods: int | None = 0*, *adjust: bool_t = True*, *ignore_na: bool_t = False*, *axis: Axis | lib.NoDefault = _NoDefault.no_default*, *times: np.ndarray | DataFrame | Series | None = None*, *method: Literal['single', 'table'] = 'single'*) → ExponentialMovingWindow

Provide exponentially weighted (EW) calculations.

Exactly one of `com`, `span`, `halflife`, or `alpha` must be provided if `times` is not provided. If `times` is provided, `halflife` and one of `com`, `span` or `alpha` may be provided.

Parameters

com
[float, optional] Specify decay in terms of center of mass

$\alpha = 1/(1 + com)$, for $com \geq 0$.

span
[float, optional] Specify decay in terms of span

$\alpha = 2/(span + 1)$, for $span \geq 1$.

halflife
[float, str, timedelta, optional] Specify decay in terms of half-life

$\alpha = 1 - \exp(-\ln(2)/halflife)$, for $halflife > 0$.

If `times` is specified, a timedelta convertible unit over which an observation decays to half its value. Only applicable to `mean()`, and halflife value will not apply to the other functions.

alpha
[float, optional] Specify smoothing factor α directly

$0 < \alpha \leq 1$.

min_periods
[int, default 0] Minimum number of observations in window required to have a value; otherwise, result is `np.nan`.

adjust
[bool, default True] Divide by decaying adjustment factor in beginning periods to account for imbalance in relative weightings (viewing EWMA as a moving average).

- When `adjust=True` (default), the EW function is calculated using weights $w_i = (1 - \alpha)^i$. For example, the EW moving average of the series $[x_0, x_1, ..., x_t]$ would be:

$$y_t = \frac{x_t + (1 - \alpha)x_{t-1} + (1 - \alpha)^2 x_{t-2} + ... + (1 - \alpha)^t x_0}{1 + (1 - \alpha) + (1 - \alpha)^2 + ... + (1 - \alpha)^t}$$

- When `adjust=False`, the exponentially weighted function is calculated recursively:

$$y_0 = x_0$$
$$y_t = (1 - \alpha)y_{t-1} + \alpha x_t,$$

ignore_na
[bool, default False] Ignore missing values when calculating weights.

- When `ignore_na=False` (default), weights are based on absolute positions. For example, the weights of x_0 and x_2 used in calculating the final weighted average of [x_0, None, x_2] are $(1-\alpha)^2$ and 1 if `adjust=True`, and $(1-\alpha)^2$ and α if `adjust=False`.
- When `ignore_na=True`, weights are based on relative positions. For example, the weights of x_0 and x_2 used in calculating the final weighted average of [x_0, None, x_2] are $1-\alpha$ and 1 if `adjust=True`, and $1-\alpha$ and α if `adjust=False`.

axis
[{0, 1}, default 0] If `0` or `'index'`, calculate across the rows.

If `1` or `'columns'`, calculate across the columns.

For *Series* this parameter is unused and defaults to 0.

times
[np.ndarray, Series, default None] Only applicable to `mean()`.

Times corresponding to the observations. Must be monotonically increasing and `datetime64[ns]` dtype.

If 1-D array like, a sequence with the same shape as the observations.

method
[str {'single', 'table'}, default 'single'] New in version 1.4.0.

Execute the rolling operation per single column or row (`'single'`) or over the entire object (`'table'`).

This argument is only implemented when specifying `engine='numba'` in the method call.

Only applicable to `mean()`

Returns

pandas.api.typing.ExponentialMovingWindow

See also:

rolling
Provides rolling window calculations.

expanding
Provides expanding transformations.

Notes

See Windowing Operations for further usage details and examples.

Examples

```
>>> df = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]})
>>> df
     B
0  0.0
1  1.0
2  2.0
3  NaN
4  4.0
```

```
>>> df.ewm(com=0.5).mean()
          B
0  0.000000
1  0.750000
2  1.615385
3  1.615385
4  3.670213
>>> df.ewm(alpha=2 / 3).mean()
          B
0  0.000000
1  0.750000
2  1.615385
3  1.615385
4  3.670213
```

adjust

```
>>> df.ewm(com=0.5, adjust=True).mean()
          B
0  0.000000
1  0.750000
2  1.615385
3  1.615385
4  3.670213
>>> df.ewm(com=0.5, adjust=False).mean()
          B
0  0.000000
1  0.666667
2  1.555556
3  1.555556
4  3.650794
```

ignore_na

```
>>> df.ewm(com=0.5, ignore_na=True).mean()
          B
0  0.000000
```

(continues on next page)

(continued from previous page)

```
1  0.750000
2  1.615385
3  1.615385
4  3.225000
>>> df.ewm(com=0.5, ignore_na=False).mean()
          B
0  0.000000
1  0.750000
2  1.615385
3  1.615385
4  3.670213
```

times

Exponentially weighted mean with weights calculated with a timedelta `halflife` relative to `times`.

```
>>> times = ['2020-01-01', '2020-01-03', '2020-01-10', '2020-01-15', '2020-01-
↪17']
>>> df.ewm(halflife='4 days', times=pd.DatetimeIndex(times)).mean()
          B
0  0.000000
1  0.585786
2  1.523889
3  1.523889
4  3.233686
```

AlloViz.AlloViz.Elements.Edges.expanding

Edges.**expanding**(*min_periods: int = 1*, *axis: int | Literal['index', 'columns', 'rows'] | Literal[_NoDefault.no_default] = _NoDefault.no_default*, *method: Literal['single', 'table'] = 'single'*) → Expanding

Provide expanding window calculations.

Parameters

min_periods
[int, default 1] Minimum number of observations in window required to have a value; otherwise, result is `np.nan`.

axis
[int or str, default 0] If `0` or `'index'`, roll across the rows.

If `1` or `'columns'`, roll across the columns.

For *Series* this parameter is unused and defaults to 0.

method
[str {'single', 'table'}, default 'single'] Execute the rolling operation per single column or row (`'single'`) or over the entire object (`'table'`).

This argument is only implemented when specifying `engine='numba'` in the method call.

New in version 1.3.0.

Returns

pandas.api.typing.Expanding

See also:

rolling
Provides rolling window calculations.

ewm
Provides exponential weighted functions.

Notes

See Windowing Operations for further usage details and examples.

Examples

```
>>> df = pd.DataFrame({"B": [0, 1, 2, np.nan, 4]})
>>> df
     B
0  0.0
1  1.0
2  2.0
3  NaN
4  4.0
```

min_periods

Expanding sum with 1 vs 3 observations needed to calculate a value.

```
>>> df.expanding(1).sum()
     B
0  0.0
1  1.0
2  3.0
3  3.0
4  7.0
>>> df.expanding(3).sum()
     B
0  NaN
1  NaN
2  3.0
3  3.0
4  7.0
```

AlloViz.AlloViz.Elements.Edges.explode

Edges.**explode**(*column: IndexLabel*, *ignore_index: bool = False*) → DataFrame

Transform each element of a list-like to a row, replicating index values.

Parameters

column
[IndexLabel] Column(s) to explode. For multiple columns, specify a non-empty list with each element be str or tuple, and all specified columns their list-like data on same row of the frame must have matching length.

New in version 1.3.0: Multi-column explode

ignore_index
[bool, default False] If True, the resulting index will be labeled 0, 1, …, n - 1.

Returns

DataFrame
Exploded lists to rows of the subset columns; index will be duplicated for these rows.

Raises

ValueError

- If columns of the frame are not unique.
- If specified columns to explode is empty list.
- If specified columns to explode have not matching count of elements rowwise in the frame.

See also:

`DataFrame.unstack`
Pivot a level of the (necessarily hierarchical) index labels.

`DataFrame.melt`
Unpivot a DataFrame from wide format to long format.

`Series.explode`
Explode a DataFrame from list-like columns to long format.

Notes

This routine will explode list-likes including lists, tuples, sets, Series, and np.ndarray. The result dtype of the subset rows will be object. Scalars will be returned unchanged, and empty list-likes will result in a np.nan for that row. In addition, the ordering of rows in the output will be non-deterministic when exploding sets.

Reference the user guide for more examples.

Examples

```
>>> df = pd.DataFrame({'A': [[0, 1, 2], 'foo', [], [3, 4]],
...                    'B': 1,
...                    'C': [['a', 'b', 'c'], np.nan, [], ['d', 'e']]})
>>> df
           A  B          C
0  [0, 1, 2]  1  [a, b, c]
1        foo  1        NaN
2         []  1         []
3     [3, 4]  1     [d, e]
```

Single-column explode.

```
>>> df.explode('A')
     A  B          C
0    0  1  [a, b, c]
0    1  1  [a, b, c]
0    2  1  [a, b, c]
1  foo  1        NaN
2  NaN  1         []
3    3  1     [d, e]
3    4  1     [d, e]
```

Multi-column explode.

```
>>> df.explode(list('AC'))
     A  B    C
0    0  1    a
0    1  1    b
0    2  1    c
1  foo  1  NaN
2  NaN  1  NaN
3    3  1    d
3    4  1    e
```

AlloViz.AlloViz.Elements.Edges.ffill

Edges.**ffill**(*, *axis: None | Axis = None*, *inplace: bool_t = False*, *limit: None | int = None*, *downcast: dict | None | lib.NoDefault = _NoDefault.no_default*) → Self | None

Fill NA/NaN values by propagating the last valid observation to next valid.

Parameters

axis
[{0 or 'index'} for Series, {0 or 'index', 1 or 'columns'} for DataFrame] Axis along which to fill missing values. For *Series* this parameter is unused and defaults to 0.

inplace
[bool, default False] If True, fill in-place. Note: this will modify any other views on this object (e.g., a no-copy slice for a column in a DataFrame).

limit
[int, default None] If method is specified, this is the maximum number of consecutive

NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled. Must be greater than 0 if not None.

downcast
[dict, default is None] A dict of item->dtype of what to downcast if possible, or the string 'infer' which will try to downcast to an appropriate equal type (e.g. float64 to int64 if possible).

Returns

Series/DataFrame or None
Object with missing values filled or None if `inplace=True`.

Examples

```
>>> df = pd.DataFrame([[np.nan, 2, np.nan, 0],
...                    [3, 4, np.nan, 1],
...                    [np.nan, np.nan, np.nan, np.nan],
...                    [np.nan, 3, np.nan, 4]],
...                   columns=list("ABCD"))
>>> df
     A    B   C    D
0  NaN  2.0 NaN  0.0
1  3.0  4.0 NaN  1.0
2  NaN  NaN NaN  NaN
3  NaN  3.0 NaN  4.0
```

```
>>> df.ffill()
     A    B   C    D
0  NaN  2.0 NaN  0.0
1  3.0  4.0 NaN  1.0
2  3.0  4.0 NaN  1.0
3  3.0  3.0 NaN  4.0
```

```
>>> ser = pd.Series([1, np.nan, 2, 3])
>>> ser.ffill()
0   1.0
1   1.0
2   2.0
3   3.0
dtype: float64
```

AlloViz.AlloViz.Elements.Edges.fillna

Edges.**fillna**(*value: Hashable | Mapping | Series | DataFrame | None = None*, ***, *method: FillnaOptions | None = None*, *axis: Axis | None = None*, *inplace: bool_t = False*, *limit: int | None = None*, *downcast: dict | None | lib.NoDefault = _NoDefault.no_default*) → Self | None

Fill NA/NaN values using the specified method.

Parameters

value
[scalar, dict, Series, or DataFrame] Value to use to fill holes (e.g. 0), alternately a dict/Series/DataFrame of values specifying which value to use for each index (for a Series) or column (for a DataFrame). Values not in the dict/Series/DataFrame will not be filled. This value cannot be a list.

method
[{'backfill', 'bfill', 'ffill', None}, default None] Method to use for filling holes in reindexed Series:

- ffill: propagate last valid observation forward to next valid.
- backfill / bfill: use next valid observation to fill gap.

Deprecated since version 2.1.0: Use ffill or bfill instead.

axis
[{0 or 'index'} for Series, {0 or 'index', 1 or 'columns'} for DataFrame] Axis along which to fill missing values. For *Series* this parameter is unused and defaults to 0.

inplace
[bool, default False] If True, fill in-place. Note: this will modify any other views on this object (e.g., a no-copy slice for a column in a DataFrame).

limit
[int, default None] If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled. Must be greater than 0 if not None.

downcast
[dict, default is None] A dict of item->dtype of what to downcast if possible, or the string 'infer' which will try to downcast to an appropriate equal type (e.g. float64 to int64 if possible).

Returns

Series/DataFrame or None
Object with missing values filled or None if `inplace=True`.

See also:

ffill
Fill values by propagating the last valid observation to next valid.

bfill
Fill values by using the next valid observation to fill the gap.

interpolate
Fill NaN values using interpolation.

reindex
Conform object to new index.

asfreq
Convert TimeSeries to specified frequency.

Examples

```
>>> df = pd.DataFrame([[np.nan, 2, np.nan, 0],
...                    [3, 4, np.nan, 1],
...                    [np.nan, np.nan, np.nan, np.nan],
...                    [np.nan, 3, np.nan, 4]],
...                   columns=list("ABCD"))
>>> df
     A    B   C    D
0  NaN  2.0 NaN  0.0
1  3.0  4.0 NaN  1.0
2  NaN  NaN NaN  NaN
3  NaN  3.0 NaN  4.0
```

Replace all NaN elements with 0s.

```
>>> df.fillna(0)
     A    B    C    D
0  0.0  2.0  0.0  0.0
1  3.0  4.0  0.0  1.0
2  0.0  0.0  0.0  0.0
3  0.0  3.0  0.0  4.0
```

Replace all NaN elements in column 'A', 'B', 'C', and 'D', with 0, 1, 2, and 3 respectively.

```
>>> values = {"A": 0, "B": 1, "C": 2, "D": 3}
>>> df.fillna(value=values)
     A    B    C    D
0  0.0  2.0  2.0  0.0
1  3.0  4.0  2.0  1.0
2  0.0  1.0  2.0  3.0
3  0.0  3.0  2.0  4.0
```

Only replace the first NaN element.

```
>>> df.fillna(value=values, limit=1)
     A    B    C    D
0  0.0  2.0  2.0  0.0
1  3.0  4.0  NaN  1.0
2  NaN  1.0  NaN  3.0
3  NaN  3.0  NaN  4.0
```

When filling using a DataFrame, replacement happens along the same column names and same indices

```
>>> df2 = pd.DataFrame(np.zeros((4, 4)), columns=list("ABCE"))
>>> df.fillna(df2)
     A    B    C    D
```

(continues on next page)

(continued from previous page)

```
0  0.0  2.0  0.0  0.0
1  3.0  4.0  0.0  1.0
2  0.0  0.0  0.0  NaN
3  0.0  3.0  0.0  4.0
```

Note that column D is not affected since it is not present in df2.

AlloViz.AlloViz.Elements.Edges.filter

Edges.**filter**(*items=None*, *like: str | None = None*, *regex: str | None = None*, *axis: int | Literal['index', 'columns', 'rows'] | None = None*) → None

Subset the dataframe rows or columns according to the specified index labels.

Note that this routine does not filter a dataframe on its contents. The filter is applied to the labels of the index.

Parameters

items
[list-like] Keep labels from axis which are in items.

like
[str] Keep labels from axis for which "like in label == True".

regex
[str (regular expression)] Keep labels from axis for which re.search(regex, label) == True.

axis
[{0 or 'index', 1 or 'columns', None}, default None] The axis to filter on, expressed either as an index (int) or axis name (str). By default this is the info axis, 'columns' for DataFrame. For *Series* this parameter is unused and defaults to *None*.

Returns

same type as input object

See also:

`DataFrame.loc`
Access a group of rows and columns by label(s) or a boolean array.

Notes

The `items`, `like`, and `regex` parameters are enforced to be mutually exclusive.

`axis` defaults to the info axis that is used when indexing with `[]`.

Examples

```
>>> df = pd.DataFrame(np.array(([1, 2, 3], [4, 5, 6])),
...                   index=['mouse', 'rabbit'],
...                   columns=['one', 'two', 'three'])
>>> df
        one  two  three
mouse     1    2      3
rabbit    4    5      6
```

```
>>> # select columns by name
>>> df.filter(items=['one', 'three'])
         one  three
mouse     1      3
rabbit    4      6
```

```
>>> # select columns by regular expression
>>> df.filter(regex='e$', axis=1)
         one  three
mouse     1      3
rabbit    4      6
```

```
>>> # select rows containing 'bbi'
>>> df.filter(like='bbi', axis=0)
         one  two  three
rabbit    4    5      6
```

AlloViz.AlloViz.Elements.Edges.first

Edges.**first**(*offset*) → None

Select initial periods of time series data based on a date offset.

For a DataFrame with a sorted DatetimeIndex, this function can select the first few rows based on a date offset.

Parameters

offset
[str, DateOffset or dateutil.relativedelta] The offset length of the data that will be selected. For instance, '1M' will display all the rows having their index within the first month.

Returns

Series or DataFrame
A subset of the caller.

Raises

TypeError
If the index is not a `DatetimeIndex`

See also:

last
Select final periods of time series based on a date offset.

at_time
Select values at a particular time of the day.

between_time
Select values between particular times of the day.

Examples

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='2D')
>>> ts = pd.DataFrame({'A': [1, 2, 3, 4]}, index=i)
>>> ts
            A
2018-04-09  1
2018-04-11  2
2018-04-13  3
2018-04-15  4
```

Get the rows for the first 3 days:

```
>>> ts.first('3D')
            A
2018-04-09  1
2018-04-11  2
```

Notice the data for 3 first calendar days were returned, not the first 3 days observed in the dataset, and therefore data for 2018-04-13 was not returned.

AlloViz.AlloViz.Elements.Edges.first_valid_index

Edges.**first_valid_index**() → Hashable | None

Return index for first non-NA value or None, if no non-NA value is found.

Returns

type of index

Notes

If all elements are non-NA/null, returns None. Also returns None for empty Series/DataFrame.

Examples

For Series:

```
>>> s = pd.Series([None, 3, 4])
>>> s.first_valid_index()
1
>>> s.last_valid_index()
2
```

For DataFrame:

```
>>> df = pd.DataFrame({'A': [None, None, 2], 'B': [None, 3, 4]})
>>> df
     A      B
0  NaN    NaN
1  NaN    3.0
2  2.0    4.0
>>> df.first_valid_index()
1
>>> df.last_valid_index()
2
```

AlloViz.AlloViz.Elements.Edges.floordiv

`Edges.floordiv`(*other*, *axis: Axis = 'columns'*, *level=None*, *fill_value=None*)

Get Integer division of dataframe and other, element-wise (binary operator *floordiv*).

Equivalent to `dataframe // other`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *rfloordiv*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *floordiv*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other
: [scalar, sequence, Series, dict or DataFrame] Any single or multiple element data structure, or list-like object.

axis
: [{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns. (1 or 'columns'). For Series input, axis to match Series index on.

level
: [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value
: [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame
: Result of the arithmetic operation.

See also:

DataFrame.add
Add DataFrames.

DataFrame.sub
Subtract DataFrames.

DataFrame.mul
Multiply DataFrames.

DataFrame.div
Divide DataFrames (float division).

DataFrame.truediv
Divide DataFrames (float division).

DataFrame.floordiv
Divide DataFrames (integer division).

DataFrame.mod
Calculate modulo (remainder after division).

DataFrame.pow
Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a dictionary by axis.

```
>>> df.mul({'angles': 0, 'degrees': 2})
            angles  degrees
circle           0      720
triangle         0      360
rectangle        0      720
```

```
>>> df.mul({'circle': 0, 'triangle': 2, 'rectangle': 3}, axis='index')
            angles  degrees
circle           0        0
triangle         6      360
rectangle       12     1080
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
```

(continues on next page)

(continued from previous page)

```
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

AlloViz.AlloViz.Elements.Edges.from_dict

classmethod Edges.**from_dict**(*data: dict*, *orient: FromDictOrient = 'columns'*, *dtype: Dtype | None = None*, *columns: Axes | None = None*) → DataFrame

Construct DataFrame from dict of array-like or dicts.

Creates DataFrame object from dictionary by columns or by index allowing dtype specification.

Parameters

data
[dict] Of the form {field : array-like} or {field : dict}.

orient
[{'columns', 'index', 'tight'}, default 'columns'] The "orientation" of the data. If the keys of the passed dict should be the columns of the resulting DataFrame, pass 'columns' (default). Otherwise if the keys should be rows, pass 'index'. If 'tight', assume a dict with keys ['index', 'columns', 'data', 'index_names', 'column_names'].

New in version 1.4.0: 'tight' as an allowed value for the `orient` argument

dtype
[dtype, default None] Data type to force after DataFrame construction, otherwise infer.

columns
[list, default None] Column labels to use when `orient='index'`. Raises a ValueError if used with `orient='columns'` or `orient='tight'`.

Returns

DataFrame

See also:

`DataFrame.from_records`
DataFrame from structured ndarray, sequence of tuples or dicts, or DataFrame.

`DataFrame`
DataFrame object creation using constructor.

`DataFrame.to_dict`
Convert the DataFrame to a dictionary.

Examples

By default the keys of the dict become the DataFrame columns:

```
>>> data = {'col_1': [3, 2, 1, 0], 'col_2': ['a', 'b', 'c', 'd']}
>>> pd.DataFrame.from_dict(data)
   col_1 col_2
0      3     a
1      2     b
2      1     c
3      0     d
```

Specify `orient='index'` to create the DataFrame using dictionary keys as rows:

```
>>> data = {'row_1': [3, 2, 1, 0], 'row_2': ['a', 'b', 'c', 'd']}
>>> pd.DataFrame.from_dict(data, orient='index')
       0  1  2  3
row_1  3  2  1  0
row_2  a  b  c  d
```

When using the 'index' orientation, the column names can be specified manually:

```
>>> pd.DataFrame.from_dict(data, orient='index',
...                        columns=['A', 'B', 'C', 'D'])
       A  B  C  D
row_1  3  2  1  0
row_2  a  b  c  d
```

Specify `orient='tight'` to create the DataFrame using a 'tight' format:

```
>>> data = {'index': [('a', 'b'), ('a', 'c')],
...         'columns': [('x', 1), ('y', 2)],
...         'data': [[1, 3], [2, 4]],
...         'index_names': ['n1', 'n2'],
...         'column_names': ['z1', 'z2']}
>>> pd.DataFrame.from_dict(data, orient='tight')
z1     x  y
z2     1  2
n1 n2
a  b   1  3
   c   2  4
```

AlloViz.AlloViz.Elements.Edges.from_records

classmethod Edges.**from_records**(*data*, *index=None*, *exclude=None*, *columns=None*, *coerce_float: bool = False*, *nrows: None | int = None*) → DataFrame

Convert structured or record ndarray to DataFrame.

Creates a DataFrame object from a structured ndarray, sequence of tuples or dicts, or DataFrame.

Parameters

data
[structured ndarray, sequence of tuples or dicts, or DataFrame] Structured input data.

Deprecated since version 2.1.0: Passing a DataFrame is deprecated.

index
[str, list of fields, array-like] Field of array to use as the index, alternately a specific set of input labels to use.

exclude
[sequence, default None] Columns or fields to exclude.

columns
[sequence, default None] Column names to use. If the passed data do not have names associated with them, this argument provides names for the columns. Otherwise this argument indicates the order of the columns in the result (any names not found in the data will become all-NA columns).

coerce_float
[bool, default False] Attempt to convert values of non-string, non-numeric objects (like decimal.Decimal) to floating point, useful for SQL result sets.

nrows
[int, default None] Number of rows to read if data is an iterator.

Returns

DataFrame

See also:

`DataFrame.from_dict`
DataFrame from dict of array-like or dicts.

`DataFrame`
DataFrame object creation using constructor.

Examples

Data can be provided as a structured ndarray:

```
>>> data = np.array([(3, 'a'), (2, 'b'), (1, 'c'), (0, 'd')],
...                 dtype=[('col_1', 'i4'), ('col_2', 'U1')])
>>> pd.DataFrame.from_records(data)
   col_1 col_2
0      3     a
1      2     b
2      1     c
3      0     d
```

Data can be provided as a list of dicts:

```
>>> data = [{'col_1': 3, 'col_2': 'a'},
...         {'col_1': 2, 'col_2': 'b'},
...         {'col_1': 1, 'col_2': 'c'},
...         {'col_1': 0, 'col_2': 'd'}]
>>> pd.DataFrame.from_records(data)
   col_1 col_2
0      3     a
1      2     b
2      1     c
3      0     d
```

Data can be provided as a list of tuples with corresponding columns:

```
>>> data = [(3, 'a'), (2, 'b'), (1, 'c'), (0, 'd')]
>>> pd.DataFrame.from_records(data, columns=['col_1', 'col_2'])
   col_1 col_2
0      3     a
1      2     b
2      1     c
3      0     d
```

AlloViz.AlloViz.Elements.Edges.ge

Edges.**ge**(*other*, *axis: Axis = 'columns'*, *level=None*)

Get Greater than or equal to of dataframe and other, element-wise (binary operator *ge*).

Among flexible wrappers (*eq*, *ne*, *le*, *lt*, *ge*, *gt*) to comparison operators.

Equivalent to ==, *!*=, <=, <, >=, > with support to choose axis (rows or columns) and level for comparison.

Parameters

other
[scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.

axis
[{0 or 'index', 1 or 'columns'}, default 'columns'] Whether to compare by the index (0 or 'index') or columns (1 or 'columns').

level
[int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

Returns

DataFrame of bool
Result of the comparison.

See also:

DataFrame.eq
Compare DataFrames for equality elementwise.

DataFrame.ne
Compare DataFrames for inequality elementwise.

DataFrame.le
Compare DataFrames for less than inequality or equality elementwise.

DataFrame.lt
Compare DataFrames for strictly less than inequality elementwise.

DataFrame.ge
Compare DataFrames for greater than inequality or equality elementwise.

DataFrame.gt
Compare DataFrames for strictly greater than inequality elementwise.

Notes

Mismatched indices will be unioned together. *NaN* values are considered different (i.e. *NaN* != *NaN*).

Examples

```
>>> df = pd.DataFrame({'cost': [250, 150, 100],
...                    'revenue': [100, 250, 300]},
...                   index=['A', 'B', 'C'])
>>> df
   cost  revenue
A   250      100
B   150      250
C   100      300
```

Comparison with a scalar, using either the operator or method:

```
>>> df == 100
    cost  revenue
A  False     True
B  False    False
C   True    False
```

```
>>> df.eq(100)
    cost  revenue
A  False     True
B  False    False
C   True    False
```

When *other* is a `Series`, the columns of a DataFrame are aligned with the index of *other* and broadcast:

```
>>> df != pd.Series([100, 250], index=["cost", "revenue"])
    cost  revenue
A   True     True
B   True    False
C  False     True
```

Use the method to control the broadcast axis:

```
>>> df.ne(pd.Series([100, 300], index=["A", "D"]), axis='index')
   cost  revenue
A  True    False
B  True     True
C  True     True
D  True     True
```

When comparing to an arbitrary sequence, the number of columns must match the number elements in *other*:

```
>>> df == [250, 100]
    cost  revenue
A   True     True
B  False    False
C  False    False
```

Use the method to control the axis:

```
>>> df.eq([250, 250, 100], axis='index')
    cost  revenue
A   True    False
B  False     True
C   True    False
```

Compare to a DataFrame of different shape.

```
>>> other = pd.DataFrame({'revenue': [300, 250, 100, 150]},
...                      index=['A', 'B', 'C', 'D'])
>>> other
   revenue
A      300
B      250
C      100
D      150
```

```
>>> df.gt(other)
    cost  revenue
A  False    False
B  False    False
C  False     True
D  False    False
```

Compare to a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'cost': [250, 150, 100, 150, 300, 220],
...                              'revenue': [100, 250, 300, 200, 175, 225]},
...                             index=[['Q1', 'Q1', 'Q1', 'Q2', 'Q2', 'Q2'],
...                                    ['A', 'B', 'C', 'A', 'B', 'C']])
>>> df_multindex
      cost  revenue
Q1 A   250      100
   B   150      250
   C   100      300
Q2 A   150      200
   B   300      175
   C   220      225
```

```
>>> df.le(df_multindex, level=1)
       cost  revenue
Q1 A   True     True
   B   True     True
   C   True     True
Q2 A  False     True
   B   True    False
   C   True    False
```

AlloViz.AlloViz.Elements.Edges.get

Edges.**get**(*key*, *default=None*)

Get item from object for given key (ex: DataFrame column).

Returns default value if not found.

Parameters

key
[object]

Returns

same type as items contained in object

Examples

```
>>> df = pd.DataFrame(
...     [
...         [24.3, 75.7, "high"],
...         [31, 87.8, "high"],
...         [22, 71.6, "medium"],
...         [35, 95, "medium"],
...     ],
...     columns=["temp_celsius", "temp_fahrenheit", "windspeed"],
...     index=pd.date_range(start="2014-02-12", end="2014-02-15", freq="D"),
... )
```

```
>>> df
            temp_celsius  temp_fahrenheit windspeed
2014-02-12          24.3             75.7      high
2014-02-13          31.0             87.8      high
2014-02-14          22.0             71.6    medium
2014-02-15          35.0             95.0    medium
```

```
>>> df.get(["temp_celsius", "windspeed"])
            temp_celsius windspeed
2014-02-12          24.3      high
2014-02-13          31.0      high
2014-02-14          22.0    medium
2014-02-15          35.0    medium
```

```
>>> ser = df['windspeed']
>>> ser.get('2014-02-13')
'high'
```

If the key isn't found, the default value will be used.

```
>>> df.get(["temp_celsius", "temp_kelvin"], default="default_value")
'default_value'
```

```
>>> ser.get('2014-02-10', '[unknown]')
'[unknown]'
```

AlloViz.AlloViz.Elements.Edges.groupby

Edges.**groupby**(*by=None*, *axis: Axis | lib.NoDefault = _NoDefault.no_default*, *level: IndexLabel | None = None*, *as_index: bool = True*, *sort: bool = True*, *group_keys: bool = True*, *observed: bool | lib.NoDefault = _NoDefault.no_default*, *dropna: bool = True*) → DataFrameGroupBy

Group DataFrame using a mapper or by a Series of columns.

A groupby operation involves some combination of splitting the object, applying a function, and combining the results. This can be used to group large amounts of data and compute operations on these groups.

Parameters

by
[mapping, function, label, pd.Grouper or list of such] Used to determine the groups for the groupby. If `by` is a function, it's called on each value of the object's index. If a dict or Series is passed, the Series or dict VALUES will be used to determine the groups (the Series' values are first aligned; see `.align()` method). If a list or ndarray of length equal to the selected axis is passed (see the groupby user guide), the values are used as-is to determine the groups. A label or list of labels may be passed to group by the columns in `self`. Notice that a tuple is interpreted as a (single) key.

axis
[{0 or 'index', 1 or 'columns'}, default 0] Split along rows (0) or columns (1). For *Series* this parameter is unused and defaults to 0.

level
[int, level name, or sequence of such, default None] If the axis is a MultiIndex (hierarchical), group by a particular level or levels. Do not specify both `by` and `level`.

as_index
[bool, default True] Return object with group labels as the index. Only relevant for DataFrame input. as_index=False is effectively "SQL-style" grouped output. This argument has no effect on filtrations (see the filtrations in the user guide), such as `head()`, `tail()`, `nth()` and in transformations (see the transformations in the user guide).

sort
[bool, default True] Sort group keys. Get better performance by turning this off. Note this does not influence the order of observations within each group. Groupby preserves the order of rows within each group. If False, the groups will appear in the same order as they did in the original DataFrame. This argument has no effect on filtrations (see the filtrations in the user guide), such as `head()`, `tail()`, `nth()` and in transformations (see the transformations in the user guide).

Changed in version 2.0.0: Specifying `sort=False` with an ordered categorical grouper will no longer sort the values.

group_keys
[bool, default True] When calling apply and the `by` argument produces a like-indexed (i.e. a transform) result, add group keys to index to identify pieces. By default group keys are not included when the result's index (and column) labels match the inputs, and are included otherwise.

Changed in version 1.5.0: Warns that `group_keys` will no longer be ignored when the result from `apply` is a like-indexed Series or DataFrame. Specify `group_keys` explicitly to include the group keys or not.

Changed in version 2.0.0: `group_keys` now defaults to `True`.

observed
[bool, default False] This only applies if any of the groupers are Categoricals. If True: only show observed values for categorical groupers. If False: show all values for categorical groupers.

Deprecated since version 2.1.0: The default value will change to True in a future version of pandas.

dropna
[bool, default True] If True, and if group keys contain NA values, NA values together with row/column will be dropped. If False, NA values will also be treated as the key in groups.

Returns

pandas.api.typing.DataFrameGroupBy
Returns a groupby object that contains information about the groups.

See also:

resample
Convenience method for frequency conversion and resampling of time series.

Notes

See the user guide for more detailed usage and examples, including splitting an object into groups, iterating through groups, selecting a group, aggregation, and more.

Examples

```
>>> df = pd.DataFrame({'Animal': ['Falcon', 'Falcon',
...                               'Parrot', 'Parrot'],
...                    'Max Speed': [380., 370., 24., 26.]})
>>> df
   Animal  Max Speed
0  Falcon      380.0
1  Falcon      370.0
2  Parrot       24.0
3  Parrot       26.0
>>> df.groupby(['Animal']).mean()
        Max Speed
Animal
Falcon      375.0
Parrot       25.0
```

Hierarchical Indexes

We can groupby different levels of a hierarchical index using the *level* parameter:

```
>>> arrays = [['Falcon', 'Falcon', 'Parrot', 'Parrot'],
...           ['Captive', 'Wild', 'Captive', 'Wild']]
>>> index = pd.MultiIndex.from_arrays(arrays, names=('Animal', 'Type'))
>>> df = pd.DataFrame({'Max Speed': [390., 350., 30., 20.]},
...                   index=index)
>>> df
```

(continues on next page)

(continued from previous page)

```
                Max Speed
Animal Type
Falcon Captive      390.0
       Wild         350.0
Parrot Captive       30.0
       Wild          20.0
>>> df.groupby(level=0).mean()
        Max Speed
Animal
Falcon      370.0
Parrot       25.0
>>> df.groupby(level="Type").mean()
         Max Speed
Type
Captive      210.0
Wild         185.0
```

We can also choose to include NA in group keys or not by setting *dropna* parameter, the default setting is *True*.

```
>>> l = [[1, 2, 3], [1, None, 4], [2, 1, 3], [1, 2, 2]]
>>> df = pd.DataFrame(l, columns=["a", "b", "c"])
```

```
>>> df.groupby(by=["b"]).sum()
    a   c
b
1.0 2   3
2.0 2   5
```

```
>>> df.groupby(by=["b"], dropna=False).sum()
    a   c
b
1.0 2   3
2.0 2   5
NaN 1   4
```

```
>>> l = [["a", 12, 12], [None, 12.3, 33.], ["b", 12.3, 123], ["a", 1, 1]]
>>> df = pd.DataFrame(l, columns=["a", "b", "c"])
```

```
>>> df.groupby(by="a").sum()
    b     c
a
a   13.0   13.0
b   12.3  123.0
```

```
>>> df.groupby(by="a", dropna=False).sum()
    b     c
a
a   13.0   13.0
b   12.3  123.0
NaN 12.3   33.0
```

When using `.apply()`, use `group_keys` to include or exclude the group keys. The `group_keys` argument defaults to `True` (include).

```
>>> df = pd.DataFrame({'Animal': ['Falcon', 'Falcon',
...                               'Parrot', 'Parrot'],
...                    'Max Speed': [380., 370., 24., 26.]})
>>> df.groupby("Animal", group_keys=True).apply(lambda x: x)
          Animal  Max Speed
Animal
Falcon 0  Falcon      380.0
       1  Falcon      370.0
Parrot 2  Parrot       24.0
       3  Parrot       26.0
```

```
>>> df.groupby("Animal", group_keys=False).apply(lambda x: x)
   Animal  Max Speed
0  Falcon      380.0
1  Falcon      370.0
2  Parrot       24.0
3  Parrot       26.0
```

AlloViz.AlloViz.Elements.Edges.gt

Edges.**gt**(*other*, *axis: Axis = 'columns'*, *level=None*)

Get Greater than of dataframe and other, element-wise (binary operator *gt*).

Among flexible wrappers (*eq*, *ne*, *le*, *lt*, *ge*, *gt*) to comparison operators.

Equivalent to ==, *!*=, <=, <, >=, > with support to choose axis (rows or columns) and level for comparison.

Parameters

other
: [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.

axis
: [{0 or 'index', 1 or 'columns'}, default 'columns'] Whether to compare by the index (0 or 'index') or columns (1 or 'columns').

level
: [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

Returns

DataFrame of bool
: Result of the comparison.

See also:

DataFrame.eq
: Compare DataFrames for equality elementwise.

DataFrame.ne
: Compare DataFrames for inequality elementwise.

DataFrame.le
Compare DataFrames for less than inequality or equality elementwise.

DataFrame.lt
Compare DataFrames for strictly less than inequality elementwise.

DataFrame.ge
Compare DataFrames for greater than inequality or equality elementwise.

DataFrame.gt
Compare DataFrames for strictly greater than inequality elementwise.

Notes

Mismatched indices will be unioned together. *NaN* values are considered different (i.e. *NaN* != *NaN*).

Examples

```
>>> df = pd.DataFrame({'cost': [250, 150, 100],
...                    'revenue': [100, 250, 300]},
...                   index=['A', 'B', 'C'])
>>> df
   cost  revenue
A   250      100
B   150      250
C   100      300
```

Comparison with a scalar, using either the operator or method:

```
>>> df == 100
    cost  revenue
A  False     True
B  False    False
C   True    False
```

```
>>> df.eq(100)
    cost  revenue
A  False     True
B  False    False
C   True    False
```

When *other* is a `Series`, the columns of a DataFrame are aligned with the index of *other* and broadcast:

```
>>> df != pd.Series([100, 250], index=["cost", "revenue"])
    cost  revenue
A   True     True
B   True    False
C  False     True
```

Use the method to control the broadcast axis:

```
>>> df.ne(pd.Series([100, 300], index=["A", "D"]), axis='index')
   cost  revenue
```

(continues on next page)

(continued from previous page)

```
A  True    False
B  True     True
C  True     True
D  True     True
```

When comparing to an arbitrary sequence, the number of columns must match the number elements in *other*:

```
>>> df == [250, 100]
    cost  revenue
A   True     True
B  False    False
C  False    False
```

Use the method to control the axis:

```
>>> df.eq([250, 250, 100], axis='index')
    cost  revenue
A   True    False
B  False     True
C   True    False
```

Compare to a DataFrame of different shape.

```
>>> other = pd.DataFrame({'revenue': [300, 250, 100, 150]},
...                      index=['A', 'B', 'C', 'D'])
>>> other
   revenue
A      300
B      250
C      100
D      150
```

```
>>> df.gt(other)
    cost  revenue
A  False    False
B  False    False
C  False     True
D  False    False
```

Compare to a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'cost': [250, 150, 100, 150, 300, 220],
...                              'revenue': [100, 250, 300, 200, 175, 225]},
...                             index=[['Q1', 'Q1', 'Q1', 'Q2', 'Q2', 'Q2'],
...                                    ['A', 'B', 'C', 'A', 'B', 'C']])
>>> df_multindex
      cost  revenue
Q1 A   250      100
   B   150      250
   C   100      300
Q2 A   150      200
```

(continues on next page)

(continued from previous page)

```
   B   300      175
   C   220      225
```

```
>>> df.le(df_multindex, level=1)
       cost  revenue
Q1 A   True     True
   B   True     True
   C   True     True
Q2 A  False     True
   B   True    False
   C   True    False
```

AlloViz.AlloViz.Elements.Edges.head

Edges.**head**(*n: int = 5*) → None

Return the first *n* rows.

This function returns the first *n* rows for the object based on position. It is useful for quickly testing if your object has the right type of data in it.

For negative values of *n*, this function returns all rows except the last |*n*| rows, equivalent to `df[:n]`.

If n is larger than the number of rows, this function returns all rows.

Parameters

n
[int, default 5] Number of rows to select.

Returns

same type as caller
The first *n* rows of the caller object.

See also:

DataFrame.tail
Returns the last *n* rows.

Examples

```
>>> df = pd.DataFrame({'animal': ['alligator', 'bee', 'falcon', 'lion',
...                    'monkey', 'parrot', 'shark', 'whale', 'zebra']})
>>> df
      animal
0  alligator
1        bee
2     falcon
3       lion
4     monkey
5     parrot
6      shark
```

(continues on next page)

(continued from previous page)

```
7      whale
8      zebra
```

Viewing the first 5 lines

```
>>> df.head()
      animal
0  alligator
1        bee
2     falcon
3       lion
4     monkey
```

Viewing the first *n* lines (three in this case)

```
>>> df.head(3)
      animal
0  alligator
1        bee
2     falcon
```

For negative values of *n*

```
>>> df.head(-3)
      animal
0  alligator
1        bee
2     falcon
3       lion
4     monkey
5     parrot
```

AlloViz.AlloViz.Elements.Edges.hist

Edges.**hist**(*column: IndexLabel | None = None*, *by=None*, *grid: bool = True*, *xlabelsize: int | None = None*, *xrot: float | None = None*, *ylabelsize: int | None = None*, *yrot: float | None = None*, *ax=None*, *sharex: bool = False*, *sharey: bool = False*, *figsize: tuple[int, int] | None = None*, *layout: tuple[int, int] | None = None*, *bins: int | Sequence[int] = 10*, *backend: str | None = None*, *legend: bool = False*, ***kwargs*)

Make a histogram of the DataFrame's columns.

A histogram is a representation of the distribution of data. This function calls `matplotlib.pyplot.hist()`, on each series in the DataFrame, resulting in one histogram per column.

Parameters

data
[DataFrame] The pandas object holding the data.

column
[str or sequence, optional] If passed, will be used to limit data to a subset of columns.

by
[object, optional] If passed, then used to form histograms for separate groups.

grid
[bool, default True] Whether to show axis grid lines.

xlabelsize
[int, default None] If specified changes the x-axis label size.

xrot
[float, default None] Rotation of x axis labels. For example, a value of 90 displays the x labels rotated 90 degrees clockwise.

ylabelsize
[int, default None] If specified changes the y-axis label size.

yrot
[float, default None] Rotation of y axis labels. For example, a value of 90 displays the y labels rotated 90 degrees clockwise.

ax
[Matplotlib axes object, default None] The axes to plot the histogram on.

sharex
[bool, default True if ax is None else False] In case subplots=True, share x axis and set some x axis labels to invisible; defaults to True if ax is None otherwise False if an ax is passed in. Note that passing in both an ax and sharex=True will alter all x axis labels for all subplots in a figure.

sharey
[bool, default False] In case subplots=True, share y axis and set some y axis labels to invisible.

figsize
[tuple, optional] The size in inches of the figure to create. Uses the value in *matplotlib.rcParams* by default.

layout
[tuple, optional] Tuple of (rows, columns) for the layout of the histograms.

bins
[int or sequence, default 10] Number of histogram bins to be used. If an integer is given, bins + 1 bin edges are calculated and returned. If bins is a sequence, gives bin edges, including left edge of first bin and right edge of last bin. In this case, bins is returned unmodified.

backend
[str, default None] Backend to use instead of the backend specified in the option `plotting.backend`. For instance, 'matplotlib'. Alternatively, to specify the `plotting.backend` for the whole session, set `pd.options.plotting.backend`.

legend
[bool, default False] Whether to show the legend.

****kwargs**
All other plotting keyword arguments to be passed to `matplotlib.pyplot.hist()`.

Returns

matplotlib.AxesSubplot or numpy.ndarray of them

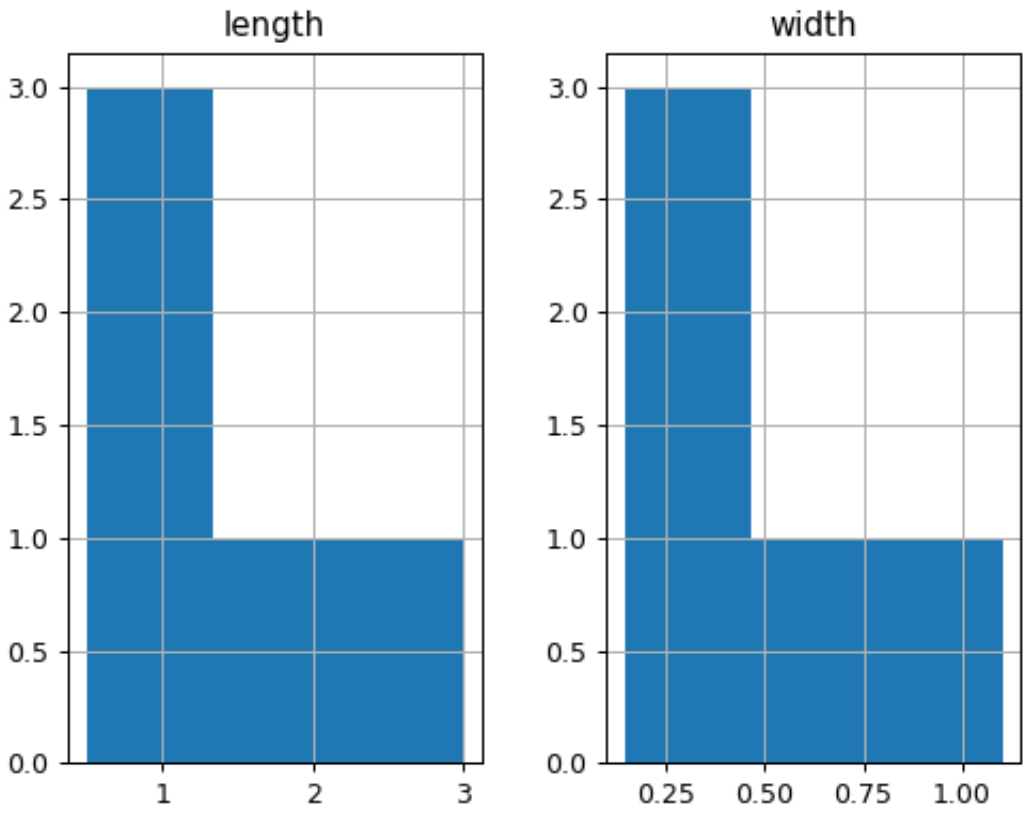
See also:

`matplotlib.pyplot.hist`
Plot a histogram using matplotlib.

Examples

This example draws a histogram based on the length and width of some animals, displayed in three bins

```
>>> df = pd.DataFrame({
...     'length': [1.5, 0.5, 1.2, 0.9, 3],
...     'width': [0.7, 0.2, 0.15, 0.2, 1.1]
...     }, index=['pig', 'rabbit', 'duck', 'chicken', 'horse'])
>>> hist = df.hist(bins=3)
```



AlloViz.AlloViz.Elements.Edges.idxmax

Edges.**idxmax**(*axis: Axis = 0*, *skipna: bool = True*, *numeric_only: bool = False*) → Series

Return index of first occurrence of maximum over requested axis.

NA/null values are excluded.

Parameters

axis

[{0 or 'index', 1 or 'columns'}, default 0] The axis to use. 0 or 'index' for row-wise, 1 or 'columns' for column-wise.

skipna
[bool, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

numeric_only
[bool, default False] Include only *float*, *int* or *boolean* data.

New in version 1.5.0.

Returns

Series
Indexes of maxima along the specified axis.

Raises

ValueError

- If the row/column is empty

See also:

`Series.idxmax`
Return index of the maximum element.

Notes

This method is the DataFrame version of `ndarray.argmax`.

Examples

Consider a dataset containing food consumption in Argentina.

```
>>> df = pd.DataFrame({'consumption': [10.51, 103.11, 55.48],
...                    'co2_emissions': [37.2, 19.66, 1712]},
...                   index=['Pork', 'Wheat Products', 'Beef'])
```

```
>>> df
                consumption  co2_emissions
Pork                  10.51         37.20
Wheat Products       103.11         19.66
Beef                  55.48       1712.00
```

By default, it returns the index for the maximum value in each column.

```
>>> df.idxmax()
consumption     Wheat Products
co2_emissions             Beef
dtype: object
```

To return the index for the maximum value in each row, use `axis="columns"`.

```
>>> df.idxmax(axis="columns")
Pork              co2_emissions
Wheat Products     consumption
Beef              co2_emissions
dtype: object
```

AlloViz.AlloViz.Elements.Edges.idxmin

Edges.**idxmin**(*axis: Axis = 0*, *skipna: bool = True*, *numeric_only: bool = False*) → Series

Return index of first occurrence of minimum over requested axis.

NA/null values are excluded.

Parameters

axis
[{0 or 'index', 1 or 'columns'}, default 0] The axis to use. 0 or 'index' for row-wise, 1 or 'columns' for column-wise.

skipna
[bool, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

numeric_only
[bool, default False] Include only *float*, *int* or *boolean* data.

New in version 1.5.0.

Returns

Series
Indexes of minima along the specified axis.

Raises

ValueError

- If the row/column is empty

See also:

Series.idxmin
Return index of the minimum element.

Notes

This method is the DataFrame version of `ndarray.argmin`.

Examples

Consider a dataset containing food consumption in Argentina.

```
>>> df = pd.DataFrame({'consumption': [10.51, 103.11, 55.48],
...                    'co2_emissions': [37.2, 19.66, 1712]},
...                   index=['Pork', 'Wheat Products', 'Beef'])
```

```
>>> df
                consumption  co2_emissions
Pork                  10.51         37.20
Wheat Products       103.11         19.66
Beef                  55.48       1712.00
```

By default, it returns the index for the minimum value in each column.

```
>>> df.idxmin()
consumption               Pork
co2_emissions    Wheat Products
dtype: object
```

To return the index for the minimum value in each row, use `axis="columns"`.

```
>>> df.idxmin(axis="columns")
Pork               consumption
Wheat Products    co2_emissions
Beef               consumption
dtype: object
```

AlloViz.AlloViz.Elements.Edges.infer_objects

Edges.**infer_objects**(*copy: bool | None = None*) → None

Attempt to infer better dtypes for object columns.

Attempts soft conversion of object-dtyped columns, leaving non-object and unconvertible columns unchanged. The inference rules are the same as during normal Series/DataFrame construction.

Parameters

copy
[bool, default True] Whether to make a copy for non-object or non-inferable columns or Series.

Returns

same type as input object

See also:

to_datetime
Convert argument to datetime.

to_timedelta
Convert argument to timedelta.

to_numeric
Convert argument to numeric type.

convert_dtypes
Convert argument to best possible dtype.

Examples

```
>>> df = pd.DataFrame({"A": ["a", 1, 2, 3]})
>>> df = df.iloc[1:]
>>> df
   A
1  1
2  2
3  3
```

```
>>> df.dtypes
A    object
dtype: object
```

```
>>> df.infer_objects().dtypes
A    int64
dtype: object
```

AlloViz.AlloViz.Elements.Edges.info

Edges.**info**(*verbose: bool | None = None*, *buf: WriteBuffer[str] | None = None*, *max_cols: int | None = None*, *memory_usage: bool | str | None = None*, *show_counts: bool | None = None*) → None

Print a concise summary of a DataFrame.

This method prints information about a DataFrame including the index dtype and columns, non-null values and memory usage.

Parameters

verbose
[bool, optional] Whether to print the full summary. By default, the setting in `pandas.options.display.max_info_columns` is followed.

buf
[writable buffer, defaults to sys.stdout] Where to send the output. By default, the output is printed to sys.stdout. Pass a writable buffer if you need to further process the output.

max_cols
[int, optional] When to switch from the verbose to the truncated output. If the DataFrame has more than *max_cols* columns, the truncated output is used. By default, the setting in `pandas.options.display.max_info_columns` is used.

memory_usage
[bool, str, optional] Specifies whether total memory usage of the DataFrame elements (including the index) should be displayed. By default, this follows the `pandas.options.display.memory_usage` setting.

True always show memory usage. False never shows memory usage. A value of 'deep' is equivalent to "True with deep introspection". Memory usage is shown in human-readable units (base-2 representation). Without deep introspection a memory estimation is made based in column dtype and number of rows assuming values consume the same memory amount for corresponding dtypes. With deep memory introspection, a real memory usage calculation is performed at the cost of computational resources. See the Frequently Asked Questions for more details.

show_counts
[bool, optional] Whether to show the non-null counts. By default, this is shown only if the DataFrame is smaller than `pandas.options.display.max_info_rows` and `pandas.options.display.max_info_columns`. A value of True always shows the counts, and False never shows the counts.

Returns

None
This method prints a summary of a DataFrame and returns None.

See also:

DataFrame.describe
Generate descriptive statistics of DataFrame columns.

DataFrame.memory_usage
Memory usage of DataFrame columns.

Examples

```
>>> int_values = [1, 2, 3, 4, 5]
>>> text_values = ['alpha', 'beta', 'gamma', 'delta', 'epsilon']
>>> float_values = [0.0, 0.25, 0.5, 0.75, 1.0]
>>> df = pd.DataFrame({"int_col": int_values, "text_col": text_values,
...                    "float_col": float_values})
>>> df
    int_col text_col  float_col
0         1    alpha       0.00
1         2     beta       0.25
2         3    gamma       0.50
3         4    delta       0.75
4         5  epsilon       1.00
```

Prints information of all columns:

```
>>> df.info(verbose=True)
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 3 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   int_col    5 non-null      int64
 1   text_col   5 non-null      object
 2   float_col  5 non-null      float64
dtypes: float64(1), int64(1), object(1)
memory usage: 248.0+ bytes
```

Prints a summary of columns count and its dtypes but not per column information:

```
>>> df.info(verbose=False)
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Columns: 3 entries, int_col to float_col
dtypes: float64(1), int64(1), object(1)
memory usage: 248.0+ bytes
```

Pipe output of DataFrame.info to buffer instead of sys.stdout, get buffer content and writes to a text file:

```
>>> import io
>>> buffer = io.StringIO()
>>> df.info(buf=buffer)
>>> s = buffer.getvalue()
>>> with open("df_info.txt", "w",
...           encoding="utf-8") as f:
...     f.write(s)
260
```

The *memory_usage* parameter allows deep introspection mode, specially useful for big DataFrames and fine-tune memory optimization:

```
>>> random_strings_array = np.random.choice(['a', 'b', 'c'], 10 ** 6)
>>> df = pd.DataFrame({
...     'column_1': np.random.choice(['a', 'b', 'c'], 10 ** 6),
...     'column_2': np.random.choice(['a', 'b', 'c'], 10 ** 6),
...     'column_3': np.random.choice(['a', 'b', 'c'], 10 ** 6)
... })
>>> df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000000 entries, 0 to 999999
Data columns (total 3 columns):
 #   Column    Non-Null Count    Dtype
---  ------    --------------    -----
 0   column_1  1000000 non-null  object
 1   column_2  1000000 non-null  object
 2   column_3  1000000 non-null  object
dtypes: object(3)
memory usage: 22.9+ MB
```

```
>>> df.info(memory_usage='deep')
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000000 entries, 0 to 999999
Data columns (total 3 columns):
 #   Column    Non-Null Count    Dtype
---  ------    --------------    -----
 0   column_1  1000000 non-null  object
 1   column_2  1000000 non-null  object
 2   column_3  1000000 non-null  object
dtypes: object(3)
memory usage: 165.9 MB
```

AlloViz.AlloViz.Elements.Edges.insert

Edges.**insert**(*loc: int*, *column: Hashable*, *value: Scalar | AnyArrayLike*, *allow_duplicates: bool | lib.NoDefault = _NoDefault.no_default*) → None

Insert column into DataFrame at specified location.

Raises a ValueError if *column* is already contained in the DataFrame, unless *allow_duplicates* is set to True.

Parameters

loc
[int] Insertion index. Must verify 0 <= loc <= len(columns).

column
[str, number, or hashable object] Label of the inserted column.

value
[Scalar, Series, or array-like]

allow_duplicates
[bool, optional, default lib.no_default]

See also:

Index.insert
Insert new item by index.

Examples

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df
   col1  col2
0     1     3
1     2     4
>>> df.insert(1, "newcol", [99, 99])
>>> df
   col1  newcol  col2
0     1      99     3
1     2      99     4
>>> df.insert(0, "col1", [100, 100], allow_duplicates=True)
>>> df
   col1  col1  newcol  col2
0   100     1      99     3
1   100     2      99     4
```

Notice that pandas uses index alignment in case of *value* from type *Series*:

```
>>> df.insert(0, "col0", pd.Series([5, 6], index=[1, 2]))
>>> df
   col0  col1  col1  newcol  col2
0   NaN   100     1      99     3
1   5.0   100     2      99     4
```

AlloViz.AlloViz.Elements.Edges.interpolate

Edges.**interpolate**(*method: InterpolateOptions = 'linear'*, ***, *axis: Axis = 0*, *limit: int | None = None*, *inplace: bool_t = False*, *limit_direction: Literal['forward', 'backward', 'both'] | None = None*, *limit_area: Literal['inside', 'outside'] | None = None*, *downcast: Literal['infer'] | None | lib.NoDefault = _NoDefault.no_default*, ***kwargs*) → Self | None

Fill NaN values using an interpolation method.

Please note that only `method='linear'` is supported for DataFrame/Series with a MultiIndex.

Parameters

method
[str, default 'linear'] Interpolation technique to use. One of:

- 'linear': Ignore the index and treat the values as equally spaced. This is the only method supported on MultiIndexes.
- 'time': Works on daily and higher resolution data to interpolate given length of interval.
- 'index', 'values': use the actual numerical values of the index.
- 'pad': Fill in NaNs using existing values.

- 'nearest', 'zero', 'slinear', 'quadratic', 'cubic', 'barycentric', 'polynomial': Passed to *scipy.interpolate.interp1d*, whereas 'spline' is passed to *scipy.interpolate.UnivariateSpline*. These methods use the numerical values of the index. Both 'polynomial' and 'spline' require that you also specify an *order* (int), e.g. `df.interpolate(method='polynomial', order=5)`. Note that, *slinear* method in Pandas refers to the Scipy first order *spline* instead of Pandas first order *spline*.
- 'krogh', 'piecewise_polynomial', 'spline', 'pchip', 'akima', 'cubicspline': Wrappers around the SciPy interpolation methods of similar names. See *Notes*.
- 'from_derivatives': Refers to *scipy.interpolate.BPoly.from_derivatives*.

axis
[{{0 or 'index', 1 or 'columns', None}}, default None] Axis to interpolate along. For *Series* this parameter is unused and defaults to 0.

limit
[int, optional] Maximum number of consecutive NaNs to fill. Must be greater than 0.

inplace
[bool, default False] Update the data in place if possible.

limit_direction
[{{'forward', 'backward', 'both'}}, Optional] Consecutive NaNs will be filled in this direction.

If limit is specified:

- If 'method' is 'pad' or 'ffill', 'limit_direction' must be 'forward'.
- If 'method' is 'backfill' or 'bfill', 'limit_direction' must be 'backwards'.

If 'limit' is not specified:

- If 'method' is 'backfill' or 'bfill', the default is 'backward'
- else the default is 'forward'

raises ValueError if *limit_direction* is 'forward' or 'both' and
method is 'backfill' or 'bfill'.

raises ValueError if *limit_direction* is 'backward' or 'both' and
method is 'pad' or 'ffill'.

limit_area
[{{*None*, 'inside', 'outside'}}, default None] If limit is specified, consecutive NaNs will be filled with this restriction.

- `None`: No fill restriction.
- 'inside': Only fill NaNs surrounded by valid values (interpolate).
- 'outside': Only fill NaNs outside valid values (extrapolate).

downcast
[optional, 'infer' or None, defaults to None] Downcast dtypes if possible.

Deprecated since version 2.1.0.

``**kwargs``
[optional] Keyword arguments to pass on to the interpolating function.

Returns

Series or DataFrame or None
Returns the same object type as the caller, interpolated at some or all `NaN` values or None if `inplace=True`.

See also:

`fillna`
Fill missing values using different methods.

`scipy.interpolate.Akima1DInterpolator`
Piecewise cubic polynomials (Akima interpolator).

`scipy.interpolate.BPoly.from_derivatives`
Piecewise polynomial in the Bernstein basis.

`scipy.interpolate.interp1d`
Interpolate a 1-D function.

`scipy.interpolate.KroghInterpolator`
Interpolate polynomial (Krogh interpolator).

`scipy.interpolate.PchipInterpolator`
PCHIP 1-d monotonic cubic interpolation.

`scipy.interpolate.CubicSpline`
Cubic spline data interpolator.

Notes

The 'krogh', 'piecewise_polynomial', 'spline', 'pchip' and 'akima' methods are wrappers around the respective SciPy implementations of similar names. These use the actual numerical values of the index. For more information on their behavior, see the SciPy documentation.

Examples

Filling in `NaN` in a `Series` via linear interpolation.

```
>>> s = pd.Series([0, 1, np.nan, 3])
>>> s
0    0.0
1    1.0
2    NaN
3    3.0
dtype: float64
>>> s.interpolate()
0    0.0
1    1.0
2    2.0
3    3.0
dtype: float64
```

Filling in `NaN` in a Series via polynomial interpolation or splines: Both 'polynomial' and 'spline' methods require that you also specify an `order` (int).

```
>>> s = pd.Series([0, 2, np.nan, 8])
>>> s.interpolate(method='polynomial', order=2)
0    0.000000
1    2.000000
2    4.666667
3    8.000000
dtype: float64
```

Fill the DataFrame forward (that is, going down) along each column using linear interpolation.

Note how the last entry in column 'a' is interpolated differently, because there is no entry after it to use for interpolation. Note how the first entry in column 'b' remains `NaN`, because there is no entry before it to use for interpolation.

```
>>> df = pd.DataFrame([(0.0, np.nan, -1.0, 1.0),
...                    (np.nan, 2.0, np.nan, np.nan),
...                    (2.0, 3.0, np.nan, 9.0),
...                    (np.nan, 4.0, -4.0, 16.0)],
...                   columns=list('abcd'))
>>> df
     a    b    c     d
0  0.0  NaN -1.0   1.0
1  NaN  2.0  NaN   NaN
2  2.0  3.0  NaN   9.0
3  NaN  4.0 -4.0  16.0
>>> df.interpolate(method='linear', limit_direction='forward', axis=0)
     a    b    c     d
0  0.0  NaN -1.0   1.0
1  1.0  2.0 -2.0   5.0
2  2.0  3.0 -3.0   9.0
3  2.0  4.0 -4.0  16.0
```

Using polynomial interpolation.

```
>>> df['d'].interpolate(method='polynomial', order=2)
0     1.0
1     4.0
2     9.0
3    16.0
Name: d, dtype: float64
```

AlloViz.AlloViz.Elements.Edges.isetitem

`Edges.`**`isetitem`**(*loc*, *value*) → None

Set the given value in the column with position *loc*.

This is a positional analogue to `__setitem__`.

Parameters

loc
[int or sequence of ints] Index position for the column.

value
[scalar or arraylike] Value(s) for the column.

Notes

`frame.isetitem(loc, value)` is an in-place method as it will modify the DataFrame in place (not returning a new object). In contrast to `frame.iloc[:, i] = value` which will try to update the existing values in place, `frame.isetitem(loc, value)` will not update the values of the column itself in place, it will instead insert a new array.

In cases where `frame.columns` is unique, this is equivalent to `frame[frame.columns[i]] = value`.

AlloViz.AlloViz.Elements.Edges.isin

`Edges.isin`(*values: Series | DataFrame | Sequence | Mapping*) → DataFrame

Whether each element in the DataFrame is contained in values.

Parameters

values
[iterable, Series, DataFrame or dict] The result will only be true at a location if all the labels match. If *values* is a Series, that's the index. If *values* is a dict, the keys must be the column names, which must match. If *values* is a DataFrame, then both the index and column labels must match.

Returns

DataFrame
DataFrame of booleans showing whether each element in the DataFrame is contained in values.

See also:

`DataFrame.eq`
Equality test for DataFrame.

`Series.isin`
Equivalent method on Series.

`Series.str.contains`
Test if pattern or regex is contained within a string of a Series or Index.

Examples

```
>>> df = pd.DataFrame({'num_legs': [2, 4], 'num_wings': [2, 0]},
...                   index=['falcon', 'dog'])
>>> df
        num_legs  num_wings
falcon         2          2
dog            4          0
```

When `values` is a list check whether every value in the DataFrame is present in the list (which animals have 0 or 2 legs or wings)

```
>>> df.isin([0, 2])
        num_legs  num_wings
falcon      True       True
dog        False       True
```

To check if `values` is *not* in the DataFrame, use the ~ operator:

```
>>> ~df.isin([0, 2])
        num_legs  num_wings
falcon     False      False
dog         True      False
```

When `values` is a dict, we can pass values to check for each column separately:

```
>>> df.isin({'num_wings': [0, 3]})
        num_legs  num_wings
falcon     False      False
dog        False       True
```

When `values` is a Series or DataFrame the index and column must match. Note that 'falcon' does not match based on the number of legs in other.

```
>>> other = pd.DataFrame({'num_legs': [8, 3], 'num_wings': [0, 2]},
...                      index=['spider', 'falcon'])
>>> df.isin(other)
        num_legs  num_wings
falcon     False       True
dog        False      False
```

AlloViz.AlloViz.Elements.Edges.isna

`Edges.isna()` → DataFrame

Detect missing values.

Return a boolean same-sized object indicating if the values are NA. NA values, such as None or `numpy.NaN`, gets mapped to True values. Everything else gets mapped to False values. Characters such as empty strings `''` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`).

Returns

DataFrame
Mask of bool values for each element in DataFrame that indicates whether an element is an NA value.

See also:

`DataFrame.isnull`
Alias of isna.

`DataFrame.notna`
Boolean inverse of isna.

`DataFrame.dropna`
Omit axes labels with missing values.

`isna`
Top-level isna.

Examples

Show which entries in a DataFrame are NA.

```
>>> df = pd.DataFrame(dict(age=[5, 6, np.nan],
...                        born=[pd.NaT, pd.Timestamp('1939-05-27'),
...                              pd.Timestamp('1940-04-25')],
...                        name=['Alfred', 'Batman', ''],
...                        toy=[None, 'Batmobile', 'Joker']))
>>> df
   age       born    name        toy
0  5.0        NaT  Alfred       None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25              Joker
```

```
>>> df.isna()
     age   born   name    toy
0  False   True  False   True
1  False  False  False  False
2   True  False  False  False
```

Show which entries in a Series are NA.

```
>>> ser = pd.Series([5, 6, np.nan])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.isna()
0    False
1    False
2     True
dtype: bool
```

AlloViz.AlloViz.Elements.Edges.isnull

Edges.**isnull**() → DataFrame

DataFrame.isnull is an alias for DataFrame.isna.

Detect missing values.

Return a boolean same-sized object indicating if the values are NA. NA values, such as None or `numpy.NaN`, gets mapped to True values. Everything else gets mapped to False values. Characters such as empty strings `''` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`).

Returns

DataFrame
Mask of bool values for each element in DataFrame that indicates whether an element is an NA value.

See also:

DataFrame.isnull
Alias of isna.

DataFrame.notna
Boolean inverse of isna.

DataFrame.dropna
Omit axes labels with missing values.

isna
Top-level isna.

Examples

Show which entries in a DataFrame are NA.

```
>>> df = pd.DataFrame(dict(age=[5, 6, np.nan],
...                        born=[pd.NaT, pd.Timestamp('1939-05-27'),
...                              pd.Timestamp('1940-04-25')],
...                        name=['Alfred', 'Batman', ''],
...                        toy=[None, 'Batmobile', 'Joker']))
>>> df
   age       born    name        toy
0  5.0        NaT  Alfred       None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25              Joker
```

```
>>> df.isna()
     age   born   name    toy
0  False   True  False   True
1  False  False  False  False
2   True  False  False  False
```

Show which entries in a Series are NA.

```
>>> ser = pd.Series([5, 6, np.nan])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.isna()
0    False
1    False
2     True
dtype: bool
```

AlloViz.AlloViz.Elements.Edges.items

Edges.**items**() → Iterable[tuple[Hashable, Series]]

Iterate over (column name, Series) pairs.

Iterates over the DataFrame columns, returning a tuple with the column name and the content as a Series.

Yields

label
[object] The column names for the DataFrame being iterated over.

content
[Series] The column entries belonging to each label, as a Series.

See also:

DataFrame.iterrows
Iterate over DataFrame rows as (index, Series) pairs.

DataFrame.itertuples
Iterate over DataFrame rows as namedtuples of the values.

Examples

```
>>> df = pd.DataFrame({'species': ['bear', 'bear', 'marsupial'],
...                    'population': [1864, 22000, 80000]},
...                    index=['panda', 'polar', 'koala'])
>>> df
        species   population
panda   bear      1864
polar   bear      22000
koala   marsupial 80000
>>> for label, content in df.items():
...     print(f'label: {label}')
...     print(f'content: {content}', sep='\n')
...
label: species
content:
panda         bear
polar         bear
koala    marsupial
Name: species, dtype: object
label: population
content:
panda     1864
polar    22000
koala    80000
Name: population, dtype: int64
```

AlloViz.AlloViz.Elements.Edges.iterrows

Edges.**iterrows**() → Iterable[tuple[Hashable, Series]]

Iterate over DataFrame rows as (index, Series) pairs.

Yields

index
[label or tuple of label] The index of the row. A tuple for a *MultiIndex*.

data
[Series] The data of the row as a Series.

See also:

DataFrame.itertuples
Iterate over DataFrame rows as namedtuples of the values.

DataFrame.items
Iterate over (column name, Series) pairs.

Notes

1. Because `iterrows` returns a Series for each row, it does **not** preserve dtypes across the rows (dtypes are preserved across columns for DataFrames).

   To preserve dtypes while iterating over the rows, it is better to use *itertuples()* which returns namedtuples of the values and which is generally faster than `iterrows`.

2. You should **never modify** something you are iterating over. This is not guaranteed to work in all cases. Depending on the data types, the iterator returns a copy and not a view, and writing to it will have no effect.

Examples

```
>>> df = pd.DataFrame([[1, 1.5]], columns=['int', 'float'])
>>> row = next(df.iterrows())[1]
>>> row
int      1.0
float    1.5
Name: 0, dtype: float64
>>> print(row['int'].dtype)
float64
>>> print(df['int'].dtype)
int64
```

AlloViz.AlloViz.Elements.Edges.itertuples

Edges.**itertuples**(*index: bool = True*, *name: str | None = 'Pandas'*) → Iterable[tuple[Any, ...]]

Iterate over DataFrame rows as namedtuples.

Parameters

index
[bool, default True] If True, return the index as the first element of the tuple.

name
[str or None, default "Pandas"] The name of the returned namedtuples or None to return regular tuples.

Returns

iterator
An object to iterate over namedtuples for each row in the DataFrame with the first field possibly being the index and following fields being the column values.

See also:

DataFrame.iterrows
Iterate over DataFrame rows as (index, Series) pairs.

DataFrame.items
Iterate over (column name, Series) pairs.

Notes

The column names will be renamed to positional names if they are invalid Python identifiers, repeated, or start with an underscore.

Examples

```
>>> df = pd.DataFrame({'num_legs': [4, 2], 'num_wings': [0, 2]},
...                   index=['dog', 'hawk'])
>>> df
      num_legs  num_wings
dog          4          0
hawk         2          2
>>> for row in df.itertuples():
...     print(row)
...
Pandas(Index='dog', num_legs=4, num_wings=0)
Pandas(Index='hawk', num_legs=2, num_wings=2)
```

By setting the *index* parameter to False we can remove the index as the first element of the tuple:

```
>>> for row in df.itertuples(index=False):
...     print(row)
...
Pandas(num_legs=4, num_wings=0)
Pandas(num_legs=2, num_wings=2)
```

With the *name* parameter set we set a custom name for the yielded namedtuples:

```
>>> for row in df.itertuples(name='Animal'):
...     print(row)
...
Animal(Index='dog', num_legs=4, num_wings=0)
Animal(Index='hawk', num_legs=2, num_wings=2)
```

AlloViz.AlloViz.Elements.Edges.join

Edges.**join**(*other: DataFrame | Series | Iterable[DataFrame | Series]*, *on: IndexLabel | None = None*, *how: MergeHow = 'left'*, *lsuffix: str = ''*, *rsuffix: str = ''*, *sort: bool = False*, *validate: JoinValidate | None = None*) → DataFrame

Join columns of another DataFrame.

Join columns with *other* DataFrame either on index or on a key column. Efficiently join multiple DataFrame objects by index at once by passing a list.

Parameters

other
[DataFrame, Series, or a list containing any combination of them] Index should be similar to one of the columns in this one. If a Series is passed, its name attribute must be set, and that will be used as the column name in the resulting joined DataFrame.

on
[str, list of str, or array-like, optional] Column or index level name(s) in the caller to join on the index in *other*, otherwise joins index-on-index. If multiple values given, the *other* DataFrame must have a MultiIndex. Can pass an array as the join key if it is not already contained in the calling DataFrame. Like an Excel VLOOKUP operation.

how
[{'left', 'right', 'outer', 'inner', 'cross'}, default 'left'] How to handle the operation of the two objects.

- left: use calling frame's index (or column if on is specified)
- right: use *other*'s index.
- outer: form union of calling frame's index (or column if on is specified) with *other*'s index, and sort it lexicographically.
- inner: form intersection of calling frame's index (or column if on is specified) with *other*'s index, preserving the order of the calling's one.
- cross: creates the cartesian product from both frames, preserves the order of the left keys.

  New in version 1.2.0.

lsuffix
[str, default ''] Suffix to use from left frame's overlapping columns.

rsuffix
[str, default ''] Suffix to use from right frame's overlapping columns.

sort
[bool, default False] Order result DataFrame lexicographically by the join key. If False, the order of the join key depends on the join type (how keyword).

validate
[str, optional] If specified, checks if join is of specified type.

- "one_to_one" or "1:1": check if join keys are unique in both left and right datasets.
- "one_to_many" or "1:m": check if join keys are unique in left dataset.
- "many_to_one" or "m:1": check if join keys are unique in right dataset.
- "many_to_many" or "m:m": allowed, but does not result in checks.

New in version 1.5.0.

Returns

DataFrame
A dataframe containing columns from both the caller and *other*.

See also:

`DataFrame.merge`
For column(s)-on-column(s) operations.

Notes

Parameters *on*, *lsuffix*, and *rsuffix* are not supported when passing a list of *DataFrame* objects.

Examples

```
>>> df = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3', 'K4', 'K5'],
...                    'A': ['A0', 'A1', 'A2', 'A3', 'A4', 'A5']})
```

```
>>> df
  key   A
0  K0  A0
1  K1  A1
2  K2  A2
3  K3  A3
4  K4  A4
5  K5  A5
```

```
>>> other = pd.DataFrame({'key': ['K0', 'K1', 'K2'],
...                       'B': ['B0', 'B1', 'B2']})
```

```
>>> other
  key   B
0  K0  B0
1  K1  B1
2  K2  B2
```

Join DataFrames using their indexes.

```
>>> df.join(other, lsuffix='_caller', rsuffix='_other')
  key_caller   A key_other    B
0         K0  A0        K0   B0
```

(continues on next page)

(continued from previous page)

```
1         K1  A1        K1   B1
2         K2  A2        K2   B2
3         K3  A3       NaN  NaN
4         K4  A4       NaN  NaN
5         K5  A5       NaN  NaN
```

If we want to join using the key columns, we need to set key to be the index in both *df* and *other*. The joined DataFrame will have key as its index.

```
>>> df.set_index('key').join(other.set_index('key'))
      A    B
key
K0   A0   B0
K1   A1   B1
K2   A2   B2
K3   A3  NaN
K4   A4  NaN
K5   A5  NaN
```

Another option to join using the key columns is to use the *on* parameter. DataFrame.join always uses *other*'s index but we can use any column in *df*. This method preserves the original DataFrame's index in the result.

```
>>> df.join(other.set_index('key'), on='key')
  key   A    B
0  K0  A0   B0
1  K1  A1   B1
2  K2  A2   B2
3  K3  A3  NaN
4  K4  A4  NaN
5  K5  A5  NaN
```

Using non-unique key values shows how they are matched.

```
>>> df = pd.DataFrame({'key': ['K0', 'K1', 'K1', 'K3', 'K0', 'K1'],
...                    'A': ['A0', 'A1', 'A2', 'A3', 'A4', 'A5']})
```

```
>>> df
  key   A
0  K0  A0
1  K1  A1
2  K1  A2
3  K3  A3
4  K0  A4
5  K1  A5
```

```
>>> df.join(other.set_index('key'), on='key', validate='m:1')
  key   A    B
0  K0  A0   B0
1  K1  A1   B1
2  K1  A2   B1
3  K3  A3  NaN
```

(continues on next page)

(continued from previous page)

```
4  K0  A4   B0
5  K1  A5   B1
```

AlloViz.AlloViz.Elements.Edges.keys

Edges.**keys**() → Index

Get the 'info axis' (see Indexing for more).

This is index for Series, columns for DataFrame.

Returns

Index
Info axis.

Examples

```
>>> d = pd.DataFrame(data={'A': [1, 2, 3], 'B': [0, 4, 8]},
...                  index=['a', 'b', 'c'])
>>> d
   A  B
a  1  0
b  2  4
c  3  8
>>> d.keys()
Index(['A', 'B'], dtype='object')
```

AlloViz.AlloViz.Elements.Edges.kurt

Edges.**kurt**(*axis: Axis | None = 0*, *skipna: bool = True*, *numeric_only: bool = False*, ***kwargs*)

Return unbiased kurtosis over requested axis.

Kurtosis obtained using Fisher's definition of kurtosis (kurtosis of normal == 0.0). Normalized by N-1.

Parameters

axis
[{index (0), columns (1)}] Axis for the function to be applied on. For *Series* this parameter is unused and defaults to 0.

For DataFrames, specifying `axis=None` will apply the aggregation across both axes.

New in version 2.0.0.

skipna
[bool, default True] Exclude NA/null values when computing the result.

numeric_only
[bool, default False] Include only float, int, boolean columns. Not implemented for Series.

****kwargs**
Additional keyword arguments to be passed to the function.

Returns

Series or scalar

Examples

```
>>> s = pd.Series([1, 2, 2, 3], index=['cat', 'dog', 'dog', 'mouse'])
>>> s
cat    1
dog    2
dog    2
mouse  3
dtype: int64
>>> s.kurt()
1.5
```

With a DataFrame

```
>>> df = pd.DataFrame({'a': [1, 2, 2, 3], 'b': [3, 4, 4, 4]},
...                   index=['cat', 'dog', 'dog', 'mouse'])
>>> df
       a   b
  cat  1   3
  dog  2   4
  dog  2   4
mouse  3   4
>>> df.kurt()
a   1.5
b   4.0
dtype: float64
```

With axis=None

```
>>> df.kurt(axis=None).round(6)
-0.988693
```

Using axis=1

```
>>> df = pd.DataFrame({'a': [1, 2], 'b': [3, 4], 'c': [3, 4], 'd': [1, 2]},
...                   index=['cat', 'dog'])
>>> df.kurt(axis=1)
cat   -6.0
dog   -6.0
dtype: float64
```

AlloViz.AlloViz.Elements.Edges.kurtosis

Edges.**kurtosis**(*axis: Axis | None = 0*, *skipna: bool = True*, *numeric_only: bool = False*, ***kwargs*)

Return unbiased kurtosis over requested axis.

Kurtosis obtained using Fisher's definition of kurtosis (kurtosis of normal == 0.0). Normalized by N-1.

Parameters

axis
[{index (0), columns (1)}] Axis for the function to be applied on. For *Series* this parameter is unused and defaults to 0.

For DataFrames, specifying `axis=None` will apply the aggregation across both axes.

New in version 2.0.0.

skipna
[bool, default True] Exclude NA/null values when computing the result.

numeric_only
[bool, default False] Include only float, int, boolean columns. Not implemented for Series.

****kwargs**
Additional keyword arguments to be passed to the function.

Returns

Series or scalar

Examples

```
>>> s = pd.Series([1, 2, 2, 3], index=['cat', 'dog', 'dog', 'mouse'])
>>> s
cat    1
dog    2
dog    2
mouse  3
dtype: int64
>>> s.kurt()
1.5
```

With a DataFrame

```
>>> df = pd.DataFrame({'a': [1, 2, 2, 3], 'b': [3, 4, 4, 4]},
...                   index=['cat', 'dog', 'dog', 'mouse'])
>>> df
       a   b
  cat  1   3
  dog  2   4
  dog  2   4
mouse  3   4
>>> df.kurt()
a   1.5
b   4.0
dtype: float64
```

With axis=None

```
>>> df.kurt(axis=None).round(6)
-0.988693
```

Using axis=1

```
>>> df = pd.DataFrame({'a': [1, 2], 'b': [3, 4], 'c': [3, 4], 'd': [1, 2]},
...                   index=['cat', 'dog'])
>>> df.kurt(axis=1)
cat   -6.0
dog   -6.0
dtype: float64
```

AlloViz.AlloViz.Elements.Edges.last

`Edges.last`(*offset*) → None

Select final periods of time series data based on a date offset.

For a DataFrame with a sorted DatetimeIndex, this function selects the last few rows based on a date offset.

Parameters

offset
[str, DateOffset, dateutil.relativedelta] The offset length of the data that will be selected. For instance, '3D' will display all the rows having their index within the last 3 days.

Returns

Series or DataFrame
A subset of the caller.

Raises

TypeError
If the index is not a `DatetimeIndex`

See also:

first
Select initial periods of time series based on a date offset.

at_time
Select values at a particular time of the day.

between_time
Select values between particular times of the day.

Notes

Deprecated since version 2.1.0: Please create a mask and filter using *.loc* instead

Examples

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='2D')
>>> ts = pd.DataFrame({'A': [1, 2, 3, 4]}, index=i)
>>> ts
            A
2018-04-09  1
2018-04-11  2
2018-04-13  3
2018-04-15  4
```

Get the rows for the last 3 days:

```
>>> ts.last('3D')
            A
2018-04-13  3
2018-04-15  4
```

Notice the data for 3 last calendar days were returned, not the last 3 observed days in the dataset, and therefore data for 2018-04-11 was not returned.

AlloViz.AlloViz.Elements.Edges.last_valid_index

Edges.**last_valid_index**() → Hashable | None

Return index for last non-NA value or None, if no non-NA value is found.

Returns

type of index

Notes

If all elements are non-NA/null, returns None. Also returns None for empty Series/DataFrame.

Examples

For Series:

```
>>> s = pd.Series([None, 3, 4])
>>> s.first_valid_index()
1
>>> s.last_valid_index()
2
```

For DataFrame:

```
>>> df = pd.DataFrame({'A': [None, None, 2], 'B': [None, 3, 4]})
>>> df
     A      B
0  NaN    NaN
1  NaN    3.0
2  2.0    4.0
>>> df.first_valid_index()
1
>>> df.last_valid_index()
2
```

AlloViz.AlloViz.Elements.Edges.le

Edges.**le**(*other*, *axis: Axis = 'columns'*, *level=None*)

Get Less than or equal to of dataframe and other, element-wise (binary operator *le*).

Among flexible wrappers (*eq*, *ne*, *le*, *lt*, *ge*, *gt*) to comparison operators.

Equivalent to ==, *!*=, <=, <, >=, > with support to choose axis (rows or columns) and level for comparison.

Parameters

other
[scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.

axis
[{0 or 'index', 1 or 'columns'}, default 'columns'] Whether to compare by the index (0 or 'index') or columns (1 or 'columns').

level
[int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

Returns

DataFrame of bool
Result of the comparison.

See also:

DataFrame.eq
Compare DataFrames for equality elementwise.

DataFrame.ne
Compare DataFrames for inequality elementwise.

DataFrame.le
Compare DataFrames for less than inequality or equality elementwise.

DataFrame.lt
Compare DataFrames for strictly less than inequality elementwise.

DataFrame.ge
Compare DataFrames for greater than inequality or equality elementwise.

DataFrame.gt
Compare DataFrames for strictly greater than inequality elementwise.

Notes

Mismatched indices will be unioned together. *NaN* values are considered different (i.e. *NaN* != *NaN*).

Examples

```
>>> df = pd.DataFrame({'cost': [250, 150, 100],
...                    'revenue': [100, 250, 300]},
...                   index=['A', 'B', 'C'])
>>> df
   cost  revenue
A   250      100
B   150      250
C   100      300
```

Comparison with a scalar, using either the operator or method:

```
>>> df == 100
    cost  revenue
A  False     True
B  False    False
C   True    False
```

```
>>> df.eq(100)
    cost  revenue
A  False     True
B  False    False
C   True    False
```

When *other* is a `Series`, the columns of a DataFrame are aligned with the index of *other* and broadcast:

```
>>> df != pd.Series([100, 250], index=["cost", "revenue"])
    cost  revenue
A   True     True
B   True    False
C  False     True
```

Use the method to control the broadcast axis:

```
>>> df.ne(pd.Series([100, 300], index=["A", "D"]), axis='index')
   cost  revenue
A  True    False
B  True     True
C  True     True
D  True     True
```

When comparing to an arbitrary sequence, the number of columns must match the number elements in *other*:

```
>>> df == [250, 100]
    cost  revenue
A   True     True
```

(continues on next page)

(continued from previous page)

```
B  False    False
C  False    False
```

Use the method to control the axis:

```
>>> df.eq([250, 250, 100], axis='index')
    cost  revenue
A   True    False
B  False     True
C   True    False
```

Compare to a DataFrame of different shape.

```
>>> other = pd.DataFrame({'revenue': [300, 250, 100, 150]},
...                      index=['A', 'B', 'C', 'D'])
>>> other
   revenue
A      300
B      250
C      100
D      150
```

```
>>> df.gt(other)
    cost  revenue
A  False    False
B  False    False
C  False     True
D  False    False
```

Compare to a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'cost': [250, 150, 100, 150, 300, 220],
...                              'revenue': [100, 250, 300, 200, 175, 225]},
...                             index=[['Q1', 'Q1', 'Q1', 'Q2', 'Q2', 'Q2'],
...                                    ['A', 'B', 'C', 'A', 'B', 'C']])
>>> df_multindex
      cost  revenue
Q1 A   250      100
   B   150      250
   C   100      300
Q2 A   150      200
   B   300      175
   C   220      225
```

```
>>> df.le(df_multindex, level=1)
       cost  revenue
Q1 A   True     True
   B   True     True
   C   True     True
Q2 A  False     True
   B   True    False
   C   True    False
```

AlloViz.AlloViz.Elements.Edges.lt

Edges.**lt**(*other*, *axis: Axis = 'columns'*, *level=None*)

Get Less than of dataframe and other, element-wise (binary operator *lt*).

Among flexible wrappers (*eq*, *ne*, *le*, *lt*, *ge*, *gt*) to comparison operators.

Equivalent to ==, *!*=, <=, <, >=, > with support to choose axis (rows or columns) and level for comparison.

Parameters

other
[scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.

axis
[{0 or 'index', 1 or 'columns'}, default 'columns'] Whether to compare by the index (0 or 'index') or columns (1 or 'columns').

level
[int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

Returns

DataFrame of bool
Result of the comparison.

See also:

DataFrame.eq
Compare DataFrames for equality elementwise.

DataFrame.ne
Compare DataFrames for inequality elementwise.

DataFrame.le
Compare DataFrames for less than inequality or equality elementwise.

DataFrame.lt
Compare DataFrames for strictly less than inequality elementwise.

DataFrame.ge
Compare DataFrames for greater than inequality or equality elementwise.

DataFrame.gt
Compare DataFrames for strictly greater than inequality elementwise.

Notes

Mismatched indices will be unioned together. *NaN* values are considered different (i.e. *NaN* != *NaN*).

Examples

```
>>> df = pd.DataFrame({'cost': [250, 150, 100],
...                    'revenue': [100, 250, 300]},
...                   index=['A', 'B', 'C'])
>>> df
   cost  revenue
A   250      100
B   150      250
C   100      300
```

Comparison with a scalar, using either the operator or method:

```
>>> df == 100
    cost  revenue
A  False     True
B  False    False
C   True    False
```

```
>>> df.eq(100)
    cost  revenue
A  False     True
B  False    False
C   True    False
```

When *other* is a `Series`, the columns of a DataFrame are aligned with the index of *other* and broadcast:

```
>>> df != pd.Series([100, 250], index=["cost", "revenue"])
    cost  revenue
A   True     True
B   True    False
C  False     True
```

Use the method to control the broadcast axis:

```
>>> df.ne(pd.Series([100, 300], index=["A", "D"]), axis='index')
   cost  revenue
A  True    False
B  True     True
C  True     True
D  True     True
```

When comparing to an arbitrary sequence, the number of columns must match the number elements in *other*:

```
>>> df == [250, 100]
    cost  revenue
A   True     True
B  False    False
C  False    False
```

Use the method to control the axis:

```
>>> df.eq([250, 250, 100], axis='index')
    cost  revenue
A   True    False
B  False     True
C   True    False
```

Compare to a DataFrame of different shape.

```
>>> other = pd.DataFrame({'revenue': [300, 250, 100, 150]},
...                      index=['A', 'B', 'C', 'D'])
>>> other
   revenue
A      300
B      250
C      100
D      150
```

```
>>> df.gt(other)
    cost  revenue
A  False    False
B  False    False
C  False     True
D  False    False
```

Compare to a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'cost': [250, 150, 100, 150, 300, 220],
...                              'revenue': [100, 250, 300, 200, 175, 225]},
...                             index=[['Q1', 'Q1', 'Q1', 'Q2', 'Q2', 'Q2'],
...                                    ['A', 'B', 'C', 'A', 'B', 'C']])
>>> df_multindex
      cost  revenue
Q1 A   250      100
   B   150      250
   C   100      300
Q2 A   150      200
   B   300      175
   C   220      225
```

```
>>> df.le(df_multindex, level=1)
       cost  revenue
Q1 A   True     True
   B   True     True
   C   True     True
Q2 A  False     True
   B   True    False
   C   True    False
```

AlloViz.AlloViz.Elements.Edges.map

Edges.**map**(*func: PythonFuncType*, *na_action: str | None = None*, ***kwargs*) → DataFrame

Apply a function to a Dataframe elementwise.

New in version 2.1.0: DataFrame.applymap was deprecated and renamed to DataFrame.map.

This method applies a function that accepts and returns a scalar to every element of a DataFrame.

Parameters

func
[callable] Python function, returns a single value from a single value.

na_action
[{None, 'ignore'}, default None] If 'ignore', propagate NaN values, without passing them to func.

****kwargs**
Additional keyword arguments to pass as keywords arguments to *func*.

Returns

DataFrame
Transformed DataFrame.

See also:

`DataFrame.apply`
Apply a function along input axis of DataFrame.

`DataFrame.replace`
Replace values given in *to_replace* with *value*.

`Series.map`
Apply a function elementwise on a Series.

Examples

```
>>> df = pd.DataFrame([[1, 2.12], [3.356, 4.567]])
>>> df
       0      1
0  1.000  2.120
1  3.356  4.567
```

```
>>> df.map(lambda x: len(str(x)))
   0  1
0  3  4
1  5  5
```

Like Series.map, NA values can be ignored:

```
>>> df_copy = df.copy()
>>> df_copy.iloc[0, 0] = pd.NA
>>> df_copy.map(lambda x: len(str(x)), na_action='ignore')
     0  1
```

(continues on next page)

(continued from previous page)

```
0  NaN  4
1  5.0  5
```

Note that a vectorized version of *func* often exists, which will be much faster. You could square each number elementwise.

```
>>> df.map(lambda x: x**2)
           0          1
0   1.000000   4.494400
1  11.262736  20.857489
```

But it's better to avoid map in that case.

```
>>> df ** 2
           0          1
0   1.000000   4.494400
1  11.262736  20.857489
```

AlloViz.AlloViz.Elements.Edges.mask

Edges.**mask**(*cond*, *other=_NoDefault.no_default*, *, *inplace: bool_t = False*, *axis: Axis | None = None*, *level: Level | None = None*) → Self | None

Replace values where the condition is True.

Parameters

cond
: [bool Series/DataFrame, array-like, or callable] Where *cond* is False, keep the original value. Where True, replace with corresponding value from *other*. If *cond* is callable, it is computed on the Series/DataFrame and should return boolean Series/DataFrame or array. The callable must not change input Series/DataFrame (though pandas doesn't check it).

other
: [scalar, Series/DataFrame, or callable] Entries where *cond* is True are replaced with corresponding value from *other*. If other is callable, it is computed on the Series/DataFrame and should return scalar or Series/DataFrame. The callable must not change input Series/DataFrame (though pandas doesn't check it). If not specified, entries will be filled with the corresponding NULL value (`np.nan` for numpy dtypes, `pd.NA` for extension dtypes).

inplace
: [bool, default False] Whether to perform the operation in place on the data.

axis
: [int, default None] Alignment axis if needed. For *Series* this parameter is unused and defaults to 0.

level
: [int, default None] Alignment level if needed.

Returns

Same type as caller or None if `inplace=True`.

See also:

DataFrame.where()
Return an object of same shape as self.

Notes

The mask method is an application of the if-then idiom. For each element in the calling DataFrame, if `cond` is `False` the element is used; otherwise the corresponding element from the DataFrame `other` is used. If the axis of `other` does not align with axis of `cond` Series/DataFrame, the misaligned index positions will be filled with True.

The signature for `DataFrame.where()` differs from `numpy.where()`. Roughly `df1.where(m, df2)` is equivalent to `np.where(m, df1, df2)`.

For further details and examples see the `mask` documentation in indexing.

The dtype of the object takes precedence. The fill value is casted to the object's dtype, if this can be done losslessly.

Examples

```
>>> s = pd.Series(range(5))
>>> s.where(s > 0)
0    NaN
1    1.0
2    2.0
3    3.0
4    4.0
dtype: float64
>>> s.mask(s > 0)
0    0.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

```
>>> s = pd.Series(range(5))
>>> t = pd.Series([True, False])
>>> s.where(t, 99)
0     0
1    99
2    99
3    99
4    99
dtype: int64
>>> s.mask(t, 99)
0    99
1     1
2    99
3    99
4    99
dtype: int64
```

```
>>> s.where(s > 1, 10)
0    10
1    10
2    2
3    3
4    4
dtype: int64
>>> s.mask(s > 1, 10)
0     0
1     1
2    10
3    10
4    10
dtype: int64
```

```
>>> df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])
>>> df
   A  B
0  0  1
1  2  3
2  4  5
3  6  7
4  8  9
>>> m = df % 3 == 0
>>> df.where(m, -df)
   A  B
0  0 -1
1 -2  3
2 -4 -5
3  6 -7
4 -8  9
>>> df.where(m, -df) == np.where(m, df, -df)
      A     B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True
>>> df.where(m, -df) == df.mask(~m, -df)
      A     B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True
```

AlloViz.AlloViz.Elements.Edges.max

Edges.**max**(*axis: Axis | None = 0*, *skipna: bool = True*, *numeric_only: bool = False*, ***kwargs*)

Return the maximum of the values over the requested axis.

If you want the *index* of the maximum, use `idxmax`. This is the equivalent of the `numpy.ndarray` method `argmax`.

Parameters

axis
[{index (0), columns (1)}] Axis for the function to be applied on. For *Series* this parameter is unused and defaults to 0.

For DataFrames, specifying `axis=None` will apply the aggregation across both axes.

New in version 2.0.0.

skipna
[bool, default True] Exclude NA/null values when computing the result.

numeric_only
[bool, default False] Include only float, int, boolean columns. Not implemented for Series.

****kwargs**
Additional keyword arguments to be passed to the function.

Returns

Series or scalar

See also:

`Series.sum`
Return the sum.

`Series.min`
Return the minimum.

`Series.max`
Return the maximum.

`Series.idxmin`
Return the index of the minimum.

`Series.idxmax`
Return the index of the maximum.

`DataFrame.sum`
Return the sum over the requested axis.

`DataFrame.min`
Return the minimum over the requested axis.

`DataFrame.max`
Return the maximum over the requested axis.

`DataFrame.idxmin`
Return the index of the minimum over the requested axis.

`DataFrame.idxmax`
Return the index of the maximum over the requested axis.

Examples

```
>>> idx = pd.MultiIndex.from_arrays([
...     ['warm', 'warm', 'cold', 'cold'],
...     ['dog', 'falcon', 'fish', 'spider']],
...     names=['blooded', 'animal'])
>>> s = pd.Series([4, 2, 0, 8], name='legs', index=idx)
>>> s
blooded  animal
warm     dog       4
         falcon    2
cold     fish      0
         spider    8
Name: legs, dtype: int64
```

```
>>> s.max()
8
```

AlloViz.AlloViz.Elements.Edges.mean

Edges.**mean**(*axis: Axis | None = 0*, *skipna: bool = True*, *numeric_only: bool = False*, ***kwargs*)

Return the mean of the values over the requested axis.

Parameters

axis
[{index (0), columns (1)}] Axis for the function to be applied on. For *Series* this parameter is unused and defaults to 0.

For DataFrames, specifying `axis=None` will apply the aggregation across both axes.

New in version 2.0.0.

skipna
[bool, default True] Exclude NA/null values when computing the result.

numeric_only
[bool, default False] Include only float, int, boolean columns. Not implemented for Series.

****kwargs**
Additional keyword arguments to be passed to the function.

Returns

Series or scalar

Examples

```
>>> s = pd.Series([1, 2, 3])
>>> s.mean()
2.0
```

With a DataFrame

```
>>> df = pd.DataFrame({'a': [1, 2], 'b': [2, 3]}, index=['tiger', 'zebra'])
>>> df
       a   b
tiger  1   2
zebra  2   3
>>> df.mean()
a   1.5
b   2.5
dtype: float64
```

Using axis=1

```
>>> df.mean(axis=1)
tiger   1.5
zebra   2.5
dtype: float64
```

In this case, *numeric_only* should be set to *True* to avoid getting an error.

```
>>> df = pd.DataFrame({'a': [1, 2], 'b': ['T', 'Z']},
...                   index=['tiger', 'zebra'])
>>> df.mean(numeric_only=True)
a   1.5
dtype: float64
```

AlloViz.AlloViz.Elements.Edges.median

Edges.**median**(*axis: Axis | None = 0*, *skipna: bool = True*, *numeric_only: bool = False*, ***kwargs*)

Return the median of the values over the requested axis.

Parameters

axis
[{index (0), columns (1)}] Axis for the function to be applied on. For *Series* this parameter is unused and defaults to 0.

For DataFrames, specifying `axis=None` will apply the aggregation across both axes.

New in version 2.0.0.

skipna
[bool, default True] Exclude NA/null values when computing the result.

numeric_only
[bool, default False] Include only float, int, boolean columns. Not implemented for Series.

****kwargs**
Additional keyword arguments to be passed to the function.

Returns

Series or scalar

Examples

```
>>> s = pd.Series([1, 2, 3])
>>> s.median()
2.0
```

With a DataFrame

```
>>> df = pd.DataFrame({'a': [1, 2], 'b': [2, 3]}, index=['tiger', 'zebra'])
>>> df
       a   b
tiger  1   2
zebra  2   3
>>> df.median()
a   1.5
b   2.5
dtype: float64
```

Using axis=1

```
>>> df.median(axis=1)
tiger   1.5
zebra   2.5
dtype: float64
```

In this case, *numeric_only* should be set to *True* to avoid getting an error.

```
>>> df = pd.DataFrame({'a': [1, 2], 'b': ['T', 'Z']},
...                   index=['tiger', 'zebra'])
>>> df.median(numeric_only=True)
a   1.5
dtype: float64
```

AlloViz.AlloViz.Elements.Edges.melt

Edges.**melt**(*id_vars=None*, *value_vars=None*, *var_name=None*, *value_name: Hashable = 'value'*, *col_level: Level | None = None*, *ignore_index: bool = True*) → DataFrame

Unpivot a DataFrame from wide to long format, optionally leaving identifiers set.

This function is useful to massage a DataFrame into a format where one or more columns are identifier variables (*id_vars*), while all other columns, considered measured variables (*value_vars*), are “unpivoted” to the row axis, leaving just two non-identifier columns, ‘variable’ and ‘value’.

Parameters

id_vars
[tuple, list, or ndarray, optional] Column(s) to use as identifier variables.

value_vars
[tuple, list, or ndarray, optional] Column(s) to unpivot. If not specified, uses all columns that are not set as *id_vars*.

var_name
[scalar] Name to use for the 'variable' column. If None it uses `frame.columns.name` or 'variable'.

value_name
[scalar, default 'value'] Name to use for the 'value' column.

col_level
[int or str, optional] If columns are a MultiIndex then use this level to melt.

ignore_index
[bool, default True] If True, original index is ignored. If False, the original index is retained. Index labels will be repeated as necessary.

Returns

DataFrame
Unpivoted DataFrame.

See also:

melt
Identical method.

pivot_table
Create a spreadsheet-style pivot table as a DataFrame.

DataFrame.pivot
Return reshaped DataFrame organized by given index / column values.

DataFrame.explode
Explode a DataFrame from list-like columns to long format.

Notes

Reference the user guide for more examples.

Examples

```
>>> df = pd.DataFrame({'A': {0: 'a', 1: 'b', 2: 'c'},
...                    'B': {0: 1, 1: 3, 2: 5},
...                    'C': {0: 2, 1: 4, 2: 6}})
>>> df
   A  B  C
0  a  1  2
1  b  3  4
2  c  5  6
```

```
>>> df.melt(id_vars=['A'], value_vars=['B'])
   A variable  value
0  a        B      1
1  b        B      3
2  c        B      5
```

```
>>> df.melt(id_vars=['A'], value_vars=['B', 'C'])
   A variable  value
0  a        B      1
1  b        B      3
2  c        B      5
3  a        C      2
4  b        C      4
5  c        C      6
```

The names of 'variable' and 'value' columns can be customized:

```
>>> df.melt(id_vars=['A'], value_vars=['B'],
...         var_name='myVarname', value_name='myValname')
   A myVarname  myValname
0  a         B          1
1  b         B          3
2  c         B          5
```

Original index values can be kept around:

```
>>> df.melt(id_vars=['A'], value_vars=['B', 'C'], ignore_index=False)
   A variable  value
0  a        B      1
1  b        B      3
2  c        B      5
0  a        C      2
1  b        C      4
2  c        C      6
```

If you have multi-index columns:

```
>>> df.columns = [list('ABC'), list('DEF')]
>>> df
   A  B  C
   D  E  F
0  a  1  2
1  b  3  4
2  c  5  6
```

```
>>> df.melt(col_level=0, id_vars=['A'], value_vars=['B'])
   A variable  value
0  a        B      1
1  b        B      3
2  c        B      5
```

```
>>> df.melt(id_vars=[('A', 'D')], value_vars=[('B', 'E')])
  (A, D) variable_0 variable_1  value
0      a          B          E      1
1      b          B          E      3
2      c          B          E      5
```

AlloViz.AlloViz.Elements.Edges.memory_usage

Edges.**memory_usage**(*index: bool = True*, *deep: bool = False*) → Series

Return the memory usage of each column in bytes.

The memory usage can optionally include the contribution of the index and elements of *object* dtype.

This value is displayed in *DataFrame.info* by default. This can be suppressed by setting `pandas.options.display.memory_usage` to False.

Parameters

index
[bool, default True] Specifies whether to include the memory usage of the DataFrame's index in returned Series. If `index=True`, the memory usage of the index is the first item in the output.

deep
[bool, default False] If True, introspect the data deeply by interrogating *object* dtypes for system-level memory consumption, and include it in the returned values.

Returns

Series
A Series whose index is the original column names and whose values is the memory usage of each column in bytes.

See also:

numpy.ndarray.nbytes
Total bytes consumed by the elements of an ndarray.

Series.memory_usage
Bytes consumed by a Series.

Categorical
Memory-efficient array for string values with many repeated values.

DataFrame.info
Concise summary of a DataFrame.

Notes

See the Frequently Asked Questions for more details.

Examples

```
>>> dtypes = ['int64', 'float64', 'complex128', 'object', 'bool']
>>> data = dict([(t, np.ones(shape=5000, dtype=int).astype(t))
...              for t in dtypes])
>>> df = pd.DataFrame(data)
>>> df.head()
   int64  float64            complex128  object  bool
0      1      1.0              1.0+0.0j       1  True
1      1      1.0              1.0+0.0j       1  True
2      1      1.0              1.0+0.0j       1  True
```

(continues on next page)

(continued from previous page)

```
3      1      1.0              1.0+0.0j       1  True
4      1      1.0              1.0+0.0j       1  True
```

```
>>> df.memory_usage()
Index           128
int64         40000
float64       40000
complex128    80000
object        40000
bool           5000
dtype: int64
```

```
>>> df.memory_usage(index=False)
int64         40000
float64       40000
complex128    80000
object        40000
bool           5000
dtype: int64
```

The memory footprint of *object* dtype columns is ignored by default:

```
>>> df.memory_usage(deep=True)
Index           128
int64         40000
float64       40000
complex128    80000
object        180000
bool           5000
dtype: int64
```

Use a Categorical for efficient storage of an object-dtype column with many repeated values.

```
>>> df['object'].astype('category').memory_usage(deep=True)
5244
```

AlloViz.AlloViz.Elements.Edges.merge

Edges.**merge**(*right: DataFrame | Series*, *how: MergeHow = 'inner'*, *on: IndexLabel | None = None*, *left_on: IndexLabel | None = None*, *right_on: IndexLabel | None = None*, *left_index: bool = False*, *right_index: bool = False*, *sort: bool = False*, *suffixes: Suffixes = ('_x', '_y')*, *copy: bool | None = None*, *indicator: str | bool = False*, *validate: MergeValidate | None = None*) → DataFrame

Merge DataFrame or named Series objects with a database-style join.

A named Series object is treated as a DataFrame with a single named column.

The join is done on columns or indexes. If joining columns on columns, the DataFrame indexes *will be ignored*. Otherwise if joining indexes on indexes or indexes on a column or columns, the index will be passed on. When performing a cross merge, no column specifications to merge on are allowed.

Warning: If both key columns contain rows where the key is a null value, those rows will be matched against each other. This is different from usual SQL join behaviour and can lead to unexpected results.

Parameters

right
[DataFrame or named Series] Object to merge with.

how
[{'left', 'right', 'outer', 'inner', 'cross'}, default 'inner'] Type of merge to be performed.

- left: use only keys from left frame, similar to a SQL left outer join; preserve key order.
- right: use only keys from right frame, similar to a SQL right outer join; preserve key order.
- outer: use union of keys from both frames, similar to a SQL full outer join; sort keys lexicographically.
- inner: use intersection of keys from both frames, similar to a SQL inner join; preserve the order of the left keys.
- cross: creates the cartesian product from both frames, preserves the order of the left keys.

  New in version 1.2.0.

on
[label or list] Column or index level names to join on. These must be found in both DataFrames. If *on* is None and not merging on indexes then this defaults to the intersection of the columns in both DataFrames.

left_on
[label or list, or array-like] Column or index level names to join on in the left DataFrame. Can also be an array or list of arrays of the length of the left DataFrame. These arrays are treated as if they are columns.

right_on
[label or list, or array-like] Column or index level names to join on in the right DataFrame. Can also be an array or list of arrays of the length of the right DataFrame. These arrays are treated as if they are columns.

left_index
[bool, default False] Use the index from the left DataFrame as the join key(s). If it is a MultiIndex, the number of keys in the other DataFrame (either the index or a number of columns) must match the number of levels.

right_index
[bool, default False] Use the index from the right DataFrame as the join key. Same caveats as left_index.

sort
[bool, default False] Sort the join keys lexicographically in the result DataFrame. If False, the order of the join keys depends on the join type (how keyword).

suffixes
[list-like, default is ("_x", "_y")] A length-2 sequence where each element is optionally a string indicating the suffix to add to overlapping column names in *left* and *right*

respectively. Pass a value of *None* instead of a string to indicate that the column name from *left* or *right* should be left as-is, with no suffix. At least one of the values must not be None.

copy
[bool, default True] If False, avoid copy if possible.

indicator
[bool or str, default False] If True, adds a column to the output DataFrame called "_merge" with information on the source of each row. The column can be given a different name by providing a string argument. The column will have a Categorical type with the value of "left_only" for observations whose merge key only appears in the left DataFrame, "right_only" for observations whose merge key only appears in the right DataFrame, and "both" if the observation's merge key is found in both DataFrames.

validate
[str, optional] If specified, checks if merge is of specified type.

- "one_to_one" or "1:1": check if merge keys are unique in both left and right datasets.
- "one_to_many" or "1:m": check if merge keys are unique in left dataset.
- "many_to_one" or "m:1": check if merge keys are unique in right dataset.
- "many_to_many" or "m:m": allowed, but does not result in checks.

Returns

DataFrame
A DataFrame of the two merged objects.

See also:

`merge_ordered`
Merge with optional filling/interpolation.

`merge_asof`
Merge on nearest keys.

`DataFrame.join`
Similar method using indices.

Examples

```
>>> df1 = pd.DataFrame({'lkey': ['foo', 'bar', 'baz', 'foo'],
...                     'value': [1, 2, 3, 5]})
>>> df2 = pd.DataFrame({'rkey': ['foo', 'bar', 'baz', 'foo'],
...                     'value': [5, 6, 7, 8]})
>>> df1
    lkey value
0   foo      1
1   bar      2
2   baz      3
3   foo      5
>>> df2
    rkey value
```

(continues on next page)

(continued from previous page)

```
0   foo      5
1   bar      6
2   baz      7
3   foo      8
```

Merge df1 and df2 on the lkey and rkey columns. The value columns have the default suffixes, _x and _y, appended.

```
>>> df1.merge(df2, left_on='lkey', right_on='rkey')
  lkey  value_x rkey  value_y
0  foo        1  foo        5
1  foo        1  foo        8
2  foo        5  foo        5
3  foo        5  foo        8
4  bar        2  bar        6
5  baz        3  baz        7
```

Merge DataFrames df1 and df2 with specified left and right suffixes appended to any overlapping columns.

```
>>> df1.merge(df2, left_on='lkey', right_on='rkey',
...           suffixes=('_left', '_right'))
  lkey  value_left rkey  value_right
0  foo           1  foo            5
1  foo           1  foo            8
2  foo           5  foo            5
3  foo           5  foo            8
4  bar           2  bar            6
5  baz           3  baz            7
```

Merge DataFrames df1 and df2, but raise an exception if the DataFrames have any overlapping columns.

```
>>> df1.merge(df2, left_on='lkey', right_on='rkey', suffixes=(False, False))
Traceback (most recent call last):
...
ValueError: columns overlap but no suffix specified:
    Index(['value'], dtype='object')
```

```
>>> df1 = pd.DataFrame({'a': ['foo', 'bar'], 'b': [1, 2]})
>>> df2 = pd.DataFrame({'a': ['foo', 'baz'], 'c': [3, 4]})
>>> df1
      a  b
0   foo  1
1   bar  2
>>> df2
      a  c
0   foo  3
1   baz  4
```

```
>>> df1.merge(df2, how='inner', on='a')
      a  b  c
0   foo  1  3
```

```
>>> df1.merge(df2, how='left', on='a')
     a  b  c
0   foo  1  3.0
1   bar  2  NaN
```

```
>>> df1 = pd.DataFrame({'left': ['foo', 'bar']})
>>> df2 = pd.DataFrame({'right': [7, 8]})
>>> df1
    left
0   foo
1   bar
>>> df2
    right
0   7
1   8
```

```
>>> df1.merge(df2, how='cross')
   left  right
0   foo      7
1   foo      8
2   bar      7
3   bar      8
```

AlloViz.AlloViz.Elements.Edges.min

Edges.**min**(*axis: Axis | None = 0*, *skipna: bool = True*, *numeric_only: bool = False*, ***kwargs*)

Return the minimum of the values over the requested axis.

If you want the *index* of the minimum, use `idxmin`. This is the equivalent of the `numpy.ndarray` method `argmin`.

Parameters

axis
[{index (0), columns (1)}] Axis for the function to be applied on. For *Series* this parameter is unused and defaults to 0.

For DataFrames, specifying `axis=None` will apply the aggregation across both axes.

New in version 2.0.0.

skipna
[bool, default True] Exclude NA/null values when computing the result.

numeric_only
[bool, default False] Include only float, int, boolean columns. Not implemented for Series.

****kwargs**
Additional keyword arguments to be passed to the function.

Returns

Series or scalar

See also:

Series.sum
Return the sum.

Series.min
Return the minimum.

Series.max
Return the maximum.

Series.idxmin
Return the index of the minimum.

Series.idxmax
Return the index of the maximum.

DataFrame.sum
Return the sum over the requested axis.

DataFrame.min
Return the minimum over the requested axis.

DataFrame.max
Return the maximum over the requested axis.

DataFrame.idxmin
Return the index of the minimum over the requested axis.

DataFrame.idxmax
Return the index of the maximum over the requested axis.

Examples

```
>>> idx = pd.MultiIndex.from_arrays([
...     ['warm', 'warm', 'cold', 'cold'],
...     ['dog', 'falcon', 'fish', 'spider']],
...     names=['blooded', 'animal'])
>>> s = pd.Series([4, 2, 0, 8], name='legs', index=idx)
>>> s
blooded  animal
warm     dog       4
         falcon    2
cold     fish      0
         spider    8
Name: legs, dtype: int64
```

```
>>> s.min()
0
```

AlloViz.AlloViz.Elements.Edges.mod

Edges.**mod**(*other*, *axis: Axis = 'columns'*, *level=None*, *fill_value=None*)

Get Modulo of dataframe and other, element-wise (binary operator *mod*).

Equivalent to `dataframe % other`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *rmod*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *floordiv*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other
[scalar, sequence, Series, dict or DataFrame] Any single or multiple element data structure, or list-like object.

axis
[{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns. (1 or 'columns'). For Series input, axis to match Series index on.

level
[int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value
[float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame
Result of the arithmetic operation.

See also:

`DataFrame.add`
Add DataFrames.

`DataFrame.sub`
Subtract DataFrames.

`DataFrame.mul`
Multiply DataFrames.

`DataFrame.div`
Divide DataFrames (float division).

`DataFrame.truediv`
Divide DataFrames (float division).

`DataFrame.floordiv`
Divide DataFrames (integer division).

`DataFrame.mod`
Calculate modulo (remainder after division).

`DataFrame.pow`
Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...         axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a dictionary by axis.

```
>>> df.mul({'angles': 0, 'degrees': 2})
            angles  degrees
circle           0      720
triangle         0      360
rectangle        0      720
```

```
>>> df.mul({'circle': 0, 'triangle': 2, 'rectangle': 3}, axis='index')
            angles  degrees
circle           0        0
triangle         6      360
rectangle       12     1080
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

AlloViz.AlloViz.Elements.Edges.mode

Edges.**mode**(*axis: Axis = 0*, *numeric_only: bool = False*, *dropna: bool = True*) → DataFrame

Get the mode(s) of each element along the selected axis.

The mode of a set of values is the value that appears most often. It can be multiple values.

Parameters

axis
[{0 or 'index', 1 or 'columns'}, default 0] The axis to iterate over while searching for the mode:

- 0 or 'index' : get mode of each column
- 1 or 'columns' : get mode of each row.

numeric_only
[bool, default False] If True, only apply to numeric columns.

dropna
[bool, default True] Don't consider counts of NaN/NaT.

Returns

DataFrame
The modes of each column or row.

See also:

Series.mode
Return the highest frequency value in a Series.

Series.value_counts
Return the counts of values in a Series.

Examples

```
>>> df = pd.DataFrame([('bird', 2, 2),
...                    ('mammal', 4, np.nan),
...                    ('arthropod', 8, 0),
...                    ('bird', 2, np.nan)],
...                   index=('falcon', 'horse', 'spider', 'ostrich'),
...                   columns=('species', 'legs', 'wings'))
>>> df
           species  legs  wings
falcon        bird     2    2.0
horse       mammal     4    NaN
spider   arthropod     8    0.0
ostrich       bird     2    NaN
```

By default, missing values are not considered, and the mode of wings are both 0 and 2. Because the resulting DataFrame has two rows, the second row of `species` and `legs` contains `NaN`.

```
>>> df.mode()
  species  legs  wings
0    bird   2.0    0.0
1     NaN   NaN    2.0
```

Setting `dropna=False` `NaN` values are considered and they can be the mode (like for wings).

```
>>> df.mode(dropna=False)
  species  legs  wings
0    bird     2    NaN
```

Setting `numeric_only=True`, only the mode of numeric columns is computed, and columns of other types are ignored.

```
>>> df.mode(numeric_only=True)
   legs  wings
0   2.0    0.0
1   NaN    2.0
```

To compute the mode over columns and not rows, use the axis parameter:

```
>>> df.mode(axis='columns', numeric_only=True)
           0    1
falcon   2.0  NaN
horse    4.0  NaN
spider   0.0  8.0
ostrich  2.0  NaN
```

AlloViz.AlloViz.Elements.Edges.mul

Edges.**mul**(*other*, *axis: Axis = 'columns'*, *level=None*, *fill_value=None*)

Get Multiplication of dataframe and other, element-wise (binary operator *mul*).

Equivalent to `dataframe * other`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *rmul*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *floordiv*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other
[scalar, sequence, Series, dict or DataFrame] Any single or multiple element data structure, or list-like object.

axis
[{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns. (1 or 'columns'). For Series input, axis to match Series index on.

level
[int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value
[float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame
Result of the arithmetic operation.

See also:

`DataFrame.add`
Add DataFrames.

`DataFrame.sub`
Subtract DataFrames.

`DataFrame.mul`
Multiply DataFrames.

`DataFrame.div`
Divide DataFrames (float division).

`DataFrame.truediv`
Divide DataFrames (float division).

`DataFrame.floordiv`
Divide DataFrames (integer division).

`DataFrame.mod`
Calculate modulo (remainder after division).

`DataFrame.pow`
Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...         axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a dictionary by axis.

```
>>> df.mul({'angles': 0, 'degrees': 2})
            angles  degrees
circle           0      720
triangle         0      360
rectangle        0      720
```

```
>>> df.mul({'circle': 0, 'triangle': 2, 'rectangle': 3}, axis='index')
            angles  degrees
circle           0        0
triangle         6      360
rectangle       12     1080
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

AlloViz.AlloViz.Elements.Edges.multiply

Edges.**multiply**(*other*, *axis: Axis = 'columns'*, *level=None*, *fill_value=None*)

Get Multiplication of dataframe and other, element-wise (binary operator *mul*).

Equivalent to `dataframe * other`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *rmul*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *floordiv*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other
: [scalar, sequence, Series, dict or DataFrame] Any single or multiple element data structure, or list-like object.

axis
: [{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns. (1 or 'columns'). For Series input, axis to match Series index on.

level
: [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value
: [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame
Result of the arithmetic operation.

See also:

DataFrame.add
Add DataFrames.

DataFrame.sub
Subtract DataFrames.

DataFrame.mul
Multiply DataFrames.

DataFrame.div
Divide DataFrames (float division).

DataFrame.truediv
Divide DataFrames (float division).

DataFrame.floordiv
Divide DataFrames (integer division).

DataFrame.mod
Calculate modulo (remainder after division).

DataFrame.pow
Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
```

(continues on next page)

(continued from previous page)

```
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a dictionary by axis.

```
>>> df.mul({'angles': 0, 'degrees': 2})
           angles  degrees
circle          0      720
triangle        0      360
rectangle       0      720
```

```
>>> df.mul({'circle': 0, 'triangle': 2, 'rectangle': 3}, axis='index')
           angles  degrees
circle          0        0
triangle        6      360
rectangle      12     1080
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

AlloViz.AlloViz.Elements.Edges.ne

Edges.**ne**(*other*, *axis: Axis = 'columns'*, *level=None*)

Get Not equal to of dataframe and other, element-wise (binary operator *ne*).

Among flexible wrappers (*eq*, *ne*, *le*, *lt*, *ge*, *gt*) to comparison operators.

Equivalent to ==, *!*=, <=, <, >=, > with support to choose axis (rows or columns) and level for comparison.

Parameters

other
[scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.

axis
[{0 or 'index', 1 or 'columns'}, default 'columns'] Whether to compare by the index (0 or 'index') or columns (1 or 'columns').

level
[int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

Returns

DataFrame of bool
Result of the comparison.

See also:

DataFrame.eq
Compare DataFrames for equality elementwise.

DataFrame.ne
Compare DataFrames for inequality elementwise.

DataFrame.le
Compare DataFrames for less than inequality or equality elementwise.

DataFrame.lt
Compare DataFrames for strictly less than inequality elementwise.

DataFrame.ge
Compare DataFrames for greater than inequality or equality elementwise.

DataFrame.gt
Compare DataFrames for strictly greater than inequality elementwise.

Notes

Mismatched indices will be unioned together. *NaN* values are considered different (i.e. *NaN* != *NaN*).

Examples

```
>>> df = pd.DataFrame({'cost': [250, 150, 100],
...                    'revenue': [100, 250, 300]},
...                   index=['A', 'B', 'C'])
>>> df
   cost  revenue
A   250      100
B   150      250
C   100      300
```

Comparison with a scalar, using either the operator or method:

```
>>> df == 100
    cost  revenue
A  False     True
B  False    False
C   True    False
```

```
>>> df.eq(100)
    cost  revenue
A  False     True
B  False    False
C   True    False
```

When *other* is a `Series`, the columns of a DataFrame are aligned with the index of *other* and broadcast:

```
>>> df != pd.Series([100, 250], index=["cost", "revenue"])
    cost  revenue
A   True     True
B   True    False
C  False     True
```

Use the method to control the broadcast axis:

```
>>> df.ne(pd.Series([100, 300], index=["A", "D"]), axis='index')
   cost  revenue
A  True    False
B  True     True
C  True     True
D  True     True
```

When comparing to an arbitrary sequence, the number of columns must match the number elements in *other*:

```
>>> df == [250, 100]
    cost  revenue
A   True     True
B  False    False
C  False    False
```

Use the method to control the axis:

```
>>> df.eq([250, 250, 100], axis='index')
    cost  revenue
A   True    False
B  False     True
C   True    False
```

Compare to a DataFrame of different shape.

```
>>> other = pd.DataFrame({'revenue': [300, 250, 100, 150]},
...                      index=['A', 'B', 'C', 'D'])
>>> other
   revenue
A      300
B      250
C      100
D      150
```

```
>>> df.gt(other)
    cost  revenue
A  False    False
B  False    False
C  False     True
D  False    False
```

Compare to a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'cost': [250, 150, 100, 150, 300, 220],
...                              'revenue': [100, 250, 300, 200, 175, 225]},
...                             index=[['Q1', 'Q1', 'Q1', 'Q2', 'Q2', 'Q2'],
...                                    ['A', 'B', 'C', 'A', 'B', 'C']])
>>> df_multindex
      cost  revenue
Q1 A   250      100
   B   150      250
   C   100      300
Q2 A   150      200
   B   300      175
   C   220      225
```

```
>>> df.le(df_multindex, level=1)
       cost  revenue
Q1 A   True     True
   B   True     True
   C   True     True
Q2 A  False     True
   B   True    False
   C   True    False
```

AlloViz.AlloViz.Elements.Edges.nlargest

Edges.**nlargest**(*n: int*, *columns: IndexLabel*, *keep: NsmallestNlargestKeep = 'first'*) → DataFrame

Return the first *n* rows ordered by *columns* in descending order.

Return the first *n* rows with the largest values in *columns*, in descending order. The columns that are not specified are returned as well, but not used for ordering.

This method is equivalent to `df.sort_values(columns, ascending=False).head(n)`, but more performant.

Parameters

n
[int] Number of rows to return.

columns
[label or list of labels] Column label(s) to order by.

keep
[{'first', 'last', 'all'}, default 'first'] Where there are duplicate values:

- `first` : prioritize the first occurrence(s)
- `last` : prioritize the last occurrence(s)
- `all` : do not drop any duplicates, even it means selecting more than *n* items.

Returns

DataFrame
The first *n* rows ordered by the given columns in descending order.

See also:

DataFrame.nsmallest
Return the first *n* rows ordered by *columns* in ascending order.

DataFrame.sort_values
Sort DataFrame by the values.

DataFrame.head
Return the first *n* rows without re-ordering.

Notes

This function cannot be used with all column types. For example, when specifying columns with *object* or *category* dtypes, `TypeError` is raised.

Examples

```
>>> df = pd.DataFrame({'population': [59000000, 65000000, 434000,
...                                   434000, 434000, 337000, 11300,
...                                   11300, 11300],
...                    'GDP': [1937894, 2583560 , 12011, 4520, 12128,
...                            17036, 182, 38, 311],
...                    'alpha-2': ["IT", "FR", "MT", "MV", "BN",
...                                "IS", "NR", "TV", "AI"]},
```

(continues on next page)

(continued from previous page)

```
...                   index=["Italy", "France", "Malta",
...                          "Maldives", "Brunei", "Iceland",
...                          "Nauru", "Tuvalu", "Anguilla"])
>>> df
          population      GDP alpha-2
Italy       59000000  1937894      IT
France      65000000  2583560      FR
Malta         434000    12011      MT
Maldives      434000     4520      MV
Brunei        434000    12128      BN
Iceland       337000    17036      IS
Nauru          11300      182      NR
Tuvalu         11300       38      TV
Anguilla       11300      311      AI
```

In the following example, we will use `nlargest` to select the three rows having the largest values in column "population".

```
>>> df.nlargest(3, 'population')
        population      GDP alpha-2
France    65000000  2583560      FR
Italy     59000000  1937894      IT
Malta       434000    12011      MT
```

When using `keep='last'`, ties are resolved in reverse order:

```
>>> df.nlargest(3, 'population', keep='last')
        population      GDP alpha-2
France    65000000  2583560      FR
Italy     59000000  1937894      IT
Brunei      434000    12128      BN
```

When using `keep='all'`, all duplicate items are maintained:

```
>>> df.nlargest(3, 'population', keep='all')
          population      GDP alpha-2
France      65000000  2583560      FR
Italy       59000000  1937894      IT
Malta         434000    12011      MT
Maldives      434000     4520      MV
Brunei        434000    12128      BN
```

To order by the largest values in column "population" and then "GDP", we can specify multiple columns like in the next example.

```
>>> df.nlargest(3, ['population', 'GDP'])
        population      GDP alpha-2
France    65000000  2583560      FR
Italy     59000000  1937894      IT
Brunei      434000    12128      BN
```

AlloViz.AlloViz.Elements.Edges.notna

Edges.**notna**() → DataFrame

Detect existing (non-missing) values.

Return a boolean same-sized object indicating if the values are not NA. Non-missing values get mapped to True. Characters such as empty strings `''` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`). NA values, such as None or `numpy.NaN`, get mapped to False values.

Returns

DataFrame
Mask of bool values for each element in DataFrame that indicates whether an element is not an NA value.

See also:

DataFrame.notnull
Alias of notna.

DataFrame.isna
Boolean inverse of notna.

DataFrame.dropna
Omit axes labels with missing values.

notna
Top-level notna.

Examples

Show which entries in a DataFrame are not NA.

```
>>> df = pd.DataFrame(dict(age=[5, 6, np.nan],
...                        born=[pd.NaT, pd.Timestamp('1939-05-27'),
...                              pd.Timestamp('1940-04-25')],
...                        name=['Alfred', 'Batman', ''],
...                        toy=[None, 'Batmobile', 'Joker']))
>>> df
   age       born    name        toy
0  5.0        NaT  Alfred       None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25              Joker
```

```
>>> df.notna()
     age   born  name    toy
0   True  False  True  False
1   True   True  True   True
2  False   True  True   True
```

Show which entries in a Series are not NA.

```
>>> ser = pd.Series([5, 6, np.nan])
>>> ser
```

(continues on next page)

(continued from previous page)

```
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.notna()
0     True
1     True
2    False
dtype: bool
```

AlloViz.AlloViz.Elements.Edges.notnull

Edges.**notnull**() → DataFrame

DataFrame.notnull is an alias for DataFrame.notna.

Detect existing (non-missing) values.

Return a boolean same-sized object indicating if the values are not NA. Non-missing values get mapped to True. Characters such as empty strings `''` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`). NA values, such as None or `numpy.NaN`, get mapped to False values.

Returns

DataFrame
Mask of bool values for each element in DataFrame that indicates whether an element is not an NA value.

See also:

DataFrame.notnull
Alias of notna.

DataFrame.isna
Boolean inverse of notna.

DataFrame.dropna
Omit axes labels with missing values.

notna
Top-level notna.

Examples

Show which entries in a DataFrame are not NA.

```
>>> df = pd.DataFrame(dict(age=[5, 6, np.nan],
...                        born=[pd.NaT, pd.Timestamp('1939-05-27'),
...                              pd.Timestamp('1940-04-25')],
...                        name=['Alfred', 'Batman', ''],
...                        toy=[None, 'Batmobile', 'Joker']))
>>> df
```

(continues on next page)

(continued from previous page)

```
   age       born    name        toy
0  5.0        NaT  Alfred       None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25              Joker
```

```
>>> df.notna()
     age   born  name    toy
0   True  False  True  False
1   True   True  True   True
2  False   True  True   True
```

Show which entries in a Series are not NA.

```
>>> ser = pd.Series([5, 6, np.nan])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.notna()
0     True
1     True
2    False
dtype: bool
```

AlloViz.AlloViz.Elements.Edges.nsmallest

Edges.**nsmallest**(*n: int*, *columns: IndexLabel*, *keep: NsmallestNlargestKeep = 'first'*) → DataFrame

Return the first *n* rows ordered by *columns* in ascending order.

Return the first *n* rows with the smallest values in *columns*, in ascending order. The columns that are not specified are returned as well, but not used for ordering.

This method is equivalent to `df.sort_values(columns, ascending=True).head(n)`, but more performant.

Parameters

n
[int] Number of items to retrieve.

columns
[list or str] Column name or names to order by.

keep
[{'first', 'last', 'all'}, default 'first'] Where there are duplicate values:

- `first` : take the first occurrence.
- `last` : take the last occurrence.
- `all` : do not drop any duplicates, even it means selecting more than *n* items.

Returns

DataFrame

See also:

DataFrame.nlargest
Return the first *n* rows ordered by *columns* in descending order.

DataFrame.sort_values
Sort DataFrame by the values.

DataFrame.head
Return the first *n* rows without re-ordering.

Examples

```
>>> df = pd.DataFrame({'population': [59000000, 65000000, 434000,
...                                   434000, 434000, 337000, 337000,
...                                   11300, 11300],
...                    'GDP': [1937894, 2583560 , 12011, 4520, 12128,
...                            17036, 182, 38, 311],
...                    'alpha-2': ["IT", "FR", "MT", "MV", "BN",
...                                "IS", "NR", "TV", "AI"]},
...                   index=["Italy", "France", "Malta",
...                          "Maldives", "Brunei", "Iceland",
...                          "Nauru", "Tuvalu", "Anguilla"])
>>> df
          population      GDP alpha-2
Italy       59000000  1937894      IT
France      65000000  2583560      FR
Malta         434000    12011      MT
Maldives      434000     4520      MV
Brunei        434000    12128      BN
Iceland       337000    17036      IS
Nauru         337000      182      NR
Tuvalu         11300       38      TV
Anguilla       11300      311      AI
```

In the following example, we will use `nsmallest` to select the three rows having the smallest values in column "population".

```
>>> df.nsmallest(3, 'population')
          population    GDP alpha-2
Tuvalu         11300     38      TV
Anguilla       11300    311      AI
Iceland       337000  17036      IS
```

When using `keep='last'`, ties are resolved in reverse order:

```
>>> df.nsmallest(3, 'population', keep='last')
          population  GDP alpha-2
Anguilla       11300  311      AI
Tuvalu         11300   38      TV
Nauru         337000  182      NR
```

When using `keep='all'`, all duplicate items are maintained:

```
>>> df.nsmallest(3, 'population', keep='all')
          population    GDP alpha-2
Tuvalu         11300     38      TV
Anguilla       11300    311      AI
Iceland       337000  17036      IS
Nauru         337000    182      NR
```

To order by the smallest values in column "population" and then "GDP", we can specify multiple columns like in the next example.

```
>>> df.nsmallest(3, ['population', 'GDP'])
          population  GDP alpha-2
Tuvalu         11300   38      TV
Anguilla       11300  311      AI
Nauru         337000  182      NR
```

AlloViz.AlloViz.Elements.Edges.nunique

Edges.**nunique**(*axis: Axis = 0*, *dropna: bool = True*) → Series

Count number of distinct elements in specified axis.

Return Series with number of distinct elements. Can ignore NaN values.

Parameters

axis
[{0 or 'index', 1 or 'columns'}, default 0] The axis to use. 0 or 'index' for row-wise, 1 or 'columns' for column-wise.

dropna
[bool, default True] Don't include NaN in the counts.

Returns

Series

See also:

Series.nunique
Method nunique for Series.

DataFrame.count
Count non-NA cells for each column or row.

Examples

```
>>> df = pd.DataFrame({'A': [4, 5, 6], 'B': [4, 1, 1]})
>>> df.nunique()
A    3
B    2
dtype: int64
```

```
>>> df.nunique(axis=1)
0    1
1    2
2    2
dtype: int64
```

AlloViz.AlloViz.Elements.Edges.pad

Edges.**pad**(*, *axis: None | Axis = None*, *inplace: bool_t = False*, *limit: None | int = None*, *downcast: dict | None | lib.NoDefault = _NoDefault.no_default*) → Self | None

Fill NA/NaN values by propagating the last valid observation to next valid.

Deprecated since version 2.0: Series/DataFrame.pad is deprecated. Use Series/DataFrame.ffill instead.

Returns

Series/DataFrame or None
Object with missing values filled or None if `inplace=True`.

Examples

Please see examples for `DataFrame.ffill()` or `Series.ffill()`.

AlloViz.AlloViz.Elements.Edges.pct_change

Edges.**pct_change**(*periods: int = 1*, *fill_method: Literal['backfill', 'bfill', 'ffill', 'pad'] | None | Literal[_NoDefault.no_default] = _NoDefault.no_default*, *limit: int | None | Literal[_NoDefault.no_default] = _NoDefault.no_default*, *freq=None*, ***kwargs*) → None

Fractional change between the current and a prior element.

Computes the fractional change from the immediately previous row by default. This is useful in comparing the fraction of change in a time series of elements.

Note: Despite the name of this method, it calculates fractional change (also known as per unit change or relative change) and not percentage change. If you need the percentage change, multiply these values by 100.

Parameters

periods
[int, default 1] Periods to shift for forming percent change.

fill_method
[{'backfill', 'bfill', 'pad', 'ffill', None}, default 'pad'] How to handle NAs **before** computing percent changes.

Deprecated since version 2.1.

limit
[int, default None] The number of consecutive NAs to fill before stopping.

Deprecated since version 2.1.

freq
[DateOffset, timedelta, or str, optional] Increment to use from time series API (e.g. 'M' or BDay()).

****kwargs**
Additional keyword arguments are passed into *DataFrame.shift* or *Series.shift*.

Returns

Series or DataFrame
The same type as the calling object.

See also:

`Series.diff`
Compute the difference of two elements in a Series.

`DataFrame.diff`
Compute the difference of two elements in a DataFrame.

`Series.shift`
Shift the index by some number of periods.

`DataFrame.shift`
Shift the index by some number of periods.

Examples

Series

```
>>> s = pd.Series([90, 91, 85])
>>> s
0    90
1    91
2    85
dtype: int64
```

```
>>> s.pct_change()
0         NaN
1    0.011111
2   -0.065934
dtype: float64
```

```
>>> s.pct_change(periods=2)
0         NaN
1         NaN
2   -0.055556
dtype: float64
```

See the percentage change in a Series where filling NAs with last valid observation forward to next valid.

```
>>> s = pd.Series([90, 91, None, 85])
>>> s
0    90.0
1    91.0
```

(continues on next page)

(continued from previous page)

```
2     NaN
3    85.0
dtype: float64
```

```
>>> s.ffill().pct_change()
0         NaN
1    0.011111
2    0.000000
3   -0.065934
dtype: float64
```

DataFrame

Percentage change in French franc, Deutsche Mark, and Italian lira from 1980-01-01 to 1980-03-01.

```
>>> df = pd.DataFrame({
...     'FR': [4.0405, 4.0963, 4.3149],
...     'GR': [1.7246, 1.7482, 1.8519],
...     'IT': [804.74, 810.01, 860.13]},
...     index=['1980-01-01', '1980-02-01', '1980-03-01'])
>>> df
                FR      GR      IT
1980-01-01  4.0405  1.7246  804.74
1980-02-01  4.0963  1.7482  810.01
1980-03-01  4.3149  1.8519  860.13
```

```
>>> df.pct_change()
                  FR        GR        IT
1980-01-01       NaN       NaN       NaN
1980-02-01  0.013810  0.013684  0.006549
1980-03-01  0.053365  0.059318  0.061876
```

Percentage of change in GOOG and APPL stock volume. Shows computing the percentage change between columns.

```
>>> df = pd.DataFrame({
...     '2016': [1769950, 30586265],
...     '2015': [1500923, 40912316],
...     '2014': [1371819, 41403351]},
...     index=['GOOG', 'APPL'])
>>> df
          2016      2015      2014
GOOG   1769950   1500923   1371819
APPL  30586265  40912316  41403351
```

```
>>> df.pct_change(axis='columns', periods=-1)
          2016      2015  2014
GOOG  0.179241  0.094112   NaN
APPL -0.252395 -0.011860   NaN
```

AlloViz.AlloViz.Elements.Edges.pipe

Edges.**pipe**(*func: Callable[[...], T] | tuple[Callable[[...], T], str]*, **args*, ***kwargs*) → T

Apply chainable functions that expect Series or DataFrames.

Parameters

func
[function] Function to apply to the Series/DataFrame. `args`, and `kwargs` are passed into `func`. Alternatively a `(callable, data_keyword)` tuple where `data_keyword` is a string indicating the keyword of `callable` that expects the Series/DataFrame.

***args**
[iterable, optional] Positional arguments passed into `func`.

****kwargs**
[mapping, optional] A dictionary of keyword arguments passed into `func`.

Returns

the return type of `func`.

See also:

`DataFrame.apply`
Apply a function along input axis of DataFrame.

`DataFrame.map`
Apply a function elementwise on a whole DataFrame.

`Series.map`
Apply a mapping correspondence on a `Series`.

Notes

Use `.pipe` when chaining together functions that expect Series, DataFrames or GroupBy objects.

Examples

Constructing a income DataFrame from a dictionary.

```
>>> data = [[8000, 1000], [9500, np.nan], [5000, 2000]]
>>> df = pd.DataFrame(data, columns=['Salary', 'Others'])
>>> df
   Salary  Others
0    8000  1000.0
1    9500     NaN
2    5000  2000.0
```

Functions that perform tax reductions on an income DataFrame.

```
>>> def subtract_federal_tax(df):
...     return df * 0.9
>>> def subtract_state_tax(df, rate):
...     return df * (1 - rate)
```

(continues on next page)

(continued from previous page)

```
>>> def subtract_national_insurance(df, rate, rate_increase):
...     new_rate = rate + rate_increase
...     return df * (1 - new_rate)
```

Instead of writing

```
>>> subtract_national_insurance(
...     subtract_state_tax(subtract_federal_tax(df), rate=0.12),
...     rate=0.05,
...     rate_increase=0.02)
```

You can write

```
>>> (
...     df.pipe(subtract_federal_tax)
...     .pipe(subtract_state_tax, rate=0.12)
...     .pipe(subtract_national_insurance, rate=0.05, rate_increase=0.02)
... )
    Salary   Others
0  5892.48   736.56
1  6997.32      NaN
2  3682.80  1473.12
```

If you have a function that takes the data as (say) the second argument, pass a tuple indicating which keyword expects the data. For example, suppose `national_insurance` takes its data as `df` in the second argument:

```
>>> def subtract_national_insurance(rate, df, rate_increase):
...     new_rate = rate + rate_increase
...     return df * (1 - new_rate)
>>> (
...     df.pipe(subtract_federal_tax)
...     .pipe(subtract_state_tax, rate=0.12)
...     .pipe(
...         (subtract_national_insurance, 'df'),
...         rate=0.05,
...         rate_increase=0.02
...     )
... )
    Salary   Others
0  5892.48   736.56
1  6997.32      NaN
2  3682.80  1473.12
```

AlloViz.AlloViz.Elements.Edges.pivot

Edges.**pivot**(*, *columns*, *index=_NoDefault.no_default*, *values=_NoDefault.no_default*) → DataFrame

Return reshaped DataFrame organized by given index / column values.

Reshape data (produce a "pivot" table) based on column values. Uses unique values from specified *index* / *columns* to form axes of the resulting DataFrame. This function does not support data aggregation, multiple values will result in a MultiIndex in the columns. See the User Guide for more on reshaping.

Parameters

columns
[str or object or a list of str] Column to use to make new frame's columns.

index
[str or object or a list of str, optional] Column to use to make new frame's index. If not given, uses existing index.

values
[str, object or a list of the previous, optional] Column(s) to use for populating new frame's values. If not specified, all remaining columns will be used and the result will have hierarchically indexed columns.

Returns

DataFrame
Returns reshaped DataFrame.

Raises

ValueError:
When there are any *index*, *columns* combinations with multiple values. *DataFrame.pivot_table* when you need to aggregate.

See also:

`DataFrame.pivot_table`
Generalization of pivot that can handle duplicate values for one index/column pair.

`DataFrame.unstack`
Pivot based on the index values instead of a column.

`wide_to_long`
Wide panel to long format. Less flexible but more user-friendly than melt.

Notes

For finer-tuned control, see hierarchical indexing documentation along with the related stack/unstack methods.

Reference the user guide for more examples.

Examples

```
>>> df = pd.DataFrame({'foo': ['one', 'one', 'one', 'two', 'two',
...                            'two'],
...                    'bar': ['A', 'B', 'C', 'A', 'B', 'C'],
...                    'baz': [1, 2, 3, 4, 5, 6],
...                    'zoo': ['x', 'y', 'z', 'q', 'w', 't']})
>>> df
    foo   bar  baz  zoo
0   one   A    1    x
1   one   B    2    y
2   one   C    3    z
3   two   A    4    q
4   two   B    5    w
5   two   C    6    t
```

```
>>> df.pivot(index='foo', columns='bar', values='baz')
bar  A   B   C
foo
one  1   2   3
two  4   5   6
```

```
>>> df.pivot(index='foo', columns='bar')['baz']
bar  A   B   C
foo
one  1   2   3
two  4   5   6
```

```
>>> df.pivot(index='foo', columns='bar', values=['baz', 'zoo'])
      baz       zoo
bar   A  B  C   A  B  C
foo
one   1  2  3   x  y  z
two   4  5  6   q  w  t
```

You could also assign a list of column names or a list of index names.

```
>>> df = pd.DataFrame({
...        "lev1": [1, 1, 1, 2, 2, 2],
...        "lev2": [1, 1, 2, 1, 1, 2],
...        "lev3": [1, 2, 1, 2, 1, 2],
...        "lev4": [1, 2, 3, 4, 5, 6],
...        "values": [0, 1, 2, 3, 4, 5]})
>>> df
    lev1 lev2 lev3 lev4 values
0   1    1    1    1    0
1   1    1    2    2    1
2   1    2    1    3    2
3   2    1    2    4    3
4   2    1    1    5    4
5   2    2    2    6    5
```

```
>>> df.pivot(index="lev1", columns=["lev2", "lev3"], values="values")
lev2    1         2
lev3    1    2    1    2
lev1
1     0.0  1.0  2.0  NaN
2     4.0  3.0  NaN  5.0
```

```
>>> df.pivot(index=["lev1", "lev2"], columns=["lev3"], values="values")
      lev3    1    2
lev1  lev2
   1     1  0.0  1.0
         2  2.0  NaN
   2     1  4.0  3.0
         2  NaN  5.0
```

A ValueError is raised if there are any duplicates.

```
>>> df = pd.DataFrame({"foo": ['one', 'one', 'two', 'two'],
...                    "bar": ['A', 'A', 'B', 'C'],
...                    "baz": [1, 2, 3, 4]})
>>> df
   foo bar  baz
0  one   A    1
1  one   A    2
2  two   B    3
3  two   C    4
```

Notice that the first two rows are the same for our *index* and *columns* arguments.

```
>>> df.pivot(index='foo', columns='bar', values='baz')
Traceback (most recent call last):
   ...
ValueError: Index contains duplicate entries, cannot reshape
```

AlloViz.AlloViz.Elements.Edges.pivot_table

Edges.**pivot_table**(*values=None*, *index=None*, *columns=None*, *aggfunc: AggFuncType = 'mean'*, *fill_value=None*, *margins: bool = False*, *dropna: bool = True*, *margins_name: Level = 'All'*, *observed: bool = False*, *sort: bool = True*) → DataFrame

Create a spreadsheet-style pivot table as a DataFrame.

The levels in the pivot table will be stored in MultiIndex objects (hierarchical indexes) on the index and columns of the result DataFrame.

Parameters

values
[list-like or scalar, optional] Column or columns to aggregate.

index
[column, Grouper, array, or list of the previous] Keys to group by on the pivot table index. If a list is passed, it can contain any of the other types (except list). If an array is passed, it must be the same length as the data and will be used in the same manner as column values.

columns
[column, Grouper, array, or list of the previous] Keys to group by on the pivot table column. If a list is passed, it can contain any of the other types (except list). If an array is passed, it must be the same length as the data and will be used in the same manner as column values.

aggfunc
[function, list of functions, dict, default "mean"] If a list of functions is passed, the resulting pivot table will have hierarchical columns whose top level are the function names (inferred from the function objects themselves). If a dict is passed, the key is column to aggregate and the value is function or list of functions. If `margin=True`, aggfunc will be used to calculate the partial aggregates.

fill_value
[scalar, default None] Value to replace missing values with (in the resulting pivot table, after aggregation).

margins
[bool, default False] If `margins=True`, special `All` columns and rows will be added with partial group aggregates across the categories on the rows and columns.

dropna
[bool, default True] Do not include columns whose entries are all NaN. If True, rows with a NaN value in any column will be omitted before computing margins.

margins_name
[str, default 'All'] Name of the row / column that will contain the totals when margins is True.

observed
[bool, default False] This only applies if any of the groupers are Categoricals. If True: only show observed values for categorical groupers. If False: show all values for categorical groupers.

sort
[bool, default True] Specifies if the result should be sorted.

New in version 1.3.0.

Returns

DataFrame
An Excel style pivot table.

See also:

`DataFrame.pivot`
Pivot without aggregation that can handle non-numeric data.

`DataFrame.melt`
Unpivot a DataFrame from wide to long format, optionally leaving identifiers set.

`wide_to_long`
Wide panel to long format. Less flexible but more user-friendly than melt.

Notes

Reference the user guide for more examples.

Examples

```
>>> df = pd.DataFrame({"A": ["foo", "foo", "foo", "foo", "foo",
...                          "bar", "bar", "bar", "bar"],
...                    "B": ["one", "one", "one", "two", "two",
...                          "one", "one", "two", "two"],
...                    "C": ["small", "large", "large", "small",
...                          "small", "large", "small", "small",
...                          "large"],
...                    "D": [1, 2, 2, 3, 3, 4, 5, 6, 7],
...                    "E": [2, 4, 5, 5, 6, 6, 8, 9, 9]})
>>> df
     A    B      C  D  E
0  foo  one  small  1  2
1  foo  one  large  2  4
2  foo  one  large  2  5
3  foo  two  small  3  5
4  foo  two  small  3  6
5  bar  one  large  4  6
6  bar  one  small  5  8
7  bar  two  small  6  9
8  bar  two  large  7  9
```

This first example aggregates values by taking the sum.

```
>>> table = pd.pivot_table(df, values='D', index=['A', 'B'],
...                        columns=['C'], aggfunc="sum")
>>> table
C        large  small
A   B
bar one    4.0    5.0
    two    7.0    6.0
foo one    4.0    1.0
    two    NaN    6.0
```

We can also fill missing values using the *fill_value* parameter.

```
>>> table = pd.pivot_table(df, values='D', index=['A', 'B'],
...                        columns=['C'], aggfunc="sum", fill_value=0)
>>> table
C        large  small
A   B
bar one      4      5
    two      7      6
foo one      4      1
    two      0      6
```

The next example aggregates by taking the mean across multiple columns.

```
>>> table = pd.pivot_table(df, values=['D', 'E'], index=['A', 'C'],
...                        aggfunc={'D': "mean", 'E': "mean"})
>>> table
                D         E
A   C
bar large  5.500000  7.500000
    small  5.500000  8.500000
foo large  2.000000  4.500000
    small  2.333333  4.333333
```

We can also calculate multiple types of aggregations for any given value column.

```
>>> table = pd.pivot_table(df, values=['D', 'E'], index=['A', 'C'],
...                        aggfunc={'D': "mean",
...                                 'E': ["min", "max", "mean"]})
>>> table
                  D   E
               mean max      mean  min
A   C
bar large  5.500000   9  7.500000    6
    small  5.500000   9  8.500000    8
foo large  2.000000   5  4.500000    4
    small  2.333333   6  4.333333    2
```

AlloViz.AlloViz.Elements.Edges.pop

Edges.**pop**(*item: Hashable*) → Series

Return item and drop from frame. Raise KeyError if not found.

Parameters

item
[label] Label of column to be popped.

Returns

Series

Examples

```
>>> df = pd.DataFrame([('falcon', 'bird', 389.0),
...                    ('parrot', 'bird', 24.0),
...                    ('lion', 'mammal', 80.5),
...                    ('monkey', 'mammal', np.nan)],
...                   columns=('name', 'class', 'max_speed'))
>>> df
     name   class  max_speed
0  falcon    bird      389.0
1  parrot    bird       24.0
2    lion  mammal       80.5
3  monkey  mammal        NaN
```

```
>>> df.pop('class')
0      bird
1      bird
2    mammal
3    mammal
Name: class, dtype: object
```

```
>>> df
     name  max_speed
0  falcon      389.0
1  parrot       24.0
2    lion       80.5
3  monkey        NaN
```

AlloViz.AlloViz.Elements.Edges.pow

Edges.**pow**(*other*, *axis: Axis = 'columns'*, *level=None*, *fill_value=None*)

Get Exponential power of dataframe and other, element-wise (binary operator *pow*).

Equivalent to `dataframe ** other`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *rpow*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *floordiv*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other
: [scalar, sequence, Series, dict or DataFrame] Any single or multiple element data structure, or list-like object.

axis
: [{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns. (1 or 'columns'). For Series input, axis to match Series index on.

level
: [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value
: [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame
: Result of the arithmetic operation.

See also:

`DataFrame.add`
: Add DataFrames.

`DataFrame.sub`
: Subtract DataFrames.

DataFrame.mul
Multiply DataFrames.

DataFrame.div
Divide DataFrames (float division).

DataFrame.truediv
Divide DataFrames (float division).

DataFrame.floordiv
Divide DataFrames (integer division).

DataFrame.mod
Calculate modulo (remainder after division).

DataFrame.pow
Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a dictionary by axis.

```
>>> df.mul({'angles': 0, 'degrees': 2})
            angles  degrees
circle           0      720
triangle         0      360
rectangle        0      720
```

```
>>> df.mul({'circle': 0, 'triangle': 2, 'rectangle': 3}, axis='index')
            angles  degrees
circle           0        0
triangle         6      360
rectangle       12     1080
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

AlloViz.AlloViz.Elements.Edges.prod

Edges.**prod**(*axis: Axis | None = 0*, *skipna: bool = True*, *numeric_only: bool = False*, *min_count: int = 0*, ***kwargs*)

Return the product of the values over the requested axis.

Parameters

axis

[{index (0), columns (1)}] Axis for the function to be applied on. For *Series* this parameter is unused and defaults to 0.

For DataFrames, specifying `axis=None` will apply the aggregation across both axes.

New in version 2.0.0.

skipna
[bool, default True] Exclude NA/null values when computing the result.

numeric_only
[bool, default False] Include only float, int, boolean columns. Not implemented for Series.

min_count
[int, default 0] The required number of valid values to perform the operation. If fewer than `min_count` non-NA values are present the result will be NA.

****kwargs**
Additional keyword arguments to be passed to the function.

Returns

Series or scalar

See also:

`Series.sum`
Return the sum.

`Series.min`
Return the minimum.

`Series.max`
Return the maximum.

`Series.idxmin`
Return the index of the minimum.

`Series.idxmax`
Return the index of the maximum.

`DataFrame.sum`
Return the sum over the requested axis.

`DataFrame.min`
Return the minimum over the requested axis.

`DataFrame.max`
Return the maximum over the requested axis.

`DataFrame.idxmin`
Return the index of the minimum over the requested axis.

`DataFrame.idxmax`
Return the index of the maximum over the requested axis.

Examples

By default, the product of an empty or all-NA Series is 1

```
>>> pd.Series([], dtype="float64").prod()
1.0
```

This can be controlled with the `min_count` parameter

```
>>> pd.Series([], dtype="float64").prod(min_count=1)
nan
```

Thanks to the `skipna` parameter, `min_count` handles all-NA and empty series identically.

```
>>> pd.Series([np.nan]).prod()
1.0
```

```
>>> pd.Series([np.nan]).prod(min_count=1)
nan
```

AlloViz.AlloViz.Elements.Edges.product

Edges.**product**(*axis: Axis | None = 0*, *skipna: bool = True*, *numeric_only: bool = False*, *min_count: int = 0*, ***kwargs*)

Return the product of the values over the requested axis.

Parameters

axis
[{index (0), columns (1)}] Axis for the function to be applied on. For *Series* this parameter is unused and defaults to 0.

For DataFrames, specifying `axis=None` will apply the aggregation across both axes.

New in version 2.0.0.

skipna
[bool, default True] Exclude NA/null values when computing the result.

numeric_only
[bool, default False] Include only float, int, boolean columns. Not implemented for Series.

min_count
[int, default 0] The required number of valid values to perform the operation. If fewer than `min_count` non-NA values are present the result will be NA.

****kwargs**
Additional keyword arguments to be passed to the function.

Returns

Series or scalar

See also:

Series.sum
Return the sum.

Series.min
Return the minimum.

Series.max
Return the maximum.

Series.idxmin
Return the index of the minimum.

Series.idxmax
Return the index of the maximum.

DataFrame.sum
Return the sum over the requested axis.

DataFrame.min
Return the minimum over the requested axis.

DataFrame.max
Return the maximum over the requested axis.

DataFrame.idxmin
Return the index of the minimum over the requested axis.

DataFrame.idxmax
Return the index of the maximum over the requested axis.

Examples

By default, the product of an empty or all-NA Series is 1

```
>>> pd.Series([], dtype="float64").prod()
1.0
```

This can be controlled with the `min_count` parameter

```
>>> pd.Series([], dtype="float64").prod(min_count=1)
nan
```

Thanks to the `skipna` parameter, `min_count` handles all-NA and empty series identically.

```
>>> pd.Series([np.nan]).prod()
1.0
```

```
>>> pd.Series([np.nan]).prod(min_count=1)
nan
```

AlloViz.AlloViz.Elements.Edges.quantile

Edges.**quantile**(*q: float | AnyArrayLike | Sequence[float] = 0.5*, *axis: Axis = 0*, *numeric_only: bool = False*, *interpolation: QuantileInterpolation = 'linear'*, *method: Literal['single', 'table'] = 'single'*) → Series | DataFrame

Return values at the given quantile over requested axis.

Parameters

q
[float or array-like, default 0.5 (50% quantile)] Value between 0 <= q <= 1, the quantile(s) to compute.

axis
[{0 or 'index', 1 or 'columns'}, default 0] Equals 0 or 'index' for row-wise, 1 or 'columns' for column-wise.

numeric_only
[bool, default False] Include only *float*, *int* or *boolean* data.

Changed in version 2.0.0: The default value of `numeric_only` is now `False`.

interpolation
[{'linear', 'lower', 'higher', 'midpoint', 'nearest'}] This optional parameter specifies the interpolation method to use, when the desired quantile lies between two data points *i* and *j*:

- linear: *i* + (*j* - *i*) * *fraction*, where *fraction* is the fractional part of the index surrounded by *i* and *j*.
- lower: *i*.
- higher: *j*.
- nearest: *i* or *j* whichever is nearest.
- midpoint: (*i* + *j*) / 2.

method
[{'single', 'table'}, default 'single'] Whether to compute quantiles per-column ('single') or over all columns ('table'). When 'table', the only allowed interpolation methods are 'nearest', 'lower', and 'higher'.

Returns

Series or DataFrame

If q is an array, a DataFrame will be returned where the
index is q, the columns are the columns of self, and the values are the quantiles.

If q is a float, a Series will be returned where the
index is the columns of self and the values are the quantiles.

See also:

`core.window.rolling.Rolling.quantile`
Rolling quantile.

`numpy.percentile`
Numpy function to compute the percentile.

Examples

```
>>> df = pd.DataFrame(np.array([[1, 1], [2, 10], [3, 100], [4, 100]]),
...                   columns=['a', 'b'])
>>> df.quantile(.1)
a    1.3
b    3.7
Name: 0.1, dtype: float64
>>> df.quantile([.1, .5])
       a     b
0.1  1.3   3.7
0.5  2.5  55.0
```

Specifying *method='table'* will compute the quantile over all columns.

```
>>> df.quantile(.1, method="table", interpolation="nearest")
a    1
b    1
Name: 0.1, dtype: int64
```

(continues on next page)

(continued from previous page)

```
>>> df.quantile([.1, .5], method="table", interpolation="nearest")
     a    b
0.1  1    1
0.5  3  100
```

Specifying *numeric_only=False* will also compute the quantile of datetime and timedelta data.

```
>>> df = pd.DataFrame({'A': [1, 2],
...                    'B': [pd.Timestamp('2010'),
...                          pd.Timestamp('2011')],
...                    'C': [pd.Timedelta('1 days'),
...                          pd.Timedelta('2 days')]})
>>> df.quantile(0.5, numeric_only=False)
A                    1.5
B    2010-07-02 12:00:00
C        1 days 12:00:00
Name: 0.5, dtype: object
```

AlloViz.AlloViz.Elements.Edges.query

`Edges.query`(*expr: str*, *, *inplace: bool = False*, ***kwargs*) → DataFrame | None

Query the columns of a DataFrame with a boolean expression.

Parameters

expr
[str] The query string to evaluate.

You can refer to variables in the environment by prefixing them with an '@' character like `@a + b`.

You can refer to column names that are not valid Python variable names by surrounding them in backticks. Thus, column names containing spaces or punctuations (besides underscores) or starting with digits must be surrounded by backticks. (For example, a column named "Area (cm^2)" would be referenced as `` `Area (cm^2)` ``). Column names which are Python keywords (like "list", "for", "import", etc) cannot be used.

For example, if one of your columns is called `a a` and you want to sum it with `b`, your query should be `` `a a` + b ``.

inplace
[bool] Whether to modify the DataFrame rather than creating a new one.

***kwargs**
See the documentation for *eval()* for complete details on the keyword arguments accepted by `DataFrame.query()`.

Returns

DataFrame or None
DataFrame resulting from the provided query expression or None if `inplace=True`.

See also:

eval
Evaluate a string describing operations on DataFrame columns.

DataFrame.eval
Evaluate a string describing operations on DataFrame columns.

Notes

The result of the evaluation of this expression is first passed to `DataFrame.loc` and if that fails because of a multidimensional key (e.g., a DataFrame) then the result will be passed to `DataFrame.__getitem__()`.

This method uses the top-level *eval()* function to evaluate the passed query.

The `query()` method uses a slightly modified Python syntax by default. For example, the `&` and `|` (bitwise) operators have the precedence of their boolean cousins, `and` and `or`. This *is* syntactically valid Python, however the semantics are different.

You can change the semantics of the expression by passing the keyword argument `parser='python'`. This enforces the same semantics as evaluation in Python space. Likewise, you can pass `engine='python'` to evaluate an expression using Python itself as a backend. This is not recommended as it is inefficient compared to using `numexpr` as the engine.

The `DataFrame.index` and `DataFrame.columns` attributes of the `DataFrame` instance are placed in the query namespace by default, which allows you to treat both the index and columns of the frame as a column in the frame. The identifier `index` is used for the frame index; you can also use the name of the index to identify it in a query. Please note that Python keywords may not be used as identifiers.

For further details and examples see the `query` documentation in indexing.

Backtick quoted variables

Backtick quoted variables are parsed as literal Python code and are converted internally to a Python valid identifier. This can lead to the following problems.

During parsing a number of disallowed characters inside the backtick quoted string are replaced by strings that are allowed as a Python identifier. These characters include all operators in Python, the space character, the question mark, the exclamation mark, the dollar sign, and the euro sign. For other characters that fall outside the ASCII range (U+0001..U+007F) and those that are not further specified in PEP 3131, the query parser will raise an error. This excludes whitespace different than the space character, but also the hashtag (as it is used for comments) and the backtick itself (backtick can also not be escaped).

In a special case, quotes that make a pair around a backtick can confuse the parser. For example, `` `it's` > `that's` `` will raise an error, as it forms a quoted string (`'s > `that'`) with a backtick inside.

See also the Python documentation about lexical analysis (https://docs.python.org/3/reference/lexical_analysis.html) in combination with the source code in `pandas.core.computation.parsing`.

Examples

```
>>> df = pd.DataFrame({'A': range(1, 6),
...                    'B': range(10, 0, -2),
...                    'C C': range(10, 5, -1)})
>>> df
   A   B  C C
0  1  10   10
1  2   8    9
2  3   6    8
3  4   4    7
4  5   2    6
```

(continues on next page)

(continued from previous page)

```
>>> df.query('A > B')
   A  B  C C
4  5  2    6
```

The previous expression is equivalent to

```
>>> df[df.A > df.B]
   A  B  C C
4  5  2    6
```

For columns with spaces in their name, you can use backtick quoting.

```
>>> df.query('B == `C C`')
   A   B  C C
0  1  10   10
```

The previous expression is equivalent to

```
>>> df[df.B == df['C C']]
   A   B  C C
0  1  10   10
```

AlloViz.AlloViz.Elements.Edges.radd

Edges.**radd**(*other*, *axis: Axis = 'columns'*, *level=None*, *fill_value=None*)

Get Addition of dataframe and other, element-wise (binary operator *radd*).

Equivalent to `other + dataframe`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *add*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *floordiv*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other
: [scalar, sequence, Series, dict or DataFrame] Any single or multiple element data structure, or list-like object.

axis
: [{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns. (1 or 'columns'). For Series input, axis to match Series index on.

level
: [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value
: [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame
: Result of the arithmetic operation.

See also:

DataFrame.add
Add DataFrames.

DataFrame.sub
Subtract DataFrames.

DataFrame.mul
Multiply DataFrames.

DataFrame.div
Divide DataFrames (float division).

DataFrame.truediv
Divide DataFrames (float division).

DataFrame.floordiv
Divide DataFrames (integer division).

DataFrame.mod
Calculate modulo (remainder after division).

DataFrame.pow
Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a dictionary by axis.

```
>>> df.mul({'angles': 0, 'degrees': 2})
            angles  degrees
circle           0      720
triangle         0      360
rectangle        0      720
```

```
>>> df.mul({'circle': 0, 'triangle': 2, 'rectangle': 3}, axis='index')
            angles  degrees
circle           0        0
triangle         6      360
rectangle       12     1080
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
```

(continues on next page)

(continued from previous page)

```
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

AlloViz.AlloViz.Elements.Edges.rank

Edges.**rank**(*axis: int | Literal['index', 'columns', 'rows'] = 0*, *method: Literal['average', 'min', 'max', 'first', 'dense'] = 'average'*, *numeric_only: bool = False*, *na_option: Literal['keep', 'top', 'bottom'] = 'keep'*, *ascending: bool = True*, *pct: bool = False*) → None

Compute numerical data ranks (1 through n) along axis.

By default, equal values are assigned a rank that is the average of the ranks of those values.

Parameters

axis
[{0 or 'index', 1 or 'columns'}, default 0] Index to direct ranking. For *Series* this parameter is unused and defaults to 0.

method
[{'average', 'min', 'max', 'first', 'dense'}, default 'average'] How to rank the group of records that have the same value (i.e. ties):

- average: average rank of the group
- min: lowest rank in the group
- max: highest rank in the group
- first: ranks assigned in order they appear in the array
- dense: like 'min', but rank always increases by 1 between groups.

numeric_only
[bool, default False] For DataFrame objects, rank only numeric columns if set to True.

Changed in version 2.0.0: The default value of `numeric_only` is now `False`.

na_option
[{'keep', 'top', 'bottom'}, default 'keep'] How to rank NaN values:

- keep: assign NaN rank to NaN values
- top: assign lowest rank to NaN values
- bottom: assign highest rank to NaN values

ascending
[bool, default True] Whether or not the elements should be ranked in ascending order.

pct
[bool, default False] Whether or not to display the returned rankings in percentile form.

Returns

same type as caller
Return a Series or DataFrame with data ranks as values.

See also:

`core.groupby.DataFrameGroupBy.rank`
Rank of values within each group.

`core.groupby.SeriesGroupBy.rank`
Rank of values within each group.

Examples

```
>>> df = pd.DataFrame(data={'Animal': ['cat', 'penguin', 'dog',
...                                    'spider', 'snake'],
...                         'Number_legs': [4, 2, 4, 8, np.nan]})
>>> df
    Animal  Number_legs
0      cat          4.0
1  penguin          2.0
2      dog          4.0
3   spider          8.0
4    snake          NaN
```

Ties are assigned the mean of the ranks (by default) for the group.

```
>>> s = pd.Series(range(5), index=list("abcde"))
>>> s["d"] = s["b"]
>>> s.rank()
a    1.0
b    2.5
c    4.0
d    2.5
e    5.0
dtype: float64
```

The following example shows how the method behaves with the above parameters:

- default_rank: this is the default behaviour obtained without using any parameter.
- max_rank: setting `method = 'max'` the records that have the same values are ranked using the highest rank (e.g.: since 'cat' and 'dog' are both in the 2nd and 3rd position, rank 3 is assigned.)
- NA_bottom: choosing `na_option = 'bottom'`, if there are records with NaN values they are placed at the bottom of the ranking.
- pct_rank: when setting `pct = True`, the ranking is expressed as percentile rank.

```
>>> df['default_rank'] = df['Number_legs'].rank()
>>> df['max_rank'] = df['Number_legs'].rank(method='max')
>>> df['NA_bottom'] = df['Number_legs'].rank(na_option='bottom')
>>> df['pct_rank'] = df['Number_legs'].rank(pct=True)
>>> df
    Animal  Number_legs  default_rank  max_rank  NA_bottom  pct_rank
0      cat          4.0           2.5       3.0        2.5     0.625
1  penguin          2.0           1.0       1.0        1.0     0.250
2      dog          4.0           2.5       3.0        2.5     0.625
3   spider          8.0           4.0       4.0        4.0     1.000
4    snake          NaN           NaN       NaN        5.0       NaN
```

AlloViz.AlloViz.Elements.Edges.rdiv

Edges.**rdiv**(*other*, *axis: Axis = 'columns'*, *level=None*, *fill_value=None*)

Get Floating division of dataframe and other, element-wise (binary operator *rtruediv*).

Equivalent to `other / dataframe`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *truediv*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *floordiv*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other
[scalar, sequence, Series, dict or DataFrame] Any single or multiple element data structure, or list-like object.

axis
[{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns. (1 or 'columns'). For Series input, axis to match Series index on.

level
[int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value
[float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame
Result of the arithmetic operation.

See also:

`DataFrame.add`
Add DataFrames.

`DataFrame.sub`
Subtract DataFrames.

`DataFrame.mul`
Multiply DataFrames.

`DataFrame.div`
Divide DataFrames (float division).

`DataFrame.truediv`
Divide DataFrames (float division).

`DataFrame.floordiv`
Divide DataFrames (integer division).

`DataFrame.mod`
Calculate modulo (remainder after division).

`DataFrame.pow`
Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...         axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a dictionary by axis.

```
>>> df.mul({'angles': 0, 'degrees': 2})
            angles  degrees
circle           0      720
triangle         0      360
rectangle        0      720
```

```
>>> df.mul({'circle': 0, 'triangle': 2, 'rectangle': 3}, axis='index')
            angles  degrees
circle           0        0
triangle         6      360
rectangle       12     1080
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

AlloViz.AlloViz.Elements.Edges.reindex

Edges.**reindex**(*labels=None*, ***, *index=None*, *columns=None*, *axis: Axis | None = None*, *method: ReindexMethod | None = None*, *copy: bool | None = None*, *level: Level | None = None*, *fill_value: Scalar | None = nan*, *limit: int | None = None*, *tolerance=None*) → DataFrame

Conform DataFrame to new index with optional filling logic.

Places NA/NaN in locations having no value in the previous index. A new object is produced unless the new index is equivalent to the current one and `copy=False`.

Parameters

labels
[array-like, optional] New labels / index to conform the axis specified by 'axis' to.

index
[array-like, optional] New labels for the index. Preferably an Index object to avoid duplicating data.

columns
[array-like, optional] New labels for the columns. Preferably an Index object to avoid duplicating data.

axis
[int or str, optional] Axis to target. Can be either the axis name ('index', 'columns') or number (0, 1).

method
[{None, 'backfill'/'bfill', 'pad'/'ffill', 'nearest'}] Method to use for filling holes in reindexed DataFrame. Please note: this is only applicable to DataFrames/Series with a monotonically increasing/decreasing index.

- None (default): don't fill gaps
- pad / ffill: Propagate last valid observation forward to next valid.
- backfill / bfill: Use next valid observation to fill gap.
- nearest: Use nearest valid observations to fill gap.

copy
[bool, default True] Return a new object, even if the passed indexes are the same.

level
[int or name] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value
[scalar, default np.nan] Value to use for missing values. Defaults to NaN, but can be any "compatible" value.

limit
[int, default None] Maximum number of consecutive elements to forward or backward fill.

tolerance
[optional] Maximum distance between original and new labels for inexact matches. The values of the index at the matching locations most satisfy the equation `abs(index[indexer] - target) <= tolerance`.

Tolerance may be a scalar value, which applies the same tolerance to all values, or list-like, which applies variable tolerance per element. List-like includes list, tuple, array, Series, and must be the same size as the index and its dtype must exactly match the index's type.

Returns

DataFrame with changed index.

See also:

`DataFrame.set_index`
Set row labels.

`DataFrame.reset_index`
Remove row labels or move them to new columns.

`DataFrame.reindex_like`
Change to same indices as other DataFrame.

Examples

`DataFrame.reindex` supports two calling conventions

- `(index=index_labels, columns=column_labels, ...)`
- `(labels, axis={'index', 'columns'}, ...)`

We *highly* recommend using keyword arguments to clarify your intent.

Create a dataframe with some fictional data.

```
>>> index = ['Firefox', 'Chrome', 'Safari', 'IE10', 'Konqueror']
>>> df = pd.DataFrame({'http_status': [200, 200, 404, 404, 301],
...                    'response_time': [0.04, 0.02, 0.07, 0.08, 1.0]},
...                    index=index)
>>> df
           http_status  response_time
Firefox            200           0.04
Chrome             200           0.02
Safari             404           0.07
IE10               404           0.08
Konqueror          301           1.00
```

Create a new index and reindex the dataframe. By default values in the new index that do not have corresponding records in the dataframe are assigned `NaN`.

```
>>> new_index = ['Safari', 'Iceweasel', 'Comodo Dragon', 'IE10',
...              'Chrome']
>>> df.reindex(new_index)
               http_status  response_time
Safari               404.0           0.07
Iceweasel              NaN            NaN
Comodo Dragon          NaN            NaN
IE10                 404.0           0.08
Chrome               200.0           0.02
```

We can fill in the missing values by passing a value to the keyword `fill_value`. Because the index is not monotonically increasing or decreasing, we cannot use arguments to the keyword `method` to fill the `NaN` values.

```
>>> df.reindex(new_index, fill_value=0)
               http_status  response_time
Safari                 404           0.07
Iceweasel                0           0.00
Comodo Dragon            0           0.00
IE10                   404           0.08
Chrome                 200           0.02
```

```
>>> df.reindex(new_index, fill_value='missing')
              http_status response_time
Safari                404          0.07
Iceweasel         missing       missing
Comodo Dragon     missing       missing
IE10                  404          0.08
Chrome                200          0.02
```

We can also reindex the columns.

```
>>> df.reindex(columns=['http_status', 'user_agent'])
           http_status  user_agent
Firefox            200         NaN
Chrome             200         NaN
Safari             404         NaN
IE10               404         NaN
```

(continues on next page)

(continued from previous page)

```
Konqueror          301         NaN
```

Or we can use "axis-style" keyword arguments

```
>>> df.reindex(['http_status', 'user_agent'], axis="columns")
           http_status  user_agent
Firefox            200         NaN
Chrome             200         NaN
Safari             404         NaN
IE10               404         NaN
Konqueror          301         NaN
```

To further illustrate the filling functionality in `reindex`, we will create a dataframe with a monotonically increasing index (for example, a sequence of dates).

```
>>> date_index = pd.date_range('1/1/2010', periods=6, freq='D')
>>> df2 = pd.DataFrame({"prices": [100, 101, np.nan, 100, 89, 88]},
...                    index=date_index)
>>> df2
            prices
2010-01-01   100.0
2010-01-02   101.0
2010-01-03     NaN
2010-01-04   100.0
2010-01-05    89.0
2010-01-06    88.0
```

Suppose we decide to expand the dataframe to cover a wider date range.

```
>>> date_index2 = pd.date_range('12/29/2009', periods=10, freq='D')
>>> df2.reindex(date_index2)
            prices
2009-12-29     NaN
2009-12-30     NaN
2009-12-31     NaN
2010-01-01   100.0
2010-01-02   101.0
2010-01-03     NaN
2010-01-04   100.0
2010-01-05    89.0
2010-01-06    88.0
2010-01-07     NaN
```

The index entries that did not have a value in the original data frame (for example, '2009-12-29') are by default filled with `NaN`. If desired, we can fill in the missing values using one of several options.

For example, to back-propagate the last valid value to fill the `NaN` values, pass `bfill` as an argument to the `method` keyword.

```
>>> df2.reindex(date_index2, method='bfill')
            prices
2009-12-29   100.0
2009-12-30   100.0
```

(continues on next page)

(continued from previous page)

```
2009-12-31   100.0
2010-01-01   100.0
2010-01-02   101.0
2010-01-03     NaN
2010-01-04   100.0
2010-01-05    89.0
2010-01-06    88.0
2010-01-07     NaN
```

Please note that the `NaN` value present in the original dataframe (at index value 2010-01-03) will not be filled by any of the value propagation schemes. This is because filling while reindexing does not look at dataframe values, but only compares the original and desired indexes. If you do want to fill in the `NaN` values present in the original dataframe, use the `fillna()` method.

See the user guide for more.

AlloViz.AlloViz.Elements.Edges.reindex_like

`Edges.reindex_like`(*other*, *method: Literal['backfill', 'bfill', 'pad', 'ffill', 'nearest'] | None = None*, *copy: bool | None = None*, *limit: None | int = None*, *tolerance=None*) → None

Return an object with matching indices as other object.

Conform the object to the same index on all axes. Optional filling logic, placing NaN in locations having no value in the previous index. A new object is produced unless the new index is equivalent to the current one and copy=False.

Parameters

other
[Object of the same data type] Its row and column indices are used to define the new indices of this object.

method
[{None, 'backfill'/'bfill', 'pad'/'ffill', 'nearest'}] Method to use for filling holes in reindexed DataFrame. Please note: this is only applicable to DataFrames/Series with a monotonically increasing/decreasing index.

- None (default): don't fill gaps
- pad / ffill: propagate last valid observation forward to next valid
- backfill / bfill: use next valid observation to fill gap
- nearest: use nearest valid observations to fill gap.

copy
[bool, default True] Return a new object, even if the passed indexes are the same.

limit
[int, default None] Maximum number of consecutive labels to fill for inexact matches.

tolerance
[optional] Maximum distance between original and new labels for inexact matches. The values of the index at the matching locations must satisfy the equation `abs(index[indexer] - target) <= tolerance`.

Tolerance may be a scalar value, which applies the same tolerance to all values, or list-like, which applies variable tolerance per element. List-like includes list, tuple,

array, Series, and must be the same size as the index and its dtype must exactly match the index's type.

Returns

Series or DataFrame
Same type as caller, but with changed indices on each axis.

See also:

DataFrame.set_index
Set row labels.

DataFrame.reset_index
Remove row labels or move them to new columns.

DataFrame.reindex
Change to new indices or expand indices.

Notes

Same as calling `.reindex(index=other.index, columns=other.columns,...)`.

Examples

```
>>> df1 = pd.DataFrame([[24.3, 75.7, 'high'],
...                     [31, 87.8, 'high'],
...                     [22, 71.6, 'medium'],
...                     [35, 95, 'medium']],
...                    columns=['temp_celsius', 'temp_fahrenheit',
...                             'windspeed'],
...                    index=pd.date_range(start='2014-02-12',
...                                        end='2014-02-15', freq='D'))
```

```
>>> df1
            temp_celsius  temp_fahrenheit windspeed
2014-02-12          24.3             75.7      high
2014-02-13          31.0             87.8      high
2014-02-14          22.0             71.6    medium
2014-02-15          35.0             95.0    medium
```

```
>>> df2 = pd.DataFrame([[28, 'low'],
...                     [30, 'low'],
...                     [35.1, 'medium']],
...                    columns=['temp_celsius', 'windspeed'],
...                    index=pd.DatetimeIndex(['2014-02-12', '2014-02-13',
...                                            '2014-02-15']))
```

```
>>> df2
            temp_celsius windspeed
2014-02-12          28.0       low
2014-02-13          30.0       low
2014-02-15          35.1    medium
```

```
>>> df2.reindex_like(df1)
            temp_celsius  temp_fahrenheit windspeed
2014-02-12          28.0              NaN       low
2014-02-13          30.0              NaN       low
2014-02-14           NaN              NaN       NaN
2014-02-15          35.1              NaN    medium
```

AlloViz.AlloViz.Elements.Edges.rename

Edges.**rename**(*mapper: Renamer | None = None*, *, *index: Renamer | None = None*, *columns: Renamer | None = None*, *axis: Axis | None = None*, *copy: bool | None = None*, *inplace: bool = False*, *level: Level | None = None*, *errors: IgnoreRaise = 'ignore'*) → DataFrame | None

Rename columns or index labels.

Function / dict values must be unique (1-to-1). Labels not contained in a dict / Series will be left as-is. Extra labels listed don't throw an error.

See the user guide for more.

Parameters

mapper
[dict-like or function] Dict-like or function transformations to apply to that axis' values. Use either `mapper` and `axis` to specify the axis to target with `mapper`, or `index` and `columns`.

index
[dict-like or function] Alternative to specifying axis (`mapper, axis=0` is equivalent to `index=mapper`).

columns
[dict-like or function] Alternative to specifying axis (`mapper, axis=1` is equivalent to `columns=mapper`).

axis
[{0 or 'index', 1 or 'columns'}, default 0] Axis to target with `mapper`. Can be either the axis name ('index', 'columns') or number (0, 1). The default is 'index'.

copy
[bool, default True] Also copy underlying data.

inplace
[bool, default False] Whether to modify the DataFrame rather than creating a new one. If True then value of copy is ignored.

level
[int or level name, default None] In case of a MultiIndex, only rename labels in the specified level.

errors
[{'ignore', 'raise'}, default 'ignore'] If 'raise', raise a *KeyError* when a dict-like *mapper*, *index*, or *columns* contains labels that are not present in the Index being transformed. If 'ignore', existing keys will be renamed and extra keys will be ignored.

Returns

DataFrame or None
DataFrame with the renamed axis labels or None if `inplace=True`.

Raises

KeyError
If any of the labels is not found in the selected axis and "errors='raise'".

See also:

`DataFrame.rename_axis`
Set the name of the axis.

Examples

`DataFrame.rename` supports two calling conventions

- `(index=index_mapper, columns=columns_mapper, ...)`
- `(mapper, axis={'index', 'columns'}, ...)`

We *highly* recommend using keyword arguments to clarify your intent.

Rename columns using a mapping:

```
>>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
>>> df.rename(columns={"A": "a", "B": "c"})
   a  c
0  1  4
1  2  5
2  3  6
```

Rename index using a mapping:

```
>>> df.rename(index={0: "x", 1: "y", 2: "z"})
   A  B
x  1  4
y  2  5
z  3  6
```

Cast index labels to a different type:

```
>>> df.index
RangeIndex(start=0, stop=3, step=1)
>>> df.rename(index=str).index
Index(['0', '1', '2'], dtype='object')
```

```
>>> df.rename(columns={"A": "a", "B": "b", "C": "c"}, errors="raise")
Traceback (most recent call last):
KeyError: ['C'] not found in axis
```

Using axis-style parameters:

```
>>> df.rename(str.lower, axis='columns')
   a  b
0  1  4
1  2  5
2  3  6
```

```
>>> df.rename({1: 2, 2: 4}, axis='index')
   A  B
0  1  4
2  2  5
4  3  6
```

AlloViz.AlloViz.Elements.Edges.rename_axis

Edges.**rename_axis**(*mapper: IndexLabel | lib.NoDefault = _NoDefault.no_default*, ***, *index=_NoDefault.no_default*, *columns=_NoDefault.no_default*, *axis: Axis = 0*, *copy: bool_t | None = None*, *inplace: bool_t = False*) → Self | None

Set the name of the axis for the index or columns.

Parameters

mapper
[scalar, list-like, optional] Value to set the axis name attribute.

index, columns
[scalar, list-like, dict-like or function, optional] A scalar, list-like, dict-like or functions transformations to apply to that axis' values. Note that the `columns` parameter is not allowed if the object is a Series. This parameter only apply for DataFrame type objects.

Use either `mapper` and `axis` to specify the axis to target with `mapper`, or `index` and/or `columns`.

axis
[{0 or 'index', 1 or 'columns'}, default 0] The axis to rename. For *Series* this parameter is unused and defaults to 0.

copy
[bool, default None] Also copy underlying data.

inplace
[bool, default False] Modifies the object directly, instead of creating a new Series or DataFrame.

Returns

Series, DataFrame, or None
The same type as the caller or None if `inplace=True`.

See also:

`Series.rename`
Alter Series index labels or name.

`DataFrame.rename`
Alter DataFrame index labels or name.

`Index.rename`
Set new names on index.

Notes

`DataFrame.rename_axis` supports two calling conventions

- `(index=index_mapper, columns=columns_mapper, ...)`
- `(mapper, axis={'index', 'columns'}, ...)`

The first calling convention will only modify the names of the index and/or the names of the Index object that is the columns. In this case, the parameter `copy` is ignored.

The second calling convention will modify the names of the corresponding index if mapper is a list or a scalar. However, if mapper is dict-like or a function, it will use the deprecated behavior of modifying the axis *labels*.

We *highly* recommend using keyword arguments to clarify your intent.

Examples

Series

```
>>> s = pd.Series(["dog", "cat", "monkey"])
>>> s
0       dog
1       cat
2    monkey
dtype: object
>>> s.rename_axis("animal")
animal
0    dog
1    cat
2    monkey
dtype: object
```

DataFrame

```
>>> df = pd.DataFrame({"num_legs": [4, 4, 2],
...                    "num_arms": [0, 0, 2]},
...                   ["dog", "cat", "monkey"])
>>> df
        num_legs  num_arms
dog            4         0
cat            4         0
monkey         2         2
>>> df = df.rename_axis("animal")
>>> df
        num_legs  num_arms
animal
dog            4         0
cat            4         0
monkey         2         2
>>> df = df.rename_axis("limbs", axis="columns")
>>> df
limbs   num_legs  num_arms
animal
```

(continues on next page)

(continued from previous page)

```
dog            4         0
cat            4         0
monkey         2         2
```

MultiIndex

```
>>> df.index = pd.MultiIndex.from_product([['mammal'],
...                                        ['dog', 'cat', 'monkey']],
...                                       names=['type', 'name'])
>>> df
limbs          num_legs  num_arms
type   name
mammal dog            4         0
       cat            4         0
       monkey         2         2
```

```
>>> df.rename_axis(index={'type': 'class'})
limbs          num_legs  num_arms
class  name
mammal dog            4         0
       cat            4         0
       monkey         2         2
```

```
>>> df.rename_axis(columns=str.upper)
LIMBS          num_legs  num_arms
type   name
mammal dog            4         0
       cat            4         0
       monkey         2         2
```

AlloViz.AlloViz.Elements.Edges.reorder_levels

Edges.**reorder_levels**(*order: Sequence[int | str]*, *axis: Axis = 0*) → DataFrame

Rearrange index levels using input order. May not drop or duplicate levels.

Parameters

order
[list of int or list of str] List representing new level order. Reference level by number (position) or by key (label).

axis
[{0 or 'index', 1 or 'columns'}, default 0] Where to reorder levels.

Returns

DataFrame

Examples

```
>>> data = {
...     "class": ["Mammals", "Mammals", "Reptiles"],
...     "diet": ["Omnivore", "Carnivore", "Carnivore"],
...     "species": ["Humans", "Dogs", "Snakes"],
... }
>>> df = pd.DataFrame(data, columns=["class", "diet", "species"])
>>> df = df.set_index(["class", "diet"])
>>> df
                                  species
class      diet
Mammals    Omnivore                Humans
           Carnivore                 Dogs
Reptiles   Carnivore               Snakes
```

Let's reorder the levels of the index:

```
>>> df.reorder_levels(["diet", "class"])
                                  species
diet      class
Omnivore  Mammals                 Humans
Carnivore Mammals                   Dogs
          Reptiles                Snakes
```

AlloViz.AlloViz.Elements.Edges.replace

Edges.**replace**(*to_replace=None*, *value=_NoDefault.no_default*, ***, *inplace: bool_t = False*, *limit: int | None = None*, *regex: bool_t = False*, *method: Literal['pad', 'ffill', 'bfill'] | lib.NoDefault = _NoDefault.no_default*) → Self | None

Replace values given in *to_replace* with *value*.

Values of the Series/DataFrame are replaced with other values dynamically. This differs from updating with `.loc` or `.iloc`, which require you to specify a location to update with some value.

Parameters

to_replace
[str, regex, list, dict, Series, int, float, or None] How to find the values that will be replaced.

- numeric, str or regex:
  - numeric: numeric values equal to *to_replace* will be replaced with *value*
  - str: string exactly matching *to_replace* will be replaced with *value*
  - regex: regexs matching *to_replace* will be replaced with *value*
- list of str, regex, or numeric:
  - First, if *to_replace* and *value* are both lists, they **must** be the same length.
  - Second, if `regex=True` then all of the strings in **both** lists will be interpreted as regexs otherwise they will match directly. This doesn't matter much for *value* since there are only a few possible substitution regexes you can use.
  - str, regex and numeric rules apply as above.

- dict:
  - Dicts can be used to specify different replacement values for different existing values. For example, `{'a': 'b', 'y': 'z'}` replaces the value 'a' with 'b' and 'y' with 'z'. To use a dict in this way, the optional *value* parameter should not be given.
  - For a DataFrame a dict can specify that different values should be replaced in different columns. For example, `{'a': 1, 'b': 'z'}` looks for the value 1 in column 'a' and the value 'z' in column 'b' and replaces these values with whatever is specified in *value*. The *value* parameter should not be `None` in this case. You can treat this as a special case of passing two lists except that you are specifying the column to search in.
  - For a DataFrame nested dictionaries, e.g., `{'a': {'b': np.nan}}`, are read as follows: look in column 'a' for the value 'b' and replace it with NaN. The optional *value* parameter should not be specified to use a nested dict in this way. You can nest regular expressions as well. Note that column names (the top-level dictionary keys in a nested dictionary) **cannot** be regular expressions.
- None:
  - This means that the *regex* argument must be a string, compiled regular expression, or list, dict, ndarray or Series of such elements. If *value* is also `None` then this **must** be a nested dictionary or Series.

See the examples section for examples of each of these.

value
[scalar, dict, list, str, regex, default None] Value to replace any values matching *to_replace* with. For a DataFrame a dict of values can be used to specify which value to use for each column (columns not in the dict will not be filled). Regular expressions, strings and lists or dicts of such objects are also allowed.

inplace
[bool, default False] If True, performs operation inplace and returns None.

limit
[int, default None] Maximum size gap to forward or backward fill.

Deprecated since version 2.1.0.

regex
[bool or same types as *to_replace*, default False] Whether to interpret *to_replace* and/or *value* as regular expressions. If this is `True` then *to_replace must* be a string. Alternatively, this could be a regular expression or a list, dict, or array of regular expressions in which case *to_replace* must be `None`.

method
[{'pad', 'ffill', 'bfill'}] The method to use when for replacement, when *to_replace* is a scalar, list or tuple and *value* is `None`.

Deprecated since version 2.1.0.

Returns

Series/DataFrame
Object after replacement.

Raises

AssertionError

- If *regex* is not a `bool` and *to_replace* is not `None`.

TypeError

- If *to_replace* is not a scalar, array-like, `dict`, or `None`
- If *to_replace* is a `dict` and *value* is not a `list`, `dict`, `ndarray`, or `Series`
- If *to_replace* is `None` and *regex* is not compilable into a regular expression or is a list, dict, ndarray, or Series.
- When replacing multiple `bool` or `datetime64` objects and the arguments to *to_replace* does not match the type of the value being replaced

ValueError

- If a `list` or an `ndarray` is passed to *to_replace* and *value* but they are not the same length.

See also:

`Series.fillna`
Fill NA values.

`DataFrame.fillna`
Fill NA values.

`Series.where`
Replace values based on boolean condition.

`DataFrame.where`
Replace values based on boolean condition.

`DataFrame.map`
Apply a function to a Dataframe elementwise.

`Series.map`
Map values of Series according to an input mapping or function.

`Series.str.replace`
Simple string replacement.

Notes

- Regex substitution is performed under the hood with `re.sub`. The rules for substitution for `re.sub` are the same.
- Regular expressions will only substitute on strings, meaning you cannot provide, for example, a regular expression matching floating point numbers and expect the columns in your frame that have a numeric dtype to be matched. However, if those floating point numbers *are* strings, then you can do this.
- This method has *a lot* of options. You are encouraged to experiment and play with this method to gain intuition about how it works.
- When dict is used as the *to_replace* value, it is like key(s) in the dict are the to_replace part and value(s) in the dict are the value parameter.

Examples

Scalar `to_replace` and `value`

```
>>> s = pd.Series([1, 2, 3, 4, 5])
>>> s.replace(1, 5)
0    5
1    2
2    3
3    4
4    5
dtype: int64
```

```
>>> df = pd.DataFrame({'A': [0, 1, 2, 3, 4],
...                    'B': [5, 6, 7, 8, 9],
...                    'C': ['a', 'b', 'c', 'd', 'e']})
>>> df.replace(0, 5)
    A  B  C
0  5  5  a
1  1  6  b
2  2  7  c
3  3  8  d
4  4  9  e
```

List-like `to_replace`

```
>>> df.replace([0, 1, 2, 3], 4)
    A  B  C
0  4  5  a
1  4  6  b
2  4  7  c
3  4  8  d
4  4  9  e
```

```
>>> df.replace([0, 1, 2, 3], [4, 3, 2, 1])
    A  B  C
0  4  5  a
1  3  6  b
2  2  7  c
3  1  8  d
4  4  9  e
```

```
>>> s.replace([1, 2], method='bfill')
0    3
1    3
2    3
3    4
4    5
dtype: int64
```

dict-like `to_replace`

```
>>> df.replace({0: 10, 1: 100})
        A  B  C
0   10  5  a
1  100  6  b
2    2  7  c
3    3  8  d
4    4  9  e
```

```
>>> df.replace({'A': 0, 'B': 5}, 100)
        A    B  C
0  100  100  a
1    1    6  b
2    2    7  c
3    3    8  d
4    4    9  e
```

```
>>> df.replace({'A': {0: 100, 4: 400}})
        A  B  C
0  100  5  a
1    1  6  b
2    2  7  c
3    3  8  d
4  400  9  e
```

Regular expression `to_replace`

```
>>> df = pd.DataFrame({'A': ['bat', 'foo', 'bait'],
...                    'B': ['abc', 'bar', 'xyz']})
>>> df.replace(to_replace=r'^ba.$', value='new', regex=True)
        A    B
0   new  abc
1   foo  new
2  bait  xyz
```

```
>>> df.replace({'A': r'^ba.$'}, {'A': 'new'}, regex=True)
        A    B
0   new  abc
1   foo  bar
2  bait  xyz
```

```
>>> df.replace(regex=r'^ba.$', value='new')
        A    B
0   new  abc
1   foo  new
2  bait  xyz
```

```
>>> df.replace(regex={r'^ba.$': 'new', 'foo': 'xyz'})
        A    B
0   new  abc
1   xyz  new
2  bait  xyz
```

```
>>> df.replace(regex=[r'^ba.$', 'foo'], value='new')
        A    B
0   new  abc
1   new  new
2  bait  xyz
```

Compare the behavior of `s.replace({'a': None})` and `s.replace('a', None)` to understand the peculiarities of the *to_replace* parameter:

```
>>> s = pd.Series([10, 'a', 'a', 'b', 'a'])
```

When one uses a dict as the *to_replace* value, it is like the value(s) in the dict are equal to the *value* parameter. `s.replace({'a': None})` is equivalent to `s.replace(to_replace={'a': None}, value=None, method=None)`:

```
>>> s.replace({'a': None})
0      10
1    None
2    None
3       b
4    None
dtype: object
```

When `value` is not explicitly passed and *to_replace* is a scalar, list or tuple, *replace* uses the method parameter (default 'pad') to do the replacement. So this is why the 'a' values are being replaced by 10 in rows 1 and 2 and 'b' in row 4 in this case.

```
>>> s.replace('a')
0    10
1    10
2    10
3     b
4     b
dtype: object
```

Deprecated since version 2.1.0: The 'method' parameter and padding behavior are deprecated.

On the other hand, if `None` is explicitly passed for `value`, it will be respected:

```
>>> s.replace('a', None)
0      10
1    None
2    None
3       b
4    None
dtype: object
```

Changed in version 1.4.0: Previously the explicit `None` was silently ignored.

AlloViz.AlloViz.Elements.Edges.resample

Edges.**resample**(*rule*, *axis: Axis | lib.NoDefault = _NoDefault.no_default*, *closed: Literal['right', 'left'] | None = None*, *label: Literal['right', 'left'] | None = None*, *convention: Literal['start', 'end', 's', 'e'] = 'start'*, *kind: Literal['timestamp', 'period'] | None = None*, *on: Level | None = None*, *level: Level | None = None*, *origin: str | TimestampConvertibleTypes = 'start_day'*, *offset: TimedeltaConvertibleTypes | None = None*, *group_keys: bool_t = False*) → Resampler

Resample time-series data.

Convenience method for frequency conversion and resampling of time series. The object must have a datetime-like index (*DatetimeIndex*, *PeriodIndex*, or *TimedeltaIndex*), or the caller must pass the label of a datetime-like series/index to the `on`/`level` keyword parameter.

Parameters

rule
[DateOffset, Timedelta or str] The offset string or object representing target conversion.

axis
[{0 or 'index', 1 or 'columns'}, default 0] Which axis to use for up- or down-sampling. For *Series* this parameter is unused and defaults to 0. Must be *DatetimeIndex*, *TimedeltaIndex* or *PeriodIndex*.

Deprecated since version 2.0.0: Use frame.T.resample(...) instead.

closed
[{'right', 'left'}, default None] Which side of bin interval is closed. The default is 'left' for all frequency offsets except for 'M', 'A', 'Q', 'BM', 'BA', 'BQ', and 'W' which all have a default of 'right'.

label
[{'right', 'left'}, default None] Which bin edge label to label bucket with. The default is 'left' for all frequency offsets except for 'M', 'A', 'Q', 'BM', 'BA', 'BQ', and 'W' which all have a default of 'right'.

convention
[{'start', 'end', 's', 'e'}, default 'start'] For *PeriodIndex* only, controls whether to use the start or end of *rule*.

kind
[{'timestamp', 'period'}, optional, default None] Pass 'timestamp' to convert the resulting index to a *DateTimeIndex* or 'period' to convert it to a *PeriodIndex*. By default the input representation is retained.

on
[str, optional] For a DataFrame, column to use instead of index for resampling. Column must be datetime-like.

level
[str or int, optional] For a MultiIndex, level (name or number) to use for resampling. *level* must be datetime-like.

origin
[Timestamp or str, default 'start_day'] The timestamp on which to adjust the grouping. The timezone of origin must match the timezone of the index. If string, must be one of the following:

- 'epoch': *origin* is 1970-01-01

- 'start': *origin* is the first value of the timeseries
- 'start_day': *origin* is the first day at midnight of the timeseries
- 'end': *origin* is the last value of the timeseries
- 'end_day': *origin* is the ceiling midnight of the last day

New in version 1.3.0.

offset
[Timedelta or str, default is None] An offset timedelta added to the origin.

group_keys
[bool, default False] Whether to include the group keys in the result index when using `.apply()` on the resampled object.

New in version 1.5.0: Not specifying `group_keys` will retain values-dependent behavior from pandas 1.4 and earlier (see pandas 1.5.0 Release notes for examples).

Changed in version 2.0.0: `group_keys` now defaults to `False`.

Returns

pandas.api.typing.Resampler
`Resampler` object.

See also:

`Series.resample`
Resample a Series.

`DataFrame.resample`
Resample a DataFrame.

`groupby`
Group Series/DataFrame by mapping, function, label, or list of labels.

`asfreq`
Reindex a Series/DataFrame with the given frequency without grouping.

Notes

See the user guide for more.

To learn more about the offset strings, please see this link.

Examples

Start by creating a series with 9 one minute timestamps.

```
>>> index = pd.date_range('1/1/2000', periods=9, freq='T')
>>> series = pd.Series(range(9), index=index)
>>> series
2000-01-01 00:00:00    0
2000-01-01 00:01:00    1
2000-01-01 00:02:00    2
2000-01-01 00:03:00    3
2000-01-01 00:04:00    4
```

(continues on next page)

(continued from previous page)

```
2000-01-01 00:05:00    5
2000-01-01 00:06:00    6
2000-01-01 00:07:00    7
2000-01-01 00:08:00    8
Freq: T, dtype: int64
```

Downsample the series into 3 minute bins and sum the values of the timestamps falling into a bin.

```
>>> series.resample('3T').sum()
2000-01-01 00:00:00     3
2000-01-01 00:03:00    12
2000-01-01 00:06:00    21
Freq: 3T, dtype: int64
```

Downsample the series into 3 minute bins as above, but label each bin using the right edge instead of the left. Please note that the value in the bucket used as the label is not included in the bucket, which it labels. For example, in the original series the bucket `2000-01-01 00:03:00` contains the value 3, but the summed value in the resampled bucket with the label `2000-01-01 00:03:00` does not include 3 (if it did, the summed value would be 6, not 3). To include this value close the right side of the bin interval as illustrated in the example below this one.

```
>>> series.resample('3T', label='right').sum()
2000-01-01 00:03:00     3
2000-01-01 00:06:00    12
2000-01-01 00:09:00    21
Freq: 3T, dtype: int64
```

Downsample the series into 3 minute bins as above, but close the right side of the bin interval.

```
>>> series.resample('3T', label='right', closed='right').sum()
2000-01-01 00:00:00     0
2000-01-01 00:03:00     6
2000-01-01 00:06:00    15
2000-01-01 00:09:00    15
Freq: 3T, dtype: int64
```

Upsample the series into 30 second bins.

```
>>> series.resample('30S').asfreq()[0:5]   # Select first 5 rows
2000-01-01 00:00:00   0.0
2000-01-01 00:00:30   NaN
2000-01-01 00:01:00   1.0
2000-01-01 00:01:30   NaN
2000-01-01 00:02:00   2.0
Freq: 30S, dtype: float64
```

Upsample the series into 30 second bins and fill the `NaN` values using the `ffill` method.

```
>>> series.resample('30S').ffill()[0:5]
2000-01-01 00:00:00    0
2000-01-01 00:00:30    0
2000-01-01 00:01:00    1
2000-01-01 00:01:30    1
```

(continues on next page)

(continued from previous page)

```
2000-01-01 00:02:00    2
Freq: 30S, dtype: int64
```

Upsample the series into 30 second bins and fill the `NaN` values using the `bfill` method.

```
>>> series.resample('30S').bfill()[0:5]
2000-01-01 00:00:00    0
2000-01-01 00:00:30    1
2000-01-01 00:01:00    1
2000-01-01 00:01:30    2
2000-01-01 00:02:00    2
Freq: 30S, dtype: int64
```

Pass a custom function via `apply`

```
>>> def custom_resampler(arraylike):
...     return np.sum(arraylike) + 5
...
>>> series.resample('3T').apply(custom_resampler)
2000-01-01 00:00:00     8
2000-01-01 00:03:00    17
2000-01-01 00:06:00    26
Freq: 3T, dtype: int64
```

For a Series with a PeriodIndex, the keyword *convention* can be used to control whether to use the start or end of *rule*.

Resample a year by quarter using 'start' *convention*. Values are assigned to the first quarter of the period.

```
>>> s = pd.Series([1, 2], index=pd.period_range('2012-01-01',
...                                             freq='A',
...                                             periods=2))
>>> s
2012    1
2013    2
Freq: A-DEC, dtype: int64
>>> s.resample('Q', convention='start').asfreq()
2012Q1    1.0
2012Q2    NaN
2012Q3    NaN
2012Q4    NaN
2013Q1    2.0
2013Q2    NaN
2013Q3    NaN
2013Q4    NaN
Freq: Q-DEC, dtype: float64
```

Resample quarters by month using 'end' *convention*. Values are assigned to the last month of the period.

```
>>> q = pd.Series([1, 2, 3, 4], index=pd.period_range('2018-01-01',
...                                                   freq='Q',
...                                                   periods=4))
>>> q
```

(continues on next page)

(continued from previous page)

```
2018Q1    1
2018Q2    2
2018Q3    3
2018Q4    4
Freq: Q-DEC, dtype: int64
>>> q.resample('M', convention='end').asfreq()
2018-03    1.0
2018-04    NaN
2018-05    NaN
2018-06    2.0
2018-07    NaN
2018-08    NaN
2018-09    3.0
2018-10    NaN
2018-11    NaN
2018-12    4.0
Freq: M, dtype: float64
```

For DataFrame objects, the keyword *on* can be used to specify the column instead of the index for resampling.

```
>>> d = {'price': [10, 11, 9, 13, 14, 18, 17, 19],
...      'volume': [50, 60, 40, 100, 50, 100, 40, 50]}
>>> df = pd.DataFrame(d)
>>> df['week_starting'] = pd.date_range('01/01/2018',
...                                     periods=8,
...                                     freq='W')
>>> df
   price  volume week_starting
0     10      50    2018-01-07
1     11      60    2018-01-14
2      9      40    2018-01-21
3     13     100    2018-01-28
4     14      50    2018-02-04
5     18     100    2018-02-11
6     17      40    2018-02-18
7     19      50    2018-02-25
>>> df.resample('M', on='week_starting').mean()
               price  volume
week_starting
2018-01-31     10.75    62.5
2018-02-28     17.00    60.0
```

For a DataFrame with MultiIndex, the keyword *level* can be used to specify on which level the resampling needs to take place.

```
>>> days = pd.date_range('1/1/2000', periods=4, freq='D')
>>> d2 = {'price': [10, 11, 9, 13, 14, 18, 17, 19],
...       'volume': [50, 60, 40, 100, 50, 100, 40, 50]}
>>> df2 = pd.DataFrame(
...     d2,
...     index=pd.MultiIndex.from_product(
```

(continues on next page)

(continued from previous page)

```
...         [days, ['morning', 'afternoon']]
...     )
... )
>>> df2
                     price  volume
2000-01-01 morning      10      50
           afternoon    11      60
2000-01-02 morning       9      40
           afternoon    13     100
2000-01-03 morning      14      50
           afternoon    18     100
2000-01-04 morning      17      40
           afternoon    19      50
>>> df2.resample('D', level=0).sum()
            price  volume
2000-01-01     21     110
2000-01-02     22     140
2000-01-03     32     150
2000-01-04     36      90
```

If you want to adjust the start of the bins based on a fixed timestamp:

```
>>> start, end = '2000-10-01 23:30:00', '2000-10-02 00:30:00'
>>> rng = pd.date_range(start, end, freq='7min')
>>> ts = pd.Series(np.arange(len(rng)) * 3, index=rng)
>>> ts
2000-10-01 23:30:00     0
2000-10-01 23:37:00     3
2000-10-01 23:44:00     6
2000-10-01 23:51:00     9
2000-10-01 23:58:00    12
2000-10-02 00:05:00    15
2000-10-02 00:12:00    18
2000-10-02 00:19:00    21
2000-10-02 00:26:00    24
Freq: 7T, dtype: int64
```

```
>>> ts.resample('17min').sum()
2000-10-01 23:14:00     0
2000-10-01 23:31:00     9
2000-10-01 23:48:00    21
2000-10-02 00:05:00    54
2000-10-02 00:22:00    24
Freq: 17T, dtype: int64
```

```
>>> ts.resample('17min', origin='epoch').sum()
2000-10-01 23:18:00     0
2000-10-01 23:35:00    18
2000-10-01 23:52:00    27
2000-10-02 00:09:00    39
2000-10-02 00:26:00    24
Freq: 17T, dtype: int64
```

```
>>> ts.resample('17W', origin='2000-01-01').sum()
2000-01-02      0
2000-04-30      0
2000-08-27      0
2000-12-24    108
Freq: 17W-SUN, dtype: int64
```

If you want to adjust the start of the bins with an *offset* Timedelta, the two following lines are equivalent:

```
>>> ts.resample('17min', origin='start').sum()
2000-10-01 23:30:00     9
2000-10-01 23:47:00    21
2000-10-02 00:04:00    54
2000-10-02 00:21:00    24
Freq: 17T, dtype: int64
```

```
>>> ts.resample('17min', offset='23h30min').sum()
2000-10-01 23:30:00     9
2000-10-01 23:47:00    21
2000-10-02 00:04:00    54
2000-10-02 00:21:00    24
Freq: 17T, dtype: int64
```

If you want to take the largest Timestamp as the end of the bins:

```
>>> ts.resample('17min', origin='end').sum()
2000-10-01 23:35:00     0
2000-10-01 23:52:00    18
2000-10-02 00:09:00    27
2000-10-02 00:26:00    63
Freq: 17T, dtype: int64
```

In contrast with the *start_day*, you can use *end_day* to take the ceiling midnight of the largest Timestamp as the end of the bins and drop the bins not containing data:

```
>>> ts.resample('17min', origin='end_day').sum()
2000-10-01 23:38:00     3
2000-10-01 23:55:00    15
2000-10-02 00:12:00    45
2000-10-02 00:29:00    45
Freq: 17T, dtype: int64
```

AlloViz.AlloViz.Elements.Edges.reset_index

Edges.**reset_index**(*level: IndexLabel | None = None*, ***, *drop: bool = False*, *inplace: bool = False*, *col_level: Hashable = 0*, *col_fill: Hashable = ''*, *allow_duplicates: bool | lib.NoDefault = _NoDefault.no_default*, *names: Hashable | Sequence[Hashable] | None = None*) → DataFrame | None

Reset the index, or a level of it.

Reset the index of the DataFrame, and use the default one instead. If the DataFrame has a MultiIndex, this method can remove one or more levels.

Parameters

level
[int, str, tuple, or list, default None] Only remove the given levels from the index. Removes all levels by default.

drop
[bool, default False] Do not try to insert index into dataframe columns. This resets the index to the default integer index.

inplace
[bool, default False] Whether to modify the DataFrame rather than creating a new one.

col_level
[int or str, default 0] If the columns have multiple levels, determines which level the labels are inserted into. By default it is inserted into the first level.

col_fill
[object, default ‘’] If the columns have multiple levels, determines how the other levels are named. If None then the index name is repeated.

allow_duplicates
[bool, optional, default lib.no_default] Allow duplicate column labels to be created.

New in version 1.5.0.

names
[int, str or 1-dimensional list, default None] Using the given string, rename the DataFrame column which contains the index data. If the DataFrame has a MultiIndex, this has to be a list or tuple with length equal to the number of levels.

New in version 1.5.0.

Returns

DataFrame or None
DataFrame with the new index or None if `inplace=True`.

See also:

`DataFrame.set_index`
Opposite of reset_index.

`DataFrame.reindex`
Change to new indices or expand indices.

`DataFrame.reindex_like`
Change to same indices as other DataFrame.

Examples

```
>>> df = pd.DataFrame([('bird', 389.0),
...                    ('bird', 24.0),
...                    ('mammal', 80.5),
...                    ('mammal', np.nan)],
...                   index=['falcon', 'parrot', 'lion', 'monkey'],
...                   columns=('class', 'max_speed'))
>>> df
         class  max_speed
```

(continues on next page)

(continued from previous page)

```
falcon    bird      389.0
parrot    bird       24.0
lion    mammal       80.5
monkey  mammal        NaN
```

When we reset the index, the old index is added as a column, and a new sequential index is used:

```
>>> df.reset_index()
    index   class  max_speed
0  falcon    bird      389.0
1  parrot    bird       24.0
2    lion  mammal       80.5
3  monkey  mammal        NaN
```

We can use the *drop* parameter to avoid the old index being added as a column:

```
>>> df.reset_index(drop=True)
    class  max_speed
0    bird      389.0
1    bird       24.0
2  mammal       80.5
3  mammal        NaN
```

You can also use *reset_index* with *MultiIndex*.

```
>>> index = pd.MultiIndex.from_tuples([('bird', 'falcon'),
...                                    ('bird', 'parrot'),
...                                    ('mammal', 'lion'),
...                                    ('mammal', 'monkey')],
...                                   names=['class', 'name'])
>>> columns = pd.MultiIndex.from_tuples([('speed', 'max'),
...                                      ('species', 'type')])
>>> df = pd.DataFrame([(389.0, 'fly'),
...                    (24.0, 'fly'),
...                    (80.5, 'run'),
...                    (np.nan, 'jump')],
...                   index=index,
...                   columns=columns)
>>> df
               speed species
                 max    type
class  name
bird   falcon  389.0     fly
       parrot   24.0     fly
mammal lion     80.5     run
       monkey    NaN    jump
```

Using the *names* parameter, choose a name for the index column:

```
>>> df.reset_index(names=['classes', 'names'])
  classes   names  speed species
                     max    type
0    bird  falcon  389.0     fly
```

(continues on next page)

(continued from previous page)

```
1    bird  parrot   24.0     fly
2  mammal    lion   80.5     run
3  mammal  monkey    NaN    jump
```

If the index has multiple levels, we can reset a subset of them:

```
>>> df.reset_index(level='class')
         class  speed species
                  max    type
name
falcon    bird  389.0     fly
parrot    bird   24.0     fly
lion    mammal   80.5     run
monkey  mammal    NaN    jump
```

If we are not dropping the index, by default, it is placed in the top level. We can place it in another level:

```
>>> df.reset_index(level='class', col_level=1)
                speed species
         class    max    type
name
falcon    bird  389.0     fly
parrot    bird   24.0     fly
lion    mammal   80.5     run
monkey  mammal    NaN    jump
```

When the index is inserted under another level, we can specify under which one with the parameter *col_fill*:

```
>>> df.reset_index(level='class', col_level=1, col_fill='species')
              species  speed species
                class    max    type
name
falcon           bird  389.0     fly
parrot           bird   24.0     fly
lion           mammal   80.5     run
monkey         mammal    NaN    jump
```

If we specify a nonexistent level for *col_fill*, it is created:

```
>>> df.reset_index(level='class', col_level=1, col_fill='genus')
                genus  speed species
                class    max    type
name
falcon           bird  389.0     fly
parrot           bird   24.0     fly
lion           mammal   80.5     run
monkey         mammal    NaN    jump
```

AlloViz.AlloViz.Elements.Edges.rfloordiv

Edges.**rfloordiv**(*other*, *axis: Axis = 'columns'*, *level=None*, *fill_value=None*)

Get Integer division of dataframe and other, element-wise (binary operator *rfloordiv*).

Equivalent to `other // dataframe`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *floordiv*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *floordiv*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other
[scalar, sequence, Series, dict or DataFrame] Any single or multiple element data structure, or list-like object.

axis
[{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns. (1 or 'columns'). For Series input, axis to match Series index on.

level
[int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value
[float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame
Result of the arithmetic operation.

See also:

`DataFrame.add`
Add DataFrames.

`DataFrame.sub`
Subtract DataFrames.

`DataFrame.mul`
Multiply DataFrames.

`DataFrame.div`
Divide DataFrames (float division).

`DataFrame.truediv`
Divide DataFrames (float division).

`DataFrame.floordiv`
Divide DataFrames (integer division).

`DataFrame.mod`
Calculate modulo (remainder after division).

`DataFrame.pow`
Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...         axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a dictionary by axis.

```
>>> df.mul({'angles': 0, 'degrees': 2})
            angles  degrees
circle           0      720
triangle         0      360
rectangle        0      720
```

```
>>> df.mul({'circle': 0, 'triangle': 2, 'rectangle': 3}, axis='index')
            angles  degrees
circle           0        0
triangle         6      360
rectangle       12     1080
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

AlloViz.AlloViz.Elements.Edges.rmod

Edges.**rmod**(*other*, *axis: Axis = 'columns'*, *level=None*, *fill_value=None*)

Get Modulo of dataframe and other, element-wise (binary operator *rmod*).

Equivalent to `other % dataframe`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *mod*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *floordiv*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other
: [scalar, sequence, Series, dict or DataFrame] Any single or multiple element data structure, or list-like object.

axis
: [{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns. (1 or 'columns'). For Series input, axis to match Series index on.

level
: [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value
: [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame
Result of the arithmetic operation.

See also:

DataFrame.add
Add DataFrames.

DataFrame.sub
Subtract DataFrames.

DataFrame.mul
Multiply DataFrames.

DataFrame.div
Divide DataFrames (float division).

DataFrame.truediv
Divide DataFrames (float division).

DataFrame.floordiv
Divide DataFrames (integer division).

DataFrame.mod
Calculate modulo (remainder after division).

DataFrame.pow
Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
```

(continues on next page)

(continued from previous page)

```
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a dictionary by axis.

```
>>> df.mul({'angles': 0, 'degrees': 2})
           angles  degrees
circle          0      720
triangle        0      360
rectangle       0      720
```

```
>>> df.mul({'circle': 0, 'triangle': 2, 'rectangle': 3}, axis='index')
           angles  degrees
circle          0        0
triangle        6      360
rectangle      12     1080
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

AlloViz.AlloViz.Elements.Edges.rmul

Edges.**rmul**(*other*, *axis: Axis = 'columns'*, *level=None*, *fill_value=None*)

Get Multiplication of dataframe and other, element-wise (binary operator *rmul*).

Equivalent to `other * dataframe`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *mul*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *floordiv*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other
[scalar, sequence, Series, dict or DataFrame] Any single or multiple element data structure, or list-like object.

axis
[{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns. (1 or 'columns'). For Series input, axis to match Series index on.

level
[int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value
[float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame
Result of the arithmetic operation.

See also:

DataFrame.add
Add DataFrames.

DataFrame.sub
Subtract DataFrames.

DataFrame.mul
Multiply DataFrames.

DataFrame.div
Divide DataFrames (float division).

DataFrame.truediv
Divide DataFrames (float division).

DataFrame.floordiv
Divide DataFrames (integer division).

DataFrame.mod
Calculate modulo (remainder after division).

DataFrame.pow
Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...         axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a dictionary by axis.

```
>>> df.mul({'angles': 0, 'degrees': 2})
            angles  degrees
circle           0      720
triangle         0      360
rectangle        0      720
```

```
>>> df.mul({'circle': 0, 'triangle': 2, 'rectangle': 3}, axis='index')
            angles  degrees
circle           0        0
triangle         6      360
rectangle       12     1080
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

AlloViz.AlloViz.Elements.Edges.rolling

Edges.**rolling**(*window: int | dt.timedelta | str | BaseOffset | BaseIndexer*, *min_periods: int | None = None*, *center: bool_t = False*, *win_type: str | None = None*, *on: str | None = None*, *axis: Axis | lib.NoDefault = _NoDefault.no_default*, *closed: IntervalClosedType | None = None*, *step: int | None = None*, *method: str = 'single'*) → Window | Rolling

Provide rolling window calculations.

Parameters

window

[int, timedelta, str, offset, or BaseIndexer subclass] Size of the moving window.

If an integer, the fixed number of observations used for each window.

If a timedelta, str, or offset, the time period of each window. Each window will be a variable sized based on the observations included in the time-period. This is only valid for datetimelike indexes. To learn more about the offsets & frequency strings, please see this link.

If a BaseIndexer subclass, the window boundaries based on the defined `get_window_bounds` method. Additional rolling keyword arguments, namely `min_periods`, `center`, `closed` and `step` will be passed to `get_window_bounds`.

min_periods

[int, default None] Minimum number of observations in window required to have a value; otherwise, result is `np.nan`.

For a window that is specified by an offset, `min_periods` will default to 1.

For a window that is specified by an integer, `min_periods` will default to the size of the window.

center
[bool, default False] If False, set the window labels as the right edge of the window index.

If True, set the window labels as the center of the window index.

win_type
[str, default None] If `None`, all points are evenly weighted.

If a string, it must be a valid scipy.signal window function.

Certain Scipy window types require additional parameters to be passed in the aggregation function. The additional parameters must match the keywords specified in the Scipy window type method signature.

on
[str, optional] For a DataFrame, a column label or Index level on which to calculate the rolling window, rather than the DataFrame's index.

Provided integer column is ignored and excluded from result since an integer index is not used to calculate the rolling window.

axis
[int or str, default 0] If `0` or `'index'`, roll across the rows.

If `1` or `'columns'`, roll across the columns.

For *Series* this parameter is unused and defaults to 0.

closed
[str, default None] If `'right'`, the first point in the window is excluded from calculations.

If `'left'`, the last point in the window is excluded from calculations.

If `'both'`, the no points in the window are excluded from calculations.

If `'neither'`, the first and last points in the window are excluded from calculations.

Default `None` (`'right'`).

Changed in version 1.2.0: The closed parameter with fixed windows is now supported.

step
[int, default None] New in version 1.5.0.

Evaluate the window at every `step` result, equivalent to slicing as `[::step]`. `window` must be an integer. Using a step argument other than None or 1 will produce a result with a different shape than the input.

method
[str {'single', 'table'}, default 'single'] New in version 1.3.0.

Execute the rolling operation per single column or row (`'single'`) or over the entire object (`'table'`).

This argument is only implemented when specifying `engine='numba'` in the method call.

Returns

pandas.api.typing.Window or pandas.api.typing.Rolling
An instance of Window is returned if `win_type` is passed. Otherwise, an instance of Rolling is returned.

See also:

expanding
Provides expanding transformations.

ewm
Provides exponential weighted functions.

Notes

See Windowing Operations for further usage details and examples.

Examples

```
>>> df = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]})
>>> df
     B
0  0.0
1  1.0
2  2.0
3  NaN
4  4.0
```

window

Rolling sum with a window length of 2 observations.

```
>>> df.rolling(2).sum()
     B
0  NaN
1  1.0
2  3.0
3  NaN
4  NaN
```

Rolling sum with a window span of 2 seconds.

```
>>> df_time = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]},
...                        index=[pd.Timestamp('20130101 09:00:00'),
...                               pd.Timestamp('20130101 09:00:02'),
...                               pd.Timestamp('20130101 09:00:03'),
...                               pd.Timestamp('20130101 09:00:05'),
...                               pd.Timestamp('20130101 09:00:06')])
```

```
>>> df_time
                     B
2013-01-01 09:00:00  0.0
2013-01-01 09:00:02  1.0
2013-01-01 09:00:03  2.0
2013-01-01 09:00:05  NaN
2013-01-01 09:00:06  4.0
```

```
>>> df_time.rolling('2s').sum()
                       B
2013-01-01 09:00:00  0.0
2013-01-01 09:00:02  1.0
2013-01-01 09:00:03  3.0
2013-01-01 09:00:05  NaN
2013-01-01 09:00:06  4.0
```

Rolling sum with forward looking windows with 2 observations.

```
>>> indexer = pd.api.indexers.FixedForwardWindowIndexer(window_size=2)
>>> df.rolling(window=indexer, min_periods=1).sum()
     B
0  1.0
1  3.0
2  2.0
3  4.0
4  4.0
```

min_periods

Rolling sum with a window length of 2 observations, but only needs a minimum of 1 observation to calculate a value.

```
>>> df.rolling(2, min_periods=1).sum()
     B
0  0.0
1  1.0
2  3.0
3  2.0
4  4.0
```

center

Rolling sum with the result assigned to the center of the window index.

```
>>> df.rolling(3, min_periods=1, center=True).sum()
     B
0  1.0
1  3.0
2  3.0
3  6.0
4  4.0
```

```
>>> df.rolling(3, min_periods=1, center=False).sum()
     B
0  0.0
1  1.0
2  3.0
3  3.0
4  6.0
```

step

Rolling sum with a window length of 2 observations, minimum of 1 observation to calculate a value, and

a step of 2.

```
>>> df.rolling(2, min_periods=1, step=2).sum()
     B
0  0.0
2  3.0
4  4.0
```

win_type

Rolling sum with a window length of 2, using the Scipy `'gaussian'` window type. `std` is required in the aggregation function.

```
>>> df.rolling(2, win_type='gaussian').sum(std=3)
          B
0       NaN
1  0.986207
2  2.958621
3       NaN
4       NaN
```

on

Rolling sum with a window length of 2 days.

```
>>> df = pd.DataFrame({
...     'A': [pd.to_datetime('2020-01-01'),
...           pd.to_datetime('2020-01-01'),
...           pd.to_datetime('2020-01-02'),],
...     'B': [1, 2, 3], },
...     index=pd.date_range('2020', periods=3))
```

```
>>> df
                   A  B
2020-01-01 2020-01-01  1
2020-01-02 2020-01-01  2
2020-01-03 2020-01-02  3
```

```
>>> df.rolling('2D', on='A').sum()
                   A    B
2020-01-01 2020-01-01  1.0
2020-01-02 2020-01-01  3.0
2020-01-03 2020-01-02  6.0
```

AlloViz.AlloViz.Elements.Edges.round

Edges.**round**(*decimals: int | dict[IndexLabel, int] | Series = 0*, **args*, ***kwargs*) → DataFrame

Round a DataFrame to a variable number of decimal places.

Parameters

decimals

[int, dict, Series] Number of decimal places to round each column to. If an int is given, round each column to the same number of places. Otherwise dict and Series round to

variable numbers of places. Column names should be in the keys if *decimals* is a dict-like, or in the index if *decimals* is a Series. Any columns not included in *decimals* will be left as is. Elements of *decimals* which are not columns of the input will be ignored.

***args**
Additional keywords have no effect but might be accepted for compatibility with numpy.

****kwargs**
Additional keywords have no effect but might be accepted for compatibility with numpy.

Returns

DataFrame
A DataFrame with the affected columns rounded to the specified number of decimal places.

See also:

`numpy.around`
Round a numpy array to the given number of decimals.

`Series.round`
Round a Series to the given number of decimals.

Examples

```
>>> df = pd.DataFrame([(.21, .32), (.01, .67), (.66, .03), (.21, .18)],
...                   columns=['dogs', 'cats'])
>>> df
    dogs  cats
0  0.21  0.32
1  0.01  0.67
2  0.66  0.03
3  0.21  0.18
```

By providing an integer each column is rounded to the same number of decimal places

```
>>> df.round(1)
    dogs  cats
0   0.2   0.3
1   0.0   0.7
2   0.7   0.0
3   0.2   0.2
```

With a dict, the number of places for specific columns can be specified with the column names as key and the number of decimal places as value

```
>>> df.round({'dogs': 1, 'cats': 0})
    dogs  cats
0   0.2   0.0
1   0.0   1.0
2   0.7   0.0
3   0.2   0.0
```

Using a Series, the number of places for specific columns can be specified with the column names as index and the number of decimal places as value

```
>>> decimals = pd.Series([0, 1], index=['cats', 'dogs'])
>>> df.round(decimals)
    dogs  cats
0   0.2   0.0
1   0.0   1.0
2   0.7   0.0
3   0.2   0.0
```

AlloViz.AlloViz.Elements.Edges.rpow

Edges.**rpow**(*other*, *axis: Axis = 'columns'*, *level=None*, *fill_value=None*)

Get Exponential power of dataframe and other, element-wise (binary operator *rpow*).

Equivalent to `other ** dataframe`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *pow*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *floordiv*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other
: [scalar, sequence, Series, dict or DataFrame] Any single or multiple element data structure, or list-like object.

axis
: [{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns. (1 or 'columns'). For Series input, axis to match Series index on.

level
: [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value
: [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame
: Result of the arithmetic operation.

See also:

DataFrame.add
: Add DataFrames.

DataFrame.sub
: Subtract DataFrames.

DataFrame.mul
: Multiply DataFrames.

DataFrame.div
: Divide DataFrames (float division).

DataFrame.truediv
Divide DataFrames (float division).

DataFrame.floordiv
Divide DataFrames (integer division).

DataFrame.mod
Calculate modulo (remainder after division).

DataFrame.pow
Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
```

(continues on next page)

(continued from previous page)

```
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a dictionary by axis.

```
>>> df.mul({'angles': 0, 'degrees': 2})
            angles  degrees
circle           0      720
triangle         0      360
rectangle        0      720
```

```
>>> df.mul({'circle': 0, 'triangle': 2, 'rectangle': 3}, axis='index')
            angles  degrees
circle           0        0
triangle         6      360
rectangle       12     1080
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
```

(continues on next page)

(continued from previous page)

```
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

AlloViz.AlloViz.Elements.Edges.rsub

Edges.**rsub**(*other*, *axis: Axis = 'columns'*, *level=None*, *fill_value=None*)

Get Subtraction of dataframe and other, element-wise (binary operator *rsub*).

Equivalent to `other - dataframe`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *sub*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *floordiv*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other
[scalar, sequence, Series, dict or DataFrame] Any single or multiple element data structure, or list-like object.

axis
[{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or

columns. (1 or 'columns'). For Series input, axis to match Series index on.

level
[int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value
[float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame
Result of the arithmetic operation.

See also:

`DataFrame.add`
Add DataFrames.

`DataFrame.sub`
Subtract DataFrames.

`DataFrame.mul`
Multiply DataFrames.

`DataFrame.div`
Divide DataFrames (float division).

`DataFrame.truediv`
Divide DataFrames (float division).

`DataFrame.floordiv`
Divide DataFrames (integer division).

`DataFrame.mod`
Calculate modulo (remainder after division).

`DataFrame.pow`
Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a dictionary by axis.

```
>>> df.mul({'angles': 0, 'degrees': 2})
            angles  degrees
circle           0      720
```

(continues on next page)

(continued from previous page)

```
triangle         0      360
rectangle        0      720
```

```
>>> df.mul({'circle': 0, 'triangle': 2, 'rectangle': 3}, axis='index')
            angles  degrees
circle           0        0
triangle         6      360
rectangle       12     1080
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
```

(continues on next page)

(continued from previous page)

```
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

AlloViz.AlloViz.Elements.Edges.rtruediv

Edges.**rtruediv**(*other*, *axis: Axis = 'columns'*, *level=None*, *fill_value=None*)

Get Floating division of dataframe and other, element-wise (binary operator *rtruediv*).

Equivalent to `other / dataframe`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *truediv*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *floordiv*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other
: [scalar, sequence, Series, dict or DataFrame] Any single or multiple element data structure, or list-like object.

axis
: [{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns. (1 or 'columns'). For Series input, axis to match Series index on.

level
: [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value
: [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame
: Result of the arithmetic operation.

See also:

`DataFrame.add`
: Add DataFrames.

`DataFrame.sub`
: Subtract DataFrames.

`DataFrame.mul`
: Multiply DataFrames.

`DataFrame.div`
: Divide DataFrames (float division).

`DataFrame.truediv`
: Divide DataFrames (float division).

DataFrame.floordiv
Divide DataFrames (integer division).

DataFrame.mod
Calculate modulo (remainder after division).

DataFrame.pow
Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a dictionary by axis.

```
>>> df.mul({'angles': 0, 'degrees': 2})
            angles  degrees
circle           0      720
triangle         0      360
rectangle        0      720
```

```
>>> df.mul({'circle': 0, 'triangle': 2, 'rectangle': 3}, axis='index')
            angles  degrees
circle           0        0
triangle         6      360
rectangle       12     1080
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
```

(continues on next page)

(continued from previous page)

```
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

AlloViz.AlloViz.Elements.Edges.sample

Edges.**sample**(*n: None | int = None*, *frac: float | None = None*, *replace: bool = False*, *weights=None*, *random_state: int | ndarray | Generator | BitGenerator | RandomState | None = None*, *axis: int | Literal['index', 'columns', 'rows'] | None = None*, *ignore_index: bool = False*) → None

Return a random sample of items from an axis of object.

You can use *random_state* for reproducibility.

Parameters

n
: [int, optional] Number of items from axis to return. Cannot be used with *frac*. Default = 1 if *frac* = None.

frac
: [float, optional] Fraction of axis items to return. Cannot be used with *n*.

replace
: [bool, default False] Allow or disallow sampling of the same row more than once.

weights
: [str or ndarray-like, optional] Default 'None' results in equal probability weighting. If passed a Series, will align with target object on index. Index values in weights not found in sampled object will be ignored and index values in sampled object not in weights will be assigned weights of zero. If called on a DataFrame, will accept the

name of a column when axis = 0. Unless weights are a Series, weights must be same length as axis being sampled. If weights do not sum to 1, they will be normalized to sum to 1. Missing values in the weights column will be treated as zero. Infinite values not allowed.

random_state
[int, array-like, BitGenerator, np.random.RandomState, np.random.Generator, optional] If int, array-like, or BitGenerator, seed for random number generator. If np.random.RandomState or np.random.Generator, use as given.

Changed in version 1.4.0: np.random.Generator objects now accepted

axis
[{0 or 'index', 1 or 'columns', None}, default None] Axis to sample. Accepts axis number or name. Default is stat axis for given data type. For *Series* this parameter is unused and defaults to *None*.

ignore_index
[bool, default False] If True, the resulting index will be labeled 0, 1, …, n - 1.

New in version 1.3.0.

Returns

Series or DataFrame
A new object of same type as caller containing *n* items randomly sampled from the caller object.

See also:

`DataFrameGroupBy.sample`
Generates random samples from each group of a DataFrame object.

`SeriesGroupBy.sample`
Generates random samples from each group of a Series object.

`numpy.random.choice`
Generates a random sample from a given 1-D numpy array.

Notes

If *frac* > 1, *replacement* should be set to *True*.

Examples

```
>>> df = pd.DataFrame({'num_legs': [2, 4, 8, 0],
...                    'num_wings': [2, 0, 0, 0],
...                    'num_specimen_seen': [10, 2, 1, 8]},
...                   index=['falcon', 'dog', 'spider', 'fish'])
>>> df
        num_legs  num_wings  num_specimen_seen
falcon         2          2                 10
dog            4          0                  2
spider         8          0                  1
fish           0          0                  8
```

Extract 3 random elements from the `Series` `df['num_legs']`: Note that we use *random_state* to ensure the reproducibility of the examples.

```
>>> df['num_legs'].sample(n=3, random_state=1)
fish      0
spider    8
falcon    2
Name: num_legs, dtype: int64
```

A random 50% sample of the `DataFrame` with replacement:

```
>>> df.sample(frac=0.5, replace=True, random_state=1)
      num_legs  num_wings  num_specimen_seen
dog          4          0                  2
fish         0          0                  8
```

An upsample sample of the `DataFrame` with replacement: Note that *replace* parameter has to be *True* for *frac* parameter > 1.

```
>>> df.sample(frac=2, replace=True, random_state=1)
        num_legs  num_wings  num_specimen_seen
dog            4          0                  2
fish           0          0                  8
falcon         2          2                 10
falcon         2          2                 10
fish           0          0                  8
dog            4          0                  2
fish           0          0                  8
dog            4          0                  2
```

Using a DataFrame column as weights. Rows with larger value in the *num_specimen_seen* column are more likely to be sampled.

```
>>> df.sample(n=2, weights='num_specimen_seen', random_state=1)
        num_legs  num_wings  num_specimen_seen
falcon         2          2                 10
fish           0          0                  8
```

AlloViz.AlloViz.Elements.Edges.select_dtypes

`Edges.`**`select_dtypes`**(*include=None*, *exclude=None*) → Self

Return a subset of the DataFrame's columns based on the column dtypes.

Parameters

include, exclude
[scalar or list-like] A selection of dtypes or strings to be included/excluded. At least one of these parameters must be supplied.

Returns

DataFrame
The subset of the frame including the dtypes in `include` and excluding the dtypes in `exclude`.

Raises

ValueError

- If both of `include` and `exclude` are empty
- If `include` and `exclude` have overlapping elements
- If any kind of string dtype is passed in.

See also:

`DataFrame.dtypes`
Return Series with the data type of each column.

Notes

- To select all *numeric* types, use `np.number` or `'number'`
- To select strings you must use the `object` dtype, but note that this will return *all* object dtype columns
- See the numpy dtype hierarchy
- To select datetimes, use `np.datetime64`, `'datetime'` or `'datetime64'`
- To select timedeltas, use `np.timedelta64`, `'timedelta'` or `'timedelta64'`
- To select Pandas categorical dtypes, use `'category'`
- To select Pandas datetimetz dtypes, use `'datetimetz'` or `'datetime64[ns, tz]'`

Examples

```
>>> df = pd.DataFrame({'a': [1, 2] * 3,
...                    'b': [True, False] * 3,
...                    'c': [1.0, 2.0] * 3})
>>> df
       a      b  c
0      1   True  1.0
1      2  False  2.0
2      1   True  1.0
3      2  False  2.0
4      1   True  1.0
5      2  False  2.0
```

```
>>> df.select_dtypes(include='bool')
   b
0  True
1  False
2  True
3  False
4  True
5  False
```

```
>>> df.select_dtypes(include=['float64'])
   c
0  1.0
1  2.0
```

(continues on next page)

(continued from previous page)

```
2  1.0
3  2.0
4  1.0
5  2.0
```

```
>>> df.select_dtypes(exclude=['int64'])
       b    c
0   True  1.0
1  False  2.0
2   True  1.0
3  False  2.0
4   True  1.0
5  False  2.0
```

AlloViz.AlloViz.Elements.Edges.sem

Edges.**sem**(*axis: Axis | None = 0*, *skipna: bool = True*, *ddof: int = 1*, *numeric_only: bool = False*, ***kwargs*)

Return unbiased standard error of the mean over requested axis.

Normalized by N-1 by default. This can be changed using the ddof argument

Parameters

axis
[{index (0), columns (1)}] For *Series* this parameter is unused and defaults to 0.

skipna
[bool, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

ddof
[int, default 1] Delta Degrees of Freedom. The divisor used in calculations is N - ddof, where N represents the number of elements.

numeric_only
[bool, default False] Include only float, int, boolean columns. Not implemented for Series.

Returns

Series or DataFrame (if level specified)

Examples

```
>>> s = pd.Series([1, 2, 3])
>>> s.sem().round(6)
0.57735
```

With a DataFrame

```
>>> df = pd.DataFrame({'a': [1, 2], 'b': [2, 3]}, index=['tiger', 'zebra'])
>>> df
       a   b
```

(continues on next page)

(continued from previous page)

```
tiger  1   2
zebra  2   3
>>> df.sem()
a   0.5
b   0.5
dtype: float64
```

Using axis=1

```
>>> df.sem(axis=1)
tiger   0.5
zebra   0.5
dtype: float64
```

In this case, *numeric_only* should be set to *True* to avoid getting an error.

```
>>> df = pd.DataFrame({'a': [1, 2], 'b': ['T', 'Z']},
...                   index=['tiger', 'zebra'])
>>> df.sem(numeric_only=True)
a   0.5
dtype: float64
```

AlloViz.AlloViz.Elements.Edges.set_axis

Edges.**set_axis**(*labels*, *, *axis: Axis = 0*, *copy: bool | None = None*) → DataFrame

Assign desired index to given axis.

Indexes for column or row labels can be changed by assigning a list-like or Index.

Parameters

labels
[list-like, Index] The values for the new index.

axis
[{0 or 'index', 1 or 'columns'}, default 0] The axis to update. The value 0 identifies the rows. For *Series* this parameter is unused and defaults to 0.

copy
[bool, default True] Whether to make a copy of the underlying data.

New in version 1.5.0.

Returns

DataFrame
An object of type DataFrame.

See also:

DataFrame.rename_axis
Alter the name of the index or columns.

Examples

```
>>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
```

Change the row labels.

```
>>> df.set_axis(['a', 'b', 'c'], axis='index')
   A  B
a  1  4
b  2  5
c  3  6
```

Change the column labels.

```
>>> df.set_axis(['I', 'II'], axis='columns')
   I  II
0  1   4
1  2   5
2  3   6
```

AlloViz.AlloViz.Elements.Edges.set_flags

Edges.**set_flags**(*, *copy: bool = False*, *allows_duplicate_labels: bool | None = None*) → None

Return a new object with updated flags.

Parameters

copy
[bool, default False] Specify if a copy of the object should be made.

allows_duplicate_labels
[bool, optional] Whether the returned object allows duplicate labels.

Returns

Series or DataFrame
The same type as the caller.

See also:

DataFrame.attrs
Global metadata applying to this dataset.

DataFrame.flags
Global flags applying to this object.

Notes

This method returns a new object that's a view on the same data as the input. Mutating the input or the output values will be reflected in the other.

This method is intended to be used in method chains.

"Flags" differ from "metadata". Flags reflect properties of the pandas object (the Series or DataFrame). Metadata refer to properties of the dataset, and should be stored in `DataFrame.attrs`.

Examples

```
>>> df = pd.DataFrame({"A": [1, 2]})
>>> df.flags.allows_duplicate_labels
True
>>> df2 = df.set_flags(allows_duplicate_labels=False)
>>> df2.flags.allows_duplicate_labels
False
```

AlloViz.AlloViz.Elements.Edges.set_index

Edges.**set_index**(*keys*, *, *drop: bool = True*, *append: bool = False*, *inplace: bool = False*, *verify_integrity: bool = False*) → DataFrame | None

Set the DataFrame index using existing columns.

Set the DataFrame index (row labels) using one or more existing columns or arrays (of the correct length). The index can replace the existing index or expand on it.

Parameters

keys
[label or array-like or list of labels/arrays] This parameter can be either a single column key, a single array of the same length as the calling DataFrame, or a list containing an arbitrary combination of column keys and arrays. Here, "array" encompasses `Series`, `Index`, `np.ndarray`, and instances of `Iterator`.

drop
[bool, default True] Delete columns to be used as the new index.

append
[bool, default False] Whether to append columns to existing index.

inplace
[bool, default False] Whether to modify the DataFrame rather than creating a new one.

verify_integrity
[bool, default False] Check the new index for duplicates. Otherwise defer the check until necessary. Setting to False will improve the performance of this method.

Returns

DataFrame or None
Changed row labels or None if `inplace=True`.

See also:

DataFrame.reset_index
Opposite of set_index.

DataFrame.reindex
Change to new indices or expand indices.

DataFrame.reindex_like
Change to same indices as other DataFrame.

Examples

```
>>> df = pd.DataFrame({'month': [1, 4, 7, 10],
...                    'year': [2012, 2014, 2013, 2014],
...                    'sale': [55, 40, 84, 31]})
>>> df
   month  year  sale
0      1  2012    55
1      4  2014    40
2      7  2013    84
3     10  2014    31
```

Set the index to become the 'month' column:

```
>>> df.set_index('month')
       year  sale
month
1      2012    55
4      2014    40
7      2013    84
10     2014    31
```

Create a MultiIndex using columns 'year' and 'month':

```
>>> df.set_index(['year', 'month'])
            sale
year  month
2012  1     55
2014  4     40
2013  7     84
2014  10    31
```

Create a MultiIndex using an Index and a column:

```
>>> df.set_index([pd.Index([1, 2, 3, 4]), 'year'])
         month  sale
   year
1  2012  1      55
2  2014  4      40
3  2013  7      84
4  2014  10     31
```

Create a MultiIndex using two Series:

```
>>> s = pd.Series([1, 2, 3, 4])
>>> df.set_index([s, s**2])
      month  year  sale
1 1       1  2012    55
2 4       4  2014    40
3 9       7  2013    84
4 16     10  2014    31
```

AlloViz.AlloViz.Elements.Edges.shift

Edges.**shift**(*periods: int | Sequence[int] = 1*, *freq: Frequency | None = None*, *axis: Axis = 0*, *fill_value: Hashable = _NoDefault.no_default*, *suffix: str | None = None*) → DataFrame

Shift index by desired number of periods with an optional time *freq*.

When *freq* is not passed, shift the index without realigning the data. If *freq* is passed (in this case, the index must be date or datetime, or it will raise a *NotImplementedError*), the index will be increased using the periods and the *freq*. *freq* can be inferred when specified as "infer" as long as either freq or inferred_freq attribute is set in the index.

Parameters

periods
: [int or Sequence] Number of periods to shift. Can be positive or negative. If an iterable of ints, the data will be shifted once by each int. This is equivalent to shifting by one value at a time and concatenating all resulting frames. The resulting columns will have the shift suffixed to their column names. For multiple periods, axis must not be 1.

freq
: [DateOffset, tseries.offsets, timedelta, or str, optional] Offset to use from the tseries module or time rule (e.g. 'EOM'). If *freq* is specified then the index values are shifted but the data is not realigned. That is, use *freq* if you would like to extend the index when shifting and preserve the original data. If *freq* is specified as "infer" then it will be inferred from the freq or inferred_freq attributes of the index. If neither of those attributes exist, a ValueError is thrown.

axis
: [{0 or 'index', 1 or 'columns', None}, default None] Shift direction. For *Series* this parameter is unused and defaults to 0.

fill_value
: [object, optional] The scalar value to use for newly introduced missing values. the default depends on the dtype of *self*. For numeric data, `np.nan` is used. For datetime, timedelta, or period data, etc. `NaT` is used. For extension dtypes, `self.dtype.na_value` is used.

suffix
: [str, optional] If str and periods is an iterable, this is added after the column name and before the shift value for each shifted column name.

Returns

DataFrame
: Copy of input object, shifted.

See also:

Index.shift
Shift values of Index.

DatetimeIndex.shift
Shift values of DatetimeIndex.

PeriodIndex.shift
Shift values of PeriodIndex.

Examples

```
>>> df = pd.DataFrame({"Col1": [10, 20, 15, 30, 45],
...                    "Col2": [13, 23, 18, 33, 48],
...                    "Col3": [17, 27, 22, 37, 52]},
...                   index=pd.date_range("2020-01-01", "2020-01-05"))
>>> df
            Col1  Col2  Col3
2020-01-01    10    13    17
2020-01-02    20    23    27
2020-01-03    15    18    22
2020-01-04    30    33    37
2020-01-05    45    48    52
```

```
>>> df.shift(periods=3)
            Col1  Col2  Col3
2020-01-01   NaN   NaN   NaN
2020-01-02   NaN   NaN   NaN
2020-01-03   NaN   NaN   NaN
2020-01-04  10.0  13.0  17.0
2020-01-05  20.0  23.0  27.0
```

```
>>> df.shift(periods=1, axis="columns")
            Col1  Col2  Col3
2020-01-01   NaN    10    13
2020-01-02   NaN    20    23
2020-01-03   NaN    15    18
2020-01-04   NaN    30    33
2020-01-05   NaN    45    48
```

```
>>> df.shift(periods=3, fill_value=0)
            Col1  Col2  Col3
2020-01-01     0     0     0
2020-01-02     0     0     0
2020-01-03     0     0     0
2020-01-04    10    13    17
2020-01-05    20    23    27
```

```
>>> df.shift(periods=3, freq="D")
            Col1  Col2  Col3
2020-01-04    10    13    17
2020-01-05    20    23    27
2020-01-06    15    18    22
```

(continues on next page)

(continued from previous page)

```
2020-01-07    30    33    37
2020-01-08    45    48    52
```

```
>>> df.shift(periods=3, freq="infer")
            Col1  Col2  Col3
2020-01-04    10    13    17
2020-01-05    20    23    27
2020-01-06    15    18    22
2020-01-07    30    33    37
2020-01-08    45    48    52
```

```
>>> df['Col1'].shift(periods=[0, 1, 2])
            Col1_0  Col1_1  Col1_2
2020-01-01      10     NaN     NaN
2020-01-02      20    10.0     NaN
2020-01-03      15    20.0    10.0
2020-01-04      30    15.0    20.0
2020-01-05      45    30.0    15.0
```

AlloViz.AlloViz.Elements.Edges.skew

Edges.**skew**(*axis: Axis | None = 0*, *skipna: bool = True*, *numeric_only: bool = False*, ***kwargs*)

Return unbiased skew over requested axis.

Normalized by N-1.

Parameters

axis
[{index (0), columns (1)}] Axis for the function to be applied on. For *Series* this parameter is unused and defaults to 0.

For DataFrames, specifying `axis=None` will apply the aggregation across both axes.

New in version 2.0.0.

skipna
[bool, default True] Exclude NA/null values when computing the result.

numeric_only
[bool, default False] Include only float, int, boolean columns. Not implemented for Series.

****kwargs**
Additional keyword arguments to be passed to the function.

Returns

Series or scalar

Examples

```
>>> s = pd.Series([1, 2, 3])
>>> s.skew()
0.0
```

With a DataFrame

```
>>> df = pd.DataFrame({'a': [1, 2, 3], 'b': [2, 3, 4], 'c': [1, 3, 5]},
...                   index=['tiger', 'zebra', 'cow'])
>>> df
        a   b   c
tiger   1   2   1
zebra   2   3   3
cow     3   4   5
>>> df.skew()
a   0.0
b   0.0
c   0.0
dtype: float64
```

Using axis=1

```
>>> df.skew(axis=1)
tiger   1.732051
zebra  -1.732051
cow     0.000000
dtype: float64
```

In this case, *numeric_only* should be set to *True* to avoid getting an error.

```
>>> df = pd.DataFrame({'a': [1, 2, 3], 'b': ['T', 'Z', 'X']},
...                   index=['tiger', 'zebra', 'cow'])
>>> df.skew(numeric_only=True)
a   0.0
dtype: float64
```

AlloViz.AlloViz.Elements.Edges.sort_index

Edges.**sort_index**(*, *axis: Axis = 0*, *level: IndexLabel | None = None*, *ascending: bool | Sequence[bool] = True*, *inplace: bool = False*, *kind: SortKind = 'quicksort'*, *na_position: NaPosition = 'last'*, *sort_remaining: bool = True*, *ignore_index: bool = False*, *key: IndexKeyFunc | None = None*) → DataFrame | None

Sort object by labels (along an axis).

Returns a new DataFrame sorted by label if *inplace* argument is `False`, otherwise updates the original DataFrame and returns None.

Parameters

axis

[{0 or 'index', 1 or 'columns'}, default 0] The axis along which to sort. The value 0 identifies the rows, and 1 identifies the columns.

level
[int or level name or list of ints or list of level names] If not None, sort on values in specified index level(s).

ascending
[bool or list-like of bools, default True] Sort ascending vs. descending. When the index is a MultiIndex the sort direction can be controlled for each level individually.

inplace
[bool, default False] Whether to modify the DataFrame rather than creating a new one.

kind
[{'quicksort', 'mergesort', 'heapsort', 'stable'}, default 'quicksort'] Choice of sorting algorithm. See also `numpy.sort()` for more information. *mergesort* and *stable* are the only stable algorithms. For DataFrames, this option is only applied when sorting on a single column or label.

na_position
[{'first', 'last'}, default 'last'] Puts NaNs at the beginning if *first*; *last* puts NaNs at the end. Not implemented for MultiIndex.

sort_remaining
[bool, default True] If True and sorting by level and index is multilevel, sort by other levels too (in order) after sorting by specified level.

ignore_index
[bool, default False] If True, the resulting axis will be labeled 0, 1, …, n - 1.

key
[callable, optional] If not None, apply the key function to the index values before sorting. This is similar to the *key* argument in the builtin `sorted()` function, with the notable difference that this *key* function should be *vectorized*. It should expect an `Index` and return an `Index` of the same shape. For MultiIndex inputs, the key is applied *per level*.

Returns

DataFrame or None
The original DataFrame sorted by the labels or None if `inplace=True`.

See also:

`Series.sort_index`
Sort Series by the index.

`DataFrame.sort_values`
Sort DataFrame by the value.

`Series.sort_values`
Sort Series by the value.

Examples

```
>>> df = pd.DataFrame([1, 2, 3, 4, 5], index=[100, 29, 234, 1, 150],
...                   columns=['A'])
>>> df.sort_index()
     A
1    4
29   2
100  1
150  5
234  3
```

By default, it sorts in ascending order, to sort in descending order, use `ascending=False`

```
>>> df.sort_index(ascending=False)
     A
234  3
150  5
100  1
29   2
1    4
```

A key function can be specified which is applied to the index before sorting. For a `MultiIndex` this is applied to each level separately.

```
>>> df = pd.DataFrame({"a": [1, 2, 3, 4]}, index=['A', 'b', 'C', 'd'])
>>> df.sort_index(key=lambda x: x.str.lower())
   a
A  1
b  2
C  3
d  4
```

AlloViz.AlloViz.Elements.Edges.sort_values

Edges.**sort_values**(*by: IndexLabel*, *, *axis: Axis = 0*, *ascending: bool | list[bool] | tuple[bool, ...] = True*, *inplace: bool = False*, *kind: SortKind = 'quicksort'*, *na_position: str = 'last'*, *ignore_index: bool = False*, *key: ValueKeyFunc | None = None*) → DataFrame | None

Sort by the values along either axis.

Parameters

by
[str or list of str] Name or list of names to sort by.

- if *axis* is 0 or *'index'* then *by* may contain index levels and/or column labels.
- if *axis* is 1 or *'columns'* then *by* may contain column levels and/or index labels.

axis
["{0 or 'index', 1 or 'columns'}", default 0] Axis to be sorted.

ascending
[bool or list of bool, default True] Sort ascending vs. descending. Specify list for multiple sort orders. If this is a list of bools, must match the length of the by.

inplace
[bool, default False] If True, perform operation in-place.

kind
[{'quicksort', 'mergesort', 'heapsort', 'stable'}, default 'quicksort'] Choice of sorting algorithm. See also `numpy.sort()` for more information. *mergesort* and *stable* are the only stable algorithms. For DataFrames, this option is only applied when sorting on a single column or label.

na_position
[{'first', 'last'}, default 'last'] Puts NaNs at the beginning if *first*; *last* puts NaNs at the end.

ignore_index
[bool, default False] If True, the resulting axis will be labeled 0, 1, …, n - 1.

key
[callable, optional] Apply the key function to the values before sorting. This is similar to the *key* argument in the builtin `sorted()` function, with the notable difference that this *key* function should be *vectorized*. It should expect a `Series` and return a Series with the same shape as the input. It will be applied to each column in *by* independently.

Returns

DataFrame or None
DataFrame with sorted values or None if `inplace=True`.

See also:

`DataFrame.sort_index`
Sort a DataFrame by the index.

`Series.sort_values`
Similar method for a Series.

Examples

```
>>> df = pd.DataFrame({
...     'col1': ['A', 'A', 'B', np.nan, 'D', 'C'],
...     'col2': [2, 1, 9, 8, 7, 4],
...     'col3': [0, 1, 9, 4, 2, 3],
...     'col4': ['a', 'B', 'c', 'D', 'e', 'F']
... })
>>> df
  col1  col2  col3 col4
0    A     2     0    a
1    A     1     1    B
2    B     9     9    c
3  NaN     8     4    D
4    D     7     2    e
5    C     4     3    F
```

Sort by col1

```
>>> df.sort_values(by=['col1'])
  col1  col2  col3 col4
```

(continues on next page)

(continued from previous page)

```
0    A     2     0    a
1    A     1     1    B
2    B     9     9    c
5    C     4     3    F
4    D     7     2    e
3  NaN     8     4    D
```

Sort by multiple columns

```
>>> df.sort_values(by=['col1', 'col2'])
  col1  col2  col3 col4
1    A     1     1    B
0    A     2     0    a
2    B     9     9    c
5    C     4     3    F
4    D     7     2    e
3  NaN     8     4    D
```

Sort Descending

```
>>> df.sort_values(by='col1', ascending=False)
  col1  col2  col3 col4
4    D     7     2    e
5    C     4     3    F
2    B     9     9    c
0    A     2     0    a
1    A     1     1    B
3  NaN     8     4    D
```

Putting NAs first

```
>>> df.sort_values(by='col1', ascending=False, na_position='first')
  col1  col2  col3 col4
3  NaN     8     4    D
4    D     7     2    e
5    C     4     3    F
2    B     9     9    c
0    A     2     0    a
1    A     1     1    B
```

Sorting with a key function

```
>>> df.sort_values(by='col4', key=lambda col: col.str.lower())
   col1  col2  col3 col4
0    A     2     0    a
1    A     1     1    B
2    B     9     9    c
3  NaN     8     4    D
4    D     7     2    e
5    C     4     3    F
```

Natural sort with the key argument, using the *natsort <https://github.com/SethMMorton/natsort>* package.

```
>>> df = pd.DataFrame({
...     "time": ['0hr', '128hr', '72hr', '48hr', '96hr'],
...     "value": [10, 20, 30, 40, 50]
... })
>>> df
    time  value
0    0hr     10
1  128hr     20
2   72hr     30
3   48hr     40
4   96hr     50
>>> from natsort import index_natsorted
>>> df.sort_values(
...     by="time",
...     key=lambda x: np.argsort(index_natsorted(df["time"]))
... )
    time  value
0    0hr     10
3   48hr     40
2   72hr     30
4   96hr     50
1  128hr     20
```

AlloViz.AlloViz.Elements.Edges.squeeze

Edges.**squeeze**(*axis: int | Literal['index', 'columns', 'rows'] | None = None*)

Squeeze 1 dimensional axis objects into scalars.

Series or DataFrames with a single element are squeezed to a scalar. DataFrames with a single column or a single row are squeezed to a Series. Otherwise the object is unchanged.

This method is most useful when you don't know if your object is a Series or DataFrame, but you do know it has just a single column. In that case you can safely call *squeeze* to ensure you have a Series.

Parameters

axis
[{0 or 'index', 1 or 'columns', None}, default None] A specific axis to squeeze. By default, all length-1 axes are squeezed. For *Series* this parameter is unused and defaults to *None*.

Returns

DataFrame, Series, or scalar
The projection after squeezing *axis* or all the axes.

See also:

Series.iloc
Integer-location based indexing for selecting scalars.

DataFrame.iloc
Integer-location based indexing for selecting Series.

Series.to_frame
Inverse of DataFrame.squeeze for a single-column DataFrame.

Examples

```
>>> primes = pd.Series([2, 3, 5, 7])
```

Slicing might produce a Series with a single value:

```
>>> even_primes = primes[primes % 2 == 0]
>>> even_primes
0    2
dtype: int64
```

```
>>> even_primes.squeeze()
2
```

Squeezing objects with more than one value in every axis does nothing:

```
>>> odd_primes = primes[primes % 2 == 1]
>>> odd_primes
1    3
2    5
3    7
dtype: int64
```

```
>>> odd_primes.squeeze()
1    3
2    5
3    7
dtype: int64
```

Squeezing is even more effective when used with DataFrames.

```
>>> df = pd.DataFrame([[1, 2], [3, 4]], columns=['a', 'b'])
>>> df
   a  b
0  1  2
1  3  4
```

Slicing a single column will produce a DataFrame with the columns having only one value:

```
>>> df_a = df[['a']]
>>> df_a
   a
0  1
1  3
```

So the columns can be squeezed down, resulting in a Series:

```
>>> df_a.squeeze('columns')
0    1
1    3
Name: a, dtype: int64
```

Slicing a single row from a single column will produce a single scalar DataFrame:

```
>>> df_0a = df.loc[df.index < 1, ['a']]
>>> df_0a
   a
0  1
```

Squeezing the rows produces a single scalar Series:

```
>>> df_0a.squeeze('rows')
a    1
Name: 0, dtype: int64
```

Squeezing all axes will project directly into a scalar:

```
>>> df_0a.squeeze()
1
```

AlloViz.AlloViz.Elements.Edges.stack

Edges.**stack**(*level: IndexLabel = -1*, *dropna: bool | lib.NoDefault = _NoDefault.no_default*, *sort: bool | lib.NoDefault = _NoDefault.no_default*, *future_stack: bool = False*)

Stack the prescribed level(s) from columns to index.

Return a reshaped DataFrame or Series having a multi-level index with one or more new inner-most levels compared to the current DataFrame. The new inner-most levels are created by pivoting the columns of the current dataframe:

- if the columns have a single level, the output is a Series;
- if the columns have multiple levels, the new index level(s) is (are) taken from the prescribed level(s) and the output is a DataFrame.

Parameters

level
[int, str, list, default -1] Level(s) to stack from the column axis onto the index axis, defined as one index or label, or a list of indices or labels.

dropna
[bool, default True] Whether to drop rows in the resulting Frame/Series with missing values. Stacking a column level onto the index axis can create combinations of index and column values that are missing from the original dataframe. See Examples section.

sort
[bool, default True] Whether to sort the levels of the resulting MultiIndex.

future_stack
[bool, default False] Whether to use the new implementation that will replace the current implementation in pandas 3.0. When True, dropna and sort have no impact on the result and must remain unspecified. See pandas 2.1.0 Release notes for more details.

Returns

DataFrame or Series
Stacked dataframe or series.

See also:

DataFrame.unstack
Unstack prescribed level(s) from index axis onto column axis.

DataFrame.pivot
Reshape dataframe from long format to wide format.

DataFrame.pivot_table
Create a spreadsheet-style pivot table as a DataFrame.

Notes

The function is named by analogy with a collection of books being reorganized from being side by side on a horizontal position (the columns of the dataframe) to being stacked vertically on top of each other (in the index of the dataframe).

Reference the user guide for more examples.

Examples

Single level columns

```
>>> df_single_level_cols = pd.DataFrame([[0, 1], [2, 3]],
...                                     index=['cat', 'dog'],
...                                     columns=['weight', 'height'])
```

Stacking a dataframe with a single level column axis returns a Series:

```
>>> df_single_level_cols
     weight height
cat       0      1
dog       2      3
>>> df_single_level_cols.stack(future_stack=True)
cat  weight    0
     height    1
dog  weight    2
     height    3
dtype: int64
```

Multi level columns: simple case

```
>>> multicol1 = pd.MultiIndex.from_tuples([('weight', 'kg'),
...                                        ('weight', 'pounds')])
>>> df_multi_level_cols1 = pd.DataFrame([[1, 2], [2, 4]],
...                                     index=['cat', 'dog'],
...                                     columns=multicol1)
```

Stacking a dataframe with a multi-level column axis:

```
>>> df_multi_level_cols1
     weight
         kg    pounds
cat       1        2
dog       2        4
>>> df_multi_level_cols1.stack(future_stack=True)
```

(continues on next page)

(continued from previous page)

```
            weight
cat kg           1
    pounds       2
dog kg           2
    pounds       4
```

Missing values

```
>>> multicol2 = pd.MultiIndex.from_tuples([('weight', 'kg'),
...                                        ('height', 'm')])
>>> df_multi_level_cols2 = pd.DataFrame([[1.0, 2.0], [3.0, 4.0]],
...                                     index=['cat', 'dog'],
...                                     columns=multicol2)
```

It is common to have missing values when stacking a dataframe with multi-level columns, as the stacked dataframe typically has more values than the original dataframe. Missing values are filled with NaNs:

```
>>> df_multi_level_cols2
    weight height
        kg      m
cat    1.0    2.0
dog    3.0    4.0
>>> df_multi_level_cols2.stack(future_stack=True)
        weight  height
cat kg     1.0     NaN
    m      NaN     2.0
dog kg     3.0     NaN
    m      NaN     4.0
```

Prescribing the level(s) to be stacked

The first parameter controls which level or levels are stacked:

```
>>> df_multi_level_cols2.stack(0, future_stack=True)
             kg    m
cat weight  1.0  NaN
    height  NaN  2.0
dog weight  3.0  NaN
    height  NaN  4.0
>>> df_multi_level_cols2.stack([0, 1], future_stack=True)
cat  weight  kg    1.0
     height  m     2.0
dog  weight  kg    3.0
     height  m     4.0
dtype: float64
```

Dropping missing values

```
>>> df_multi_level_cols3 = pd.DataFrame([[None, 1.0], [2.0, 3.0]],
...                                     index=['cat', 'dog'],
...                                     columns=multicol2)
```

Note that rows where all values are missing are dropped by default but this behaviour can be controlled via the dropna keyword parameter:

```
>>> df_multi_level_cols3
    weight height
        kg      m
cat    NaN    1.0
dog    2.0    3.0
>>> df_multi_level_cols3.stack(dropna=False)
        weight  height
cat kg     NaN     NaN
    m      NaN     1.0
dog kg     2.0     NaN
    m      NaN     3.0
>>> df_multi_level_cols3.stack(dropna=True)
        weight  height
cat m      NaN     1.0
dog kg     2.0     NaN
    m      NaN     3.0
```

AlloViz.AlloViz.Elements.Edges.std

Edges.**std**(*axis: Axis | None = 0*, *skipna: bool = True*, *ddof: int = 1*, *numeric_only: bool = False*, ***kwargs*)

Return sample standard deviation over requested axis.

Normalized by N-1 by default. This can be changed using the ddof argument.

Parameters

axis
[{index (0), columns (1)}] For *Series* this parameter is unused and defaults to 0.

skipna
[bool, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

ddof
[int, default 1] Delta Degrees of Freedom. The divisor used in calculations is N - ddof, where N represents the number of elements.

numeric_only
[bool, default False] Include only float, int, boolean columns. Not implemented for Series.

Returns

Series or DataFrame (if level specified)

Notes

To have the same behaviour as *numpy.std*, use *ddof=0* (instead of the default *ddof=1*)

Examples

```
>>> df = pd.DataFrame({'person_id': [0, 1, 2, 3],
...                    'age': [21, 25, 62, 43],
...                    'height': [1.61, 1.87, 1.49, 2.01]}
...                   ).set_index('person_id')
>>> df
           age  height
person_id
0           21    1.61
1           25    1.87
2           62    1.49
3           43    2.01
```

The standard deviation of the columns can be found as follows:

```
>>> df.std()
age       18.786076
height     0.237417
dtype: float64
```

Alternatively, *ddof=0* can be set to normalize by N instead of N-1:

```
>>> df.std(ddof=0)
age       16.269219
height     0.205609
dtype: float64
```

AlloViz.AlloViz.Elements.Edges.sub

`Edges.`**`sub`**(*other*, *axis: Axis = 'columns'*, *level=None*, *fill_value=None*)

Get Subtraction of dataframe and other, element-wise (binary operator *sub*).

Equivalent to `dataframe - other`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *rsub*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *floordiv*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other
: [scalar, sequence, Series, dict or DataFrame] Any single or multiple element data structure, or list-like object.

axis
: [{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns. (1 or 'columns'). For Series input, axis to match Series index on.

level
[int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value
[float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame
Result of the arithmetic operation.

See also:

`DataFrame.add`
Add DataFrames.

`DataFrame.sub`
Subtract DataFrames.

`DataFrame.mul`
Multiply DataFrames.

`DataFrame.div`
Divide DataFrames (float division).

`DataFrame.truediv`
Divide DataFrames (float division).

`DataFrame.floordiv`
Divide DataFrames (integer division).

`DataFrame.mod`
Calculate modulo (remainder after division).

`DataFrame.pow`
Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a dictionary by axis.

```
>>> df.mul({'angles': 0, 'degrees': 2})
            angles  degrees
circle           0      720
```

(continues on next page)

(continued from previous page)

```
triangle         0      360
rectangle        0      720
```

```
>>> df.mul({'circle': 0, 'triangle': 2, 'rectangle': 3}, axis='index')
           angles  degrees
circle          0        0
triangle        6      360
rectangle      12     1080
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
```

(continues on next page)

(continued from previous page)

```
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

AlloViz.AlloViz.Elements.Edges.subtract

Edges.**subtract**(*other*, *axis: Axis = 'columns'*, *level=None*, *fill_value=None*)

Get Subtraction of dataframe and other, element-wise (binary operator *sub*).

Equivalent to `dataframe - other`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *rsub*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *floordiv*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other
[scalar, sequence, Series, dict or DataFrame] Any single or multiple element data structure, or list-like object.

axis
[{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns. (1 or 'columns'). For Series input, axis to match Series index on.

level
[int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value
[float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame
Result of the arithmetic operation.

See also:

`DataFrame.add`
Add DataFrames.

`DataFrame.sub`
Subtract DataFrames.

`DataFrame.mul`
Multiply DataFrames.

`DataFrame.div`
Divide DataFrames (float division).

`DataFrame.truediv`
Divide DataFrames (float division).

DataFrame.floordiv
Divide DataFrames (integer division).

DataFrame.mod
Calculate modulo (remainder after division).

DataFrame.pow
Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a dictionary by axis.

```
>>> df.mul({'angles': 0, 'degrees': 2})
            angles  degrees
circle           0      720
triangle         0      360
rectangle        0      720
```

```
>>> df.mul({'circle': 0, 'triangle': 2, 'rectangle': 3}, axis='index')
            angles  degrees
circle           0        0
triangle         6      360
rectangle       12     1080
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
```

(continues on next page)

(continued from previous page)

```
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

AlloViz.AlloViz.Elements.Edges.sum

Edges.**sum**(*axis: Axis | None = 0*, *skipna: bool = True*, *numeric_only: bool = False*, *min_count: int = 0*, ***kwargs*)

Return the sum of the values over the requested axis.

This is equivalent to the method `numpy.sum`.

Parameters

axis
[{index (0), columns (1)}] Axis for the function to be applied on. For *Series* this parameter is unused and defaults to 0.

For DataFrames, specifying `axis=None` will apply the aggregation across both axes.

New in version 2.0.0.

skipna
[bool, default True] Exclude NA/null values when computing the result.

numeric_only
[bool, default False] Include only float, int, boolean columns. Not implemented for Series.

min_count
[int, default 0] The required number of valid values to perform the operation. If fewer than `min_count` non-NA values are present the result will be NA.

****kwargs**
Additional keyword arguments to be passed to the function.

Returns

Series or scalar

See also:

`Series.sum`
Return the sum.

`Series.min`
Return the minimum.

`Series.max`
Return the maximum.

`Series.idxmin`
Return the index of the minimum.

`Series.idxmax`
Return the index of the maximum.

`DataFrame.sum`
Return the sum over the requested axis.

`DataFrame.min`
Return the minimum over the requested axis.

`DataFrame.max`
Return the maximum over the requested axis.

`DataFrame.idxmin`
Return the index of the minimum over the requested axis.

`DataFrame.idxmax`
Return the index of the maximum over the requested axis.

Examples

```
>>> idx = pd.MultiIndex.from_arrays([
...     ['warm', 'warm', 'cold', 'cold'],
...     ['dog', 'falcon', 'fish', 'spider']],
...     names=['blooded', 'animal'])
>>> s = pd.Series([4, 2, 0, 8], name='legs', index=idx)
>>> s
blooded  animal
warm     dog       4
         falcon    2
cold     fish      0
         spider    8
Name: legs, dtype: int64
```

```
>>> s.sum()
14
```

By default, the sum of an empty or all-NA Series is `0`.

```
>>> pd.Series([], dtype="float64").sum()  # min_count=0 is the default
0.0
```

This can be controlled with the `min_count` parameter. For example, if you'd like the sum of an empty series to be NaN, pass `min_count=1`.

```
>>> pd.Series([], dtype="float64").sum(min_count=1)
nan
```

Thanks to the `skipna` parameter, `min_count` handles all-NA and empty series identically.

```
>>> pd.Series([np.nan]).sum()
0.0
```

```
>>> pd.Series([np.nan]).sum(min_count=1)
nan
```

AlloViz.AlloViz.Elements.Edges.swapaxes

`Edges.`**`swapaxes`**(*axis1: int | Literal['index', 'columns', 'rows']*, *axis2: int | Literal['index', 'columns', 'rows']*, *copy: bool | None = None*) → None

Interchange axes and swap values axes appropriately.

Deprecated since version 2.1.0: `swapaxes` is deprecated and will be removed. Please use `transpose` instead.

Returns

same as input

Examples

Please see examples for `DataFrame.transpose()`.

AlloViz.AlloViz.Elements.Edges.swaplevel

`Edges.`**`swaplevel`**(*i: Axis = -2*, *j: Axis = -1*, *axis: Axis = 0*) → DataFrame

Swap levels i and j in a `MultiIndex`.

Default is to swap the two innermost levels of the index.

Parameters

i, j
: [int or str] Levels of the indices to be swapped. Can pass level name as string.

axis
: [{0 or 'index', 1 or 'columns'}, default 0] The axis to swap levels on. 0 or 'index' for row-wise, 1 or 'columns' for column-wise.

Returns

DataFrame
DataFrame with levels swapped in MultiIndex.

Examples

```
>>> df = pd.DataFrame(
...     {"Grade": ["A", "B", "A", "C"]},
...     index=[
...         ["Final exam", "Final exam", "Coursework", "Coursework"],
...         ["History", "Geography", "History", "Geography"],
...         ["January", "February", "March", "April"],
...     ],
... )
>>> df
                                Grade
Final exam  History     January      A
            Geography   February     B
Coursework  History     March        A
            Geography   April        C
```

In the following example, we will swap the levels of the indices. Here, we will swap the levels column-wise, but levels can be swapped row-wise in a similar manner. Note that column-wise is the default behaviour. By not supplying any arguments for i and j, we swap the last and second to last indices.

```
>>> df.swaplevel()
                                Grade
Final exam  January     History      A
            February    Geography    B
Coursework  March       History      A
            April       Geography    C
```

By supplying one argument, we can choose which index to swap the last index with. We can for example swap the first index with the last one as follows.

```
>>> df.swaplevel(0)
                                Grade
January     History     Final exam   A
February    Geography   Final exam   B
March       History     Coursework   A
April       Geography   Coursework   C
```

We can also define explicitly which indices we want to swap by supplying values for both i and j. Here, we for example swap the first and second indices.

```
>>> df.swaplevel(0, 1)
                                Grade
History     Final exam  January      A
Geography   Final exam  February     B
History     Coursework  March        A
Geography   Coursework  April        C
```

AlloViz.AlloViz.Elements.Edges.tail

Edges.**tail**(*n: int = 5*) → None

Return the last *n* rows.

This function returns last *n* rows from the object based on position. It is useful for quickly verifying data, for example, after sorting or appending rows.

For negative values of *n*, this function returns all rows except the first |*n*| rows, equivalent to `df[|n|:]`.

If n is larger than the number of rows, this function returns all rows.

Parameters

n
[int, default 5] Number of rows to select.

Returns

type of caller
The last *n* rows of the caller object.

See also:

DataFrame.head
The first *n* rows of the caller object.

Examples

```
>>> df = pd.DataFrame({'animal': ['alligator', 'bee', 'falcon', 'lion',
...                    'monkey', 'parrot', 'shark', 'whale', 'zebra']})
>>> df
      animal
0  alligator
1        bee
2     falcon
3       lion
4     monkey
5     parrot
6      shark
7      whale
8      zebra
```

Viewing the last 5 lines

```
>>> df.tail()
   animal
4  monkey
5  parrot
6   shark
7   whale
8   zebra
```

Viewing the last *n* lines (three in this case)

```
>>> df.tail(3)
  animal
6  shark
7  whale
8  zebra
```

For negative values of *n*

```
>>> df.tail(-3)
   animal
3    lion
4  monkey
5  parrot
6   shark
7   whale
8   zebra
```

AlloViz.AlloViz.Elements.Edges.take

Edges.**take**(*indices*, *axis: int | Literal['index', 'columns', 'rows'] = 0*, ***kwargs*) → None

Return the elements in the given *positional* indices along an axis.

This means that we are not indexing according to actual values in the index attribute of the object. We are indexing according to the actual position of the element in the object.

Parameters

indices
[array-like] An array of ints indicating which positions to take.

axis
[{0 or 'index', 1 or 'columns', None}, default 0] The axis on which to select elements. `0` means that we are selecting rows, `1` means that we are selecting columns. For *Series* this parameter is unused and defaults to 0.

****kwargs**
For compatibility with `numpy.take()`. Has no effect on the output.

Returns

same type as caller
An array-like containing the elements taken from the object.

See also:

`DataFrame.loc`
Select a subset of a DataFrame by labels.

`DataFrame.iloc`
Select a subset of a DataFrame by positions.

`numpy.take`
Take elements from an array along an axis.

Examples

```
>>> df = pd.DataFrame([('falcon', 'bird', 389.0),
...                    ('parrot', 'bird', 24.0),
...                    ('lion', 'mammal', 80.5),
...                    ('monkey', 'mammal', np.nan)],
...                   columns=['name', 'class', 'max_speed'],
...                   index=[0, 2, 3, 1])
>>> df
     name   class  max_speed
0  falcon    bird      389.0
2  parrot    bird       24.0
3    lion  mammal       80.5
1  monkey  mammal        NaN
```

Take elements at positions 0 and 3 along the axis 0 (default).

Note how the actual indices selected (0 and 1) do not correspond to our selected indices 0 and 3. That's because we are selecting the 0th and 3rd rows, not rows whose indices equal 0 and 3.

```
>>> df.take([0, 3])
     name   class  max_speed
0  falcon    bird      389.0
1  monkey  mammal        NaN
```

Take elements at indices 1 and 2 along the axis 1 (column selection).

```
>>> df.take([1, 2], axis=1)
    class  max_speed
0    bird      389.0
2    bird       24.0
3  mammal       80.5
1  mammal        NaN
```

We may take elements using negative integers for positive indices, starting from the end of the object, just like with Python lists.

```
>>> df.take([-1, -2])
     name   class  max_speed
1  monkey  mammal        NaN
3    lion  mammal       80.5
```

AlloViz.AlloViz.Elements.Edges.to_clipboard

Edges.**to_clipboard**(*excel: bool = True*, *sep: str | None = None*, ***kwargs*) → None

Copy object to the system clipboard.

Write a text representation of object to the system clipboard. This can be pasted into Excel, for example.

Parameters

excel

[bool, default True] Produce output in a csv format for easy pasting into excel.

- True, use the provided separator for csv pasting.

- False, write a string representation of the object to the clipboard.

sep
[str, default `'\t'`] Field delimiter.

****kwargs**
These parameters will be passed to DataFrame.to_csv.

See also:

`DataFrame.to_csv`
Write a DataFrame to a comma-separated values (csv) file.

`read_clipboard`
Read text from clipboard and pass to read_csv.

Notes

Requirements for your platform.

- Linux : *xclip*, or *xsel* (with *PyQt4* modules)
- Windows : none
- macOS : none

This method uses the processes developed for the package *pyperclip*. A solution to render any output string format is given in the examples.

Examples

Copy the contents of a DataFrame to the clipboard.

```
>>> df = pd.DataFrame([[1, 2, 3], [4, 5, 6]], columns=['A', 'B', 'C'])
```

```
>>> df.to_clipboard(sep=',')
... # Wrote the following to the system clipboard:
... # ,A,B,C
... # 0,1,2,3
... # 1,4,5,6
```

We can omit the index by passing the keyword *index* and setting it to false.

```
>>> df.to_clipboard(sep=',', index=False)
... # Wrote the following to the system clipboard:
... # A,B,C
... # 1,2,3
... # 4,5,6
```

Using the original *pyperclip* package for any string output format.

```
import pyperclip
html = df.style.to_html()
pyperclip.copy(html)
```

AlloViz.AlloViz.Elements.Edges.to_csv

Edges.**to_csv**(*path_or_buf: FilePath | WriteBuffer[bytes] | WriteBuffer[str] | None = None*, *sep: str = ','*, *na_rep: str = ''*, *float_format: str | Callable | None = None*, *columns: Sequence[Hashable] | None = None*, *header: bool_t | list[str] = True*, *index: bool_t = True*, *index_label: IndexLabel | None = None*, *mode: str = 'w'*, *encoding: str | None = None*, *compression: CompressionOptions = 'infer'*, *quoting: int | None = None*, *quotechar: str = '"'*, *lineterminator: str | None = None*, *chunksize: int | None = None*, *date_format: str | None = None*, *doublequote: bool_t = True*, *escapechar: str | None = None*, *decimal: str = '.'*, *errors: OpenFileErrors = 'strict'*, *storage_options: StorageOptions | None = None*) → str | None

Write object to a comma-separated values (csv) file.

Parameters

path_or_buf
[str, path object, file-like object, or None, default None] String, path object (implementing os.PathLike[str]), or file-like object implementing a write() function. If None, the result is returned as a string. If a non-binary file object is passed, it should be opened with *newline=''*, disabling universal newlines. If a binary file object is passed, *mode* might need to contain a *'b'*.

Changed in version 1.2.0: Support for binary file objects was introduced.

sep
[str, default ','] String of length 1. Field delimiter for the output file.

na_rep
[str, default ''] Missing data representation.

float_format
[str, Callable, default None] Format string for floating point numbers. If a Callable is given, it takes precedence over other numeric formatting parameters, like decimal.

columns
[sequence, optional] Columns to write.

header
[bool or list of str, default True] Write out the column names. If a list of strings is given it is assumed to be aliases for the column names.

index
[bool, default True] Write row names (index).

index_label
[str or sequence, or False, default None] Column label for index column(s) if desired. If None is given, and *header* and *index* are True, then the index names are used. A sequence should be given if the object uses MultiIndex. If False do not print fields for index names. Use index_label=False for easier importing in R.

mode
[{'w', 'x', 'a'}, default 'w'] Forwarded to either *open(mode=)* or *fsspec.open(mode=)* to control the file opening. Typical values include:

- 'w', truncate the file first.
- 'x', exclusive creation, failing if the file already exists.
- 'a', append to the end of file if it exists.

encoding
[str, optional] A string representing the encoding to use in the output file, defaults to 'utf-8'. *encoding* is not supported if *path_or_buf* is a non-binary file object.

compression
[str or dict, default 'infer'] For on-the-fly compression of the output data. If 'infer' and 'path_or_buf' is path-like, then detect compression from the following extensions: '.gz', '.bz2', '.zip', '.xz', '.zst', '.tar', '.tar.gz', '.tar.xz' or '.tar.bz2' (otherwise no compression). Set to `None` for no compression. Can also be a dict with key `'method'` set to one of {`'zip'`, `'gzip'`, `'bz2'`, `'zstd'`, `'xz'`, `'tar'`} and other key-value pairs are forwarded to `zipfile.ZipFile`, `gzip.GzipFile`, `bz2.BZ2File`, `zstandard.ZstdCompressor`, `lzma.LZMAFile` or `tarfile.TarFile`, respectively. As an example, the following could be passed for faster compression and to create a reproducible gzip archive: `compression={'method': 'gzip', 'compresslevel': 1, 'mtime': 1}`.

New in version 1.5.0: Added support for *.tar* files.

May be a dict with key 'method' as compression mode and other entries as additional compression options if compression mode is 'zip'.

Passing compression options as keys in dict is supported for compression modes 'gzip', 'bz2', 'zstd', and 'zip'.

Changed in version 1.2.0: Compression is supported for binary file objects.

Changed in version 1.2.0: Previous versions forwarded dict entries for 'gzip' to *gzip.open* instead of *gzip.GzipFile* which prevented setting *mtime*.

quoting
[optional constant from csv module] Defaults to csv.QUOTE_MINIMAL. If you have set a *float_format* then floats are converted to strings and thus csv.QUOTE_NONNUMERIC will treat them as non-numeric.

quotechar
[str, default '"'] String of length 1. Character used to quote fields.

lineterminator
[str, optional] The newline character or character sequence to use in the output file. Defaults to *os.linesep*, which depends on the OS in which this method is called ('\n' for linux, '\r\n' for Windows, i.e.).

Changed in version 1.5.0: Previously was line_terminator, changed for consistency with read_csv and the standard library 'csv' module.

chunksize
[int or None] Rows to write at a time.

date_format
[str, default None] Format string for datetime objects.

doublequote
[bool, default True] Control quoting of *quotechar* inside a field.

escapechar
[str, default None] String of length 1. Character used to escape *sep* and *quotechar* when appropriate.

decimal
[str, default '.'] Character recognized as decimal separator. E.g. use ',' for European data.

errors
[str, default 'strict'] Specifies how encoding and decoding errors are to be handled. See the errors argument for `open()` for a full list of options.

storage_options
[dict, optional] Extra options that make sense for a particular storage connection, e.g. host, port, username, password, etc. For HTTP(S) URLs the key-value pairs are forwarded to `urllib.request.Request` as header options. For other URLs (e.g. starting with "s3://", and "gcs://") the key-value pairs are forwarded to `fsspec.open`. Please see `fsspec` and `urllib` for more details, and for more examples on storage options refer here.

New in version 1.2.0.

Returns

None or str
If path_or_buf is None, returns the resulting csv format as a string. Otherwise returns None.

See also:

`read_csv`
Load a CSV file into a DataFrame.

`to_excel`
Write DataFrame to an Excel file.

Examples

```
>>> df = pd.DataFrame({'name': ['Raphael', 'Donatello'],
...                    'mask': ['red', 'purple'],
...                    'weapon': ['sai', 'bo staff']})
>>> df.to_csv(index=False)
'name,mask,weapon\nRaphael,red,sai\nDonatello,purple,bo staff\n'
```

Create 'out.zip' containing 'out.csv'

```
>>> compression_opts = dict(method='zip',
...                         archive_name='out.csv')
>>> df.to_csv('out.zip', index=False,
...           compression=compression_opts)
```

To write a csv file to a new folder or nested folder you will first need to create it using either Pathlib or os:

```
>>> from pathlib import Path
>>> filepath = Path('folder/subfolder/out.csv')
>>> filepath.parent.mkdir(parents=True, exist_ok=True)
>>> df.to_csv(filepath)
```

```
>>> import os
>>> os.makedirs('folder/subfolder', exist_ok=True)
>>> df.to_csv('folder/subfolder/out.csv')
```

AlloViz.AlloViz.Elements.Edges.to_dict

Edges.**to_dict**(*orient: ~typing.Literal['dict', 'list', 'series', 'split', 'tight', 'records', 'index'] = 'dict', into: type[dict] = <class 'dict'>, index: bool = True*) → dict | list[dict]

Convert the DataFrame to a dictionary.

The type of the key-value pairs can be customized with the parameters (see below).

Parameters

orient
[str {'dict', 'list', 'series', 'split', 'tight', 'records', 'index'}] Determines the type of the values of the dictionary.

- 'dict' (default) : dict like {column -> {index -> value}}
- 'list' : dict like {column -> [values]}
- 'series' : dict like {column -> Series(values)}
- 'split' : dict like {'index' -> [index], 'columns' -> [columns], 'data' -> [values]}
- 'tight' : dict like {'index' -> [index], 'columns' -> [columns], 'data' -> [values], 'index_names' -> [index.names], 'column_names' -> [column.names]}
- 'records' : list like [{column -> value}, ... , {column -> value}]
- 'index' : dict like {index -> {column -> value}}

New in version 1.4.0: 'tight' as an allowed value for the `orient` argument

into
[class, default dict] The collections.abc.Mapping subclass used for all Mappings in the return value. Can be the actual class or an empty instance of the mapping type you want. If you want a collections.defaultdict, you must pass it initialized.

index
[bool, default True] Whether to include the index item (and index_names item if *orient* is 'tight') in the returned dictionary. Can only be `False` when *orient* is 'split' or 'tight'.

New in version 2.0.0.

Returns

dict, list or collections.abc.Mapping
Return a collections.abc.Mapping object representing the DataFrame. The resulting transformation depends on the *orient* parameter.

See also:

`DataFrame.from_dict`
Create a DataFrame from a dictionary.

`DataFrame.to_json`
Convert a DataFrame to JSON format.

Examples

```
>>> df = pd.DataFrame({'col1': [1, 2],
...                    'col2': [0.5, 0.75]},
...                   index=['row1', 'row2'])
>>> df
      col1  col2
row1     1  0.50
row2     2  0.75
>>> df.to_dict()
{'col1': {'row1': 1, 'row2': 2}, 'col2': {'row1': 0.5, 'row2': 0.75}}
```

You can specify the return orientation.

```
>>> df.to_dict('series')
{'col1': row1    1
         row2    2
Name: col1, dtype: int64,
'col2': row1    0.50
        row2    0.75
Name: col2, dtype: float64}
```

```
>>> df.to_dict('split')
{'index': ['row1', 'row2'], 'columns': ['col1', 'col2'],
 'data': [[1, 0.5], [2, 0.75]]}
```

```
>>> df.to_dict('records')
[{'col1': 1, 'col2': 0.5}, {'col1': 2, 'col2': 0.75}]
```

```
>>> df.to_dict('index')
{'row1': {'col1': 1, 'col2': 0.5}, 'row2': {'col1': 2, 'col2': 0.75}}
```

```
>>> df.to_dict('tight')
{'index': ['row1', 'row2'], 'columns': ['col1', 'col2'],
 'data': [[1, 0.5], [2, 0.75]], 'index_names': [None], 'column_names': [None]}
```

You can also specify the mapping type.

```
>>> from collections import OrderedDict, defaultdict
>>> df.to_dict(into=OrderedDict)
OrderedDict([('col1', OrderedDict([('row1', 1), ('row2', 2)])),
             ('col2', OrderedDict([('row1', 0.5), ('row2', 0.75)]))])
```

If you want a *defaultdict*, you need to initialize it:

```
>>> dd = defaultdict(list)
>>> df.to_dict('records', into=dd)
[defaultdict(<class 'list'>, {'col1': 1, 'col2': 0.5}),
 defaultdict(<class 'list'>, {'col1': 2, 'col2': 0.75})]
```

AlloViz.AlloViz.Elements.Edges.to_excel

Edges.**to_excel**(*excel_writer: FilePath | WriteExcelBuffer | ExcelWriter*, *sheet_name: str = 'Sheet1'*, *na_rep: str = ''*, *float_format: str | None = None*, *columns: Sequence[Hashable] | None = None*, *header: Sequence[Hashable] | bool_t = True*, *index: bool_t = True*, *index_label: IndexLabel | None = None*, *startrow: int = 0*, *startcol: int = 0*, *engine: Literal['openpyxl', 'xlsxwriter'] | None = None*, *merge_cells: bool_t = True*, *inf_rep: str = 'inf'*, *freeze_panes: tuple[int, int] | None = None*, *storage_options: StorageOptions | None = None*, *engine_kwargs: dict[str, Any] | None = None*) → None

Write object to an Excel sheet.

To write a single object to an Excel .xlsx file it is only necessary to specify a target file name. To write to multiple sheets it is necessary to create an *ExcelWriter* object with a target file name, and specify a sheet in the file to write to.

Multiple sheets may be written to by specifying unique *sheet_name*. With all data written to the file it is necessary to save the changes. Note that creating an *ExcelWriter* object with a file name that already exists will result in the contents of the existing file being erased.

Parameters

excel_writer
[path-like, file-like, or ExcelWriter object] File path or existing ExcelWriter.

sheet_name
[str, default 'Sheet1'] Name of sheet which will contain DataFrame.

na_rep
[str, default ''] Missing data representation.

float_format
[str, optional] Format string for floating point numbers. For example `float_format="%.2f"` will format 0.1234 to 0.12.

columns
[sequence or list of str, optional] Columns to write.

header
[bool or list of str, default True] Write out the column names. If a list of string is given it is assumed to be aliases for the column names.

index
[bool, default True] Write row names (index).

index_label
[str or sequence, optional] Column label for index column(s) if desired. If not specified, and *header* and *index* are True, then the index names are used. A sequence should be given if the DataFrame uses MultiIndex.

startrow
[int, default 0] Upper left cell row to dump data frame.

startcol
[int, default 0] Upper left cell column to dump data frame.

engine
[str, optional] Write engine to use, 'openpyxl' or 'xlsxwriter'. You can also set this via the options `io.excel.xlsx.writer` or `io.excel.xlsm.writer`.

merge_cells
[bool, default True] Write MultiIndex and Hierarchical Rows as merged cells.

inf_rep
[str, default 'inf'] Representation for infinity (there is no native representation for infinity in Excel).

freeze_panes
[tuple of int (length 2), optional] Specifies the one-based bottommost row and rightmost column that is to be frozen.

storage_options
[dict, optional] Extra options that make sense for a particular storage connection, e.g. host, port, username, password, etc. For HTTP(S) URLs the key-value pairs are forwarded to `urllib.request.Request` as header options. For other URLs (e.g. starting with "s3://", and "gcs://") the key-value pairs are forwarded to `fsspec.open`. Please see `fsspec` and `urllib` for more details, and for more examples on storage options refer here.

New in version 1.2.0.

engine_kwargs
[dict, optional] Arbitrary keyword arguments passed to excel engine.

See also:

to_csv
Write DataFrame to a comma-separated values (csv) file.

ExcelWriter
Class for writing DataFrame objects into excel sheets.

read_excel
Read an Excel file into a pandas DataFrame.

read_csv
Read a comma-separated values (csv) file into DataFrame.

io.formats.style.Styler.to_excel
Add styles to Excel sheet.

Notes

For compatibility with `to_csv()`, to_excel serializes lists and dicts to strings before writing.

Once a workbook has been saved it is not possible to write further data without rewriting the whole workbook.

Examples

Create, write to and save a workbook:

```
>>> df1 = pd.DataFrame([['a', 'b'], ['c', 'd']],
...                    index=['row 1', 'row 2'],
...                    columns=['col 1', 'col 2'])
>>> df1.to_excel("output.xlsx")
```

To specify the sheet name:

```
>>> df1.to_excel("output.xlsx",
...              sheet_name='Sheet_name_1')
```

If you wish to write to more than one sheet in the workbook, it is necessary to specify an ExcelWriter object:

```
>>> df2 = df1.copy()
>>> with pd.ExcelWriter('output.xlsx') as writer:
...     df1.to_excel(writer, sheet_name='Sheet_name_1')
...     df2.to_excel(writer, sheet_name='Sheet_name_2')
```

ExcelWriter can also be used to append to an existing Excel file:

```
>>> with pd.ExcelWriter('output.xlsx',
...                     mode='a') as writer:
...     df1.to_excel(writer, sheet_name='Sheet_name_3')
```

To set the library that is used to write the Excel file, you can pass the *engine* keyword (the default engine is automatically chosen depending on the file extension):

```
>>> df1.to_excel('output1.xlsx', engine='xlsxwriter')
```

AlloViz.AlloViz.Elements.Edges.to_feather

`Edges.to_feather`(*path: FilePath | WriteBuffer[bytes]*, ***kwargs*) → None

Write a DataFrame to the binary Feather format.

Parameters

path
: [str, path object, file-like object] String, path object (implementing `os.PathLike[str]`), or file-like object implementing a binary `write()` function. If a string or a path, it will be used as Root Directory path when writing a partitioned dataset.

****kwargs**
: Additional keywords passed to `pyarrow.feather.write_feather()`. This includes the *compression*, *compression_level*, *chunksize* and *version* keywords.

Notes

This function writes the dataframe as a feather file. Requires a default index. For saving the DataFrame with your custom index use a method that supports custom indices e.g. *to_parquet*.

Examples

```
>>> df = pd.DataFrame([[1, 2, 3], [4, 5, 6]])
>>> df.to_feather("file.feather")
```

AlloViz.AlloViz.Elements.Edges.to_gbq

Edges.**to_gbq**(*destination_table: str*, *project_id: str | None = None*, *chunksize: int | None = None*, *reauth: bool = False*, *if_exists: ToGbqIfexist = 'fail'*, *auth_local_webserver: bool = True*, *table_schema: list[dict[str, str]] | None = None*, *location: str | None = None*, *progress_bar: bool = True*, *credentials=None*) → None

Write a DataFrame to a Google BigQuery table.

This function requires the pandas-gbq package.

See the How to authenticate with Google BigQuery guide for authentication instructions.

Parameters

destination_table
[str] Name of table to be written, in the form `dataset.tablename`.

project_id
[str, optional] Google BigQuery Account project ID. Optional when available from the environment.

chunksize
[int, optional] Number of rows to be inserted in each chunk from the dataframe. Set to `None` to load the whole dataframe at once.

reauth
[bool, default False] Force Google BigQuery to re-authenticate the user. This is useful if multiple accounts are used.

if_exists
[str, default 'fail'] Behavior when the destination table exists. Value can be one of:

`'fail'`
If table exists raise pandas_gbq.gbq.TableCreationError.

`'replace'`
If table exists, drop it, recreate it, and insert data.

`'append'`
If table exists, insert data. Create if does not exist.

auth_local_webserver
[bool, default True] Use the local webserver flow instead of the console flow when getting user credentials.

New in version 0.2.0 of pandas-gbq.

Changed in version 1.5.0: Default value is changed to `True`. Google has deprecated the `auth_local_webserver = False` "out of band" (copy-paste) flow.

table_schema
[list of dicts, optional] List of BigQuery table fields to which according DataFrame columns conform to, e.g. `[{'name': 'col1', 'type': 'STRING'},...]`. If schema is not provided, it will be generated according to dtypes of DataFrame columns. See BigQuery API documentation on available names of a field.

New in version 0.3.1 of pandas-gbq.

location
[str, optional] Location where the load job should run. See the BigQuery locations documentation for a list of available locations. The location must match that of the target dataset.

New in version 0.5.0 of pandas-gbq.

progress_bar
[bool, default True] Use the library *tqdm* to show the progress bar for the upload, chunk by chunk.

New in version 0.5.0 of pandas-gbq.

credentials
[google.auth.credentials.Credentials, optional] Credentials for accessing Google APIs. Use this parameter to override default credentials, such as to use Compute Engine `google.auth.compute_engine.Credentials` or Service Account `google.oauth2.service_account.Credentials` directly.

New in version 0.8.0 of pandas-gbq.

See also:

pandas_gbq.to_gbq
This function in the pandas-gbq library.

read_gbq
Read a DataFrame from Google BigQuery.

Examples

Example taken from Google BigQuery documentation

```
>>> project_id = "my-project"
>>> table_id = 'my_dataset.my_table'
>>> df = pd.DataFrame({
...                    "my_string": ["a", "b", "c"],
...                    "my_int64": [1, 2, 3],
...                    "my_float64": [4.0, 5.0, 6.0],
...                    "my_bool1": [True, False, True],
...                    "my_bool2": [False, True, False],
...                    "my_dates": pd.date_range("now", periods=3),
...                    }
...                    )
```

```
>>> df.to_gbq(table_id, project_id=project_id)
```

AlloViz.AlloViz.Elements.Edges.to_hdf

Edges.**to_hdf**(*path_or_buf: FilePath | HDFStore, key: str, mode: Literal['a', 'w', 'r+'] = 'a', complevel: int | None = None, complib: Literal['zlib', 'lzo', 'bzip2', 'blosc'] | None = None, append: bool_t = False, format: Literal['fixed', 'table'] | None = None, index: bool_t = True, min_itemsize: int | dict[str, int] | None = None, nan_rep=None, dropna: bool_t | None = None, data_columns: Literal[True] | list[str] | None = None, errors: OpenFileErrors = 'strict', encoding: str = 'UTF-8'*) → None

Write the contained data to an HDF5 file using HDFStore.

Hierarchical Data Format (HDF) is self-describing, allowing an application to interpret the structure and contents of a file with no outside information. One HDF file can hold a mix of related objects which can be accessed as a group or as individual objects.

In order to add another DataFrame or Series to an existing HDF file please use append mode and a different a key.

Warning: One can store a subclass of `DataFrame` or `Series` to HDF5, but the type of the subclass is lost upon storing.

For more information see the user guide.

Parameters

path_or_buf
[str or pandas.HDFStore] File path or HDFStore object.

key
[str] Identifier for the group in the store.

mode
[{'a', 'w', 'r+'}, default 'a'] Mode to open file:

- 'w': write, a new file is created (an existing file with the same name would be deleted).
- 'a': append, an existing file is opened for reading and writing, and if the file does not exist it is created.
- 'r+': similar to 'a', but the file must already exist.

complevel
[{0-9}, default None] Specifies a compression level for data. A value of 0 or None disables compression.

complib
[{'zlib', 'lzo', 'bzip2', 'blosc'}, default 'zlib'] Specifies the compression library to be used. These additional compressors for Blosc are supported (default if no compressor specified: 'blosc:blosclz'): {'blosc:blosclz', 'blosc:lz4', 'blosc:lz4hc', 'blosc:snappy', 'blosc:zlib', 'blosc:zstd'}. Specifying a compression library which is not available issues a ValueError.

append
[bool, default False] For Table formats, append the input data to the existing.

format
[{'fixed', 'table', None}, default 'fixed'] Possible values:

- 'fixed': Fixed format. Fast writing/reading. Not-appendable, nor searchable.
- 'table': Table format. Write as a PyTables Table structure which may perform worse but allow more flexible operations like searching / selecting subsets of the data.
- If None, pd.get_option('io.hdf.default_format') is checked, followed by fallback to "fixed".

index
[bool, default True] Write DataFrame index as a column.

min_itemsize
[dict or int, optional] Map column names to minimum string sizes for columns.

nan_rep
[Any, optional] How to represent null values as str. Not allowed with append=True.

dropna
[bool, default False, optional] Remove missing values.

data_columns
[list of columns or True, optional] List of columns to create as indexed data columns for on-disk queries, or True to use all columns. By default only the axes of the object are indexed. See Query via data columns. for more information. Applicable only to format='table'.

errors
[str, default 'strict'] Specifies how encoding and decoding errors are to be handled. See the errors argument for `open()` for a full list of options.

encoding
[str, default "UTF-8"]

See also:

`read_hdf`
Read from HDF file.

`DataFrame.to_orc`
Write a DataFrame to the binary orc format.

`DataFrame.to_parquet`
Write a DataFrame to the binary parquet format.

`DataFrame.to_sql`
Write to a SQL table.

`DataFrame.to_feather`
Write out feather-format for DataFrames.

`DataFrame.to_csv`
Write out to a csv file.

Examples

```
>>> df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]},
...                   index=['a', 'b', 'c'])
>>> df.to_hdf('data.h5', key='df', mode='w')
```

We can add another object to the same file:

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s.to_hdf('data.h5', key='s')
```

Reading from HDF file:

```
>>> pd.read_hdf('data.h5', 'df')
A  B
a  1  4
b  2  5
c  3  6
>>> pd.read_hdf('data.h5', 's')
0    1
1    2
```

(continues on next page)

(continued from previous page)

```
2    3
3    4
dtype: int64
```

AlloViz.AlloViz.Elements.Edges.to_html

Edges.**to_html**(*buf: FilePath | WriteBuffer[str] | None = None*, *columns: Axes | None = None*, *col_space: ColspaceArgType | None = None*, *header: bool = True*, *index: bool = True*, *na_rep: str = 'NaN'*, *formatters: FormattersType | None = None*, *float_format: FloatFormatType | None = None*, *sparsify: bool | None = None*, *index_names: bool = True*, *justify: str | None = None*, *max_rows: int | None = None*, *max_cols: int | None = None*, *show_dimensions: bool | str = False*, *decimal: str = '.'*, *bold_rows: bool = True*, *classes: str | list | tuple | None = None*, *escape: bool = True*, *notebook: bool = False*, *border: int | bool | None = None*, *table_id: str | None = None*, *render_links: bool = False*, *encoding: str | None = None*) → str | None

Render a DataFrame as an HTML table.

Parameters

buf
[str, Path or StringIO-like, optional, default None] Buffer to write to. If None, the output is returned as a string.

columns
[array-like, optional, default None] The subset of columns to write. Writes all columns by default.

col_space
[str or int, list or dict of int or str, optional] The minimum width of each column in CSS length units. An int is assumed to be px units..

header
[bool, optional] Whether to print column labels, default True.

index
[bool, optional, default True] Whether to print index (row) labels.

na_rep
[str, optional, default 'NaN'] String representation of `NaN` to use.

formatters
[list, tuple or dict of one-param. functions, optional] Formatter functions to apply to columns' elements by position or name. The result of each function must be a unicode string. List/tuple must be of length equal to the number of columns.

float_format
[one-parameter function, optional, default None] Formatter function to apply to columns' elements if they are floats. This function must return a unicode string and will be applied only to the non-`NaN` elements, with `NaN` being handled by `na_rep`.

Changed in version 1.2.0.

sparsify
[bool, optional, default True] Set to False for a DataFrame with a hierarchical index to print every multiindex key at each row.

index_names
[bool, optional, default True] Prints the names of the indexes.

justify
[str, default None] How to justify the column labels. If None uses the option from the print configuration (controlled by set_option), 'right' out of the box. Valid values are

- left
- right
- center
- justify
- justify-all
- start
- end
- inherit
- match-parent
- initial
- unset.

max_rows
[int, optional] Maximum number of rows to display in the console.

max_cols
[int, optional] Maximum number of columns to display in the console.

show_dimensions
[bool, default False] Display DataFrame dimensions (number of rows by number of columns).

decimal
[str, default '.'] Character recognized as decimal separator, e.g. ',' in Europe.

bold_rows
[bool, default True] Make the row labels bold in the output.

classes
[str or list or tuple, default None] CSS class(es) to apply to the resulting html table.

escape
[bool, default True] Convert the characters <, >, and & to HTML-safe sequences.

notebook
[{True, False}, default False] Whether the generated HTML is for IPython Notebook.

border
[int] A `border=border` attribute is included in the opening <*table*> tag. Default `pd.options.display.html.border`.

table_id
[str, optional] A css id is included in the opening <*table*> tag if specified.

render_links
[bool, default False] Convert URLs to HTML links.

encoding
[str, default "utf-8"] Set character encoding.

Returns

str or None
If buf is None, returns the result as a string. Otherwise returns None.

See also:

to_string
Convert DataFrame to a string.

Examples

```
>>> df = pd.DataFrame(data={'col1': [1, 2], 'col2': [4, 3]})
>>> html_string = '''<table border="1" class="dataframe">
...   <thead>
...     <tr style="text-align: right;">
...       <th></th>
...       <th>col1</th>
...       <th>col2</th>
...     </tr>
...   </thead>
...   <tbody>
...     <tr>
...       <th>0</th>
...       <td>1</td>
...       <td>4</td>
...     </tr>
...     <tr>
...       <th>1</th>
...       <td>2</td>
...       <td>3</td>
...     </tr>
...   </tbody>
... </table>'''
>>> assert html_string == df.to_html()
```

AlloViz.AlloViz.Elements.Edges.to_json

Edges.**to_json**(*path_or_buf: FilePath | WriteBuffer[bytes] | WriteBuffer[str] | None = None*, *orient: Literal['split', 'records', 'index', 'table', 'columns', 'values'] | None = None*, *date_format: str | None = None*, *double_precision: int = 10*, *force_ascii: bool_t = True*, *date_unit: TimeUnit = 'ms'*, *default_handler: Callable[[Any], JSONSerializable] | None = None*, *lines: bool_t = False*, *compression: CompressionOptions = 'infer'*, *index: bool_t | None = None*, *indent: int | None = None*, *storage_options: StorageOptions | None = None*, *mode: Literal['a', 'w'] = 'w'*) → str | None

Convert the object to a JSON string.

Note NaN's and None will be converted to null and datetime objects will be converted to UNIX timestamps.

Parameters

path_or_buf
[str, path object, file-like object, or None, default None] String, path object (implementing os.PathLike[str]), or file-like object implementing a write() function. If None, the result is returned as a string.

orient
[str] Indication of expected JSON string format.

- Series:
  - default is 'index'
  - allowed values are: {'split', 'records', 'index', 'table'}.
- DataFrame:
  - default is 'columns'
  - allowed values are: {'split', 'records', 'index', 'columns', 'values', 'table'}.
- The format of the JSON string:
  - 'split' : dict like {'index' -> [index], 'columns' -> [columns], 'data' -> [values]}
  - 'records' : list like [{column -> value}, ... , {column -> value}]
  - 'index' : dict like {index -> {column -> value}}
  - 'columns' : dict like {column -> {index -> value}}
  - 'values' : just the values array
  - 'table' : dict like {'schema': {schema}, 'data': {data}}

  Describing the data, where data component is like `orient='records'`.

date_format
[{None, 'epoch', 'iso'}] Type of date conversion. 'epoch' = epoch milliseconds, 'iso' = ISO8601. The default depends on the *orient*. For `orient='table'`, the default is 'iso'. For all other orients, the default is 'epoch'.

double_precision
[int, default 10] The number of decimal places to use when encoding floating point values. The possible maximal value is 15. Passing double_precision greater than 15 will raise a ValueError.

force_ascii
[bool, default True] Force encoded string to be ASCII.

date_unit
[str, default 'ms' (milliseconds)] The time unit to encode to, governs timestamp and ISO8601 precision. One of 's', 'ms', 'us', 'ns' for second, millisecond, microsecond, and nanosecond respectively.

default_handler
[callable, default None] Handler to call if object cannot otherwise be converted to a suitable format for JSON. Should receive a single argument which is the object to convert and return a serialisable object.

lines
[bool, default False] If 'orient' is 'records' write out line-delimited json format. Will throw ValueError if incorrect 'orient' since others are not list-like.

compression
[str or dict, default 'infer'] For on-the-fly compression of the output data. If 'infer' and 'path_or_buf' is path-like, then detect compression from the following extensions: '.gz', '.bz2', '.zip', '.xz', '.zst', '.tar', '.tar.gz', '.tar.xz' or '.tar.bz2' (otherwise no compression). Set to `None` for no compression. Can also be a dict with key `'method'` set

to one of {'zip', 'gzip', 'bz2', 'zstd', 'xz', 'tar'} and other key-value pairs are forwarded to zipfile.ZipFile, gzip.GzipFile, bz2.BZ2File, zstandard.ZstdCompressor, lzma.LZMAFile or tarfile.TarFile, respectively. As an example, the following could be passed for faster compression and to create a reproducible gzip archive: compression={'method': 'gzip', 'compresslevel': 1, 'mtime': 1}.

New in version 1.5.0: Added support for *.tar* files.

Changed in version 1.4.0: Zstandard support.

index
[bool or None, default None] The index is only used when 'orient' is 'split', 'index', 'column', or 'table'. Of these, 'index' and 'column' do not support *index=False*.

indent
[int, optional] Length of whitespace used to indent each record.

storage_options
[dict, optional] Extra options that make sense for a particular storage connection, e.g. host, port, username, password, etc. For HTTP(S) URLs the key-value pairs are forwarded to urllib.request.Request as header options. For other URLs (e.g. starting with "s3://", and "gcs://") the key-value pairs are forwarded to fsspec.open. Please see fsspec and urllib for more details, and for more examples on storage options refer here.

New in version 1.2.0.

mode
[str, default 'w' (writing)] Specify the IO mode for output when supplying a path_or_buf. Accepted args are 'w' (writing) and 'a' (append) only. mode='a' is only supported when lines is True and orient is 'records'.

Returns

None or str
If path_or_buf is None, returns the resulting json format as a string. Otherwise returns None.

See also:

read_json
Convert a JSON string to pandas object.

Notes

The behavior of indent=0 varies from the stdlib, which does not indent the output but does insert newlines. Currently, indent=0 and the default indent=None are equivalent in pandas, though this may change in a future release.

orient='table' contains a 'pandas_version' field under 'schema'. This stores the version of *pandas* used in the latest revision of the schema.

Examples

```
>>> from json import loads, dumps
>>> df = pd.DataFrame(
...     [["a", "b"], ["c", "d"]],
...     index=["row 1", "row 2"],
...     columns=["col 1", "col 2"],
... )
```

```
>>> result = df.to_json(orient="split")
>>> parsed = loads(result)
>>> dumps(parsed, indent=4)
{
    "columns": [
        "col 1",
        "col 2"
    ],
    "index": [
        "row 1",
        "row 2"
    ],
    "data": [
        [
            "a",
            "b"
        ],
        [
            "c",
            "d"
        ]
    ]
}
```

Encoding/decoding a Dataframe using `'records'` formatted JSON. Note that index labels are not preserved with this encoding.

```
>>> result = df.to_json(orient="records")
>>> parsed = loads(result)
>>> dumps(parsed, indent=4)
[
    {
        "col 1": "a",
        "col 2": "b"
    },
    {
        "col 1": "c",
        "col 2": "d"
    }
]
```

Encoding/decoding a Dataframe using `'index'` formatted JSON:

```
>>> result = df.to_json(orient="index")
```

(continues on next page)

(continued from previous page)

```
>>> parsed = loads(result)
>>> dumps(parsed, indent=4)
{
    "row 1": {
        "col 1": "a",
        "col 2": "b"
    },
    "row 2": {
        "col 1": "c",
        "col 2": "d"
    }
}
```

Encoding/decoding a Dataframe using `'columns'` formatted JSON:

```
>>> result = df.to_json(orient="columns")
>>> parsed = loads(result)
>>> dumps(parsed, indent=4)
{
    "col 1": {
        "row 1": "a",
        "row 2": "c"
    },
    "col 2": {
        "row 1": "b",
        "row 2": "d"
    }
}
```

Encoding/decoding a Dataframe using `'values'` formatted JSON:

```
>>> result = df.to_json(orient="values")
>>> parsed = loads(result)
>>> dumps(parsed, indent=4)
[
    [
        "a",
        "b"
    ],
    [
        "c",
        "d"
    ]
]
```

Encoding with Table Schema:

```
>>> result = df.to_json(orient="table")
>>> parsed = loads(result)
>>> dumps(parsed, indent=4)
{
    "schema": {
        "fields": [
```

(continues on next page)

(continued from previous page)

```
            {
                "name": "index",
                "type": "string"
            },
            {
                "name": "col 1",
                "type": "string"
            },
            {
                "name": "col 2",
                "type": "string"
            }
        ],
        "primaryKey": [
            "index"
        ],
        "pandas_version": "1.4.0"
    },
    "data": [
        {
            "index": "row 1",
            "col 1": "a",
            "col 2": "b"
        },
        {
            "index": "row 2",
            "col 1": "c",
            "col 2": "d"
        }
    ]
}
```

AlloViz.AlloViz.Elements.Edges.to_latex

Edges.**to_latex**(*buf: FilePath | WriteBuffer[str] | None = None*, *columns: Sequence[Hashable] | None = None*, *header: bool_t | list[str] = True*, *index: bool_t = True*, *na_rep: str = 'NaN'*, *formatters: FormattersType | None = None*, *float_format: FloatFormatType | None = None*, *sparsify: bool_t | None = None*, *index_names: bool_t = True*, *bold_rows: bool_t = False*, *column_format: str | None = None*, *longtable: bool_t | None = None*, *escape: bool_t | None = None*, *encoding: str | None = None*, *decimal: str = '.'*, *multicolumn: bool_t | None = None*, *multicolumn_format: str | None = None*, *multirow: bool_t | None = None*, *caption: str | tuple[str, str] | None = None*, *label: str | None = None*, *position: str | None = None*) → str | None

Render object to a LaTeX tabular, longtable, or nested table.

Requires `\usepackage{{booktabs}}`. The output can be copy/pasted into a main LaTeX document or read from an external file with `\input{{table.tex}}`.

Changed in version 1.2.0: Added position argument, changed meaning of caption argument.

Changed in version 2.0.0: Refactored to use the Styler implementation via jinja2 templating.

Parameters

buf
[str, Path or StringIO-like, optional, default None] Buffer to write to. If None, the output is returned as a string.

columns
[list of label, optional] The subset of columns to write. Writes all columns by default.

header
[bool or list of str, default True] Write out the column names. If a list of strings is given, it is assumed to be aliases for the column names.

index
[bool, default True] Write row names (index).

na_rep
[str, default 'NaN'] Missing data representation.

formatters
[list of functions or dict of {{str: function}}, optional] Formatter functions to apply to columns' elements by position or name. The result of each function must be a unicode string. List must be of length equal to the number of columns.

float_format
[one-parameter function or str, optional, default None] Formatter for floating point numbers. For example `float_format="%.2f"` and `float_format="{{:0.2f}}".format` will both result in 0.1234 being formatted as 0.12.

sparsify
[bool, optional] Set to False for a DataFrame with a hierarchical index to print every multiindex key at each row. By default, the value will be read from the config module.

index_names
[bool, default True] Prints the names of the indexes.

bold_rows
[bool, default False] Make the row labels bold in the output.

column_format
[str, optional] The columns format as specified in LaTeX table format e.g. 'rcl' for 3 columns. By default, 'l' will be used for all columns except columns of numbers, which default to 'r'.

longtable
[bool, optional] Use a longtable environment instead of tabular. Requires adding a usepackage{{longtable}} to your LaTeX preamble. By default, the value will be read from the pandas config module, and set to *True* if the option `styler.latex.environment` is *"longtable"*.

Changed in version 2.0.0: The pandas option affecting this argument has changed.

escape
[bool, optional] By default, the value will be read from the pandas config module and set to *True* if the option `styler.format.escape` is *"latex"*. When set to False prevents from escaping latex special characters in column names.

Changed in version 2.0.0: The pandas option affecting this argument has changed, as has the default value to *False*.

encoding
[str, optional] A string representing the encoding to use in the output file, defaults to 'utf-8'.

decimal
[str, default '.'] Character recognized as decimal separator, e.g. ',' in Europe.

multicolumn
[bool, default True] Use multicolumn to enhance MultiIndex columns. The default will be read from the config module, and is set as the option `styler.sparse.columns`.

Changed in version 2.0.0: The pandas option affecting this argument has changed.

multicolumn_format
[str, default 'r'] The alignment for multicolumns, similar to *column_format* The default will be read from the config module, and is set as the option `styler.latex.multicol_align`.

Changed in version 2.0.0: The pandas option affecting this argument has changed, as has the default value to "r".

multirow
[bool, default True] Use multirow to enhance MultiIndex rows. Requires adding a usepackage{{multirow}} to your LaTeX preamble. Will print centered labels (instead of top-aligned) across the contained rows, separating groups via clines. The default will be read from the pandas config module, and is set as the option `styler.sparse.index`.

Changed in version 2.0.0: The pandas option affecting this argument has changed, as has the default value to *True*.

caption
[str or tuple, optional] Tuple (full_caption, short_caption), which results in `\caption[short_caption]{{full_caption}}`; if a single string is passed, no short caption will be set.

Changed in version 1.2.0: Optionally allow caption to be a tuple `(full_caption, short_caption)`.

label
[str, optional] The LaTeX label to be placed inside `\label{{}}` in the output. This is used with `\ref{{}}` in the main `.tex` file.

position
[str, optional] The LaTeX positional argument for tables, to be placed after `\begin{{}}` in the output.

New in version 1.2.0.

Returns

str or None
If buf is None, returns the result as a string. Otherwise returns None.

See also:

`io.formats.style.Styler.to_latex`
Render a DataFrame to LaTeX with conditional formatting.

`DataFrame.to_string`
Render a DataFrame to a console-friendly tabular output.

DataFrame.to_html
Render a DataFrame as an HTML table.

Notes

As of v2.0.0 this method has changed to use the Styler implementation as part of `Styler.to_latex()` via `jinja2` templating. This means that `jinja2` is a requirement, and needs to be installed, for this method to function. It is advised that users switch to using Styler, since that implementation is more frequently updated and contains much more flexibility with the output.

Examples

Convert a general DataFrame to LaTeX with formatting:

```
>>> df = pd.DataFrame(dict(name=['Raphael', 'Donatello'],
...                        age=[26, 45],
...                        height=[181.23, 177.65]))
>>> print(df.to_latex(index=False,
...                   formatters={"name": str.upper},
...                   float_format="{:.1f}".format,
... ))
\begin{tabular}{lrr}
\toprule
name & age & height \\
\midrule
RAPHAEL & 26 & 181.2 \\
DONATELLO & 45 & 177.7 \\
\bottomrule
\end{tabular}
```

AlloViz.AlloViz.Elements.Edges.to_markdown

Edges.**to_markdown**(*buf: FilePath | WriteBuffer[str] | None = None*, *mode: str = 'wt'*, *index: bool = True*, *storage_options: StorageOptions | None = None*, ***kwargs*) → str | None

Print DataFrame in Markdown-friendly format.

Parameters

buf
[str, Path or StringIO-like, optional, default None] Buffer to write to. If None, the output is returned as a string.

mode
[str, optional] Mode in which file is opened, "wt" by default.

index
[bool, optional, default True] Add index (row) labels.

storage_options
[dict, optional] Extra options that make sense for a particular storage connection, e.g. host, port, username, password, etc. For HTTP(S) URLs the key-value pairs are forwarded to `urllib.request.Request` as header options. For other URLs (e.g.

starting with "s3://", and "gcs://") the key-value pairs are forwarded to `fsspec.open`. Please see `fsspec` and `urllib` for more details, and for more examples on storage options refer here.

New in version 1.2.0.

****kwargs**
These parameters will be passed to tabulate.

Returns

str
DataFrame in Markdown-friendly format.

Notes

Requires the tabulate package.

Examples

```
>>> df = pd.DataFrame(
...     data={"animal_1": ["elk", "pig"], "animal_2": ["dog", "quetzal"]}
... )
>>> print(df.to_markdown())
|    | animal_1   | animal_2   |
|---:|:-----------|:-----------|
|  0 | elk        | dog        |
|  1 | pig        | quetzal    |
```

Output markdown with a tabulate option.

```
>>> print(df.to_markdown(tablefmt="grid"))
+----+------------+------------+
|    | animal_1   | animal_2   |
+====+============+============+
|  0 | elk        | dog        |
+----+------------+------------+
|  1 | pig        | quetzal    |
+----+------------+------------+
```

AlloViz.AlloViz.Elements.Edges.to_numpy

Edges.**to_numpy**(*dtype: npt.DTypeLike | None = None*, *copy: bool = False*, *na_value: object = _NoDefault.no_default*) → np.ndarray

Convert the DataFrame to a NumPy array.

By default, the dtype of the returned array will be the common NumPy dtype of all types in the DataFrame. For example, if the dtypes are `float16` and `float32`, the results dtype will be `float32`. This may require copying data and coercing values, which may be expensive.

Parameters

dtype
[str or numpy.dtype, optional] The dtype to pass to `numpy.asarray()`.

copy
: [bool, default False] Whether to ensure that the returned value is not a view on another array. Note that `copy=False` does not *ensure* that `to_numpy()` is no-copy. Rather, `copy=True` ensure that a copy is made, even if not strictly necessary.

na_value
: [Any, optional] The value to use for missing values. The default value depends on *dtype* and the dtypes of the DataFrame columns.

Returns

numpy.ndarray

See also:

`Series.to_numpy`
: Similar method for Series.

Examples

```
>>> pd.DataFrame({"A": [1, 2], "B": [3, 4]}).to_numpy()
array([[1, 3],
       [2, 4]])
```

With heterogeneous data, the lowest common type will have to be used.

```
>>> df = pd.DataFrame({"A": [1, 2], "B": [3.0, 4.5]})
>>> df.to_numpy()
array([[1. , 3. ],
       [2. , 4.5]])
```

For a mix of numeric and non-numeric types, the output array will have object dtype.

```
>>> df['C'] = pd.date_range('2000', periods=2)
>>> df.to_numpy()
array([[1, 3.0, Timestamp('2000-01-01 00:00:00')],
       [2, 4.5, Timestamp('2000-01-02 00:00:00')]], dtype=object)
```

AlloViz.AlloViz.Elements.Edges.to_orc

`Edges.to_orc`(*path: FilePath | WriteBuffer[bytes] | None = None*, ***, *engine: Literal['pyarrow'] = 'pyarrow'*, *index: bool | None = None*, *engine_kwargs: dict[str, Any] | None = None*) → bytes | None

Write a DataFrame to the ORC format.

New in version 1.5.0.

Parameters

path
: [str, file-like object or None, default None] If a string, it will be used as Root Directory path when writing a partitioned dataset. By file-like object, we refer to objects with a write() method, such as a file handle (e.g. via builtin open function). If path is None, a bytes object is returned.

engine
: [{'pyarrow'}, default 'pyarrow'] ORC library to use. Pyarrow must be >= 7.0.0.

index
[bool, optional] If `True`, include the dataframe's index(es) in the file output. If `False`, they will not be written to the file. If `None`, similar to `infer` the dataframe's index(es) will be saved. However, instead of being saved as values, the RangeIndex will be stored as a range in the metadata so it doesn't require much space and is faster. Other indexes will be included as columns in the file output.

engine_kwargs
[dict[str, Any] or None, default None] Additional keyword arguments passed to `pyarrow.orc.write_table()`.

Returns

bytes if no path argument is provided else None

Raises

NotImplementedError
Dtype of one or more columns is category, unsigned integers, interval, period or sparse.

ValueError
engine is not pyarrow.

See also:

`read_orc`
Read a ORC file.

`DataFrame.to_parquet`
Write a parquet file.

`DataFrame.to_csv`
Write a csv file.

`DataFrame.to_sql`
Write to a sql table.

`DataFrame.to_hdf`
Write to hdf.

Notes

- Before using this function you should read the user guide about ORC and install optional dependencies.
- This function requires pyarrow library.
- For supported dtypes please refer to supported ORC features in Arrow.
- Currently timezones in datetime columns are not preserved when a dataframe is converted into ORC files.

Examples

```
>>> df = pd.DataFrame(data={'col1': [1, 2], 'col2': [4, 3]})
>>> df.to_orc('df.orc')
>>> pd.read_orc('df.orc')
   col1  col2
0     1     4
1     2     3
```

If you want to get a buffer to the orc content you can write it to io.BytesIO

```
>>> import io
>>> b = io.BytesIO(df.to_orc())
>>> b.seek(0)
0
>>> content = b.read()
```

AlloViz.AlloViz.Elements.Edges.to_parquet

Edges.**to_parquet**(*path: FilePath | WriteBuffer[bytes] | None = None*, *engine: Literal['auto', 'pyarrow', 'fastparquet'] = 'auto'*, *compression: str | None = 'snappy'*, *index: bool | None = None*, *partition_cols: list[str] | None = None*, *storage_options: StorageOptions | None = None*, ***kwargs*) → bytes | None

Write a DataFrame to the binary parquet format.

This function writes the dataframe as a parquet file. You can choose different parquet backends, and have the option of compression. See the user guide for more details.

Parameters

path
[str, path object, file-like object, or None, default None] String, path object (implementing `os.PathLike[str]`), or file-like object implementing a binary `write()` function. If None, the result is returned as bytes. If a string or path, it will be used as Root Directory path when writing a partitioned dataset.

Changed in version 1.2.0.

Previously this was "fname"

engine
[{'auto', 'pyarrow', 'fastparquet'}, default 'auto'] Parquet library to use. If 'auto', then the option `io.parquet.engine` is used. The default `io.parquet.engine` behavior is to try 'pyarrow', falling back to 'fastparquet' if 'pyarrow' is unavailable.

compression
[str or None, default 'snappy'] Name of the compression to use. Use `None` for no compression. Supported options: 'snappy', 'gzip', 'brotli', 'lz4', 'zstd'.

index
[bool, default None] If `True`, include the dataframe's index(es) in the file output. If `False`, they will not be written to the file. If `None`, similar to `True` the dataframe's index(es) will be saved. However, instead of being saved as values, the RangeIndex will be stored as a range in the metadata so it doesn't require much space and is faster. Other indexes will be included as columns in the file output.

partition_cols
[list, optional, default None] Column names by which to partition the dataset. Columns are partitioned in the order they are given. Must be None if path is not a string.

storage_options
[dict, optional] Extra options that make sense for a particular storage connection, e.g. host, port, username, password, etc. For HTTP(S) URLs the key-value pairs are forwarded to `urllib.request.Request` as header options. For other URLs (e.g. starting with "s3://", and "gcs://") the key-value pairs are forwarded to `fsspec.open`. Please see `fsspec` and `urllib` for more details, and for more examples on storage options refer here.

New in version 1.2.0.

****kwargs**
Additional arguments passed to the parquet library. See pandas io for more details.

Returns

bytes if no path argument is provided else None

See also:

`read_parquet`
Read a parquet file.

`DataFrame.to_orc`
Write an orc file.

`DataFrame.to_csv`
Write a csv file.

`DataFrame.to_sql`
Write to a sql table.

`DataFrame.to_hdf`
Write to hdf.

Notes

This function requires either the fastparquet or pyarrow library.

Examples

```
>>> df = pd.DataFrame(data={'col1': [1, 2], 'col2': [3, 4]})
>>> df.to_parquet('df.parquet.gzip',
...               compression='gzip')
>>> pd.read_parquet('df.parquet.gzip')
   col1  col2
0     1     3
1     2     4
```

If you want to get a buffer to the parquet content you can use a io.BytesIO object, as long as you don't use partition_cols, which creates multiple files.

```
>>> import io
>>> f = io.BytesIO()
>>> df.to_parquet(f)
>>> f.seek(0)
0
>>> content = f.read()
```

AlloViz.AlloViz.Elements.Edges.to_period

Edges.**to_period**(*freq: Frequency | None = None*, *axis: Axis = 0*, *copy: bool | None = None*) → DataFrame

Convert DataFrame from DatetimeIndex to PeriodIndex.

Convert DataFrame from DatetimeIndex to PeriodIndex with desired frequency (inferred from index if not passed).

Parameters

freq
[str, default] Frequency of the PeriodIndex.

axis
[{0 or 'index', 1 or 'columns'}, default 0] The axis to convert (the index by default).

copy
[bool, default True] If False then underlying input data is not copied.

Returns

DataFrame
The DataFrame has a PeriodIndex.

Examples

```
>>> idx = pd.to_datetime(
...     [
...         "2001-03-31 00:00:00",
...         "2002-05-31 00:00:00",
...         "2003-08-31 00:00:00",
...     ]
... )
```

```
>>> idx
DatetimeIndex(['2001-03-31', '2002-05-31', '2003-08-31'],
dtype='datetime64[ns]', freq=None)
```

```
>>> idx.to_period("M")
PeriodIndex(['2001-03', '2002-05', '2003-08'], dtype='period[M]')
```

For the yearly frequency

```
>>> idx.to_period("Y")
PeriodIndex(['2001', '2002', '2003'], dtype='period[A-DEC]')
```

AlloViz.AlloViz.Elements.Edges.to_pickle

Edges.**to_pickle**(*path: FilePath | WriteBuffer[bytes]*, *compression: CompressionOptions = 'infer'*, *protocol: int = 5*, *storage_options: StorageOptions | None = None*) → None

Pickle (serialize) object to file.

Parameters

path
[str, path object, or file-like object] String, path object (implementing `os.PathLike[str]`), or file-like object implementing a binary `write()` function. File path where the pickled object will be stored.

compression
[str or dict, default 'infer'] For on-the-fly compression of the output data. If 'infer' and 'path' is path-like, then detect compression from the following extensions: '.gz', '.bz2', '.zip', '.xz', '.zst', '.tar', '.tar.gz', '.tar.xz' or '.tar.bz2' (otherwise no compression). Set to `None` for no compression. Can also be a dict with key `'method'` set to one of {`'zip'`, `'gzip'`, `'bz2'`, `'zstd'`, `'xz'`, `'tar'`} and other key-value pairs are forwarded to `zipfile.ZipFile`, `gzip.GzipFile`, `bz2.BZ2File`, `zstandard.ZstdCompressor`, `lzma.LZMAFile` or `tarfile.TarFile`, respectively. As an example, the following could be passed for faster compression and to create a reproducible gzip archive: `compression={'method': 'gzip', 'compresslevel': 1, 'mtime': 1}`.

New in version 1.5.0: Added support for *.tar* files.

protocol
[int] Int which indicates which protocol should be used by the pickler, default HIGHEST_PROTOCOL (see [1] paragraph 12.1.2). The possible values are 0, 1, 2, 3, 4, 5. A negative value for the protocol parameter is equivalent to setting its value to HIGHEST_PROTOCOL.

storage_options
[dict, optional] Extra options that make sense for a particular storage connection, e.g. host, port, username, password, etc. For HTTP(S) URLs the key-value pairs are forwarded to `urllib.request.Request` as header options. For other URLs (e.g. starting with "s3://", and "gcs://") the key-value pairs are forwarded to `fsspec.open`. Please see `fsspec` and `urllib` for more details, and for more examples on storage options refer here.

New in version 1.2.0.

See also:

`read_pickle`
Load pickled pandas object (or any object) from file.

`DataFrame.to_hdf`
Write DataFrame to an HDF5 file.

`DataFrame.to_sql`
Write DataFrame to a SQL database.

`DataFrame.to_parquet`
Write a DataFrame to the binary parquet format.

Examples

```
>>> original_df = pd.DataFrame({"foo": range(5), "bar": range(5, 10)})
>>> original_df
   foo  bar
0    0    5
1    1    6
2    2    7
3    3    8
4    4    9
>>> original_df.to_pickle("./dummy.pkl")
```

```
>>> unpickled_df = pd.read_pickle("./dummy.pkl")
>>> unpickled_df
   foo  bar
0    0    5
1    1    6
2    2    7
3    3    8
4    4    9
```

AlloViz.AlloViz.Elements.Edges.to_records

`Edges.to_records`(*index: bool = True*, *column_dtypes=None*, *index_dtypes=None*) → recarray

Convert DataFrame to a NumPy record array.

Index will be included as the first field of the record array if requested.

Parameters

index
: [bool, default True] Include index in resulting record array, stored in 'index' field or using the index label, if set.

column_dtypes
: [str, type, dict, default None] If a string or type, the data type to store all columns. If a dictionary, a mapping of column names and indices (zero-indexed) to specific data types.

index_dtypes
: [str, type, dict, default None] If a string or type, the data type to store all index levels. If a dictionary, a mapping of index level names and indices (zero-indexed) to specific data types.

  This mapping is applied only if *index=True*.

Returns

numpy.recarray
: NumPy ndarray with the DataFrame labels as fields and each row of the DataFrame as entries.

See also:

`DataFrame.from_records`
: Convert structured or record ndarray to DataFrame.

numpy.recarray
An ndarray that allows field access using attributes, analogous to typed columns in a spreadsheet.

Examples

```
>>> df = pd.DataFrame({'A': [1, 2], 'B': [0.5, 0.75]},
...                   index=['a', 'b'])
>>> df
   A     B
a  1  0.50
b  2  0.75
>>> df.to_records()
rec.array([('a', 1, 0.5 ), ('b', 2, 0.75)],
          dtype=[('index', 'O'), ('A', '<i8'), ('B', '<f8')])
```

If the DataFrame index has no label then the recarray field name is set to 'index'. If the index has a label then this is used as the field name:

```
>>> df.index = df.index.rename("I")
>>> df.to_records()
rec.array([('a', 1, 0.5 ), ('b', 2, 0.75)],
          dtype=[('I', 'O'), ('A', '<i8'), ('B', '<f8')])
```

The index can be excluded from the record array:

```
>>> df.to_records(index=False)
rec.array([(1, 0.5 ), (2, 0.75)],
          dtype=[('A', '<i8'), ('B', '<f8')])
```

Data types can be specified for the columns:

```
>>> df.to_records(column_dtypes={"A": "int32"})
rec.array([('a', 1, 0.5 ), ('b', 2, 0.75)],
          dtype=[('I', 'O'), ('A', '<i4'), ('B', '<f8')])
```

As well as for the index:

```
>>> df.to_records(index_dtypes="<S2")
rec.array([(b'a', 1, 0.5 ), (b'b', 2, 0.75)],
          dtype=[('I', 'S2'), ('A', '<i8'), ('B', '<f8')])
```

```
>>> index_dtypes = f"<S{df.index.str.len().max()}"
>>> df.to_records(index_dtypes=index_dtypes)
rec.array([(b'a', 1, 0.5 ), (b'b', 2, 0.75)],
          dtype=[('I', 'S1'), ('A', '<i8'), ('B', '<f8')])
```

AlloViz.AlloViz.Elements.Edges.to_sql

Edges.**to_sql**(*name: str*, *con*, ***, *schema: str | None = None*, *if_exists: Literal['fail', 'replace', 'append'] = 'fail'*, *index: bool = True*, *index_label: Hashable | Sequence[Hashable] | None = None*, *chunksize: int | None = None*, *dtype: ExtensionDtype | str | dtype | Type[str | complex | bool | object] | dict[Hashable, ExtensionDtype | str | dtype | Type[str | complex | bool | object]] | None = None*, *method: Literal['multi'] | Callable | None = None*) → int | None

Write records stored in a DataFrame to a SQL database.

Databases supported by SQLAlchemy [1] are supported. Tables can be newly created, appended to, or overwritten.

Parameters

name
[str] Name of SQL table.

con
[sqlalchemy.engine.(Engine or Connection) or sqlite3.Connection] Using SQLAlchemy makes it possible to use any DB supported by that library. Legacy support is provided for sqlite3.Connection objects. The user is responsible for engine disposal and connection closure for the SQLAlchemy connectable. See here. If passing a sqlalchemy.engine.Connection which is already in a transaction, the transaction will not be committed. If passing a sqlite3.Connection, it will not be possible to roll back the record insertion.

schema
[str, optional] Specify the schema (if database flavor supports this). If None, use default schema.

if_exists
[{'fail', 'replace', 'append'}, default 'fail'] How to behave if the table already exists.

- fail: Raise a ValueError.
- replace: Drop the table before inserting new values.
- append: Insert new values to the existing table.

index
[bool, default True] Write DataFrame index as a column. Uses *index_label* as the column name in the table.

index_label
[str or sequence, default None] Column label for index column(s). If None is given (default) and *index* is True, then the index names are used. A sequence should be given if the DataFrame uses MultiIndex.

chunksize
[int, optional] Specify the number of rows in each batch to be written at a time. By default, all rows will be written at once.

dtype
[dict or scalar, optional] Specifying the datatype for columns. If a dictionary is used, the keys should be the column names and the values should be the SQLAlchemy types or strings for the sqlite3 legacy mode. If a scalar is provided, it will be applied to all columns.

method
[{None, 'multi', callable}, optional] Controls the SQL insertion clause used:

- None : Uses standard SQL `INSERT` clause (one per row).
- 'multi': Pass multiple values in a single `INSERT` clause.
- callable with signature `(pd_table, conn, keys, data_iter)`.

Details and a sample callable implementation can be found in the section insert method.

Returns

None or int
Number of rows affected by to_sql. None is returned if the callable passed into `method` does not return an integer number of rows.

The number of returned rows affected is the sum of the `rowcount` attribute of `sqlite3.Cursor` or SQLAlchemy connectable which may not reflect the exact number of written rows as stipulated in the sqlite3 or SQLAlchemy.

New in version 1.4.0.

Raises

ValueError
When the table already exists and *if_exists* is 'fail' (the default).

See also:

`read_sql`
Read a DataFrame from a table.

Notes

Timezone aware datetime columns will be written as `Timestamp with timezone` type with SQLAlchemy if supported by the database. Otherwise, the datetimes will be stored as timezone unaware timestamps local to the original timezone.

References

[1], [2]

Examples

Create an in-memory SQLite database.

```
>>> from sqlalchemy import create_engine
>>> engine = create_engine('sqlite://', echo=False)
```

Create a table from scratch with 3 rows.

```
>>> df = pd.DataFrame({'name' : ['User 1', 'User 2', 'User 3']})
>>> df
     name
0  User 1
1  User 2
2  User 3
```

```
>>> df.to_sql(name='users', con=engine)
3
>>> from sqlalchemy import text
>>> with engine.connect() as conn:
...     conn.execute(text("SELECT * FROM users")).fetchall()
[(0, 'User 1'), (1, 'User 2'), (2, 'User 3')]
```

An *sqlalchemy.engine.Connection* can also be passed to *con*:

```
>>> with engine.begin() as connection:
...     df1 = pd.DataFrame({'name' : ['User 4', 'User 5']})
...     df1.to_sql(name='users', con=connection, if_exists='append')
2
```

This is allowed to support operations that require that the same DBAPI connection is used for the entire operation.

```
>>> df2 = pd.DataFrame({'name' : ['User 6', 'User 7']})
>>> df2.to_sql(name='users', con=engine, if_exists='append')
2
>>> with engine.connect() as conn:
...     conn.execute(text("SELECT * FROM users")).fetchall()
[(0, 'User 1'), (1, 'User 2'), (2, 'User 3'),
 (0, 'User 4'), (1, 'User 5'), (0, 'User 6'),
 (1, 'User 7')]
```

Overwrite the table with just `df2`.

```
>>> df2.to_sql(name='users', con=engine, if_exists='replace',
...            index_label='id')
2
>>> with engine.connect() as conn:
...     conn.execute(text("SELECT * FROM users")).fetchall()
[(0, 'User 6'), (1, 'User 7')]
```

Use `method` to define a callable insertion method to do nothing if there's a primary key conflict on a table in a PostgreSQL database.

```
>>> from sqlalchemy.dialects.postgresql import insert
>>> def insert_on_conflict_nothing(table, conn, keys, data_iter):
...     # "a" is the primary key in "conflict_table"
...     data = [dict(zip(keys, row)) for row in data_iter]
...     stmt = insert(table.table).values(data).on_conflict_do_nothing(index_
↪elements=["a"])
...     result = conn.execute(stmt)
...     return result.rowcount
>>> df_conflict.to_sql(name="conflict_table", con=conn, if_exists="append",␣
↪method=insert_on_conflict_nothing)
0
```

For MySQL, a callable to update columns `b` and `c` if there's a conflict on a primary key.

```
>>> from sqlalchemy.dialects.mysql import insert
>>> def insert_on_conflict_update(table, conn, keys, data_iter):
```

(continues on next page)

(continued from previous page)

```
...     # update columns "b" and "c" on primary key conflict
...     data = [dict(zip(keys, row)) for row in data_iter]
...     stmt = (
...         insert(table.table)
...         .values(data)
...     )
...     stmt = stmt.on_duplicate_key_update(b=stmt.inserted.b, c=stmt.inserted.
↪c)
...     result = conn.execute(stmt)
...     return result.rowcount
>>> df_conflict.to_sql(name="conflict_table", con=conn, if_exists="append",␣
↪method=insert_on_conflict_update)
2
```

Specify the dtype (especially useful for integers with missing values). Notice that while pandas is forced to store the data as floating point, the database supports nullable integers. When fetching the data with Python, we get back integer scalars.

```
>>> df = pd.DataFrame({"A": [1, None, 2]})
>>> df
     A
0  1.0
1  NaN
2  2.0
```

```
>>> from sqlalchemy.types import Integer
>>> df.to_sql(name='integers', con=engine, index=False,
...           dtype={"A": Integer()})
3
```

```
>>> with engine.connect() as conn:
...   conn.execute(text("SELECT * FROM integers")).fetchall()
[(1,), (None,), (2,)]
```

AlloViz.AlloViz.Elements.Edges.to_stata

Edges.**to_stata**(*path: FilePath | WriteBuffer[bytes]*, ***, *convert_dates: dict[Hashable, str] | None = None*, *write_index: bool = True*, *byteorder: ToStataByteorder | None = None*, *time_stamp: datetime.datetime | None = None*, *data_label: str | None = None*, *variable_labels: dict[Hashable, str] | None = None*, *version: int | None = 114*, *convert_strl: Sequence[Hashable] | None = None*, *compression: CompressionOptions = 'infer'*, *storage_options: StorageOptions | None = None*, *value_labels: dict[Hashable, dict[float, str]] | None = None*) → None

Export DataFrame object to Stata dta format.

Writes the DataFrame to a Stata dataset file. “dta” files contain a Stata dataset.

Parameters

path

[str, path object, or buffer] String, path object (implementing `os.PathLike[str]`), or file-like object implementing a binary `write()` function.

convert_dates
[dict] Dictionary mapping columns containing datetime types to stata internal format to use when writing the dates. Options are 'tc', 'td', 'tm', 'tw', 'th', 'tq', 'ty'. Column can be either an integer or a name. Datetime columns that do not have a conversion type specified will be converted to 'tc'. Raises NotImplementedError if a datetime column has timezone information.

write_index
[bool] Write the index to Stata dataset.

byteorder
[str] Can be ">", "<", "little", or "big". default is *sys.byteorder*.

time_stamp
[datetime] A datetime to use as file creation date. Default is the current time.

data_label
[str, optional] A label for the data set. Must be 80 characters or smaller.

variable_labels
[dict] Dictionary containing columns as keys and variable labels as values. Each label must be 80 characters or smaller.

version
[{114, 117, 118, 119, None}, default 114] Version to use in the output dta file. Set to None to let pandas decide between 118 or 119 formats depending on the number of columns in the frame. Version 114 can be read by Stata 10 and later. Version 117 can be read by Stata 13 or later. Version 118 is supported in Stata 14 and later. Version 119 is supported in Stata 15 and later. Version 114 limits string variables to 244 characters or fewer while versions 117 and later allow strings with lengths up to 2,000,000 characters. Versions 118 and 119 support Unicode characters, and version 119 supports more than 32,767 variables.

Version 119 should usually only be used when the number of variables exceeds the capacity of dta format 118. Exporting smaller datasets in format 119 may have unintended consequences, and, as of November 2020, Stata SE cannot read version 119 files.

convert_strl
[list, optional] List of column names to convert to string columns to Stata StrL format. Only available if version is 117. Storing strings in the StrL format can produce smaller dta files if strings have more than 8 characters and values are repeated.

compression
[str or dict, default 'infer'] For on-the-fly compression of the output data. If 'infer' and 'path' is path-like, then detect compression from the following extensions: '.gz', '.bz2', '.zip', '.xz', '.zst', '.tar', '.tar.gz', '.tar.xz' or '.tar.bz2' (otherwise no compression). Set to `None` for no compression. Can also be a dict with key `'method'` set to one of {`'zip'`, `'gzip'`, `'bz2'`, `'zstd'`, `'xz'`, `'tar'`} and other key-value pairs are forwarded to `zipfile.ZipFile`, `gzip.GzipFile`, `bz2.BZ2File`, `zstandard.ZstdCompressor`, `lzma.LZMAFile` or `tarfile.TarFile`, respectively. As an example, the following could be passed for faster compression and to create a reproducible gzip archive: `compression={'method': 'gzip', 'compresslevel': 1, 'mtime': 1}`.

New in version 1.5.0: Added support for *.tar* files.

Changed in version 1.4.0: Zstandard support.

storage_options
[dict, optional] Extra options that make sense for a particular storage connection, e.g. host, port, username, password, etc. For HTTP(S) URLs the key-value pairs are forwarded to `urllib.request.Request` as header options. For other URLs (e.g. starting with "s3://", and "gcs://") the key-value pairs are forwarded to `fsspec.open`. Please see `fsspec` and `urllib` for more details, and for more examples on storage options refer here.

New in version 1.2.0.

value_labels
[dict of dicts] Dictionary containing columns as keys and dictionaries of column value to labels as values. Labels for a single variable must be 32,000 characters or smaller.

New in version 1.4.0.

Raises

NotImplementedError

- If datetimes contain timezone information
- Column dtype is not representable in Stata

ValueError

- Columns listed in convert_dates are neither datetime64[ns] or datetime.datetime
- Column listed in convert_dates is not in DataFrame
- Categorical label contains more than 32,000 characters

See also:

`read_stata`
Import Stata data files.

`io.stata.StataWriter`
Low-level writer for Stata data files.

`io.stata.StataWriter117`
Low-level writer for version 117 files.

Examples

```
>>> df = pd.DataFrame({'animal': ['falcon', 'parrot', 'falcon',
...                               'parrot'],
...                    'speed': [350, 18, 361, 15]})
>>> df.to_stata('animals.dta')
```

AlloViz.AlloViz.Elements.Edges.to_string

Edges.**to_string**(*buf: FilePath | WriteBuffer[str] | None = None*, *columns: Axes | None = None*, *col_space: int | list[int] | dict[Hashable, int] | None = None*, *header: bool | list[str] = True*, *index: bool = True*, *na_rep: str = 'NaN'*, *formatters: fmt.FormattersType | None = None*, *float_format: fmt.FloatFormatType | None = None*, *sparsify: bool | None = None*, *index_names: bool = True*, *justify: str | None = None*, *max_rows: int | None = None*, *max_cols: int | None = None*, *show_dimensions: bool = False*, *decimal: str = '.'*, *line_width: int | None = None*, *min_rows: int | None = None*, *max_colwidth: int | None = None*, *encoding: str | None = None*) → str | None

Render a DataFrame to a console-friendly tabular output.

Parameters

buf
[str, Path or StringIO-like, optional, default None] Buffer to write to. If None, the output is returned as a string.

columns
[array-like, optional, default None] The subset of columns to write. Writes all columns by default.

col_space
[int, list or dict of int, optional] The minimum width of each column. If a list of ints is given every integers corresponds with one column. If a dict is given, the key references the column, while the value defines the space to use..

header
[bool or list of str, optional] Write out the column names. If a list of columns is given, it is assumed to be aliases for the column names.

index
[bool, optional, default True] Whether to print index (row) labels.

na_rep
[str, optional, default 'NaN'] String representation of `NaN` to use.

formatters
[list, tuple or dict of one-param. functions, optional] Formatter functions to apply to columns' elements by position or name. The result of each function must be a unicode string. List/tuple must be of length equal to the number of columns.

float_format
[one-parameter function, optional, default None] Formatter function to apply to columns' elements if they are floats. This function must return a unicode string and will be applied only to the non-`NaN` elements, with `NaN` being handled by `na_rep`.

Changed in version 1.2.0.

sparsify
[bool, optional, default True] Set to False for a DataFrame with a hierarchical index to print every multiindex key at each row.

index_names
[bool, optional, default True] Prints the names of the indexes.

justify
[str, default None] How to justify the column labels. If None uses the option from the print configuration (controlled by set_option), 'right' out of the box. Valid values are

- left
- right
- center
- justify
- justify-all
- start
- end
- inherit
- match-parent
- initial
- unset.

max_rows
[int, optional] Maximum number of rows to display in the console.

max_cols
[int, optional] Maximum number of columns to display in the console.

show_dimensions
[bool, default False] Display DataFrame dimensions (number of rows by number of columns).

decimal
[str, default '.'] Character recognized as decimal separator, e.g. ',' in Europe.

line_width
[int, optional] Width to wrap a line in characters.

min_rows
[int, optional] The number of rows to display in the console in a truncated repr (when number of rows is above *max_rows*).

max_colwidth
[int, optional] Max width to truncate each column in characters. By default, no limit.

encoding
[str, default "utf-8"] Set character encoding.

Returns

str or None
If buf is None, returns the result as a string. Otherwise returns None.

See also:

to_html
Convert DataFrame to HTML.

Examples

```
>>> d = {'col1': [1, 2, 3], 'col2': [4, 5, 6]}
>>> df = pd.DataFrame(d)
>>> print(df.to_string())
   col1  col2
0     1     4
1     2     5
2     3     6
```

AlloViz.AlloViz.Elements.Edges.to_timestamp

Edges.**to_timestamp**(*freq: Frequency | None = None*, *how: ToTimestampHow = 'start'*, *axis: Axis = 0*, *copy: bool | None = None*) → DataFrame

Cast to DatetimeIndex of timestamps, at *beginning* of period.

Parameters

freq
[str, default frequency of PeriodIndex] Desired frequency.

how
[{'s', 'e', 'start', 'end'}] Convention for converting period to timestamp; start of period vs. end.

axis
[{0 or 'index', 1 or 'columns'}, default 0] The axis to convert (the index by default).

copy
[bool, default True] If False then underlying input data is not copied.

Returns

DataFrame
The DataFrame has a DatetimeIndex.

Examples

```
>>> idx = pd.PeriodIndex(['2023', '2024'], freq='Y')
>>> d = {'col1': [1, 2], 'col2': [3, 4]}
>>> df1 = pd.DataFrame(data=d, index=idx)
>>> df1
      col1   col2
2023     1      3
2024     2      4
```

The resulting timestamps will be at the beginning of the year in this case

```
>>> df1 = df1.to_timestamp()
>>> df1
            col1   col2
2023-01-01     1      3
2024-01-01     2      4
```

(continues on next page)

(continued from previous page)

```
>>> df1.index
DatetimeIndex(['2023-01-01', '2024-01-01'], dtype='datetime64[ns]', freq=None)
```

Using *freq* which is the offset that the Timestamps will have

```
>>> df2 = pd.DataFrame(data=d, index=idx)
>>> df2 = df2.to_timestamp(freq='M')
>>> df2
            col1   col2
2023-01-31     1      3
2024-01-31     2      4
>>> df2.index
DatetimeIndex(['2023-01-31', '2024-01-31'], dtype='datetime64[ns]', freq=None)
```

AlloViz.AlloViz.Elements.Edges.to_xarray

Edges.**to_xarray**()

Return an xarray object from the pandas object.

Returns

xarray.DataArray or xarray.Dataset
Data in the pandas structure converted to Dataset if the object is a DataFrame, or a DataArray if the object is a Series.

See also:

DataFrame.to_hdf
Write DataFrame to an HDF5 file.

DataFrame.to_parquet
Write a DataFrame to the binary parquet format.

Notes

See the xarray docs

Examples

```
>>> df = pd.DataFrame([('falcon', 'bird', 389.0, 2),
...                    ('parrot', 'bird', 24.0, 2),
...                    ('lion', 'mammal', 80.5, 4),
...                    ('monkey', 'mammal', np.nan, 4)],
...                   columns=['name', 'class', 'max_speed',
...                            'num_legs'])
>>> df
     name   class  max_speed  num_legs
0  falcon    bird      389.0         2
1  parrot    bird       24.0         2
2    lion  mammal       80.5         4
3  monkey  mammal        NaN         4
```

```
>>> df.to_xarray()
<xarray.Dataset>
Dimensions:    (index: 4)
Coordinates:
  * index      (index) int64 0 1 2 3
Data variables:
    name       (index) object 'falcon' 'parrot' 'lion' 'monkey'
    class      (index) object 'bird' 'bird' 'mammal' 'mammal'
    max_speed  (index) float64 389.0 24.0 80.5 nan
    num_legs   (index) int64 2 2 4 4
```

```
>>> df['max_speed'].to_xarray()
<xarray.DataArray 'max_speed' (index: 4)>
array([389. ,  24. ,  80.5,   nan])
Coordinates:
  * index    (index) int64 0 1 2 3
```

```
>>> dates = pd.to_datetime(['2018-01-01', '2018-01-01',
...                         '2018-01-02', '2018-01-02'])
>>> df_multiindex = pd.DataFrame({'date': dates,
...                               'animal': ['falcon', 'parrot',
...                                          'falcon', 'parrot'],
...                               'speed': [350, 18, 361, 15]})
>>> df_multiindex = df_multiindex.set_index(['date', 'animal'])
```

```
>>> df_multiindex
                   speed
date       animal
2018-01-01 falcon    350
           parrot     18
2018-01-02 falcon    361
           parrot     15
```

```
>>> df_multiindex.to_xarray()
<xarray.Dataset>
Dimensions:  (date: 2, animal: 2)
Coordinates:
  * date     (date) datetime64[ns] 2018-01-01 2018-01-02
  * animal   (animal) object 'falcon' 'parrot'
Data variables:
    speed    (date, animal) int64 350 18 361 15
```

AlloViz.AlloViz.Elements.Edges.to_xml

Edges.**to_xml**(*path_or_buffer: FilePath | WriteBuffer[bytes] | WriteBuffer[str] | None = None*, *index: bool = True*, *root_name: str | None = 'data'*, *row_name: str | None = 'row'*, *na_rep: str | None = None*, *attr_cols: list[str] | None = None*, *elem_cols: list[str] | None = None*, *namespaces: dict[str | None, str] | None = None*, *prefix: str | None = None*, *encoding: str = 'utf-8'*, *xml_declaration: bool | None = True*, *pretty_print: bool | None = True*, *parser: XMLParsers | None = 'lxml'*, *stylesheet: FilePath | ReadBuffer[str] | ReadBuffer[bytes] | None = None*, *compression: CompressionOptions = 'infer'*, *storage_options: StorageOptions | None = None*) → str | None

Render a DataFrame to an XML document.

New in version 1.3.0.

Parameters

path_or_buffer
[str, path object, file-like object, or None, default None] String, path object (implementing `os.PathLike[str]`), or file-like object implementing a `write()` function. If None, the result is returned as a string.

index
[bool, default True] Whether to include index in XML document.

root_name
[str, default 'data'] The name of root element in XML document.

row_name
[str, default 'row'] The name of row element in XML document.

na_rep
[str, optional] Missing data representation.

attr_cols
[list-like, optional] List of columns to write as attributes in row element. Hierarchical columns will be flattened with underscore delimiting the different levels.

elem_cols
[list-like, optional] List of columns to write as children in row element. By default, all columns output as children of row element. Hierarchical columns will be flattened with underscore delimiting the different levels.

namespaces
[dict, optional] All namespaces to be defined in root element. Keys of dict should be prefix names and values of dict corresponding URIs. Default namespaces should be given empty string key. For example,

```
namespaces = {"": "https://example.com"}
```

prefix
[str, optional] Namespace prefix to be used for every element and/or attribute in document. This should be one of the keys in `namespaces` dict.

encoding
[str, default 'utf-8'] Encoding of the resulting document.

xml_declaration
[bool, default True] Whether to include the XML declaration at start of document.

pretty_print
[bool, default True] Whether output should be pretty printed with indentation and line breaks.

parser
[{'lxml','etree'}, default 'lxml'] Parser module to use for building of tree. Only 'lxml' and 'etree' are supported. With 'lxml', the ability to use XSLT stylesheet is supported.

stylesheet
[str, path object or file-like object, optional] A URL, file-like object, or a raw string containing an XSLT script used to transform the raw XML output. Script should use layout of elements and attributes from original output. This argument requires `lxml` to be installed. Only XSLT 1.0 scripts and not later versions is currently supported.

compression
[str or dict, default 'infer'] For on-the-fly compression of the output data. If 'infer' and 'path_or_buffer' is path-like, then detect compression from the following extensions: '.gz', '.bz2', '.zip', '.xz', '.zst', '.tar', '.tar.gz', '.tar.xz' or '.tar.bz2' (otherwise no compression). Set to `None` for no compression. Can also be a dict with key `'method'` set to one of {`'zip'`, `'gzip'`, `'bz2'`, `'zstd'`, `'xz'`, `'tar'`} and other key-value pairs are forwarded to `zipfile.ZipFile`, `gzip.GzipFile`, `bz2.BZ2File`, `zstandard.ZstdCompressor`, `lzma.LZMAFile` or `tarfile.TarFile`, respectively. As an example, the following could be passed for faster compression and to create a reproducible gzip archive: `compression={'method': 'gzip', 'compresslevel': 1, 'mtime': 1}`.

New in version 1.5.0: Added support for *.tar* files.

Changed in version 1.4.0: Zstandard support.

storage_options
[dict, optional] Extra options that make sense for a particular storage connection, e.g. host, port, username, password, etc. For HTTP(S) URLs the key-value pairs are forwarded to `urllib.request.Request` as header options. For other URLs (e.g. starting with "s3://", and "gcs://") the key-value pairs are forwarded to `fsspec.open`. Please see `fsspec` and `urllib` for more details, and for more examples on storage options refer here.

Returns

None or str
If `io` is None, returns the resulting XML format as a string. Otherwise returns None.

See also:

to_json
Convert the pandas object to a JSON string.

to_html
Convert DataFrame to a html.

Examples

```
>>> df = pd.DataFrame({'shape': ['square', 'circle', 'triangle'],
...                    'degrees': [360, 360, 180],
...                    'sides': [4, np.nan, 3]})
```

```
>>> df.to_xml()
<?xml version='1.0' encoding='utf-8'?>
<data>
  <row>
    <index>0</index>
    <shape>square</shape>
    <degrees>360</degrees>
    <sides>4.0</sides>
  </row>
  <row>
    <index>1</index>
    <shape>circle</shape>
    <degrees>360</degrees>
    <sides/>
  </row>
  <row>
    <index>2</index>
    <shape>triangle</shape>
    <degrees>180</degrees>
    <sides>3.0</sides>
  </row>
</data>
```

```
>>> df.to_xml(attr_cols=[
...           'index', 'shape', 'degrees', 'sides'
...           ])
<?xml version='1.0' encoding='utf-8'?>
<data>
  <row index="0" shape="square" degrees="360" sides="4.0"/>
  <row index="1" shape="circle" degrees="360"/>
  <row index="2" shape="triangle" degrees="180" sides="3.0"/>
</data>
```

```
>>> df.to_xml(namespaces={"doc": "https://example.com"},
...           prefix="doc")
<?xml version='1.0' encoding='utf-8'?>
<doc:data xmlns:doc="https://example.com">
  <doc:row>
    <doc:index>0</doc:index>
    <doc:shape>square</doc:shape>
    <doc:degrees>360</doc:degrees>
    <doc:sides>4.0</doc:sides>
  </doc:row>
  <doc:row>
    <doc:index>1</doc:index>
    <doc:shape>circle</doc:shape>
```

(continues on next page)

(continued from previous page)

```
    <doc:degrees>360</doc:degrees>
    <doc:sides/>
  </doc:row>
  <doc:row>
    <doc:index>2</doc:index>
    <doc:shape>triangle</doc:shape>
    <doc:degrees>180</doc:degrees>
    <doc:sides>3.0</doc:sides>
  </doc:row>
</doc:data>
```

AlloViz.AlloViz.Elements.Edges.transform

Edges.**transform**(*func: AggFuncType*, *axis: Axis = 0*, **args*, ***kwargs*) → DataFrame

Call `func` on self producing a DataFrame with the same axis shape as self.

Parameters

func
[function, str, list-like or dict-like] Function to use for transforming the data. If a function, must either work when passed a DataFrame or when passed to DataFrame.apply. If func is both list-like and dict-like, dict-like behavior takes precedence.

Accepted combinations are:

- function
- string function name
- list-like of functions and/or function names, e.g. `[np.exp, 'sqrt']`
- dict-like of axis labels -> functions, function names or list-like of such.

axis
[{0 or 'index', 1 or 'columns'}, default 0] If 0 or 'index': apply function to each column. If 1 or 'columns': apply function to each row.

***args**
Positional arguments to pass to *func*.

****kwargs**
Keyword arguments to pass to *func*.

Returns

DataFrame
A DataFrame that must have the same length as self.

Raises

ValueError
[If the returned DataFrame has a different length than self.]

See also:

`DataFrame.agg`
Only perform aggregating type operations.

DataFrame.apply
Invoke function on a DataFrame.

Notes

Functions that mutate the passed object can produce unexpected behavior or errors and are not supported. See Mutating with User Defined Function (UDF) methods for more details.

Examples

```
>>> df = pd.DataFrame({'A': range(3), 'B': range(1, 4)})
>>> df
   A  B
0  0  1
1  1  2
2  2  3
>>> df.transform(lambda x: x + 1)
   A  B
0  1  2
1  2  3
2  3  4
```

Even though the resulting DataFrame must have the same length as the input DataFrame, it is possible to provide several input functions:

```
>>> s = pd.Series(range(3))
>>> s
0    0
1    1
2    2
dtype: int64
>>> s.transform([np.sqrt, np.exp])
       sqrt        exp
0  0.000000   1.000000
1  1.000000   2.718282
2  1.414214   7.389056
```

You can call transform on a GroupBy object:

```
>>> df = pd.DataFrame({
...     "Date": [
...         "2015-05-08", "2015-05-07", "2015-05-06", "2015-05-05",
...         "2015-05-08", "2015-05-07", "2015-05-06", "2015-05-05"],
...     "Data": [5, 8, 6, 1, 50, 100, 60, 120],
... })
>>> df
         Date  Data
0  2015-05-08     5
1  2015-05-07     8
2  2015-05-06     6
3  2015-05-05     1
4  2015-05-08    50
```

(continues on next page)

(continued from previous page)

```
5  2015-05-07   100
6  2015-05-06    60
7  2015-05-05   120
>>> df.groupby('Date')['Data'].transform('sum')
0     55
1    108
2     66
3    121
4     55
5    108
6     66
7    121
Name: Data, dtype: int64
```

```
>>> df = pd.DataFrame({
...     "c": [1, 1, 1, 2, 2, 2, 2],
...     "type": ["m", "n", "o", "m", "m", "n", "n"]
... })
>>> df
   c type
0  1    m
1  1    n
2  1    o
3  2    m
4  2    m
5  2    n
6  2    n
>>> df['size'] = df.groupby('c')['type'].transform(len)
>>> df
   c type size
0  1    m    3
1  1    n    3
2  1    o    3
3  2    m    4
4  2    m    4
5  2    n    4
6  2    n    4
```

AlloViz.AlloViz.Elements.Edges.transpose

Edges.**transpose**(**args*, *copy: bool = False*) → DataFrame

Transpose index and columns.

Reflect the DataFrame over its main diagonal by writing rows as columns and vice-versa. The property `T` is an accessor to the method *`transpose()`*.

Parameters

***args**
[tuple, optional] Accepted for compatibility with NumPy.

copy
[bool, default False] Whether to copy the data after transposing, even for

DataFrames with a single dtype.

Note that a copy is always required for mixed dtype DataFrames, or for DataFrames with any extension types.

Returns

DataFrame
The transposed DataFrame.

See also:

numpy.transpose
Permute the dimensions of a given array.

Notes

Transposing a DataFrame with mixed dtypes will result in a homogeneous DataFrame with the *object* dtype. In such a case, a copy of the data is always made.

Examples

Square DataFrame with homogeneous dtype

```
>>> d1 = {'col1': [1, 2], 'col2': [3, 4]}
>>> df1 = pd.DataFrame(data=d1)
>>> df1
   col1  col2
0     1     3
1     2     4
```

```
>>> df1_transposed = df1.T  # or df1.transpose()
>>> df1_transposed
      0  1
col1  1  2
col2  3  4
```

When the dtype is homogeneous in the original DataFrame, we get a transposed DataFrame with the same dtype:

```
>>> df1.dtypes
col1    int64
col2    int64
dtype: object
>>> df1_transposed.dtypes
0    int64
1    int64
dtype: object
```

Non-square DataFrame with mixed dtypes

```
>>> d2 = {'name': ['Alice', 'Bob'],
...       'score': [9.5, 8],
...       'employed': [False, True],
```

(continues on next page)

(continued from previous page)

```
...        'kids': [0, 0]}
>>> df2 = pd.DataFrame(data=d2)
>>> df2
    name  score  employed  kids
0  Alice    9.5     False     0
1    Bob    8.0      True     0
```

```
>>> df2_transposed = df2.T  # or df2.transpose()
>>> df2_transposed
              0     1
name      Alice   Bob
score       9.5   8.0
employed  False  True
kids          0     0
```

When the DataFrame has mixed dtypes, we get a transposed DataFrame with the *object* dtype:

```
>>> df2.dtypes
name         object
score       float64
employed       bool
kids          int64
dtype: object
>>> df2_transposed.dtypes
0    object
1    object
dtype: object
```

AlloViz.AlloViz.Elements.Edges.truediv

Edges.**truediv**(*other*, *axis: Axis = 'columns'*, *level=None*, *fill_value=None*)

Get Floating division of dataframe and other, element-wise (binary operator *truediv*).

Equivalent to `dataframe / other`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *rtruediv*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *floordiv*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other
: [scalar, sequence, Series, dict or DataFrame] Any single or multiple element data structure, or list-like object.

axis
: [{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns. (1 or 'columns'). For Series input, axis to match Series index on.

level
: [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value
[float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame
Result of the arithmetic operation.

See also:

`DataFrame.add`
Add DataFrames.

`DataFrame.sub`
Subtract DataFrames.

`DataFrame.mul`
Multiply DataFrames.

`DataFrame.div`
Divide DataFrames (float division).

`DataFrame.truediv`
Divide DataFrames (float division).

`DataFrame.floordiv`
Divide DataFrames (integer division).

`DataFrame.mod`
Calculate modulo (remainder after division).

`DataFrame.pow`
Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
```

(continues on next page)

(continued from previous page)

```
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a dictionary by axis.

```
>>> df.mul({'angles': 0, 'degrees': 2})
           angles  degrees
circle          0      720
triangle        0      360
rectangle       0      720
```

```
>>> df.mul({'circle': 0, 'triangle': 2, 'rectangle': 3}, axis='index')
           angles  degrees
circle          0        0
triangle        6      360
rectangle      12     1080
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

AlloViz.AlloViz.Elements.Edges.truncate

Edges.**truncate**(*before=None*, *after=None*, *axis: int | Literal['index', 'columns', 'rows'] | None = None*, *copy: bool | None = None*) → None

Truncate a Series or DataFrame before and after some index value.

This is a useful shorthand for boolean indexing based on index values above or below certain thresholds.

Parameters

before
[date, str, int] Truncate all rows before this index value.

after
[date, str, int] Truncate all rows after this index value.

axis
[{0 or 'index', 1 or 'columns'}, optional] Axis to truncate. Truncates the index (rows) by default. For *Series* this parameter is unused and defaults to 0.

copy
[bool, default is True,] Return a copy of the truncated section.

Returns

type of caller
The truncated Series or DataFrame.

See also:

DataFrame.loc
Select a subset of a DataFrame by label.

DataFrame.iloc
Select a subset of a DataFrame by position.

Notes

If the index being truncated contains only datetime values, *before* and *after* may be specified as strings instead of Timestamps.

Examples

```
>>> df = pd.DataFrame({'A': ['a', 'b', 'c', 'd', 'e'],
...                    'B': ['f', 'g', 'h', 'i', 'j'],
...                    'C': ['k', 'l', 'm', 'n', 'o']},
...                   index=[1, 2, 3, 4, 5])
>>> df
   A  B  C
1  a  f  k
2  b  g  l
3  c  h  m
4  d  i  n
5  e  j  o
```

```
>>> df.truncate(before=2, after=4)
   A  B  C
2  b  g  l
3  c  h  m
4  d  i  n
```

The columns of a DataFrame can be truncated.

```
>>> df.truncate(before="A", after="B", axis="columns")
   A  B
1  a  f
2  b  g
3  c  h
4  d  i
5  e  j
```

For Series, only rows can be truncated.

```
>>> df['A'].truncate(before=2, after=4)
2    b
3    c
4    d
Name: A, dtype: object
```

The index values in `truncate` can be datetimes or string dates.

```
>>> dates = pd.date_range('2016-01-01', '2016-02-01', freq='s')
>>> df = pd.DataFrame(index=dates, data={'A': 1})
>>> df.tail()
                     A
2016-01-31 23:59:56  1
2016-01-31 23:59:57  1
2016-01-31 23:59:58  1
2016-01-31 23:59:59  1
2016-02-01 00:00:00  1
```

```
>>> df.truncate(before=pd.Timestamp('2016-01-05'),
...             after=pd.Timestamp('2016-01-10')).tail()
                     A
2016-01-09 23:59:56  1
2016-01-09 23:59:57  1
2016-01-09 23:59:58  1
2016-01-09 23:59:59  1
2016-01-10 00:00:00  1
```

Because the index is a DatetimeIndex containing only dates, we can specify *before* and *after* as strings. They will be coerced to Timestamps before truncation.

```
>>> df.truncate('2016-01-05', '2016-01-10').tail()
                     A
2016-01-09 23:59:56  1
2016-01-09 23:59:57  1
2016-01-09 23:59:58  1
```

(continues on next page)

(continued from previous page)

```
2016-01-09 23:59:59  1
2016-01-10 00:00:00  1
```

Note that `truncate` assumes a 0 value for any unspecified time component (midnight). This differs from partial string slicing, which returns any partially matching dates.

```
>>> df.loc['2016-01-05':'2016-01-10', :].tail()
                     A
2016-01-10 23:59:55  1
2016-01-10 23:59:56  1
2016-01-10 23:59:57  1
2016-01-10 23:59:58  1
2016-01-10 23:59:59  1
```

AlloViz.AlloViz.Elements.Edges.tz_convert

Edges.**tz_convert**(*tz*, *axis: int | Literal['index', 'columns', 'rows'] = 0*, *level=None*, *copy: bool | None = None*) → None

Convert tz-aware axis to target time zone.

Parameters

tz
: [str or tzinfo object or None] Target time zone. Passing `None` will convert to UTC and remove the timezone information.

axis
: [{0 or 'index', 1 or 'columns'}, default 0] The axis to convert

level
: [int, str, default None] If axis is a MultiIndex, convert a specific level. Otherwise must be None.

copy
: [bool, default True] Also make a copy of the underlying data.

Returns

Series/DataFrame
: Object with time zone converted axis.

Raises

TypeError
: If the axis is tz-naive.

Examples

Change to another time zone:

```
>>> s = pd.Series(
...     [1],
...     index=pd.DatetimeIndex(['2018-09-15 01:30:00+02:00']),
... )
>>> s.tz_convert('Asia/Shanghai')
2018-09-15 07:30:00+08:00    1
dtype: int64
```

Pass None to convert to UTC and get a tz-naive index:

```
>>> s = pd.Series([1],
...     index=pd.DatetimeIndex(['2018-09-15 01:30:00+02:00']))
>>> s.tz_convert(None)
2018-09-14 23:30:00    1
dtype: int64
```

AlloViz.AlloViz.Elements.Edges.tz_localize

`Edges.tz_localize`(*tz*, *axis: Axis = 0*, *level=None*, *copy: bool_t | None = None*, *ambiguous: TimeAmbiguous = 'raise'*, *nonexistent: TimeNonexistent = 'raise'*) → Self

Localize tz-naive index of a Series or DataFrame to target time zone.

This operation localizes the Index. To localize the values in a timezone-naive Series, use `Series.dt.tz_localize()`.

Parameters

tz
: [str or tzinfo or None] Time zone to localize. Passing `None` will remove the time zone information and preserve local time.

axis
: [{0 or 'index', 1 or 'columns'}, default 0] The axis to localize

level
: [int, str, default None] If axis ia a MultiIndex, localize a specific level. Otherwise must be None.

copy
: [bool, default True] Also make a copy of the underlying data.

ambiguous
: ['infer', bool-ndarray, 'NaT', default 'raise'] When clocks moved backward due to DST, ambiguous times may arise. For example in Central European Time (UTC+01), when going from 03:00 DST to 02:00 non-DST, 02:30:00 local time occurs both at 00:30:00 UTC and at 01:30:00 UTC. In such a situation, the *ambiguous* parameter dictates how ambiguous times should be handled.

- 'infer' will attempt to infer fall dst-transition hours based on order
- bool-ndarray where True signifies a DST time, False designates a non-DST time (note that this flag is only applicable for ambiguous times)
- 'NaT' will return NaT where there are ambiguous times

- 'raise' will raise an AmbiguousTimeError if there are ambiguous times.

nonexistent
[str, default 'raise'] A nonexistent time does not exist in a particular timezone where clocks moved forward due to DST. Valid values are:

- 'shift_forward' will shift the nonexistent time forward to the closest existing time
- 'shift_backward' will shift the nonexistent time backward to the closest existing time
- 'NaT' will return NaT where there are nonexistent times
- timedelta objects will shift nonexistent times by the timedelta
- 'raise' will raise an NonExistentTimeError if there are nonexistent times.

Returns

Series/DataFrame
Same type as the input.

Raises

TypeError
If the TimeSeries is tz-aware and tz is not None.

Examples

Localize local times:

```
>>> s = pd.Series(
...     [1],
...     index=pd.DatetimeIndex(['2018-09-15 01:30:00']),
... )
>>> s.tz_localize('CET')
2018-09-15 01:30:00+02:00    1
dtype: int64
```

Pass None to convert to tz-naive index and preserve local time:

```
>>> s = pd.Series([1],
...     index=pd.DatetimeIndex(['2018-09-15 01:30:00+02:00']))
>>> s.tz_localize(None)
2018-09-15 01:30:00    1
dtype: int64
```

Be careful with DST changes. When there is sequential data, pandas can infer the DST time:

```
>>> s = pd.Series(range(7),
...               index=pd.DatetimeIndex(['2018-10-28 01:30:00',
...                                       '2018-10-28 02:00:00',
...                                       '2018-10-28 02:30:00',
...                                       '2018-10-28 02:00:00',
...                                       '2018-10-28 02:30:00',
...                                       '2018-10-28 03:00:00',
...                                       '2018-10-28 03:30:00']))
```

(continues on next page)

(continued from previous page)

```
>>> s.tz_localize('CET', ambiguous='infer')
2018-10-28 01:30:00+02:00    0
2018-10-28 02:00:00+02:00    1
2018-10-28 02:30:00+02:00    2
2018-10-28 02:00:00+01:00    3
2018-10-28 02:30:00+01:00    4
2018-10-28 03:00:00+01:00    5
2018-10-28 03:30:00+01:00    6
dtype: int64
```

In some cases, inferring the DST is impossible. In such cases, you can pass an ndarray to the ambiguous parameter to set the DST explicitly

```
>>> s = pd.Series(range(3),
...               index=pd.DatetimeIndex(['2018-10-28 01:20:00',
...                                       '2018-10-28 02:36:00',
...                                       '2018-10-28 03:46:00']))
>>> s.tz_localize('CET', ambiguous=np.array([True, True, False]))
2018-10-28 01:20:00+02:00    0
2018-10-28 02:36:00+02:00    1
2018-10-28 03:46:00+01:00    2
dtype: int64
```

If the DST transition causes nonexistent times, you can shift these dates forward or backward with a timedelta object or *'shift_forward'* or *'shift_backward'*.

```
>>> s = pd.Series(range(2),
...               index=pd.DatetimeIndex(['2015-03-29 02:30:00',
...                                       '2015-03-29 03:30:00']))
>>> s.tz_localize('Europe/Warsaw', nonexistent='shift_forward')
2015-03-29 03:00:00+02:00    0
2015-03-29 03:30:00+02:00    1
dtype: int64
>>> s.tz_localize('Europe/Warsaw', nonexistent='shift_backward')
2015-03-29 01:59:59.999999999+01:00    0
2015-03-29 03:30:00+02:00              1
dtype: int64
>>> s.tz_localize('Europe/Warsaw', nonexistent=pd.Timedelta('1H'))
2015-03-29 03:30:00+02:00    0
2015-03-29 03:30:00+02:00    1
dtype: int64
```

AlloViz.AlloViz.Elements.Edges.unstack

Edges.**unstack**(*level: IndexLabel = -1*, *fill_value=None*, *sort: bool = True*)

Pivot a level of the (necessarily hierarchical) index labels.

Returns a DataFrame having a new level of column labels whose inner-most level consists of the pivoted index labels.

If the index is not a MultiIndex, the output will be a Series (the analogue of stack when the columns are not a MultiIndex).

Parameters

level
[int, str, or list of these, default -1 (last level)] Level(s) of index to unstack, can pass level name.

fill_value
[int, str or dict] Replace NaN with this value if the unstack produces missing values.

sort
[bool, default True] Sort the level(s) in the resulting MultiIndex columns.

Returns

Series or DataFrame

See also:

`DataFrame.pivot`
Pivot a table based on column values.

`DataFrame.stack`
Pivot a level of the column labels (inverse operation from *unstack*).

Notes

Reference the user guide for more examples.

Examples

```
>>> index = pd.MultiIndex.from_tuples([('one', 'a'), ('one', 'b'),
...                                    ('two', 'a'), ('two', 'b')])
>>> s = pd.Series(np.arange(1.0, 5.0), index=index)
>>> s
one  a   1.0
     b   2.0
two  a   3.0
     b   4.0
dtype: float64
```

```
>>> s.unstack(level=-1)
     a   b
one  1.0  2.0
two  3.0  4.0
```

```
>>> s.unstack(level=0)
   one  two
a  1.0   3.0
b  2.0   4.0
```

```
>>> df = s.unstack(level=0)
>>> df.unstack()
one  a  1.0
     b  2.0
```

(continues on next page)

(continued from previous page)

```
two  a  3.0
     b  4.0
dtype: float64
```

AlloViz.AlloViz.Elements.Edges.update

Edges.**update**(*other*, *join: UpdateJoin = 'left'*, *overwrite: bool = True*, *filter_func=None*, *errors: IgnoreRaise = 'ignore'*) → None

Modify in place using non-NA values from another DataFrame.

Aligns on indices. There is no return value.

Parameters

other
[DataFrame, or object coercible into a DataFrame] Should have at least one matching index/column label with the original DataFrame. If a Series is passed, its name attribute must be set, and that will be used as the column name to align with the original DataFrame.

join
[{'left'}, default 'left'] Only left join is implemented, keeping the index and columns of the original object.

overwrite
[bool, default True] How to handle non-NA values for overlapping keys:

- True: overwrite original DataFrame's values with values from *other*.
- False: only update values that are NA in the original DataFrame.

filter_func
[callable(1d-array) -> bool 1d-array, optional] Can choose to replace values other than NA. Return True for values that should be updated.

errors
[{'raise', 'ignore'}, default 'ignore'] If 'raise', will raise a ValueError if the DataFrame and *other* both contain non-NA data in the same place.

Returns

None
This method directly changes calling object.

Raises

ValueError

- When *errors='raise'* and there's overlapping non-NA data.
- When *errors* is not either *'ignore'* or *'raise'*

NotImplementedError

- If *join != 'left'*

See also:

`dict.update`
Similar method for dictionaries.

DataFrame.merge
For column(s)-on-column(s) operations.

Examples

```
>>> df = pd.DataFrame({'A': [1, 2, 3],
...                    'B': [400, 500, 600]})
>>> new_df = pd.DataFrame({'B': [4, 5, 6],
...                        'C': [7, 8, 9]})
>>> df.update(new_df)
>>> df
   A  B
0  1  4
1  2  5
2  3  6
```

The DataFrame's length does not increase as a result of the update, only values at matching index/column labels are updated.

```
>>> df = pd.DataFrame({'A': ['a', 'b', 'c'],
...                    'B': ['x', 'y', 'z']})
>>> new_df = pd.DataFrame({'B': ['d', 'e', 'f', 'g', 'h', 'i']})
>>> df.update(new_df)
>>> df
   A  B
0  a  d
1  b  e
2  c  f
```

For Series, its name attribute must be set.

```
>>> df = pd.DataFrame({'A': ['a', 'b', 'c'],
...                    'B': ['x', 'y', 'z']})
>>> new_column = pd.Series(['d', 'e'], name='B', index=[0, 2])
>>> df.update(new_column)
>>> df
   A  B
0  a  d
1  b  y
2  c  e
>>> df = pd.DataFrame({'A': ['a', 'b', 'c'],
...                    'B': ['x', 'y', 'z']})
>>> new_df = pd.DataFrame({'B': ['d', 'e']}, index=[1, 2])
>>> df.update(new_df)
>>> df
   A  B
0  a  x
1  b  d
2  c  e
```

If *other* contains NaNs the corresponding values are not updated in the original dataframe.

```
>>> df = pd.DataFrame({'A': [1, 2, 3],
...                    'B': [400, 500, 600]})
>>> new_df = pd.DataFrame({'B': [4, np.nan, 6]})
>>> df.update(new_df)
>>> df
   A    B
0  1    4
1  2  500
2  3    6
```

AlloViz.AlloViz.Elements.Edges.value_counts

Edges.**value_counts**(*subset: IndexLabel | None = None*, *normalize: bool = False*, *sort: bool = True*, *ascending: bool = False*, *dropna: bool = True*) → Series

Return a Series containing the frequency of each distinct row in the Dataframe.

Parameters

subset
: [label or list of labels, optional] Columns to use when counting unique combinations.

normalize
: [bool, default False] Return proportions rather than frequencies.

sort
: [bool, default True] Sort by frequencies when True. Sort by DataFrame column values when False.

ascending
: [bool, default False] Sort in ascending order.

dropna
: [bool, default True] Don't include counts of rows that contain NA values.

  New in version 1.3.0.

Returns

Series

See also:

Series.value_counts
: Equivalent method on Series.

Notes

The returned Series will have a MultiIndex with one level per input column but an Index (non-multi) for a single label. By default, rows that contain any NA values are omitted from the result. By default, the resulting Series will be in descending order so that the first element is the most frequently-occurring row.

Examples

```
>>> df = pd.DataFrame({'num_legs': [2, 4, 4, 6],
...                    'num_wings': [2, 0, 0, 0]},
...                   index=['falcon', 'dog', 'cat', 'ant'])
>>> df
        num_legs  num_wings
falcon         2          2
dog            4          0
cat            4          0
ant            6          0
```

```
>>> df.value_counts()
num_legs  num_wings
4         0            2
2         2            1
6         0            1
Name: count, dtype: int64
```

```
>>> df.value_counts(sort=False)
num_legs  num_wings
2         2            1
4         0            2
6         0            1
Name: count, dtype: int64
```

```
>>> df.value_counts(ascending=True)
num_legs  num_wings
2         2            1
6         0            1
4         0            2
Name: count, dtype: int64
```

```
>>> df.value_counts(normalize=True)
num_legs  num_wings
4         0            0.50
2         2            0.25
6         0            0.25
Name: proportion, dtype: float64
```

With *dropna* set to *False* we can also count rows with NA values.

```
>>> df = pd.DataFrame({'first_name': ['John', 'Anne', 'John', 'Beth'],
...                    'middle_name': ['Smith', pd.NA, pd.NA, 'Louise']})
>>> df
  first_name middle_name
0       John       Smith
1       Anne        <NA>
2       John        <NA>
3       Beth      Louise
```

```
>>> df.value_counts()
first_name  middle_name
Beth        Louise         1
John        Smith          1
Name: count, dtype: int64
```

```
>>> df.value_counts(dropna=False)
first_name  middle_name
Anne        NaN            1
Beth        Louise         1
John        Smith          1
            NaN            1
Name: count, dtype: int64
```

```
>>> df.value_counts("first_name")
first_name
John    2
Anne    1
Beth    1
Name: count, dtype: int64
```

AlloViz.AlloViz.Elements.Edges.var

Edges.**var**(*axis: Axis | None = 0*, *skipna: bool = True*, *ddof: int = 1*, *numeric_only: bool = False*, ***kwargs*)

Return unbiased variance over requested axis.

Normalized by N-1 by default. This can be changed using the ddof argument.

Parameters

axis
[{index (0), columns (1)}] For *Series* this parameter is unused and defaults to 0.

skipna
[bool, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

ddof
[int, default 1] Delta Degrees of Freedom. The divisor used in calculations is N - ddof, where N represents the number of elements.

numeric_only
[bool, default False] Include only float, int, boolean columns. Not implemented for Series.

Returns

Series or DataFrame (if level specified)

Examples

```
>>> df = pd.DataFrame({'person_id': [0, 1, 2, 3],
...                    'age': [21, 25, 62, 43],
...                    'height': [1.61, 1.87, 1.49, 2.01]}
...                   ).set_index('person_id')
>>> df
           age  height
person_id
0           21    1.61
1           25    1.87
2           62    1.49
3           43    2.01
```

```
>>> df.var()
age       352.916667
height      0.056367
dtype: float64
```

Alternatively, `ddof=0` can be set to normalize by N instead of N-1:

```
>>> df.var(ddof=0)
age       264.687500
height      0.042275
dtype: float64
```

AlloViz.AlloViz.Elements.Edges.view

`Edges.`**`view`**(*metric*, *num=20*, *colors=['orange', 'turquoise']*, *nv=None*)

Represent the selected metric in the structure

Retrieves the analyzed data corresponding to the present Element (depending on the class) and from it the corresponding metric column from the DataFrame. It is used to obtain the elements' colors, sizes (inv. proportional to errors, if available) and names (resnames); and the parent instance attribute is used to retrieve the structure for representation using `nglview.NGLWidget`.

Data is sorted according to the selected metric in absolute value and descending order, a `LinearSegmentedColormap` is made with the passed colors. The colormap is represented in a colorbar through `_show_cbar()` and is used to establish the elements' colors through `_get_colors()`.

Errors, if available (i.e., if more than one trajectory has been used and averages were calculated), are used to establish the elements' sizes to be inversely proportional to them (interpolated between 1 and 0.1), thus directly proportional to the "confidence" in the calculated value in a way.

Elements are shown on a representation of the selected structure or added to the passed `NGLWidget` if applicable.

Parameters

metric
[str] Metric/Name of the column in the object's df attribute to represent.

num
[int, default: 20] Number of (each of the) network elements to show on the structure.

colors
[list, default: [“orange”, “turquoise”]] List of two colors to assign to the minimum and maximum values of the network to be represented, respectively. Middle value is assigned “white” and it will be the mean of the network values or 0 if the network has both negative and positive values.

nv
[nglview.NGLWidget, optional] A structure representation into which the shapes representing the chosen network elements will be added.

See also:

AlloViz.Protein.view

AlloViz.AlloViz.Elements.Edges.where

Edges.**where**(*cond*, *other=nan*, ***, *inplace: bool_t = False*, *axis: Axis | None = None*, *level: Level | None = None*) → Self | None

Replace values where the condition is False.

Parameters

cond
[bool Series/DataFrame, array-like, or callable] Where *cond* is True, keep the original value. Where False, replace with corresponding value from *other*. If *cond* is callable, it is computed on the Series/DataFrame and should return boolean Series/DataFrame or array. The callable must not change input Series/DataFrame (though pandas doesn’t check it).

other
[scalar, Series/DataFrame, or callable] Entries where *cond* is False are replaced with corresponding value from *other*. If other is callable, it is computed on the Series/DataFrame and should return scalar or Series/DataFrame. The callable must not change input Series/DataFrame (though pandas doesn’t check it). If not specified, entries will be filled with the corresponding NULL value (`np.nan` for numpy dtypes, `pd.NA` for extension dtypes).

inplace
[bool, default False] Whether to perform the operation in place on the data.

axis
[int, default None] Alignment axis if needed. For *Series* this parameter is unused and defaults to 0.

level
[int, default None] Alignment level if needed.

Returns

Same type as caller or None if `inplace=True`.

See also:

`DataFrame.mask()`
Return an object of same shape as self.

Notes

The where method is an application of the if-then idiom. For each element in the calling DataFrame, if `cond` is `True` the element is used; otherwise the corresponding element from the DataFrame `other` is used. If the axis of `other` does not align with axis of `cond` Series/DataFrame, the misaligned index positions will be filled with False.

The signature for `DataFrame.where()` differs from `numpy.where()`. Roughly `df1.where(m, df2)` is equivalent to `np.where(m, df1, df2)`.

For further details and examples see the `where` documentation in indexing.

The dtype of the object takes precedence. The fill value is casted to the object's dtype, if this can be done losslessly.

Examples

```
>>> s = pd.Series(range(5))
>>> s.where(s > 0)
0    NaN
1    1.0
2    2.0
3    3.0
4    4.0
dtype: float64
>>> s.mask(s > 0)
0    0.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

```
>>> s = pd.Series(range(5))
>>> t = pd.Series([True, False])
>>> s.where(t, 99)
0     0
1    99
2    99
3    99
4    99
dtype: int64
>>> s.mask(t, 99)
0    99
1     1
2    99
3    99
4    99
dtype: int64
```

```
>>> s.where(s > 1, 10)
0    10
1    10
```

(continues on next page)

(continued from previous page)

```
2    2
3    3
4    4
dtype: int64
>>> s.mask(s > 1, 10)
0     0
1     1
2    10
3    10
4    10
dtype: int64
```

```
>>> df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])
>>> df
   A  B
0  0  1
1  2  3
2  4  5
3  6  7
4  8  9
>>> m = df % 3 == 0
>>> df.where(m, -df)
   A  B
0  0 -1
1 -2  3
2 -4 -5
3  6 -7
4 -8  9
>>> df.where(m, -df) == np.where(m, df, -df)
      A     B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True
>>> df.where(m, -df) == df.mask(~m, -df)
      A     B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True
```

AlloViz.AlloViz.Elements.Edges.xs

Edges.**xs**(*key: Hashable | Sequence[Hashable]*, *axis: int | Literal['index', 'columns', 'rows'] = 0*, *level: Hashable | Sequence[Hashable] | None = None*, *drop_level: bool = True*) → None

Return cross-section from the Series/DataFrame.

This method takes a *key* argument to select data at a particular level of a MultiIndex.

Parameters

key
[label or tuple of label] Label contained in the index, or partially in a MultiIndex.

axis
[{0 or 'index', 1 or 'columns'}, default 0] Axis to retrieve cross-section on.

level
[object, defaults to first n levels (n=1 or len(key))] In case of a key partially contained in a MultiIndex, indicate which levels are used. Levels can be referred by label or position.

drop_level
[bool, default True] If False, returns object with same levels as self.

Returns

Series or DataFrame
Cross-section from the original Series or DataFrame corresponding to the selected index levels.

See also:

DataFrame.loc
Access a group of rows and columns by label(s) or a boolean array.

DataFrame.iloc
Purely integer-location based indexing for selection by position.

Notes

xs can not be used to set values.

MultiIndex Slicers is a generic way to get/set values on any level or levels. It is a superset of *xs* functionality, see MultiIndex Slicers.

Examples

```
>>> d = {'num_legs': [4, 4, 2, 2],
...      'num_wings': [0, 0, 2, 2],
...      'class': ['mammal', 'mammal', 'mammal', 'bird'],
...      'animal': ['cat', 'dog', 'bat', 'penguin'],
...      'locomotion': ['walks', 'walks', 'flies', 'walks']}
>>> df = pd.DataFrame(data=d)
>>> df = df.set_index(['class', 'animal', 'locomotion'])
>>> df
                           num_legs  num_wings
```

(continues on next page)

(continued from previous page)

```
class  animal  locomotion
mammal cat     walks              4          0
       dog     walks              4          0
       bat     flies              2          2
bird   penguin walks              2          2
```

Get values at specified index

```
>>> df.xs('mammal')
                   num_legs  num_wings
animal locomotion
cat    walks              4          0
dog    walks              4          0
bat    flies              2          2
```

Get values at several indexes

```
>>> df.xs(('mammal', 'dog', 'walks'))
num_legs     4
num_wings    0
Name: (mammal, dog, walks), dtype: int64
```

Get values at specified index and level

```
>>> df.xs('cat', level=1)
                   num_legs  num_wings
class  locomotion
mammal walks              4          0
```

Get values at several indexes and levels

```
>>> df.xs(('bird', 'walks'),
...       level=[0, 'locomotion'])
         num_legs  num_wings
animal
penguin         2          2
```

Get values at specified column and axis

```
>>> df.xs('num_wings', axis=1)
class   animal   locomotion
mammal  cat      walks         0
        dog      walks         0
        bat      flies         2
bird    penguin  walks         2
Name: num_wings, dtype: int64
```

AlloViz.AlloViz.Elements.Element

class AlloViz.AlloViz.Elements.**Element**(*data*, *parent*, *index=None*, *columns=None*, *dtype=None*, *copy=True*)

Bases: DataFrame

Base class for network storage and representation

This class extends pandas.DataFrame with a *view()* method (and related private methods) to visualize the networks on the protein structures, besides storing the analyzed data. It uses the private methods `_get_colors()` and `_show_cbar()` to establish the color scale and represent it as a colorbar, respectively, and `_get_nv()` to retrieve information about the protein structure to show and/or the visualization widget to add the network elements to. The different elements, cylinders for *Edges* and spheres for *Nodes*, are represented using the corresponding child class' *_add_element* private method.

Once a class instance is created, the private attribute *_parent* must be established with the *Protein* or *Delta* object the information belongs to, as it is needed for representation.

Attributes

T
The transpose of the DataFrame.

at
Access a single value for a row/column label pair.

attrs
Dictionary of global attributes of this dataset.

axes
Return a list representing the axes of the DataFrame.

columns
The column labels of the DataFrame.

```
>>> df = pd.DataFrame({'A': [1, 2], 'B': [3, 4]})
>>> df
     A  B
0    1  3
1    2  4
>>> df.columns
Index(['A', 'B'], dtype='object')
```

dtypes
Return the dtypes in the DataFrame.

empty
Indicator whether Series/DataFrame is empty.

flags
Get the properties associated with this pandas object.

iat
Access a single value for a row/column pair by integer position.

iloc
Purely integer-location based indexing for selection by position.

index
The index (row labels) of the DataFrame.

The index of a DataFrame is a series of labels that identify each row. The labels can be integers, strings, or any other hashable type. The index is used for label-based access and alignment, and can be accessed or modified using this attribute.

pandas.Index
The index labels of the DataFrame.

DataFrame.columns : The column labels of the DataFrame. DataFrame.to_numpy : Convert the DataFrame to a NumPy array.

```
>>> df = pd.DataFrame({'Name': ['Alice', 'Bob', 'Aritra'],
...                    'Age': [25, 30, 35],
...                    'Location': ['Seattle', 'New York', 'Kona
↪']},
...                   index=([10, 20, 30]))
>>> df.index
Index([10, 20, 30], dtype='int64')
```

In this example, we create a DataFrame with 3 rows and 3 columns, including Name, Age, and Location information. We set the index labels to be the integers 10, 20, and 30. We then access the *index* attribute of the DataFrame, which returns an *Index* object containing the index labels.

```
>>> df.index = [100, 200, 300]
>>> df
    Name  Age Location
100  Alice   25  Seattle
200    Bob   30 New York
300  Aritra  35    Kona
```

In this example, we modify the index labels of the DataFrame by assigning a new list of labels to the *index* attribute. The DataFrame is then updated with the new labels, and the output shows the modified DataFrame.

loc
Access a group of rows and columns by label(s) or a boolean array.

ndim
Return an int representing the number of axes / array dimensions.

shape
Return a tuple representing the dimensionality of the DataFrame.

size
Return an int representing the number of elements in this object.

style
Returns a Styler object.

values
Return a Numpy representation of the DataFrame.

Methods

abs()	Return a Series/DataFrame with absolute numeric value of each element.
add(other[, axis, level, fill_value])	Get Addition of dataframe and other, element-wise (binary operator *add*).
add_prefix(prefix[, axis])	Prefix labels with string *prefix*.
add_suffix(suffix[, axis])	Suffix labels with string *suffix*.
agg([func, axis])	Aggregate using one or more operations over the specified axis.
aggregate([func, axis])	Aggregate using one or more operations over the specified axis.
align(other[, join, axis, level, copy, ...])	Align two objects on their axes with the specified join method.
all([axis, bool_only, skipna])	Return whether all elements are True, potentially over an axis.
any(*[, axis, bool_only, skipna])	Return whether any element is True, potentially over an axis.
apply(func[, axis, raw, result_type, args, ...])	Apply a function along an axis of the DataFrame.
applymap(func[, na_action])	Apply a function to a Dataframe elementwise.
asfreq(freq[, method, how, normalize, ...])	Convert time series to specified frequency.
asof(where[, subset])	Return the last row(s) without any NaNs before *where*.
assign(**kwargs)	Assign new columns to a DataFrame.
astype(dtype[, copy, errors])	Cast a pandas object to a specified dtype `dtype`.
at_time(time[, asof, axis])	Select values at particular time of day (e.g., 9:30AM).
backfill(*[, axis, inplace, limit, downcast])	Fill NA/NaN values by using the next valid observation to fill the gap.
between_time(start_time, end_time[, ...])	Select values between particular times of the day (e.g., 9:00-9:30 AM).
bfill(*[, axis, inplace, limit, downcast])	Fill NA/NaN values by using the next valid observation to fill the gap.
bool()	Return the bool of a single element Series or DataFrame.
boxplot([column, by, ax, fontsize, rot, ...])	Make a box plot from DataFrame columns.
clip([lower, upper, axis, inplace])	Trim values at input threshold(s).
combine(other, func[, fill_value, overwrite])	Perform column-wise combine with another DataFrame.
combine_first(other)	Update null elements with value in the same location in *other*.
compare(other[, align_axis, keep_shape, ...])	Compare to another DataFrame and show the differences.
convert_dtypes([infer_objects, ...])	Convert columns to the best possible dtypes using dtypes supporting `pd.NA`.
copy([deep])	Make a copy of this object's indices and data.
corr([method, min_periods, numeric_only])	Compute pairwise correlation of columns, excluding NA/null values.
corrwith(other[, axis, drop, method, ...])	Compute pairwise correlation.
count([axis, numeric_only])	Count non-NA cells for each column or row.
cov([min_periods, ddof, numeric_only])	Compute pairwise covariance of columns, excluding NA/null values.

continues on next page

Table 2 – continued from previous page

cummax([axis, skipna])	Return cumulative maximum over a DataFrame or Series axis.
cummin([axis, skipna])	Return cumulative minimum over a DataFrame or Series axis.
cumprod([axis, skipna])	Return cumulative product over a DataFrame or Series axis.
cumsum([axis, skipna])	Return cumulative sum over a DataFrame or Series axis.
describe([percentiles, include, exclude])	Generate descriptive statistics.
diff([periods, axis])	First discrete difference of element.
div(other[, axis, level, fill_value])	Get Floating division of dataframe and other, element-wise (binary operator *truediv*).
divide(other[, axis, level, fill_value])	Get Floating division of dataframe and other, element-wise (binary operator *truediv*).
dot(other)	Compute the matrix multiplication between the DataFrame and other.
drop([labels, axis, index, columns, level, ...])	Drop specified labels from rows or columns.
drop_duplicates([subset, keep, inplace, ...])	Return DataFrame with duplicate rows removed.
droplevel(level[, axis])	Return Series/DataFrame with requested index / column level(s) removed.
dropna(*[, axis, how, thresh, subset, ...])	Remove missing values.
duplicated([subset, keep])	Return boolean Series denoting duplicate rows.
eq(other[, axis, level])	Get Equal to of dataframe and other, element-wise (binary operator *eq*).
equals(other)	Test whether two objects contain the same elements.
eval(expr, *[, inplace])	Evaluate a string describing operations on DataFrame columns.
ewm([com, span, halflife, alpha, ...])	Provide exponentially weighted (EW) calculations.
expanding([min_periods, axis, method])	Provide expanding window calculations.
explode(column[, ignore_index])	Transform each element of a list-like to a row, replicating index values.
ffill(*[, axis, inplace, limit, downcast])	Fill NA/NaN values by propagating the last valid observation to next valid.
fillna([value, method, axis, inplace, ...])	Fill NA/NaN values using the specified method.
filter([items, like, regex, axis])	Subset the dataframe rows or columns according to the specified index labels.
first(offset)	Select initial periods of time series data based on a date offset.
first_valid_index()	Return index for first non-NA value or None, if no non-NA value is found.
floordiv(other[, axis, level, fill_value])	Get Integer division of dataframe and other, element-wise (binary operator *floordiv*).
from_dict(data[, orient, dtype, columns])	Construct DataFrame from dict of array-like or dicts.
from_records(data[, index, exclude, ...])	Convert structured or record ndarray to DataFrame.
ge(other[, axis, level])	Get Greater than or equal to of dataframe and other, element-wise (binary operator *ge*).
get(key[, default])	Get item from object for given key (ex: DataFrame column).
groupby([by, axis, level, as_index, sort, ...])	Group DataFrame using a mapper or by a Series of columns.
gt(other[, axis, level])	Get Greater than of dataframe and other, element-wise (binary operator *gt*).

continues on next page

Table 2 – continued from previous page

head([n])	Return the first *n* rows.
hist([column, by, grid, xlabelsize, xrot, ...])	Make a histogram of the DataFrame's columns.
idxmax([axis, skipna, numeric_only])	Return index of first occurrence of maximum over requested axis.
idxmin([axis, skipna, numeric_only])	Return index of first occurrence of minimum over requested axis.
infer_objects([copy])	Attempt to infer better dtypes for object columns.
info([verbose, buf, max_cols, memory_usage, ...])	Print a concise summary of a DataFrame.
insert(loc, column, value[, allow_duplicates])	Insert column into DataFrame at specified location.
interpolate([method, axis, limit, inplace, ...])	Fill NaN values using an interpolation method.
isetitem(loc, value)	Set the given value in the column with position *loc*.
isin(values)	Whether each element in the DataFrame is contained in values.
isna()	Detect missing values.
isnull()	DataFrame.isnull is an alias for DataFrame.isna.
items()	Iterate over (column name, Series) pairs.
iterrows()	Iterate over DataFrame rows as (index, Series) pairs.
itertuples([index, name])	Iterate over DataFrame rows as namedtuples.
join(other[, on, how, lsuffix, rsuffix, ...])	Join columns of another DataFrame.
keys()	Get the 'info axis' (see Indexing for more).
kurt([axis, skipna, numeric_only])	Return unbiased kurtosis over requested axis.
kurtosis([axis, skipna, numeric_only])	Return unbiased kurtosis over requested axis.
last(offset)	Select final periods of time series data based on a date offset.
last_valid_index()	Return index for last non-NA value or None, if no non-NA value is found.
le(other[, axis, level])	Get Less than or equal to of dataframe and other, element-wise (binary operator *le*).
lt(other[, axis, level])	Get Less than of dataframe and other, element-wise (binary operator *lt*).
map(func[, na_action])	Apply a function to a Dataframe elementwise.
mask(cond[, other, inplace, axis, level])	Replace values where the condition is True.
max([axis, skipna, numeric_only])	Return the maximum of the values over the requested axis.
mean([axis, skipna, numeric_only])	Return the mean of the values over the requested axis.
median([axis, skipna, numeric_only])	Return the median of the values over the requested axis.
melt([id_vars, value_vars, var_name, ...])	Unpivot a DataFrame from wide to long format, optionally leaving identifiers set.
memory_usage([index, deep])	Return the memory usage of each column in bytes.
merge(right[, how, on, left_on, right_on, ...])	Merge DataFrame or named Series objects with a database-style join.
min([axis, skipna, numeric_only])	Return the minimum of the values over the requested axis.
mod(other[, axis, level, fill_value])	Get Modulo of dataframe and other, element-wise (binary operator *mod*).
mode([axis, numeric_only, dropna])	Get the mode(s) of each element along the selected axis.
mul(other[, axis, level, fill_value])	Get Multiplication of dataframe and other, element-wise (binary operator *mul*).
multiply(other[, axis, level, fill_value])	Get Multiplication of dataframe and other, element-wise (binary operator *mul*).

continues on next page

Table 2 – continued from previous page

ne(other[, axis, level])	Get Not equal to of dataframe and other, element-wise (binary operator *ne*).
nlargest(n, columns[, keep])	Return the first *n* rows ordered by *columns* in descending order.
notna()	Detect existing (non-missing) values.
notnull()	DataFrame.notnull is an alias for DataFrame.notna.
nsmallest(n, columns[, keep])	Return the first *n* rows ordered by *columns* in ascending order.
nunique([axis, dropna])	Count number of distinct elements in specified axis.
pad(*[, axis, inplace, limit, downcast])	Fill NA/NaN values by propagating the last valid observation to next valid.
pct_change([periods, fill_method, limit, freq])	Fractional change between the current and a prior element.
pipe(func, *args, **kwargs)	Apply chainable functions that expect Series or DataFrames.
pivot(*, columns[, index, values])	Return reshaped DataFrame organized by given index / column values.
pivot_table([values, index, columns, ...])	Create a spreadsheet-style pivot table as a DataFrame.
plot	alias of PlotAccessor
pop(item)	Return item and drop from frame.
pow(other[, axis, level, fill_value])	Get Exponential power of dataframe and other, element-wise (binary operator *pow*).
prod([axis, skipna, numeric_only, min_count])	Return the product of the values over the requested axis.
product([axis, skipna, numeric_only, min_count])	Return the product of the values over the requested axis.
quantile([q, axis, numeric_only, ...])	Return values at the given quantile over requested axis.
query(expr, *[, inplace])	Query the columns of a DataFrame with a boolean expression.
radd(other[, axis, level, fill_value])	Get Addition of dataframe and other, element-wise (binary operator *radd*).
rank([axis, method, numeric_only, ...])	Compute numerical data ranks (1 through n) along axis.
rdiv(other[, axis, level, fill_value])	Get Floating division of dataframe and other, element-wise (binary operator *rtruediv*).
reindex([labels, index, columns, axis, ...])	Conform DataFrame to new index with optional filling logic.
reindex_like(other[, method, copy, limit, ...])	Return an object with matching indices as other object.
rename([mapper, index, columns, axis, copy, ...])	Rename columns or index labels.
rename_axis([mapper, index, columns, axis, ...])	Set the name of the axis for the index or columns.
reorder_levels(order[, axis])	Rearrange index levels using input order.
replace([to_replace, value, inplace, limit, ...])	Replace values given in *to_replace* with *value*.
resample(rule[, axis, closed, label, ...])	Resample time-series data.
reset_index([level, drop, inplace, ...])	Reset the index, or a level of it.
rfloordiv(other[, axis, level, fill_value])	Get Integer division of dataframe and other, element-wise (binary operator *rfloordiv*).
rmod(other[, axis, level, fill_value])	Get Modulo of dataframe and other, element-wise (binary operator *rmod*).
rmul(other[, axis, level, fill_value])	Get Multiplication of dataframe and other, element-wise (binary operator *rmul*).

continues on next page

Table 2 – continued from previous page

rolling(window[, min_periods, center, ...])	Provide rolling window calculations.
round([decimals])	Round a DataFrame to a variable number of decimal places.
rpow(other[, axis, level, fill_value])	Get Exponential power of dataframe and other, element-wise (binary operator *rpow*).
rsub(other[, axis, level, fill_value])	Get Subtraction of dataframe and other, element-wise (binary operator *rsub*).
rtruediv(other[, axis, level, fill_value])	Get Floating division of dataframe and other, element-wise (binary operator *rtruediv*).
sample([n, frac, replace, weights, ...])	Return a random sample of items from an axis of object.
select_dtypes([include, exclude])	Return a subset of the DataFrame's columns based on the column dtypes.
sem([axis, skipna, ddof, numeric_only])	Return unbiased standard error of the mean over requested axis.
set_axis(labels, *[, axis, copy])	Assign desired index to given axis.
set_flags(*[, copy, allows_duplicate_labels])	Return a new object with updated flags.
set_index(keys, *[, drop, append, inplace, ...])	Set the DataFrame index using existing columns.
shift([periods, freq, axis, fill_value, suffix])	Shift index by desired number of periods with an optional time *freq*.
skew([axis, skipna, numeric_only])	Return unbiased skew over requested axis.
sort_index(*[, axis, level, ascending, ...])	Sort object by labels (along an axis).
sort_values(by, *[, axis, ascending, ...])	Sort by the values along either axis.
sparse	alias of `SparseFrameAccessor`
squeeze([axis])	Squeeze 1 dimensional axis objects into scalars.
stack([level, dropna, sort, future_stack])	Stack the prescribed level(s) from columns to index.
std([axis, skipna, ddof, numeric_only])	Return sample standard deviation over requested axis.
sub(other[, axis, level, fill_value])	Get Subtraction of dataframe and other, element-wise (binary operator *sub*).
subtract(other[, axis, level, fill_value])	Get Subtraction of dataframe and other, element-wise (binary operator *sub*).
sum([axis, skipna, numeric_only, min_count])	Return the sum of the values over the requested axis.
swapaxes(axis1, axis2[, copy])	Interchange axes and swap values axes appropriately.
swaplevel([i, j, axis])	Swap levels i and j in a `MultiIndex`.
tail([n])	Return the last *n* rows.
take(indices[, axis])	Return the elements in the given *positional* indices along an axis.
to_clipboard([excel, sep])	Copy object to the system clipboard.
to_csv([path_or_buf, sep, na_rep, ...])	Write object to a comma-separated values (csv) file.
to_dict([orient, into, index])	Convert the DataFrame to a dictionary.
to_excel(excel_writer[, sheet_name, na_rep, ...])	Write object to an Excel sheet.
to_feather(path, **kwargs)	Write a DataFrame to the binary Feather format.
to_gbq(destination_table[, project_id, ...])	Write a DataFrame to a Google BigQuery table.
to_hdf(path_or_buf, key[, mode, complevel, ...])	Write the contained data to an HDF5 file using HDFStore.
to_html([buf, columns, col_space, header, ...])	Render a DataFrame as an HTML table.
to_json([path_or_buf, orient, date_format, ...])	Convert the object to a JSON string.
to_latex([buf, columns, header, index, ...])	Render object to a LaTeX tabular, longtable, or nested table.
to_markdown([buf, mode, index, storage_options])	Print DataFrame in Markdown-friendly format.
to_numpy([dtype, copy, na_value])	Convert the DataFrame to a NumPy array.
to_orc([path, engine, index, engine_kwargs])	Write a DataFrame to the ORC format.

continues on next page

Table 2 – continued from previous page

to_parquet([path, engine, compression, ...])	Write a DataFrame to the binary parquet format.
to_period([freq, axis, copy])	Convert DataFrame from DatetimeIndex to PeriodIndex.
to_pickle(path[, compression, protocol, ...])	Pickle (serialize) object to file.
to_records([index, column_dtypes, index_dtypes])	Convert DataFrame to a NumPy record array.
to_sql(name, con, *[, schema, if_exists, ...])	Write records stored in a DataFrame to a SQL database.
to_stata(path, *[, convert_dates, ...])	Export DataFrame object to Stata dta format.
to_string([buf, columns, col_space, header, ...])	Render a DataFrame to a console-friendly tabular output.
to_timestamp([freq, how, axis, copy])	Cast to DatetimeIndex of timestamps, at *beginning* of period.
to_xarray()	Return an xarray object from the pandas object.
to_xml([path_or_buffer, index, root_name, ...])	Render a DataFrame to an XML document.
transform(func[, axis])	Call `func` on self producing a DataFrame with the same axis shape as self.
transpose(*args[, copy])	Transpose index and columns.
truediv(other[, axis, level, fill_value])	Get Floating division of dataframe and other, element-wise (binary operator *truediv*).
truncate([before, after, axis, copy])	Truncate a Series or DataFrame before and after some index value.
tz_convert(tz[, axis, level, copy])	Convert tz-aware axis to target time zone.
tz_localize(tz[, axis, level, copy, ...])	Localize tz-naive index of a Series or DataFrame to target time zone.
unstack([level, fill_value, sort])	Pivot a level of the (necessarily hierarchical) index labels.
update(other[, join, overwrite, ...])	Modify in place using non-NA values from another DataFrame.
value_counts([subset, normalize, sort, ...])	Return a Series containing the frequency of each distinct row in the Dataframe.
var([axis, skipna, ddof, numeric_only])	Return unbiased variance over requested axis.
view(metric[, num, colors, nv])	Represent the selected metric in the structure
where(cond[, other, inplace, axis, level])	Replace values where the condition is False.
xs(key[, axis, level, drop_level])	Return cross-section from the Series/DataFrame.

AlloViz.AlloViz.Elements.Element.abs

Element.**abs**() → None

Return a Series/DataFrame with absolute numeric value of each element.

This function only applies to elements that are all numeric.

Returns

abs
Series/DataFrame containing the absolute value of each element.

See also:

numpy.absolute
Calculate the absolute value element-wise.

Notes

For `complex` inputs, `1.2 + 1j`, the absolute value is $\sqrt{a^2 + b^2}$.

Examples

Absolute numeric values in a Series.

```
>>> s = pd.Series([-1.10, 2, -3.33, 4])
>>> s.abs()
0    1.10
1    2.00
2    3.33
3    4.00
dtype: float64
```

Absolute numeric values in a Series with complex numbers.

```
>>> s = pd.Series([1.2 + 1j])
>>> s.abs()
0    1.56205
dtype: float64
```

Absolute numeric values in a Series with a Timedelta element.

```
>>> s = pd.Series([pd.Timedelta('1 days')])
>>> s.abs()
0   1 days
dtype: timedelta64[ns]
```

Select rows with data closest to certain value using argsort (from StackOverflow).

```
>>> df = pd.DataFrame({
...     'a': [4, 5, 6, 7],
...     'b': [10, 20, 30, 40],
...     'c': [100, 50, -30, -50]
... })
>>> df
     a    b    c
0    4   10  100
1    5   20   50
2    6   30  -30
3    7   40  -50
>>> df.loc[(df.c - 43).abs().argsort()]
     a    b    c
1    5   20   50
0    4   10  100
2    6   30  -30
3    7   40  -50
```

AlloViz.AlloViz.Elements.Element.add

Element.**add**(*other*, *axis: Axis = 'columns'*, *level=None*, *fill_value=None*)

Get Addition of dataframe and other, element-wise (binary operator *add*).

Equivalent to `dataframe + other`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *radd*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *floordiv*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other
[scalar, sequence, Series, dict or DataFrame] Any single or multiple element data structure, or list-like object.

axis
[{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns. (1 or 'columns'). For Series input, axis to match Series index on.

level
[int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value
[float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame
Result of the arithmetic operation.

See also:

`DataFrame.add`
Add DataFrames.

`DataFrame.sub`
Subtract DataFrames.

`DataFrame.mul`
Multiply DataFrames.

`DataFrame.div`
Divide DataFrames (float division).

`DataFrame.truediv`
Divide DataFrames (float division).

`DataFrame.floordiv`
Divide DataFrames (integer division).

`DataFrame.mod`
Calculate modulo (remainder after division).

`DataFrame.pow`
Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...         axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a dictionary by axis.

```
>>> df.mul({'angles': 0, 'degrees': 2})
            angles  degrees
circle           0      720
triangle         0      360
rectangle        0      720
```

```
>>> df.mul({'circle': 0, 'triangle': 2, 'rectangle': 3}, axis='index')
            angles  degrees
circle           0        0
triangle         6      360
rectangle       12     1080
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

AlloViz.AlloViz.Elements.Element.add_prefix

Element.**add_prefix**(*prefix: str*, *axis: int | Literal['index', 'columns', 'rows'] | None = None*) → None

Prefix labels with string *prefix*.

For Series, the row labels are prefixed. For DataFrame, the column labels are prefixed.

Parameters

prefix
[str] The string to add before each label.

axis
[{0 or 'index', 1 or 'columns', None}, default None] Axis to add prefix on

New in version 2.0.0.

Returns

Series or DataFrame
New Series or DataFrame with updated labels.

See also:

Series.add_suffix
Suffix row labels with string *suffix*.

DataFrame.add_suffix
Suffix column labels with string *suffix*.

Examples

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s
0    1
1    2
2    3
3    4
dtype: int64
```

```
>>> s.add_prefix('item_')
item_0    1
item_1    2
item_2    3
item_3    4
dtype: int64
```

```
>>> df = pd.DataFrame({'A': [1, 2, 3, 4], 'B': [3, 4, 5, 6]})
>>> df
   A  B
0  1  3
1  2  4
2  3  5
3  4  6
```

```
>>> df.add_prefix('col_')
     col_A  col_B
0       1       3
1       2       4
2       3       5
3       4       6
```

AlloViz.AlloViz.Elements.Element.add_suffix

Element.**add_suffix**(*suffix: str*, *axis: int | Literal['index', 'columns', 'rows'] | None = None*) → None

Suffix labels with string *suffix*.

For Series, the row labels are suffixed. For DataFrame, the column labels are suffixed.

Parameters

suffix
[str] The string to add after each label.

axis
[{0 or 'index', 1 or 'columns', None}, default None] Axis to add suffix on

New in version 2.0.0.

Returns

Series or DataFrame
New Series or DataFrame with updated labels.

See also:

Series.add_prefix
Prefix row labels with string *prefix*.

DataFrame.add_prefix
Prefix column labels with string *prefix*.

Examples

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s
0    1
1    2
2    3
3    4
dtype: int64
```

```
>>> s.add_suffix('_item')
0_item    1
1_item    2
2_item    3
3_item    4
dtype: int64
```

```
>>> df = pd.DataFrame({'A': [1, 2, 3, 4], 'B': [3, 4, 5, 6]})
>>> df
   A  B
0  1  3
1  2  4
2  3  5
3  4  6
```

```
>>> df.add_suffix('_col')
     A_col  B_col
0       1       3
1       2       4
2       3       5
3       4       6
```

AlloViz.AlloViz.Elements.Element.agg

`Element.agg`(*func=None*, *axis: Axis = 0*, **args*, ***kwargs*)

Aggregate using one or more operations over the specified axis.

Parameters

func
[function, str, list or dict] Function to use for aggregating the data. If a function, must either work when passed a DataFrame or when passed to DataFrame.apply.

Accepted combinations are:

- function

- string function name
- list of functions and/or function names, e.g. `[np.sum, 'mean']`
- dict of axis labels -> functions, function names or list of such.

axis
[{0 or 'index', 1 or 'columns'}, default 0] If 0 or 'index': apply function to each column. If 1 or 'columns': apply function to each row.

***args**
Positional arguments to pass to *func*.

****kwargs**
Keyword arguments to pass to *func*.

Returns

scalar, Series or DataFrame
The return can be:

- scalar : when Series.agg is called with single function
- Series : when DataFrame.agg is called with a single function
- DataFrame : when DataFrame.agg is called with several functions

Return scalar, Series or DataFrame.

See also:

`DataFrame.apply`
Perform any type of operations.

`DataFrame.transform`
Perform transformation type operations.

`core.groupby.GroupBy`
Perform operations over groups.

`core.resample.Resampler`
Perform operations over resampled bins.

`core.window.Rolling`
Perform operations over rolling window.

`core.window.Expanding`
Perform operations over expanding window.

`core.window.ExponentialMovingWindow`
Perform operation over exponential weighted window.

Notes

The aggregation operations are always performed over an axis, either the index (default) or the column axis. This behavior is different from *numpy* aggregation functions (*mean*, *median*, *prod*, *sum*, *std*, *var*), where the default is to compute the aggregation of the flattened array, e.g., `numpy.mean(arr_2d)` as opposed to `numpy.mean(arr_2d, axis=0)`.

agg is an alias for *aggregate*. Use the alias.

Functions that mutate the passed object can produce unexpected behavior or errors and are not supported. See Mutating with User Defined Function (UDF) methods for more details.

A passed user-defined-function will be passed a Series for evaluation.

Examples

```
>>> df = pd.DataFrame([[1, 2, 3],
...                    [4, 5, 6],
...                    [7, 8, 9],
...                    [np.nan, np.nan, np.nan]],
...                   columns=['A', 'B', 'C'])
```

Aggregate these functions over the rows.

```
>>> df.agg(['sum', 'min'])
        A     B     C
sum  12.0  15.0  18.0
min   1.0   2.0   3.0
```

Different aggregations per column.

```
>>> df.agg({'A' : ['sum', 'min'], 'B' : ['min', 'max']})
        A    B
sum  12.0  NaN
min   1.0  2.0
max   NaN  8.0
```

Aggregate different functions over the columns and rename the index of the resulting DataFrame.

```
>>> df.agg(x=('A', 'max'), y=('B', 'min'), z=('C', 'mean'))
     A    B    C
x  7.0  NaN  NaN
y  NaN  2.0  NaN
z  NaN  NaN  6.0
```

Aggregate over the columns.

```
>>> df.agg("mean", axis="columns")
0    2.0
1    5.0
2    8.0
3    NaN
dtype: float64
```

AlloViz.AlloViz.Elements.Element.aggregate

`Element.`**`aggregate`**(*func=None*, *axis: Axis = 0*, **args*, ***kwargs*)

Aggregate using one or more operations over the specified axis.

Parameters

func

[function, str, list or dict] Function to use for aggregating the data. If a function, must either work when passed a DataFrame or when passed to DataFrame.apply.

Accepted combinations are:

- function
- string function name
- list of functions and/or function names, e.g. `[np.sum, 'mean']`
- dict of axis labels -> functions, function names or list of such.

axis
[{0 or 'index', 1 or 'columns'}, default 0] If 0 or 'index': apply function to each column. If 1 or 'columns': apply function to each row.

***args**
Positional arguments to pass to *func*.

****kwargs**
Keyword arguments to pass to *func*.

Returns

scalar, Series or DataFrame
The return can be:

- scalar : when Series.agg is called with single function
- Series : when DataFrame.agg is called with a single function
- DataFrame : when DataFrame.agg is called with several functions

Return scalar, Series or DataFrame.

See also:

`DataFrame.apply`
Perform any type of operations.

`DataFrame.transform`
Perform transformation type operations.

`core.groupby.GroupBy`
Perform operations over groups.

`core.resample.Resampler`
Perform operations over resampled bins.

`core.window.Rolling`
Perform operations over rolling window.

`core.window.Expanding`
Perform operations over expanding window.

`core.window.ExponentialMovingWindow`
Perform operation over exponential weighted window.

Notes

The aggregation operations are always performed over an axis, either the index (default) or the column axis. This behavior is different from *numpy* aggregation functions (*mean*, *median*, *prod*, *sum*, *std*, *var*), where the default is to compute the aggregation of the flattened array, e.g., `numpy.mean(arr_2d)` as opposed to `numpy.mean(arr_2d, axis=0)`.

agg is an alias for *aggregate*. Use the alias.

Functions that mutate the passed object can produce unexpected behavior or errors and are not supported. See Mutating with User Defined Function (UDF) methods for more details.

A passed user-defined-function will be passed a Series for evaluation.

Examples

```
>>> df = pd.DataFrame([[1, 2, 3],
...                    [4, 5, 6],
...                    [7, 8, 9],
...                    [np.nan, np.nan, np.nan]],
...                   columns=['A', 'B', 'C'])
```

Aggregate these functions over the rows.

```
>>> df.agg(['sum', 'min'])
        A     B     C
sum  12.0  15.0  18.0
min   1.0   2.0   3.0
```

Different aggregations per column.

```
>>> df.agg({'A' : ['sum', 'min'], 'B' : ['min', 'max']})
        A    B
sum  12.0  NaN
min   1.0  2.0
max   NaN  8.0
```

Aggregate different functions over the columns and rename the index of the resulting DataFrame.

```
>>> df.agg(x=('A', 'max'), y=('B', 'min'), z=('C', 'mean'))
     A    B    C
x  7.0  NaN  NaN
y  NaN  2.0  NaN
z  NaN  NaN  6.0
```

Aggregate over the columns.

```
>>> df.agg("mean", axis="columns")
0    2.0
1    5.0
2    8.0
3    NaN
dtype: float64
```

AlloViz.AlloViz.Elements.Element.align

Element.**align**(*other: NDFrameT*, *join: AlignJoin = 'outer'*, *axis: Axis | None = None*, *level: Level | None = None*, *copy: bool_t | None = None*, *fill_value: Hashable | None = None*, *method: FillnaOptions | None | lib.NoDefault = _NoDefault.no_default*, *limit: int | None | lib.NoDefault = _NoDefault.no_default*, *fill_axis: Axis | lib.NoDefault = _NoDefault.no_default*, *broadcast_axis: Axis | None | lib.NoDefault = _NoDefault.no_default*) → tuple[Self, NDFrameT]

Align two objects on their axes with the specified join method.

Join method is specified for each axis Index.

Parameters

other
[DataFrame or Series]

join
[{'outer', 'inner', 'left', 'right'}, default 'outer'] Type of alignment to be performed.

- left: use only keys from left frame, preserve key order.
- right: use only keys from right frame, preserve key order.
- outer: use union of keys from both frames, sort keys lexicographically.
- inner: use intersection of keys from both frames, preserve the order of the left keys.

axis
[allowed axis of the other object, default None] Align on index (0), columns (1), or both (None).

level
[int or level name, default None] Broadcast across a level, matching Index values on the passed MultiIndex level.

copy
[bool, default True] Always returns new objects. If copy=False and no reindexing is required then original objects are returned.

fill_value
[scalar, default np.nan] Value to use for missing values. Defaults to NaN, but can be any "compatible" value.

method
[{'backfill', 'bfill', 'pad', 'ffill', None}, default None] Method to use for filling holes in reindexed Series:

- pad / ffill: propagate last valid observation forward to next valid.
- backfill / bfill: use NEXT valid observation to fill gap.

Deprecated since version 2.1.

limit
[int, default None] If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled. Must be greater than 0 if not None.

Deprecated since version 2.1.

fill_axis
[{0 or 'index'} for Series, {0 or 'index', 1 or 'columns'} for DataFrame, default 0] Filling axis, method and limit.

Deprecated since version 2.1.

broadcast_axis
[{0 or 'index'} for Series, {0 or 'index', 1 or 'columns'} for DataFrame, default None] Broadcast values along this axis, if aligning two objects of different dimensions.

Deprecated since version 2.1.

Returns

tuple of (Series/DataFrame, type of other)
Aligned objects.

Examples

```
>>> df = pd.DataFrame(
...     [[1, 2, 3, 4], [6, 7, 8, 9]], columns=["D", "B", "E", "A"], index=[1,␣
↪2]
... )
>>> other = pd.DataFrame(
...     [[10, 20, 30, 40], [60, 70, 80, 90], [600, 700, 800, 900]],
...     columns=["A", "B", "C", "D"],
...     index=[2, 3, 4],
... )
>>> df
   D  B  E  A
1  1  2  3  4
2  6  7  8  9
>>> other
    A    B    C    D
2   10   20   30   40
3   60   70   80   90
4  600  700  800  900
```

Align on columns:

```
>>> left, right = df.align(other, join="outer", axis=1)
>>> left
   A  B   C  D  E
1  4  2 NaN  1  3
2  9  7 NaN  6  8
>>> right
    A    B    C    D   E
2   10   20   30   40 NaN
3   60   70   80   90 NaN
4  600  700  800  900 NaN
```

We can also align on the index:

```
>>> left, right = df.align(other, join="outer", axis=0)
>>> left
    D    B    E    A
1  1.0  2.0  3.0  4.0
2  6.0  7.0  8.0  9.0
3  NaN  NaN  NaN  NaN
4  NaN  NaN  NaN  NaN
>>> right
    A      B      C      D
1   NaN    NaN    NaN    NaN
2  10.0   20.0   30.0   40.0
3  60.0   70.0   80.0   90.0
4 600.0  700.0  800.0  900.0
```

Finally, the default *axis=None* will align on both index and columns:

```
>>> left, right = df.align(other, join="outer", axis=None)
>>> left
     A    B   C    D    E
1  4.0  2.0 NaN  1.0  3.0
2  9.0  7.0 NaN  6.0  8.0
3  NaN  NaN NaN  NaN  NaN
4  NaN  NaN NaN  NaN  NaN
>>> right
       A      B      C      D   E
1   NaN    NaN    NaN    NaN NaN
2  10.0   20.0   30.0   40.0 NaN
3  60.0   70.0   80.0   90.0 NaN
4 600.0  700.0  800.0  900.0 NaN
```

AlloViz.AlloViz.Elements.Element.all

`Element.all`(*axis: Axis = 0*, *bool_only: bool = False*, *skipna: bool = True*, ***kwargs*) → Series | bool

Return whether all elements are True, potentially over an axis.

Returns True unless there at least one element within a series or along a Dataframe axis that is False or equivalent (e.g. zero or empty).

Parameters

axis
[{0 or 'index', 1 or 'columns', None}, default 0] Indicate which axis or axes should be reduced. For *Series* this parameter is unused and defaults to 0.

- 0 / 'index' : reduce the index, return a Series whose index is the original column labels.
- 1 / 'columns' : reduce the columns, return a Series whose index is the original index.
- None : reduce all axes, return a scalar.

bool_only
[bool, default False] Include only boolean columns. Not implemented for Series.

skipna
[bool, default True] Exclude NA/null values. If the entire row/column is NA and skipna is True, then the result will be True, as for an empty row/column. If skipna is False, then NA are treated as True, because these are not equal to zero.

****kwargs**
[any, default None] Additional keywords have no effect but might be accepted for compatibility with NumPy.

Returns

Series or DataFrame
If level is specified, then, DataFrame is returned; otherwise, Series is returned.

See also:

`Series.all`
Return True if all elements are True.

`DataFrame.any`
Return True if one (or more) elements are True.

Examples

Series

```
>>> pd.Series([True, True]).all()
True
>>> pd.Series([True, False]).all()
False
>>> pd.Series([], dtype="float64").all()
True
>>> pd.Series([np.nan]).all()
True
>>> pd.Series([np.nan]).all(skipna=False)
True
```

DataFrames

Create a dataframe from a dictionary.

```
>>> df = pd.DataFrame({'col1': [True, True], 'col2': [True, False]})
>>> df
   col1   col2
0  True   True
1  True  False
```

Default behaviour checks if values in each column all return True.

```
>>> df.all()
col1     True
col2    False
dtype: bool
```

Specify `axis='columns'` to check if values in each row all return True.

```
>>> df.all(axis='columns')
0     True
1    False
dtype: bool
```

Or `axis=None` for whether every value is True.

```
>>> df.all(axis=None)
False
```

AlloViz.AlloViz.Elements.Element.any

`Element.any`(*, *axis: Axis = 0*, *bool_only: bool = False*, *skipna: bool = True*, ***kwargs*) → Series | bool

Return whether any element is True, potentially over an axis.

Returns False unless there is at least one element within a series or along a Dataframe axis that is True or equivalent (e.g. non-zero or non-empty).

Parameters

axis
[{0 or 'index', 1 or 'columns', None}, default 0] Indicate which axis or axes should be reduced. For *Series* this parameter is unused and defaults to 0.

- 0 / 'index' : reduce the index, return a Series whose index is the original column labels.
- 1 / 'columns' : reduce the columns, return a Series whose index is the original index.
- None : reduce all axes, return a scalar.

bool_only
[bool, default False] Include only boolean columns. Not implemented for Series.

skipna
[bool, default True] Exclude NA/null values. If the entire row/column is NA and skipna is True, then the result will be False, as for an empty row/column. If skipna is False, then NA are treated as True, because these are not equal to zero.

****kwargs**
[any, default None] Additional keywords have no effect but might be accepted for compatibility with NumPy.

Returns

Series or DataFrame
If level is specified, then, DataFrame is returned; otherwise, Series is returned.

See also:

`numpy.any`
Numpy version of this method.

`Series.any`
Return whether any element is True.

`Series.all`
Return whether all elements are True.

DataFrame.any
Return whether any element is True over requested axis.

DataFrame.all
Return whether all elements are True over requested axis.

Examples

Series

For Series input, the output is a scalar indicating whether any element is True.

```
>>> pd.Series([False, False]).any()
False
>>> pd.Series([True, False]).any()
True
>>> pd.Series([], dtype="float64").any()
False
>>> pd.Series([np.nan]).any()
False
>>> pd.Series([np.nan]).any(skipna=False)
True
```

DataFrame

Whether each column contains at least one True element (the default).

```
>>> df = pd.DataFrame({"A": [1, 2], "B": [0, 2], "C": [0, 0]})
>>> df
   A  B  C
0  1  0  0
1  2  2  0
```

```
>>> df.any()
A     True
B     True
C    False
dtype: bool
```

Aggregating over the columns.

```
>>> df = pd.DataFrame({"A": [True, False], "B": [1, 2]})
>>> df
       A  B
0   True  1
1  False  2
```

```
>>> df.any(axis='columns')
0    True
1    True
dtype: bool
```

```
>>> df = pd.DataFrame({"A": [True, False], "B": [1, 0]})
>>> df
```

(continues on next page)

(continued from previous page)

```
       A  B
0   True  1
1  False  0
```

```
>>> df.any(axis='columns')
0    True
1    False
dtype: bool
```

Aggregating over the entire DataFrame with `axis=None`.

```
>>> df.any(axis=None)
True
```

any for an empty DataFrame is an empty Series.

```
>>> pd.DataFrame([]).any()
Series([], dtype: bool)
```

AlloViz.AlloViz.Elements.Element.apply

`Element.apply`(*func: AggFuncType*, *axis: Axis = 0*, *raw: bool = False*, *result_type: Literal['expand', 'reduce', 'broadcast'] | None = None*, *args=()*, *by_row: Literal[False, 'compat'] = 'compat'*, ***kwargs*)

Apply a function along an axis of the DataFrame.

Objects passed to the function are Series objects whose index is either the DataFrame's index (`axis=0`) or the DataFrame's columns (`axis=1`). By default (`result_type=None`), the final return type is inferred from the return type of the applied function. Otherwise, it depends on the *result_type* argument.

Parameters

func
: [function] Function to apply to each column or row.

axis
: [{0 or 'index', 1 or 'columns'}, default 0] Axis along which the function is applied:

- 0 or 'index': apply function to each column.
- 1 or 'columns': apply function to each row.

raw
: [bool, default False] Determines if row or column is passed as a Series or ndarray object:

- `False` : passes each row or column as a Series to the function.
- `True` : the passed function will receive ndarray objects instead. If you are just applying a NumPy reduction function this will achieve much better performance.

result_type
: [{'expand', 'reduce', 'broadcast', None}, default None] These only act when `axis=1` (columns):

- 'expand' : list-like results will be turned into columns.

- 'reduce' : returns a Series if possible rather than expanding list-like results. This is the opposite of 'expand'.
- 'broadcast' : results will be broadcast to the original shape of the DataFrame, the original index and columns will be retained.

The default behaviour (None) depends on the return value of the applied function: list-like results will be returned as a Series of those. However if the apply function returns a Series these are expanded to columns.

args
[tuple] Positional arguments to pass to *func* in addition to the array/series.

by_row
[False or "compat", default "compat"] Only has an effect when `func` is a listlike or dictlike of funcs and the func isn't a string. If "compat", will if possible first translate the func into pandas methods (e.g. `Series().apply(np.sum)` will be translated to `Series().sum()`). If that doesn't work, will try call to apply again with `by_row=True` and if that fails, will call apply again with `by_row=False` (backward compatible). If False, the funcs will be passed the whole Series at once.

New in version 2.1.0.

****kwargs**
Additional keyword arguments to pass as keywords arguments to *func*.

Returns

Series or DataFrame
Result of applying `func` along the given axis of the DataFrame.

See also:

`DataFrame.map`
For elementwise operations.

`DataFrame.aggregate`
Only perform aggregating type operations.

`DataFrame.transform`
Only perform transforming type operations.

Notes

Functions that mutate the passed object can produce unexpected behavior or errors and are not supported. See Mutating with User Defined Function (UDF) methods for more details.

Examples

```
>>> df = pd.DataFrame([[4, 9]] * 3, columns=['A', 'B'])
>>> df
   A  B
0  4  9
1  4  9
2  4  9
```

Using a numpy universal function (in this case the same as `np.sqrt(df)`):

```
>>> df.apply(np.sqrt)
     A    B
0  2.0  3.0
1  2.0  3.0
2  2.0  3.0
```

Using a reducing function on either axis

```
>>> df.apply(np.sum, axis=0)
A    12
B    27
dtype: int64
```

```
>>> df.apply(np.sum, axis=1)
0    13
1    13
2    13
dtype: int64
```

Returning a list-like will result in a Series

```
>>> df.apply(lambda x: [1, 2], axis=1)
0    [1, 2]
1    [1, 2]
2    [1, 2]
dtype: object
```

Passing `result_type='expand'` will expand list-like results to columns of a Dataframe

```
>>> df.apply(lambda x: [1, 2], axis=1, result_type='expand')
   0  1
0  1  2
1  1  2
2  1  2
```

Returning a Series inside the function is similar to passing `result_type='expand'`. The resulting column names will be the Series index.

```
>>> df.apply(lambda x: pd.Series([1, 2], index=['foo', 'bar']), axis=1)
   foo  bar
0    1    2
1    1    2
2    1    2
```

Passing `result_type='broadcast'` will ensure the same shape result, whether list-like or scalar is returned by the function, and broadcast it along the axis. The resulting column names will be the originals.

```
>>> df.apply(lambda x: [1, 2], axis=1, result_type='broadcast')
   A  B
0  1  2
1  1  2
2  1  2
```

AlloViz.AlloViz.Elements.Element.applymap

Element.**applymap**(*func: PythonFuncType*, *na_action: NaAction | None = None*, ***kwargs*) → DataFrame

Apply a function to a Dataframe elementwise.

Deprecated since version 2.1.0: DataFrame.applymap has been deprecated. Use DataFrame.map instead.

This method applies a function that accepts and returns a scalar to every element of a DataFrame.

Parameters

func
[callable] Python function, returns a single value from a single value.

na_action
[{None, 'ignore'}, default None] If 'ignore', propagate NaN values, without passing them to func.

****kwargs**
Additional keyword arguments to pass as keywords arguments to *func*.

Returns

DataFrame
Transformed DataFrame.

See also:

DataFrame.apply
Apply a function along input axis of DataFrame.

DataFrame.map
Apply a function along input axis of DataFrame.

DataFrame.replace
Replace values given in *to_replace* with *value*.

Examples

```
>>> df = pd.DataFrame([[1, 2.12], [3.356, 4.567]])
>>> df
       0      1
0  1.000  2.120
1  3.356  4.567
```

```
>>> df.map(lambda x: len(str(x)))
   0  1
0  3  4
1  5  5
```

AlloViz.AlloViz.Elements.Element.asfreq

Element.**asfreq**(*freq: Frequency*, *method: FillnaOptions | None = None*, *how: Literal['start', 'end'] | None = None*, *normalize: bool_t = False*, *fill_value: Hashable | None = None*) → Self

Convert time series to specified frequency.

Returns the original data conformed to a new index with the specified frequency.

If the index of this Series/DataFrame is a `PeriodIndex`, the new index is the result of transforming the original index with `PeriodIndex.asfreq` (so the original index will map one-to-one to the new index).

Otherwise, the new index will be equivalent to `pd.date_range(start, end, freq=freq)` where `start` and `end` are, respectively, the first and last entries in the original index (see `pandas.date_range()`). The values corresponding to any timesteps in the new index which were not present in the original index will be null (`NaN`), unless a method for filling such unknowns is provided (see the `method` parameter below).

The *`resample()`* method is more appropriate if an operation on each group of timesteps (such as an aggregate) is necessary to represent the data at the new frequency.

Parameters

freq
[DateOffset or str] Frequency DateOffset or string.

method
[{'backfill'/'bfill', 'pad'/'ffill'}, default None] Method to use for filling holes in reindexed Series (note this does not fill NaNs that already were present):

- 'pad' / 'ffill': propagate last valid observation forward to next valid
- 'backfill' / 'bfill': use NEXT valid observation to fill.

how
[{'start', 'end'}, default end] For PeriodIndex only (see PeriodIndex.asfreq).

normalize
[bool, default False] Whether to reset output index to midnight.

fill_value
[scalar, optional] Value to use for missing values, applied during upsampling (note this does not fill NaNs that already were present).

Returns

Series/DataFrame
Series/DataFrame object reindexed to the specified frequency.

See also:

`reindex`
Conform DataFrame to new index with optional filling logic.

Notes

To learn more about the frequency strings, please see this link.

Examples

Start by creating a series with 4 one minute timestamps.

```
>>> index = pd.date_range('1/1/2000', periods=4, freq='T')
>>> series = pd.Series([0.0, None, 2.0, 3.0], index=index)
>>> df = pd.DataFrame({'s': series})
>>> df
                     s
2000-01-01 00:00:00  0.0
2000-01-01 00:01:00  NaN
2000-01-01 00:02:00  2.0
2000-01-01 00:03:00  3.0
```

Upsample the series into 30 second bins.

```
>>> df.asfreq(freq='30S')
                     s
2000-01-01 00:00:00  0.0
2000-01-01 00:00:30  NaN
2000-01-01 00:01:00  NaN
2000-01-01 00:01:30  NaN
2000-01-01 00:02:00  2.0
2000-01-01 00:02:30  NaN
2000-01-01 00:03:00  3.0
```

Upsample again, providing a `fill value`.

```
>>> df.asfreq(freq='30S', fill_value=9.0)
                     s
2000-01-01 00:00:00  0.0
2000-01-01 00:00:30  9.0
2000-01-01 00:01:00  NaN
2000-01-01 00:01:30  9.0
2000-01-01 00:02:00  2.0
2000-01-01 00:02:30  9.0
2000-01-01 00:03:00  3.0
```

Upsample again, providing a `method`.

```
>>> df.asfreq(freq='30S', method='bfill')
                     s
2000-01-01 00:00:00  0.0
2000-01-01 00:00:30  NaN
2000-01-01 00:01:00  NaN
2000-01-01 00:01:30  2.0
2000-01-01 00:02:00  2.0
2000-01-01 00:02:30  3.0
2000-01-01 00:03:00  3.0
```

AlloViz.AlloViz.Elements.Element.asof

`Element.asof`(*where*, *subset=None*)

Return the last row(s) without any NaNs before *where*.

The last row (for each element in *where*, if list) without any NaN is taken. In case of a `DataFrame`, the last row without NaN considering only the subset of columns (if not *None*)

If there is no good value, NaN is returned for a Series or a Series of NaN values for a DataFrame

Parameters

where
[date or array-like of dates] Date(s) before which the last row(s) are returned.

subset
[str or array-like of str, default *None*] For DataFrame, if not *None*, only use these columns to check for NaNs.

Returns

scalar, Series, or DataFrame
The return can be:

- scalar : when *self* is a Series and *where* is a scalar
- Series: when *self* is a Series and *where* is an array-like, or when *self* is a DataFrame and *where* is a scalar
- DataFrame : when *self* is a DataFrame and *where* is an array-like

Return scalar, Series, or DataFrame.

See also:

merge_asof
Perform an asof merge. Similar to left join.

Notes

Dates are assumed to be sorted. Raises if this is not the case.

Examples

A Series and a scalar *where*.

```
>>> s = pd.Series([1, 2, np.nan, 4], index=[10, 20, 30, 40])
>>> s
10    1.0
20    2.0
30    NaN
40    4.0
dtype: float64
```

```
>>> s.asof(20)
2.0
```

For a sequence *where*, a Series is returned. The first value is NaN, because the first element of *where* is before the first index value.

```
>>> s.asof([5, 20])
5     NaN
20    2.0
dtype: float64
```

Missing values are not considered. The following is `2.0`, not NaN, even though NaN is at the index location for `30`.

```
>>> s.asof(30)
2.0
```

Take all columns into consideration

```
>>> df = pd.DataFrame({'a': [10., 20., 30., 40., 50.],
...                    'b': [None, None, None, None, 500]},
...                   index=pd.DatetimeIndex(['2018-02-27 09:01:00',
...                                           '2018-02-27 09:02:00',
...                                           '2018-02-27 09:03:00',
...                                           '2018-02-27 09:04:00',
...                                           '2018-02-27 09:05:00']))
>>> df.asof(pd.DatetimeIndex(['2018-02-27 09:03:30',
...                           '2018-02-27 09:04:30']))
                     a   b
2018-02-27 09:03:30 NaN NaN
2018-02-27 09:04:30 NaN NaN
```

Take a single column into consideration

```
>>> df.asof(pd.DatetimeIndex(['2018-02-27 09:03:30',
...                           '2018-02-27 09:04:30']),
...         subset=['a'])
                      a   b
2018-02-27 09:03:30  30.0 NaN
2018-02-27 09:04:30  40.0 NaN
```

AlloViz.AlloViz.Elements.Element.assign

`Element.`**`assign`**(***kwargs*) → DataFrame

Assign new columns to a DataFrame.

Returns a new object with all original columns in addition to new ones. Existing columns that are re-assigned will be overwritten.

Parameters

****kwargs**

[dict of {str: callable or Series}] The column names are keywords. If the values are callable, they are computed on the DataFrame and assigned to the new columns. The callable must not change input DataFrame (though pandas doesn't check it). If the values are not callable, (e.g. a Series, scalar, or array), they are simply assigned.

Returns

DataFrame
A new DataFrame with the new columns in addition to all the existing columns.

Notes

Assigning multiple columns within the same `assign` is possible. Later items in '**kwargs' may refer to newly created or modified columns in 'df'; items are computed and assigned into 'df' in order.

Examples

```
>>> df = pd.DataFrame({'temp_c': [17.0, 25.0]},
...                   index=['Portland', 'Berkeley'])
>>> df
          temp_c
Portland    17.0
Berkeley    25.0
```

Where the value is a callable, evaluated on *df*:

```
>>> df.assign(temp_f=lambda x: x.temp_c * 9 / 5 + 32)
          temp_c  temp_f
Portland    17.0    62.6
Berkeley    25.0    77.0
```

Alternatively, the same behavior can be achieved by directly referencing an existing Series or sequence:

```
>>> df.assign(temp_f=df['temp_c'] * 9 / 5 + 32)
          temp_c  temp_f
Portland    17.0    62.6
Berkeley    25.0    77.0
```

You can create multiple columns within the same assign where one of the columns depends on another one defined within the same assign:

```
>>> df.assign(temp_f=lambda x: x['temp_c'] * 9 / 5 + 32,
...           temp_k=lambda x: (x['temp_f'] + 459.67) * 5 / 9)
          temp_c  temp_f  temp_k
Portland    17.0    62.6  290.15
Berkeley    25.0    77.0  298.15
```

AlloViz.AlloViz.Elements.Element.astype

`Element.`**`astype`**(*dtype*, *copy: bool | None = None*, *errors: Literal['ignore', 'raise'] = 'raise'*) → None

Cast a pandas object to a specified dtype `dtype`.

Parameters

dtype
[str, data type, Series or Mapping of column name -> data type] Use a str, numpy.dtype, pandas.ExtensionDtype or Python type to cast entire pandas object to the same type. Alternatively, use a mapping, e.g. {col: dtype, ...}, where col is

a column label and dtype is a numpy.dtype or Python type to cast one or more of the DataFrame's columns to column-specific types.

copy
[bool, default True] Return a copy when `copy=True` (be very careful setting `copy=False` as changes to values then may propagate to other pandas objects).

errors
[{'raise', 'ignore'}, default 'raise'] Control raising of exceptions on invalid data for provided dtype.

- `raise` : allow exceptions to be raised
- `ignore` : suppress exceptions. On error return original object.

Returns

same type as caller

See also:

`to_datetime`
Convert argument to datetime.

`to_timedelta`
Convert argument to timedelta.

`to_numeric`
Convert argument to a numeric type.

`numpy.ndarray.astype`
Cast a numpy array to a specified type.

Notes

Changed in version 2.0.0: Using `astype` to convert from timezone-naive dtype to timezone-aware dtype will raise an exception. Use `Series.dt.tz_localize()` instead.

Examples

Create a DataFrame:

```
>>> d = {'col1': [1, 2], 'col2': [3, 4]}
>>> df = pd.DataFrame(data=d)
>>> df.dtypes
col1    int64
col2    int64
dtype: object
```

Cast all columns to int32:

```
>>> df.astype('int32').dtypes
col1    int32
col2    int32
dtype: object
```

Cast col1 to int32 using a dictionary:

```
>>> df.astype({'col1': 'int32'}).dtypes
col1    int32
col2    int64
dtype: object
```

Create a series:

```
>>> ser = pd.Series([1, 2], dtype='int32')
>>> ser
0    1
1    2
dtype: int32
>>> ser.astype('int64')
0    1
1    2
dtype: int64
```

Convert to categorical type:

```
>>> ser.astype('category')
0    1
1    2
dtype: category
Categories (2, int32): [1, 2]
```

Convert to ordered categorical type with custom ordering:

```
>>> from pandas.api.types import CategoricalDtype
>>> cat_dtype = CategoricalDtype(
...     categories=[2, 1], ordered=True)
>>> ser.astype(cat_dtype)
0    1
1    2
dtype: category
Categories (2, int64): [2 < 1]
```

Create a series of dates:

```
>>> ser_date = pd.Series(pd.date_range('20200101', periods=3))
>>> ser_date
0   2020-01-01
1   2020-01-02
2   2020-01-03
dtype: datetime64[ns]
```

AlloViz.AlloViz.Elements.Element.at_time

Element.**at_time**(*time*, *asof: bool = False*, *axis: int | Literal['index', 'columns', 'rows'] | None = None*) → None

Select values at particular time of day (e.g., 9:30AM).

Parameters

time
[datetime.time or str] The values to select.

axis
[{0 or 'index', 1 or 'columns'}, default 0] For *Series* this parameter is unused and defaults to 0.

Returns

Series or DataFrame

Raises

TypeError
If the index is not a `DatetimeIndex`

See also:

between_time
Select values between particular times of the day.

first
Select initial periods of time series based on a date offset.

last
Select final periods of time series based on a date offset.

DatetimeIndex.indexer_at_time
Get just the index locations for values at particular time of the day.

Examples

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='12H')
>>> ts = pd.DataFrame({'A': [1, 2, 3, 4]}, index=i)
>>> ts
                     A
2018-04-09 00:00:00  1
2018-04-09 12:00:00  2
2018-04-10 00:00:00  3
2018-04-10 12:00:00  4
```

```
>>> ts.at_time('12:00')
                     A
2018-04-09 12:00:00  2
2018-04-10 12:00:00  4
```

AlloViz.AlloViz.Elements.Element.backfill

Element.**backfill**(*, *axis: None | Axis = None*, *inplace: bool_t = False*, *limit: None | int = None*, *downcast: dict | None | lib.NoDefault = _NoDefault.no_default*) → Self | None

Fill NA/NaN values by using the next valid observation to fill the gap.

Deprecated since version 2.0: Series/DataFrame.backfill is deprecated. Use Series/DataFrame.bfill instead.

Returns

Series/DataFrame or None
Object with missing values filled or None if `inplace=True`.

Examples

Please see examples for `DataFrame.bfill()` or `Series.bfill()`.

AlloViz.AlloViz.Elements.Element.between_time

Element.**between_time**(*start_time*, *end_time*, *inclusive: Literal['left', 'right'] | Literal['both', 'neither'] = 'both'*, *axis: int | Literal['index', 'columns', 'rows'] | None = None*) → None

Select values between particular times of the day (e.g., 9:00-9:30 AM).

By setting `start_time` to be later than `end_time`, you can get the times that are *not* between the two times.

Parameters

start_time
[datetime.time or str] Initial time as a time filter limit.

end_time
[datetime.time or str] End time as a time filter limit.

inclusive
[{"both", "neither", "left", "right"}, default "both"] Include boundaries; whether to set each bound as closed or open.

axis
[{0 or 'index', 1 or 'columns'}, default 0] Determine range time on index or columns value. For *Series* this parameter is unused and defaults to 0.

Returns

Series or DataFrame
Data from the original object filtered to the specified dates range.

Raises

TypeError
If the index is not a `DatetimeIndex`

See also:

at_time
Select values at a particular time of the day.

first
Select initial periods of time series based on a date offset.

last
Select final periods of time series based on a date offset.

DatetimeIndex.indexer_between_time
Get just the index locations for values between particular times of the day.

Examples

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='1D20min')
>>> ts = pd.DataFrame({'A': [1, 2, 3, 4]}, index=i)
>>> ts
                     A
2018-04-09 00:00:00  1
2018-04-10 00:20:00  2
2018-04-11 00:40:00  3
2018-04-12 01:00:00  4
```

```
>>> ts.between_time('0:15', '0:45')
                     A
2018-04-10 00:20:00  2
2018-04-11 00:40:00  3
```

You get the times that are *not* between two times by setting `start_time` later than `end_time`:

```
>>> ts.between_time('0:45', '0:15')
                     A
2018-04-09 00:00:00  1
2018-04-12 01:00:00  4
```

AlloViz.AlloViz.Elements.Element.bfill

Element.**bfill**(*, *axis: None | Axis = None*, *inplace: bool_t = False*, *limit: None | int = None*, *downcast: dict | None | lib.NoDefault = _NoDefault.no_default*) → Self | None

Fill NA/NaN values by using the next valid observation to fill the gap.

Parameters

axis
[{0 or 'index'} for Series, {0 or 'index', 1 or 'columns'} for DataFrame] Axis along which to fill missing values. For *Series* this parameter is unused and defaults to 0.

inplace
[bool, default False] If True, fill in-place. Note: this will modify any other views on this object (e.g., a no-copy slice for a column in a DataFrame).

limit
[int, default None] If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled. Must be greater than 0 if not None.

downcast
[dict, default is None] A dict of item->dtype of what to downcast if possible, or the string 'infer' which will try to downcast to an appropriate equal type (e.g. float64 to int64 if possible).

Returns

Series/DataFrame or None
Object with missing values filled or None if `inplace=True`.

Examples

For Series:

```
>>> s = pd.Series([1, None, None, 2])
>>> s.bfill()
0    1.0
1    2.0
2    2.0
3    2.0
dtype: float64
>>> s.bfill(limit=1)
0    1.0
1    NaN
2    2.0
3    2.0
dtype: float64
```

With DataFrame:

```
>>> df = pd.DataFrame({'A': [1, None, None, 4], 'B': [None, 5, None, 7]})
>>> df
     A     B
0   1.0   NaN
1   NaN   5.0
2   NaN   NaN
3   4.0   7.0
>>> df.bfill()
     A     B
0   1.0   5.0
1   4.0   5.0
2   4.0   7.0
3   4.0   7.0
>>> df.bfill(limit=1)
     A     B
0   1.0   5.0
1   NaN   5.0
2   4.0   7.0
3   4.0   7.0
```

AlloViz.AlloViz.Elements.Element.bool

Element.**bool**() → bool

Return the bool of a single element Series or DataFrame.

Deprecated since version 2.1.0: bool is deprecated and will be removed in future version of pandas

This must be a boolean scalar value, either True or False. It will raise a ValueError if the Series or DataFrame does not have exactly 1 element, or that element is not boolean (integer values 0 and 1 will also raise an exception).

Returns

bool
The value in the Series or DataFrame.

See also:

Series.astype
Change the data type of a Series, including to boolean.

DataFrame.astype
Change the data type of a DataFrame, including to boolean.

numpy.bool_
NumPy boolean data type, used by pandas for boolean values.

Examples

The method will only work for single element objects with a boolean value:

```
>>> pd.Series([True]).bool()
True
>>> pd.Series([False]).bool()
False
```

```
>>> pd.DataFrame({'col': [True]}).bool()
True
>>> pd.DataFrame({'col': [False]}).bool()
False
```

AlloViz.AlloViz.Elements.Element.boxplot

Element.**boxplot**(*column=None*, *by=None*, *ax=None*, *fontsize: None | int = None*, *rot: int = 0*, *grid: bool = True*, *figsize: tuple[float, float] | None = None*, *layout=None*, *return_type=None*, *backend=None*, ***kwargs*)

Make a box plot from DataFrame columns.

Make a box-and-whisker plot from DataFrame columns, optionally grouped by some other columns. A box plot is a method for graphically depicting groups of numerical data through their quartiles. The box extends from the Q1 to Q3 quartile values of the data, with a line at the median (Q2). The whiskers extend from the edges of box to show the range of the data. By default, they extend no more than *1.5 * IQR (IQR = Q3 - Q1)* from the edges of the box, ending at the farthest data point within that interval. Outliers are plotted as separate dots.

For further details see Wikipedia's entry for boxplot.

Parameters

column
[str or list of str, optional] Column name or list of names, or vector. Can be any valid input to `pandas.DataFrame.groupby()`.

by
[str or array-like, optional] Column in the DataFrame to `pandas.DataFrame.groupby()`. One box-plot will be done per value of columns in *by*.

ax
[object of class matplotlib.axes.Axes, optional] The matplotlib axes to be used by boxplot.

fontsize
[float or str] Tick label font size in points or as a string (e.g., *large*).

rot
[float, default 0] The rotation angle of labels (in degrees) with respect to the screen coordinate system.

grid
[bool, default True] Setting this to True will show the grid.

figsize
[A tuple (width, height) in inches] The size of the figure to create in matplotlib.

layout
[tuple (rows, columns), optional] For example, (3, 5) will display the subplots using 3 rows and 5 columns, starting from the top-left.

return_type
[{'axes', 'dict', 'both'} or None, default 'axes'] The kind of object to return. The default is `axes`.

- 'axes' returns the matplotlib axes the boxplot is drawn on.
- 'dict' returns a dictionary whose values are the matplotlib Lines of the boxplot.
- 'both' returns a namedtuple with the axes and dict.
- when grouping with `by`, a Series mapping columns to `return_type` is returned.

  If `return_type` is *None*, a NumPy array of axes with the same shape as `layout` is returned.

backend
[str, default None] Backend to use instead of the backend specified in the option `plotting.backend`. For instance, 'matplotlib'. Alternatively, to specify the `plotting.backend` for the whole session, set `pd.options.plotting.backend`.

****kwargs**
All other plotting keyword arguments to be passed to `matplotlib.pyplot.boxplot()`.

Returns

result
See Notes.

See also:

`pandas.Series.plot.hist`
: Make a histogram.

`matplotlib.pyplot.boxplot`
: Matplotlib equivalent plot.

Notes

The return type depends on the *return_type* parameter:

- 'axes' : object of class matplotlib.axes.Axes
- 'dict' : dict of matplotlib.lines.Line2D objects
- 'both' : a namedtuple with structure (ax, lines)

For data grouped with `by`, return a Series of the above or a numpy array:

- `Series`
- `array` (for `return_type = None`)

Use `return_type='dict'` when you want to tweak the appearance of the lines after plotting. In this case a dict containing the Lines making up the boxes, caps, fliers, medians, and whiskers is returned.

Examples

Boxplots can be created for every column in the dataframe by `df.boxplot()` or indicating the columns to be used:

```
>>> np.random.seed(1234)
>>> df = pd.DataFrame(np.random.randn(10, 4),
...                   columns=['Col1', 'Col2', 'Col3', 'Col4'])
>>> boxplot = df.boxplot(column=['Col1', 'Col2', 'Col3'])
```

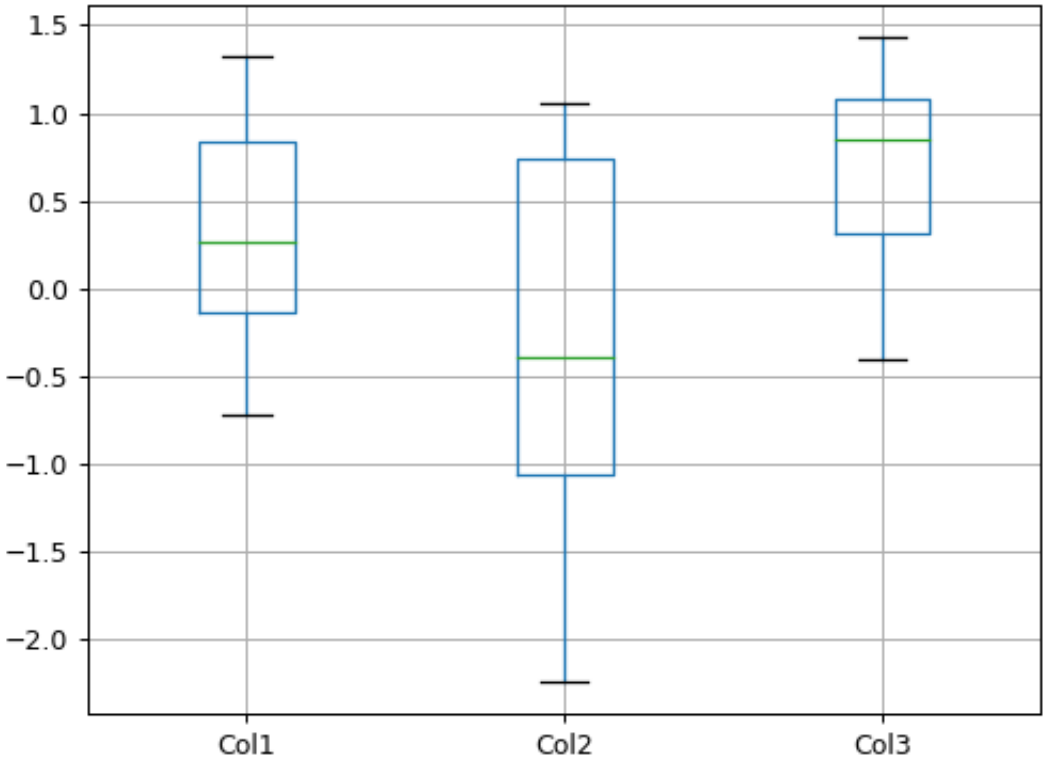
Boxplots of variables distributions grouped by the values of a third variable can be created using the option `by`. For instance:

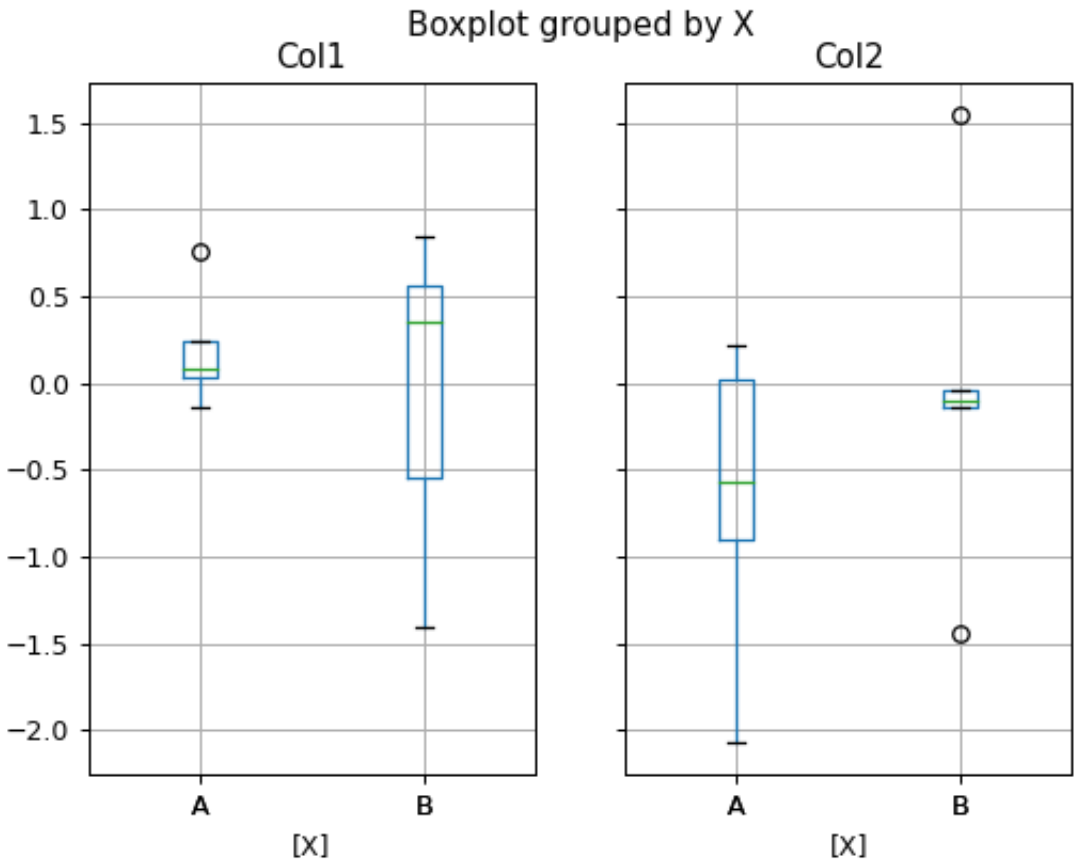
```
>>> df = pd.DataFrame(np.random.randn(10, 2),
...                   columns=['Col1', 'Col2'])
>>> df['X'] = pd.Series(['A', 'A', 'A', 'A', 'A',
...                      'B', 'B', 'B', 'B', 'B'])
>>> boxplot = df.boxplot(by='X')
```

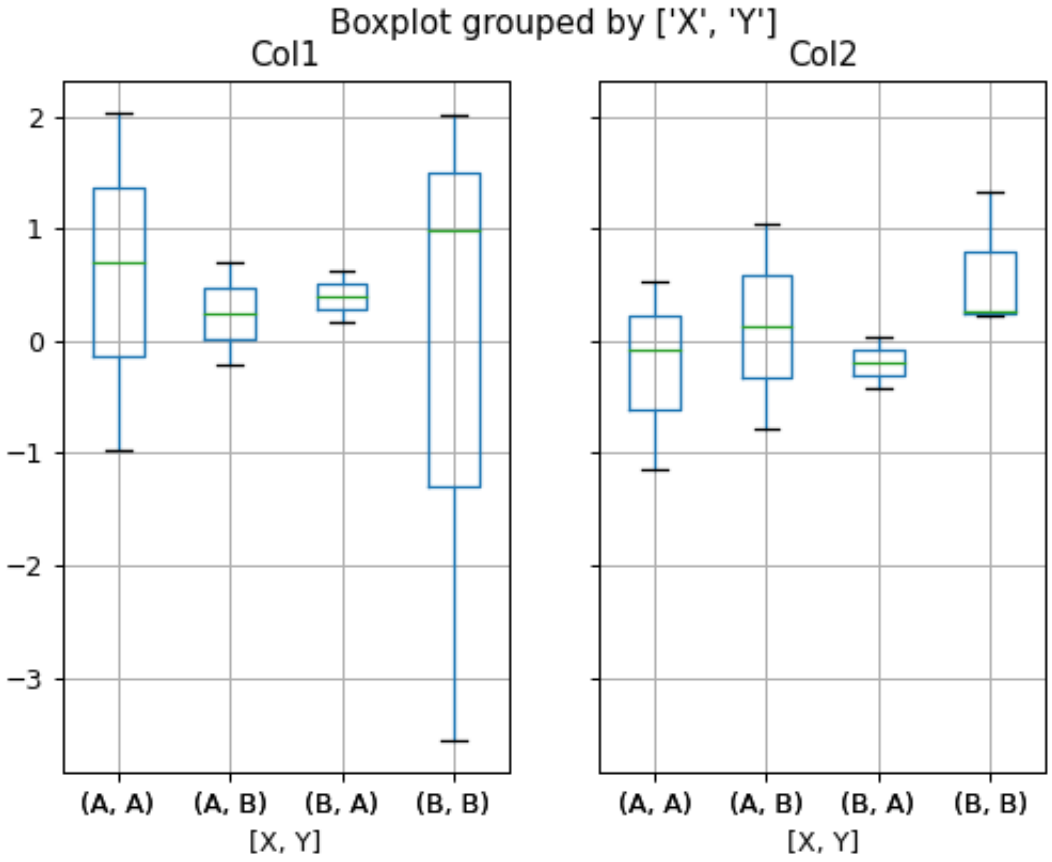
A list of strings (i.e. `['X', 'Y']`) can be passed to boxplot in order to group the data by combination of the variables in the x-axis:

```
>>> df = pd.DataFrame(np.random.randn(10, 3),
...                   columns=['Col1', 'Col2', 'Col3'])
>>> df['X'] = pd.Series(['A', 'A', 'A', 'A', 'A',
...                      'B', 'B', 'B', 'B', 'B'])
>>> df['Y'] = pd.Series(['A', 'B', 'A', 'B', 'A',
...                      'B', 'A', 'B', 'A', 'B'])
>>> boxplot = df.boxplot(column=['Col1', 'Col2'], by=['X', 'Y'])
```

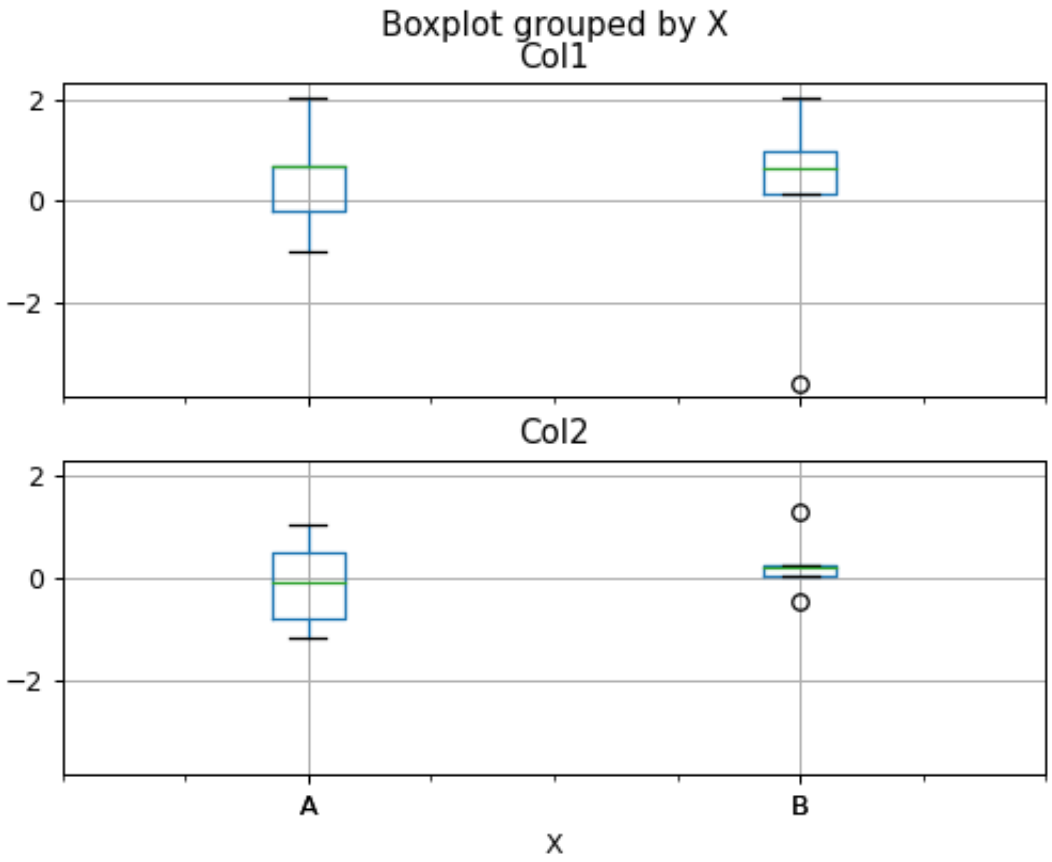
The layout of boxplot can be adjusted giving a tuple to `layout`:


1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
Col1
Col2
Col3


Boxplot grouped by X
Col1
Col2
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
A
B
[X]
A
B
[X]


Boxplot grouped by ['X', 'Y']
Col1
Col2
2
1
0
−1
−2
−3
(A, A)
(A, B)
(B, A)
(B, B)
[X, Y]
(A, A)
(A, B)
(B, A)
(B, B)
[X, Y]

```
>>> boxplot = df.boxplot(column=['Col1', 'Col2'], by='X',
...                      layout=(2, 1))
```



Additional formatting can be done to the boxplot, like suppressing the grid (`grid=False`), rotating the labels in the x-axis (i.e. `rot=45`) or changing the fontsize (i.e. `fontsize=15`):

```
>>> boxplot = df.boxplot(grid=False, rot=45, fontsize=15)
```

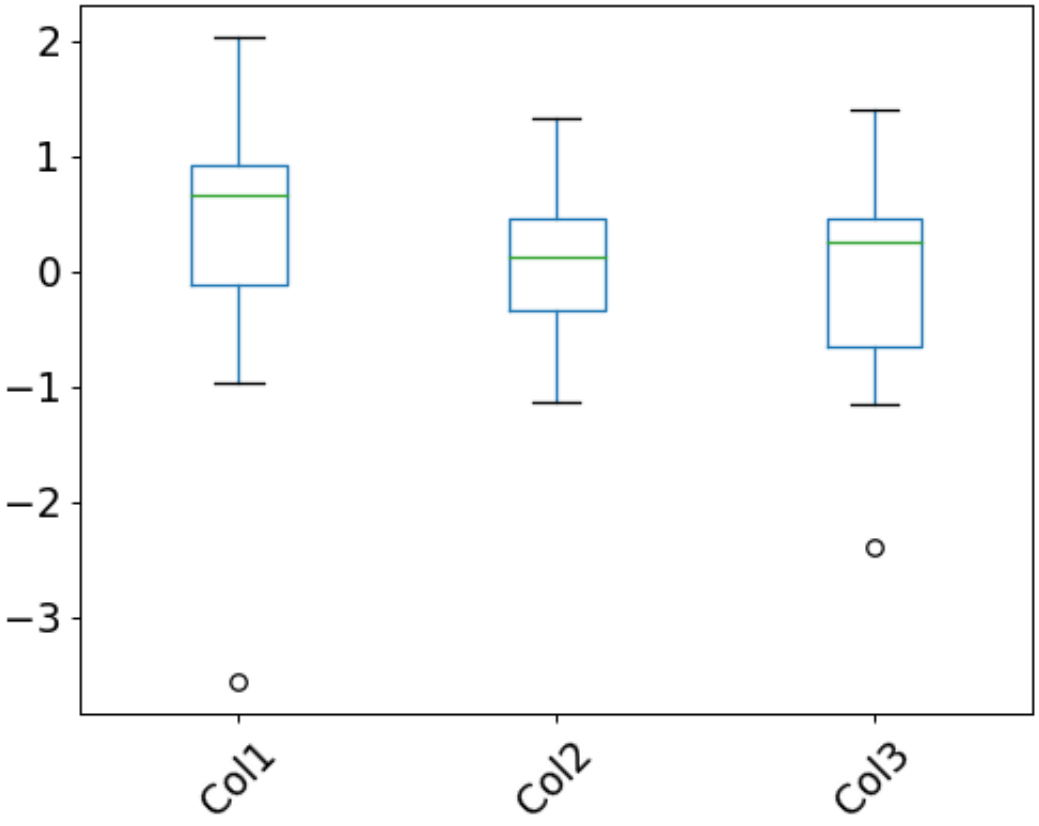
The parameter `return_type` can be used to select the type of element returned by *boxplot*. When `return_type='axes'` is selected, the matplotlib axes on which the boxplot is drawn are returned:

```
>>> boxplot = df.boxplot(column=['Col1', 'Col2'], return_type='axes')
>>> type(boxplot)
<class 'matplotlib.axes._axes.Axes'>
```

When grouping with `by`, a Series mapping columns to `return_type` is returned:

```
>>> boxplot = df.boxplot(column=['Col1', 'Col2'], by='X',
...                      return_type='axes')
>>> type(boxplot)
<class 'pandas.core.series.Series'>
```

If `return_type` is *None*, a NumPy array of axes with the same shape as `layout` is returned:


2
1
0
−1
−2
−3
Col1
Col2
Col3

```
>>> boxplot = df.boxplot(column=['Col1', 'Col2'], by='X',
...                      return_type=None)
>>> type(boxplot)
<class 'numpy.ndarray'>
```

AlloViz.AlloViz.Elements.Element.clip

Element.**clip**(*lower=None*, *upper=None*, *, *axis: Axis | None = None*, *inplace: bool_t = False*, ***kwargs*) → Self | None

Trim values at input threshold(s).

Assigns values outside boundary to boundary values. Thresholds can be singular values or array like, and in the latter case the clipping is performed element-wise in the specified axis.

Parameters

lower
[float or array-like, default None] Minimum threshold value. All values below this threshold will be set to it. A missing threshold (e.g *NA*) will not clip the value.

upper
[float or array-like, default None] Maximum threshold value. All values above this threshold will be set to it. A missing threshold (e.g *NA*) will not clip the value.

axis
[{{0 or 'index', 1 or 'columns', None}}, default None] Align object with lower and upper along the given axis. For *Series* this parameter is unused and defaults to *None*.

inplace
[bool, default False] Whether to perform the operation in place on the data.

***args, **kwargs**
Additional keywords have no effect but might be accepted for compatibility with numpy.

Returns

Series or DataFrame or None
Same type as calling object with the values outside the clip boundaries replaced or None if `inplace=True`.

See also:

`Series.clip`
Trim values at input threshold in series.

`DataFrame.clip`
Trim values at input threshold in dataframe.

`numpy.clip`
Clip (limit) the values in an array.

Examples

```
>>> data = {'col_0': [9, -3, 0, -1, 5], 'col_1': [-2, -7, 6, 8, -5]}
>>> df = pd.DataFrame(data)
>>> df
   col_0  col_1
0      9     -2
1     -3     -7
2      0      6
3     -1      8
4      5     -5
```

Clips per column using lower and upper thresholds:

```
>>> df.clip(-4, 6)
   col_0  col_1
0      6     -2
1     -3     -4
2      0      6
3     -1      6
4      5     -4
```

Clips using specific lower and upper thresholds per column element:

```
>>> t = pd.Series([2, -4, -1, 6, 3])
>>> t
0    2
1   -4
2   -1
3    6
4    3
dtype: int64
```

```
>>> df.clip(t, t + 4, axis=0)
   col_0  col_1
0      6      2
1     -3     -4
2      0      3
3      6      8
4      5      3
```

Clips using specific lower threshold per column element, with missing values:

```
>>> t = pd.Series([2, -4, np.nan, 6, 3])
>>> t
0    2.0
1   -4.0
2    NaN
3    6.0
4    3.0
dtype: float64
```

```
>>> df.clip(t, axis=0)
col_0  col_1
0      9      2
1     -3     -4
2      0      6
3      6      8
4      5      3
```

AlloViz.AlloViz.Elements.Element.combine

Element.**combine**(*other: DataFrame*, *func: Callable[[Series, Series], Series | Hashable]*, *fill_value=None*, *overwrite: bool = True*) → DataFrame

Perform column-wise combine with another DataFrame.

Combines a DataFrame with *other* DataFrame using *func* to element-wise combine columns. The row and column indexes of the resulting DataFrame will be the union of the two.

Parameters

other
[DataFrame] The DataFrame to merge column-wise.

func
[function] Function that takes two series as inputs and return a Series or a scalar. Used to merge the two dataframes column by columns.

fill_value
[scalar value, default None] The value to fill NaNs with prior to passing any column to the merge func.

overwrite
[bool, default True] If True, columns in *self* that do not exist in *other* will be overwritten with NaNs.

Returns

DataFrame
Combination of the provided DataFrames.

See also:

DataFrame.combine_first
Combine two DataFrame objects and default to non-null values in frame calling the method.

Examples

Combine using a simple function that chooses the smaller column.

```
>>> df1 = pd.DataFrame({'A': [0, 0], 'B': [4, 4]})
>>> df2 = pd.DataFrame({'A': [1, 1], 'B': [3, 3]})
>>> take_smaller = lambda s1, s2: s1 if s1.sum() < s2.sum() else s2
>>> df1.combine(df2, take_smaller)
   A  B
0  0  3
1  0  3
```

Example using a true element-wise combine function.

```
>>> df1 = pd.DataFrame({'A': [5, 0], 'B': [2, 4]})
>>> df2 = pd.DataFrame({'A': [1, 1], 'B': [3, 3]})
>>> df1.combine(df2, np.minimum)
   A  B
0  1  2
1  0  3
```

Using *fill_value* fills Nones prior to passing the column to the merge function.

```
>>> df1 = pd.DataFrame({'A': [0, 0], 'B': [None, 4]})
>>> df2 = pd.DataFrame({'A': [1, 1], 'B': [3, 3]})
>>> df1.combine(df2, take_smaller, fill_value=-5)
   A    B
0  0 -5.0
1  0  4.0
```

However, if the same element in both dataframes is None, that None is preserved

```
>>> df1 = pd.DataFrame({'A': [0, 0], 'B': [None, 4]})
>>> df2 = pd.DataFrame({'A': [1, 1], 'B': [None, 3]})
>>> df1.combine(df2, take_smaller, fill_value=-5)
    A    B
0  0 -5.0
1  0  3.0
```

Example that demonstrates the use of *overwrite* and behavior when the axis differ between the dataframes.

```
>>> df1 = pd.DataFrame({'A': [0, 0], 'B': [4, 4]})
>>> df2 = pd.DataFrame({'B': [3, 3], 'C': [-10, 1], }, index=[1, 2])
>>> df1.combine(df2, take_smaller)
     A    B     C
0  NaN  NaN   NaN
1  NaN  3.0 -10.0
2  NaN  3.0   1.0
```

```
>>> df1.combine(df2, take_smaller, overwrite=False)
     A    B     C
0  0.0  NaN   NaN
1  0.0  3.0 -10.0
2  NaN  3.0   1.0
```

Demonstrating the preference of the passed in dataframe.

```
>>> df2 = pd.DataFrame({'B': [3, 3], 'C': [1, 1], }, index=[1, 2])
>>> df2.combine(df1, take_smaller)
   A    B   C
0  0.0  NaN NaN
1  0.0  3.0 NaN
2  NaN  3.0 NaN
```

```
>>> df2.combine(df1, take_smaller, overwrite=False)
     A    B   C
```

(continues on next page)

(continued from previous page)

```
0  0.0  NaN NaN
1  0.0  3.0 1.0
2  NaN  3.0 1.0
```

AlloViz.AlloViz.Elements.Element.combine_first

Element.**combine_first**(*other: DataFrame*) → DataFrame

Update null elements with value in the same location in *other*.

Combine two DataFrame objects by filling null values in one DataFrame with non-null values from other DataFrame. The row and column indexes of the resulting DataFrame will be the union of the two. The resulting dataframe contains the 'first' dataframe values and overrides the second one values where both first.loc[index, col] and second.loc[index, col] are not missing values, upon calling first.combine_first(second).

Parameters

other
[DataFrame] Provided DataFrame to use to fill null values.

Returns

DataFrame
The result of combining the provided DataFrame with the other object.

See also:

DataFrame.combine
Perform series-wise operation on two DataFrames using a given function.

Examples

```
>>> df1 = pd.DataFrame({'A': [None, 0], 'B': [None, 4]})
>>> df2 = pd.DataFrame({'A': [1, 1], 'B': [3, 3]})
>>> df1.combine_first(df2)
     A    B
0  1.0  3.0
1  0.0  4.0
```

Null values still persist if the location of that null value does not exist in *other*

```
>>> df1 = pd.DataFrame({'A': [None, 0], 'B': [4, None]})
>>> df2 = pd.DataFrame({'B': [3, 3], 'C': [1, 1]}, index=[1, 2])
>>> df1.combine_first(df2)
     A    B    C
0  NaN  4.0  NaN
1  0.0  3.0  1.0
2  NaN  3.0  1.0
```

AlloViz.AlloViz.Elements.Element.compare

Element.**compare**(*other: DataFrame*, *align_axis: Axis = 1*, *keep_shape: bool = False*, *keep_equal: bool = False*, *result_names: Suffixes = ('self', 'other')*) → DataFrame

Compare to another DataFrame and show the differences.

Parameters

other
[DataFrame] Object to compare with.

align_axis
[{0 or 'index', 1 or 'columns'}, default 1] Determine which axis to align the comparison on.

- **0, or 'index'**
  [Resulting differences are stacked vertically] with rows drawn alternately from self and other.
- **1, or 'columns'**
  [Resulting differences are aligned horizontally] with columns drawn alternately from self and other.

keep_shape
[bool, default False] If true, all rows and columns are kept. Otherwise, only the ones with different values are kept.

keep_equal
[bool, default False] If true, the result keeps values that are equal. Otherwise, equal values are shown as NaNs.

result_names
[tuple, default ('self', 'other')] Set the dataframes names in the comparison.

New in version 1.5.0.

Returns

DataFrame
DataFrame that shows the differences stacked side by side.

The resulting index will be a MultiIndex with 'self' and 'other' stacked alternately at the inner level.

Raises

ValueError
When the two DataFrames don't have identical labels or shape.

See also:

`Series.compare`
Compare with another Series and show differences.

`DataFrame.equals`
Test whether two objects contain the same elements.

Notes

Matching NaNs will not appear as a difference.

Can only compare identically-labeled (i.e. same shape, identical row and column labels) DataFrames

Examples

```
>>> df = pd.DataFrame(
...     {
...         "col1": ["a", "a", "b", "b", "a"],
...         "col2": [1.0, 2.0, 3.0, np.nan, 5.0],
...         "col3": [1.0, 2.0, 3.0, 4.0, 5.0]
...     },
...     columns=["col1", "col2", "col3"],
... )
>>> df
  col1  col2  col3
0    a   1.0   1.0
1    a   2.0   2.0
2    b   3.0   3.0
3    b   NaN   4.0
4    a   5.0   5.0
```

```
>>> df2 = df.copy()
>>> df2.loc[0, 'col1'] = 'c'
>>> df2.loc[2, 'col3'] = 4.0
>>> df2
  col1  col2  col3
0    c   1.0   1.0
1    a   2.0   2.0
2    b   3.0   4.0
3    b   NaN   4.0
4    a   5.0   5.0
```

Align the differences on columns

```
>>> df.compare(df2)
  col1       col3
  self other self other
0    a     c  NaN   NaN
2  NaN   NaN  3.0   4.0
```

Assign result_names

```
>>> df.compare(df2, result_names=("left", "right"))
  col1       col3
  left right left right
0    a     c  NaN   NaN
2  NaN   NaN  3.0   4.0
```

Stack the differences on rows

```
>>> df.compare(df2, align_axis=0)
        col1  col3
0 self     a   NaN
  other    c   NaN
2 self   NaN   3.0
  other  NaN   4.0
```

Keep the equal values

```
>>> df.compare(df2, keep_equal=True)
  col1       col3
  self other self other
0    a     c  1.0   1.0
2    b     b  3.0   4.0
```

Keep all original rows and columns

```
>>> df.compare(df2, keep_shape=True)
  col1       col2       col3
  self other self other self other
0    a     c  NaN   NaN  NaN   NaN
1  NaN   NaN  NaN   NaN  NaN   NaN
2  NaN   NaN  NaN   NaN  3.0   4.0
3  NaN   NaN  NaN   NaN  NaN   NaN
4  NaN   NaN  NaN   NaN  NaN   NaN
```

Keep all original rows and columns and also all original values

```
>>> df.compare(df2, keep_shape=True, keep_equal=True)
  col1       col2       col3
  self other self other self other
0    a     c  1.0   1.0  1.0   1.0
1    a     a  2.0   2.0  2.0   2.0
2    b     b  3.0   3.0  3.0   4.0
3    b     b  NaN   NaN  4.0   4.0
4    a     a  5.0   5.0  5.0   5.0
```

AlloViz.AlloViz.Elements.Element.convert_dtypes

Element.**convert_dtypes**(*infer_objects: bool = True*, *convert_string: bool = True*, *convert_integer: bool = True*, *convert_boolean: bool = True*, *convert_floating: bool = True*, *dtype_backend: Literal['pyarrow', 'numpy_nullable'] = 'numpy_nullable'*) → None

Convert columns to the best possible dtypes using dtypes supporting `pd.NA`.

Parameters

infer_objects
[bool, default True] Whether object dtypes should be converted to the best possible types.

convert_string
[bool, default True] Whether object dtypes should be converted to `StringDtype()`.

convert_integer
[bool, default True] Whether, if possible, conversion can be done to integer extension types.

convert_boolean
[bool, defaults True] Whether object dtypes should be converted to `BooleanDtypes()`.

convert_floating
[bool, defaults True] Whether, if possible, conversion can be done to floating extension types. If *convert_integer* is also True, preference will be give to integer dtypes if the floats can be faithfully casted to integers.

New in version 1.2.0.

dtype_backend
[{'numpy_nullable', 'pyarrow'}, default 'numpy_nullable'] Back-end data type applied to the resultant `DataFrame` (still experimental). Behaviour is as follows:

- `"numpy_nullable"`: returns nullable-dtype-backed `DataFrame` (default).
- `"pyarrow"`: returns pyarrow-backed nullable `ArrowDtype` DataFrame.

New in version 2.0.

Returns

Series or DataFrame
Copy of input object with new dtype.

See also:

`infer_objects`
Infer dtypes of objects.

`to_datetime`
Convert argument to datetime.

`to_timedelta`
Convert argument to timedelta.

`to_numeric`
Convert argument to a numeric type.

Notes

By default, `convert_dtypes` will attempt to convert a Series (or each Series in a DataFrame) to dtypes that support `pd.NA`. By using the options `convert_string`, `convert_integer`, `convert_boolean` and `convert_floating`, it is possible to turn off individual conversions to `StringDtype`, the integer extension types, `BooleanDtype` or floating extension types, respectively.

For object-dtyped columns, if `infer_objects` is `True`, use the inference rules as during normal Series/DataFrame construction. Then, if possible, convert to `StringDtype`, `BooleanDtype` or an appropriate integer or floating extension type, otherwise leave as `object`.

If the dtype is integer, convert to an appropriate integer extension type.

If the dtype is numeric, and consists of all integers, convert to an appropriate integer extension type. Otherwise, convert to an appropriate floating extension type.

Changed in version 1.2: Starting with pandas 1.2, this method also converts float columns to the nullable floating extension type.

In the future, as new dtypes are added that support `pd.NA`, the results of this method will change to support those new dtypes.

Examples

```
>>> df = pd.DataFrame(
...     {
...         "a": pd.Series([1, 2, 3], dtype=np.dtype("int32")),
...         "b": pd.Series(["x", "y", "z"], dtype=np.dtype("O")),
...         "c": pd.Series([True, False, np.nan], dtype=np.dtype("O")),
...         "d": pd.Series(["h", "i", np.nan], dtype=np.dtype("O")),
...         "e": pd.Series([10, np.nan, 20], dtype=np.dtype("float")),
...         "f": pd.Series([np.nan, 100.5, 200], dtype=np.dtype("float")),
...     }
... )
```

Start with a DataFrame with default dtypes.

```
>>> df
   a  b      c    d     e      f
0  1  x   True    h  10.0    NaN
1  2  y  False    i   NaN  100.5
2  3  z    NaN  NaN  20.0  200.0
```

```
>>> df.dtypes
a      int32
b     object
c     object
d     object
e    float64
f    float64
dtype: object
```

Convert the DataFrame to use best possible dtypes.

```
>>> dfn = df.convert_dtypes()
>>> dfn
   a  b      c     d     e      f
0  1  x   True     h    10   <NA>
1  2  y  False     i  <NA>  100.5
2  3  z   <NA>  <NA>    20  200.0
```

```
>>> dfn.dtypes
a             Int32
b    string[python]
c           boolean
d    string[python]
e             Int64
f           Float64
dtype: object
```

Start with a Series of strings and missing data represented by `np.nan`.

```
>>> s = pd.Series(["a", "b", np.nan])
>>> s
0      a
1      b
2    NaN
dtype: object
```

Obtain a Series with dtype `StringDtype`.

```
>>> s.convert_dtypes()
0       a
1       b
2    <NA>
dtype: string
```

AlloViz.AlloViz.Elements.Element.copy

`Element.copy`(*deep: bool | None = True*) → None

Make a copy of this object's indices and data.

When `deep=True` (default), a new object will be created with a copy of the calling object's data and indices. Modifications to the data or indices of the copy will not be reflected in the original object (see notes below).

When `deep=False`, a new object will be created without copying the calling object's data or index (only references to the data and index are copied). Any changes to the data of the original will be reflected in the shallow copy (and vice versa).

Parameters

deep
[bool, default True] Make a deep copy, including a copy of the data and the indices. With `deep=False` neither the indices nor the data are copied.

Returns

Series or DataFrame
Object type matches caller.

Notes

When `deep=True`, data is copied but actual Python objects will not be copied recursively, only the reference to the object. This is in contrast to *copy.deepcopy* in the Standard Library, which recursively copies object data (see examples below).

While `Index` objects are copied when `deep=True`, the underlying numpy array is not copied for performance reasons. Since `Index` is immutable, the underlying data can be safely shared and a copy is not needed.

Since pandas is not thread safe, see the gotchas when copying in a threading environment.

When `copy_on_write` in pandas config is set to `True`, the `copy_on_write` config takes effect even when `deep=False`. This means that any changes to the copied data would make a new copy of the data upon write (and vice versa). Changes made to either the original or copied variable would not be reflected in the counterpart. See Copy_on_Write for more information.

Examples

```
>>> s = pd.Series([1, 2], index=["a", "b"])
>>> s
a    1
b    2
dtype: int64
```

```
>>> s_copy = s.copy()
>>> s_copy
a    1
b    2
dtype: int64
```

Shallow copy versus default (deep) copy:

```
>>> s = pd.Series([1, 2], index=["a", "b"])
>>> deep = s.copy()
>>> shallow = s.copy(deep=False)
```

Shallow copy shares data and index with original.

```
>>> s is shallow
False
>>> s.values is shallow.values and s.index is shallow.index
True
```

Deep copy has own copy of data and index.

```
>>> s is deep
False
>>> s.values is deep.values or s.index is deep.index
False
```

Updates to the data shared by shallow copy and original is reflected in both; deep copy remains unchanged.

```
>>> s.iloc[0] = 3
>>> shallow.iloc[1] = 4
>>> s
a    3
b    4
dtype: int64
>>> shallow
a    3
b    4
dtype: int64
>>> deep
a    1
b    2
dtype: int64
```

Note that when copying an object containing Python objects, a deep copy will copy the data, but will not do so recursively. Updating a nested data object will be reflected in the deep copy.

```
>>> s = pd.Series([[1, 2], [3, 4]])
>>> deep = s.copy()
>>> s[0][0] = 10
>>> s
0    [10, 2]
1     [3, 4]
dtype: object
>>> deep
0    [10, 2]
1     [3, 4]
dtype: object
```

** Copy-on-Write is set to true: **

```
>>> with pd.option_context("mode.copy_on_write", True):
...     s = pd.Series([1, 2], index=["a", "b"])
...     copy = s.copy(deep=False)
...     s.iloc[0] = 100
...     s
a    100
b      2
dtype: int64
>>> copy
a    1
b    2
dtype: int64
```

AlloViz.AlloViz.Elements.Element.corr

Element.**corr**(*method: CorrelationMethod = 'pearson'*, *min_periods: int = 1*, *numeric_only: bool = False*) → DataFrame

Compute pairwise correlation of columns, excluding NA/null values.

Parameters

method
[{'pearson', 'kendall', 'spearman'} or callable] Method of correlation:

- pearson : standard correlation coefficient
- kendall : Kendall Tau correlation coefficient
- spearman : Spearman rank correlation
- **callable: callable with input two 1d ndarrays**
  and returning a float. Note that the returned matrix from corr will have 1 along the diagonals and will be symmetric regardless of the callable's behavior.

min_periods
[int, optional] Minimum number of observations required per pair of columns to have a valid result. Currently only available for Pearson and Spearman correlation.

numeric_only
[bool, default False] Include only *float*, *int* or *boolean* data.

New in version 1.5.0.

Changed in version 2.0.0: The default value of `numeric_only` is now `False`.

Returns

DataFrame
Correlation matrix.

See also:

`DataFrame.corrwith`
Compute pairwise correlation with another DataFrame or Series.

`Series.corr`
Compute the correlation between two Series.

Notes

Pearson, Kendall and Spearman correlation are currently computed using pairwise complete observations.

- Pearson correlation coefficient
- Kendall rank correlation coefficient
- Spearman's rank correlation coefficient

Examples

```
>>> def histogram_intersection(a, b):
...     v = np.minimum(a, b).sum().round(decimals=1)
...     return v
>>> df = pd.DataFrame([(.2, .3), (.0, .6), (.6, .0), (.2, .1)],
...                   columns=['dogs', 'cats'])
>>> df.corr(method=histogram_intersection)
      dogs  cats
dogs   1.0   0.3
cats   0.3   1.0
```

```
>>> df = pd.DataFrame([(1, 1), (2, np.nan), (np.nan, 3), (4, 4)],
...                   columns=['dogs', 'cats'])
>>> df.corr(min_periods=3)
      dogs  cats
dogs   1.0   NaN
cats   NaN   1.0
```

AlloViz.AlloViz.Elements.Element.corrwith

Element.**corrwith**(*other: DataFrame | Series*, *axis: Axis = 0*, *drop: bool = False*, *method: CorrelationMethod = 'pearson'*, *numeric_only: bool = False*) → Series

Compute pairwise correlation.

Pairwise correlation is computed between rows or columns of DataFrame with rows or columns of Series or DataFrame. DataFrames are first aligned along both axes before computing the correlations.

Parameters

other
[DataFrame, Series] Object with which to compute correlations.

axis
[{0 or 'index', 1 or 'columns'}, default 0] The axis to use. 0 or 'index' to compute row-wise, 1 or 'columns' for column-wise.

drop
[bool, default False] Drop missing indices from result.

method
[{'pearson', 'kendall', 'spearman'} or callable] Method of correlation:

- pearson : standard correlation coefficient
- kendall : Kendall Tau correlation coefficient
- spearman : Spearman rank correlation
- **callable: callable with input two 1d ndarrays**
  and returning a float.

numeric_only
[bool, default False] Include only *float*, *int* or *boolean* data.

New in version 1.5.0.

Changed in version 2.0.0: The default value of `numeric_only` is now `False`.

Returns

Series
Pairwise correlations.

See also:

DataFrame.corr
Compute pairwise correlation of columns.

Examples

```
>>> index = ["a", "b", "c", "d", "e"]
>>> columns = ["one", "two", "three", "four"]
>>> df1 = pd.DataFrame(np.arange(20).reshape(5, 4), index=index,␣
↪columns=columns)
>>> df2 = pd.DataFrame(np.arange(16).reshape(4, 4), index=index[:4],␣
↪columns=columns)
>>> df1.corrwith(df2)
```

(continues on next page)

(continued from previous page)

```
one      1.0
two      1.0
three    1.0
four     1.0
dtype: float64
```

```
>>> df2.corrwith(df1, axis=1)
a    1.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

AlloViz.AlloViz.Elements.Element.count

Element.**count**(*axis: Axis = 0*, *numeric_only: bool = False*)

Count non-NA cells for each column or row.

The values *None*, *NaN*, *NaT*, `pandas.NA` are considered NA.

Parameters

axis
[{0 or 'index', 1 or 'columns'}, default 0] If 0 or 'index' counts are generated for each column. If 1 or 'columns' counts are generated for each row.

numeric_only
[bool, default False] Include only *float*, *int* or *boolean* data.

Returns

Series
For each column/row the number of non-NA/null entries.

See also:

`Series.count`
Number of non-NA elements in a Series.

`DataFrame.value_counts`
Count unique combinations of columns.

`DataFrame.shape`
Number of DataFrame rows and columns (including NA elements).

`DataFrame.isna`
Boolean same-sized DataFrame showing places of NA elements.

Examples

Constructing DataFrame from a dictionary:

```
>>> df = pd.DataFrame({"Person":
...                    ["John", "Myla", "Lewis", "John", "Myla"],
...                    "Age": [24., np.nan, 21., 33, 26],
...                    "Single": [False, True, True, True, False]})
>>> df
   Person   Age  Single
0    John  24.0   False
1    Myla   NaN    True
2   Lewis  21.0    True
3    John  33.0    True
4    Myla  26.0   False
```

Notice the uncounted NA values:

```
>>> df.count()
Person    5
Age       4
Single    5
dtype: int64
```

Counts for each **row**:

```
>>> df.count(axis='columns')
0    3
1    2
2    3
3    3
4    3
dtype: int64
```

AlloViz.AlloViz.Elements.Element.cov

`Element.cov`(*min_periods: None | int = None*, *ddof: int | None = 1*, *numeric_only: bool = False*) → DataFrame

Compute pairwise covariance of columns, excluding NA/null values.

Compute the pairwise covariance among the series of a DataFrame. The returned data frame is the covariance matrix of the columns of the DataFrame.

Both NA and null values are automatically excluded from the calculation. (See the note below about bias from missing values.) A threshold can be set for the minimum number of observations for each value created. Comparisons with observations below this threshold will be returned as `NaN`.

This method is generally used for the analysis of time series data to understand the relationship between different measures across time.

Parameters

min_periods
[int, optional] Minimum number of observations required per pair of columns to have a valid result.

ddof
[int, default 1] Delta degrees of freedom. The divisor used in calculations is `N - ddof`, where `N` represents the number of elements. This argument is applicable only when no `nan` is in the dataframe.

numeric_only
[bool, default False] Include only *float*, *int* or *boolean* data.

New in version 1.5.0.

Changed in version 2.0.0: The default value of `numeric_only` is now `False`.

Returns

DataFrame
The covariance matrix of the series of the DataFrame.

See also:

`Series.cov`
Compute covariance with another Series.

`core.window.ewm.ExponentialMovingWindow.cov`
Exponential weighted sample covariance.

`core.window.expanding.Expanding.cov`
Expanding sample covariance.

`core.window.rolling.Rolling.cov`
Rolling sample covariance.

Notes

Returns the covariance matrix of the DataFrame's time series. The covariance is normalized by N-ddof.

For DataFrames that have Series that are missing data (assuming that data is missing at random) the returned covariance matrix will be an unbiased estimate of the variance and covariance between the member Series.

However, for many applications this estimate may not be acceptable because the estimate covariance matrix is not guaranteed to be positive semi-definite. This could lead to estimate correlations having absolute values which are greater than one, and/or a non-invertible covariance matrix. See Estimation of covariance matrices for more details.

Examples

```
>>> df = pd.DataFrame([(1, 2), (0, 3), (2, 0), (1, 1)],
...                   columns=['dogs', 'cats'])
>>> df.cov()
          dogs      cats
dogs  0.666667 -1.000000
cats -1.000000  1.666667
```

```
>>> np.random.seed(42)
>>> df = pd.DataFrame(np.random.randn(1000, 5),
...                   columns=['a', 'b', 'c', 'd', 'e'])
>>> df.cov()
```

(continues on next page)

(continued from previous page)

```
          a         b         c         d         e
a  0.998438 -0.020161  0.059277 -0.008943  0.014144
b -0.020161  1.059352 -0.008543 -0.024738  0.009826
c  0.059277 -0.008543  1.010670 -0.001486 -0.000271
d -0.008943 -0.024738 -0.001486  0.921297 -0.013692
e  0.014144  0.009826 -0.000271 -0.013692  0.977795
```

Minimum number of periods

This method also supports an optional `min_periods` keyword that specifies the required minimum number of non-NA observations for each column pair in order to have a valid result:

```
>>> np.random.seed(42)
>>> df = pd.DataFrame(np.random.randn(20, 3),
...                   columns=['a', 'b', 'c'])
>>> df.loc[df.index[:5], 'a'] = np.nan
>>> df.loc[df.index[5:10], 'b'] = np.nan
>>> df.cov(min_periods=12)
          a         b         c
a  0.316741       NaN -0.150812
b       NaN  1.248003  0.191417
c -0.150812  0.191417  0.895202
```

AlloViz.AlloViz.Elements.Element.cummax

`Element.`**`cummax`**(*axis: Axis | None = None*, *skipna: bool = True*, **args*, ***kwargs*)

Return cumulative maximum over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative maximum.

Parameters

axis
: [{0 or 'index', 1 or 'columns'}, default 0] The index or the name of the axis. 0 is equivalent to None or 'index'. For *Series* this parameter is unused and defaults to 0.

skipna
: [bool, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

***args, **kwargs**
: Additional keywords have no effect but might be accepted for compatibility with NumPy.

Returns

Series or DataFrame
: Return cumulative maximum of Series or DataFrame.

See also:

`core.window.expanding.Expanding.max`
: Similar functionality but ignores `NaN` values.

DataFrame.max
Return the maximum over DataFrame axis.

DataFrame.cummax
Return cumulative maximum over DataFrame axis.

DataFrame.cummin
Return cumulative minimum over DataFrame axis.

DataFrame.cumsum
Return cumulative sum over DataFrame axis.

DataFrame.cumprod
Return cumulative product over DataFrame axis.

Examples

Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cummax()
0    2.0
1    NaN
2    5.0
3    5.0
4    5.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cummax(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                   columns=list('AB'))
>>> df
```

(continues on next page)

(continued from previous page)

```
     A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the maximum in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cummax()
     A    B
0  2.0  1.0
1  3.0  NaN
2  3.0  1.0
```

To iterate over columns and find the maximum in each row, use `axis=1`

```
>>> df.cummax(axis=1)
     A    B
0  2.0  2.0
1  3.0  NaN
2  1.0  1.0
```

AlloViz.AlloViz.Elements.Element.cummin

`Element.`**`cummin`**(*axis: Axis | None = None*, *skipna: bool = True*, **args*, ***kwargs*)

Return cumulative minimum over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative minimum.

Parameters

axis
: [{0 or 'index', 1 or 'columns'}, default 0] The index or the name of the axis. 0 is equivalent to None or 'index'. For *Series* this parameter is unused and defaults to 0.

skipna
: [bool, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

***args, **kwargs**
: Additional keywords have no effect but might be accepted for compatibility with NumPy.

Returns

Series or DataFrame
: Return cumulative minimum of Series or DataFrame.

See also:

`core.window.expanding.Expanding.min`
: Similar functionality but ignores `NaN` values.

`DataFrame.min`
: Return the minimum over DataFrame axis.

DataFrame.cummax
Return cumulative maximum over DataFrame axis.

DataFrame.cummin
Return cumulative minimum over DataFrame axis.

DataFrame.cumsum
Return cumulative sum over DataFrame axis.

DataFrame.cumprod
Return cumulative product over DataFrame axis.

Examples

Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cummin()
0    2.0
1    NaN
2    2.0
3   -1.0
4   -1.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cummin(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                   columns=list('AB'))
>>> df
     A    B
0  2.0  1.0
```

(continues on next page)

(continued from previous page)

```
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the minimum in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cummin()
     A    B
0  2.0  1.0
1  2.0  NaN
2  1.0  0.0
```

To iterate over columns and find the minimum in each row, use `axis=1`

```
>>> df.cummin(axis=1)
     A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

AlloViz.AlloViz.Elements.Element.cumprod

`Element.`**`cumprod`**(*axis: Axis | None = None*, *skipna: bool = True*, **args*, ***kwargs*)

Return cumulative product over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative product.

Parameters

axis
: [{0 or 'index', 1 or 'columns'}, default 0] The index or the name of the axis. 0 is equivalent to None or 'index'. For *Series* this parameter is unused and defaults to 0.

skipna
: [bool, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

***args, **kwargs**
: Additional keywords have no effect but might be accepted for compatibility with NumPy.

Returns

Series or DataFrame
: Return cumulative product of Series or DataFrame.

See also:

`core.window.expanding.Expanding.prod`
: Similar functionality but ignores `NaN` values.

`DataFrame.prod`
: Return the product over DataFrame axis.

DataFrame.cummax
Return cumulative maximum over DataFrame axis.

DataFrame.cummin
Return cumulative minimum over DataFrame axis.

DataFrame.cumsum
Return cumulative sum over DataFrame axis.

DataFrame.cumprod
Return cumulative product over DataFrame axis.

Examples

Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cumprod()
0     2.0
1     NaN
2    10.0
3   -10.0
4    -0.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cumprod(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                   columns=list('AB'))
>>> df
     A    B
0  2.0  1.0
```

(continues on next page)

(continued from previous page)

```
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the product in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cumprod()
     A    B
0  2.0  1.0
1  6.0  NaN
2  6.0  0.0
```

To iterate over columns and find the product in each row, use `axis=1`

```
>>> df.cumprod(axis=1)
     A    B
0  2.0  2.0
1  3.0  NaN
2  1.0  0.0
```

AlloViz.AlloViz.Elements.Element.cumsum

`Element.`**`cumsum`**(*axis: Axis | None = None*, *skipna: bool = True*, **args*, ***kwargs*)

Return cumulative sum over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative sum.

Parameters

axis
: [{0 or 'index', 1 or 'columns'}, default 0] The index or the name of the axis. 0 is equivalent to None or 'index'. For *Series* this parameter is unused and defaults to 0.

skipna
: [bool, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

***args, **kwargs**
: Additional keywords have no effect but might be accepted for compatibility with NumPy.

Returns

Series or DataFrame
: Return cumulative sum of Series or DataFrame.

See also:

`core.window.expanding.Expanding.sum`
: Similar functionality but ignores `NaN` values.

`DataFrame.sum`
: Return the sum over DataFrame axis.

DataFrame.cummax
Return cumulative maximum over DataFrame axis.

DataFrame.cummin
Return cumulative minimum over DataFrame axis.

DataFrame.cumsum
Return cumulative sum over DataFrame axis.

DataFrame.cumprod
Return cumulative product over DataFrame axis.

Examples

Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cumsum()
0    2.0
1    NaN
2    7.0
3    6.0
4    6.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cumsum(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                   columns=list('AB'))
>>> df
     A    B
0  2.0  1.0
```

(continues on next page)

(continued from previous page)

```
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the sum in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cumsum()
     A    B
0  2.0  1.0
1  5.0  NaN
2  6.0  1.0
```

To iterate over columns and find the sum in each row, use `axis=1`

```
>>> df.cumsum(axis=1)
     A    B
0  2.0  3.0
1  3.0  NaN
2  1.0  1.0
```

AlloViz.AlloViz.Elements.Element.describe

`Element.`**`describe`**(*percentiles=None*, *include=None*, *exclude=None*) → None

Generate descriptive statistics.

Descriptive statistics include those that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding `NaN` values.

Analyzes both numeric and object series, as well as `DataFrame` column sets of mixed data types. The output will vary depending on what is provided. Refer to the notes below for more detail.

Parameters

percentiles
[list-like of numbers, optional] The percentiles to include in the output. All should fall between 0 and 1. The default is `[.25, .5, .75]`, which returns the 25th, 50th, and 75th percentiles.

include
['all', list-like of dtypes or None (default), optional] A white list of data types to include in the result. Ignored for `Series`. Here are the options:

- 'all' : All columns of the input will be included in the output.
- A list-like of dtypes : Limits the results to the provided data types. To limit the result to numeric types submit `numpy.number`. To limit it instead to object columns submit the `numpy.object` data type. Strings can also be used in the style of `select_dtypes` (e.g. `df.describe(include=['O'])`). To select pandas categorical columns, use `'category'`
- None (default) : The result will include all numeric columns.

exclude
[list-like of dtypes or None (default), optional,] A black list of data types to omit from the result. Ignored for `Series`. Here are the options:

- A list-like of dtypes : Excludes the provided data types from the result. To exclude numeric types submit `numpy.number`. To exclude object columns submit the data type `numpy.object`. Strings can also be used in the style of `select_dtypes` (e.g. `df.describe(exclude=['O'])`). To exclude pandas categorical columns, use `'category'`
- None (default) : The result will exclude nothing.

Returns

Series or DataFrame
Summary statistics of the Series or Dataframe provided.

See also:

`DataFrame.count`
Count number of non-NA/null observations.

`DataFrame.max`
Maximum of the values in the object.

`DataFrame.min`
Minimum of the values in the object.

`DataFrame.mean`
Mean of the values.

`DataFrame.std`
Standard deviation of the observations.

`DataFrame.select_dtypes`
Subset of a DataFrame including/excluding columns based on their dtype.

Notes

For numeric data, the result's index will include `count`, `mean`, `std`, `min`, `max` as well as lower, `50` and upper percentiles. By default the lower percentile is `25` and the upper percentile is `75`. The `50` percentile is the same as the median.

For object data (e.g. strings or timestamps), the result's index will include `count`, `unique`, `top`, and `freq`. The `top` is the most common value. The `freq` is the most common value's frequency. Timestamps also include the `first` and `last` items.

If multiple object values have the highest count, then the `count` and `top` results will be arbitrarily chosen from among those with the highest count.

For mixed data types provided via a `DataFrame`, the default is to return only an analysis of numeric columns. If the dataframe consists only of object and categorical data without any numeric columns, the default is to return an analysis of both the object and categorical columns. If `include='all'` is provided as an option, the result will include a union of attributes of each type.

The *include* and *exclude* parameters can be used to limit which columns in a `DataFrame` are analyzed for the output. The parameters are ignored when analyzing a `Series`.

Examples

Describing a numeric `Series`.

```
>>> s = pd.Series([1, 2, 3])
>>> s.describe()
count    3.0
mean     2.0
std      1.0
min      1.0
25%      1.5
50%      2.0
75%      2.5
max      3.0
dtype: float64
```

Describing a categorical `Series`.

```
>>> s = pd.Series(['a', 'a', 'b', 'c'])
>>> s.describe()
count     4
unique    3
top       a
freq      2
dtype: object
```

Describing a timestamp `Series`.

```
>>> s = pd.Series([
...     np.datetime64("2000-01-01"),
...     np.datetime64("2010-01-01"),
...     np.datetime64("2010-01-01")
... ])
>>> s.describe()
count                      3
mean     2006-09-01 08:00:00
min      2000-01-01 00:00:00
25%      2004-12-31 12:00:00
50%      2010-01-01 00:00:00
75%      2010-01-01 00:00:00
max      2010-01-01 00:00:00
dtype: object
```

Describing a `DataFrame`. By default only numeric fields are returned.

```
>>> df = pd.DataFrame({'categorical': pd.Categorical(['d','e','f']),
...                    'numeric': [1, 2, 3],
...                    'object': ['a', 'b', 'c']
...                   })
>>> df.describe()
       numeric
count      3.0
mean       2.0
std        1.0
```

(continues on next page)

(continued from previous page)

```
min        1.0
25%        1.5
50%        2.0
75%        2.5
max        3.0
```

Describing all columns of a `DataFrame` regardless of data type.

```
>>> df.describe(include='all')
       categorical  numeric object
count            3      3.0      3
unique           3      NaN      3
top              f      NaN      a
freq             1      NaN      1
mean           NaN      2.0    NaN
std            NaN      1.0    NaN
min            NaN      1.0    NaN
25%            NaN      1.5    NaN
50%            NaN      2.0    NaN
75%            NaN      2.5    NaN
max            NaN      3.0    NaN
```

Describing a column from a `DataFrame` by accessing it as an attribute.

```
>>> df.numeric.describe()
count    3.0
mean     2.0
std      1.0
min      1.0
25%      1.5
50%      2.0
75%      2.5
max      3.0
Name: numeric, dtype: float64
```

Including only numeric columns in a `DataFrame` description.

```
>>> df.describe(include=[np.number])
       numeric
count      3.0
mean       2.0
std        1.0
min        1.0
25%        1.5
50%        2.0
75%        2.5
max        3.0
```

Including only string columns in a `DataFrame` description.

```
>>> df.describe(include=[object])
       object
count       3
```

(continues on next page)

(continued from previous page)

```
unique      3
top         a
freq        1
```

Including only categorical columns from a `DataFrame` description.

```
>>> df.describe(include=['category'])
       categorical
count            3
unique           3
top              d
freq             1
```

Excluding numeric columns from a `DataFrame` description.

```
>>> df.describe(exclude=[np.number])
       categorical object
count            3      3
unique           3      3
top              f      a
freq             1      1
```

Excluding object columns from a `DataFrame` description.

```
>>> df.describe(exclude=[object])
       categorical  numeric
count            3      3.0
unique           3      NaN
top              f      NaN
freq             1      NaN
mean           NaN      2.0
std            NaN      1.0
min            NaN      1.0
25%            NaN      1.5
50%            NaN      2.0
75%            NaN      2.5
max            NaN      3.0
```

AlloViz.AlloViz.Elements.Element.diff

`Element.`**`diff`**(*periods: int = 1*, *axis: Axis = 0*) → DataFrame

First discrete difference of element.

Calculates the difference of a DataFrame element compared with another element in the DataFrame (default is element in previous row).

Parameters

periods
[int, default 1] Periods to shift for calculating difference, accepts negative values.

axis
[{0 or 'index', 1 or 'columns'}, default 0] Take difference over rows (0) or columns (1).

Returns

DataFrame
First differences of the Series.

See also:

`DataFrame.pct_change`
Percent change over given number of periods.

`DataFrame.shift`
Shift index by desired number of periods with an optional time freq.

`Series.diff`
First discrete difference of object.

Notes

For boolean dtypes, this uses `operator.xor()` rather than `operator.sub()`. The result is calculated according to current dtype in DataFrame, however dtype of the result is always float64.

Examples

Difference with previous row

```
>>> df = pd.DataFrame({'a': [1, 2, 3, 4, 5, 6],
...                    'b': [1, 1, 2, 3, 5, 8],
...                    'c': [1, 4, 9, 16, 25, 36]})
>>> df
   a  b   c
0  1  1   1
1  2  1   4
2  3  2   9
3  4  3  16
4  5  5  25
5  6  8  36
```

```
>>> df.diff()
     a    b     c
0  NaN  NaN   NaN
1  1.0  0.0   3.0
2  1.0  1.0   5.0
3  1.0  1.0   7.0
4  1.0  2.0   9.0
5  1.0  3.0  11.0
```

Difference with previous column

```
>>> df.diff(axis=1)
    a  b   c
0 NaN  0   0
1 NaN -1   3
2 NaN -1   7
3 NaN -1  13
```

(continues on next page)

(continued from previous page)

```
4 NaN  0  20
5 NaN  2  28
```

Difference with 3rd previous row

```
>>> df.diff(periods=3)
     a    b     c
0  NaN  NaN   NaN
1  NaN  NaN   NaN
2  NaN  NaN   NaN
3  3.0  2.0  15.0
4  3.0  4.0  21.0
5  3.0  6.0  27.0
```

Difference with following row

```
>>> df.diff(periods=-1)
     a    b     c
0 -1.0  0.0  -3.0
1 -1.0 -1.0  -5.0
2 -1.0 -1.0  -7.0
3 -1.0 -2.0  -9.0
4 -1.0 -3.0 -11.0
5  NaN  NaN   NaN
```

Overflow in input dtype

```
>>> df = pd.DataFrame({'a': [1, 0]}, dtype=np.uint8)
>>> df.diff()
       a
0    NaN
1  255.0
```

AlloViz.AlloViz.Elements.Element.div

`Element.`**`div`**(*other*, *axis: Axis = 'columns'*, *level=None*, *fill_value=None*)

Get Floating division of dataframe and other, element-wise (binary operator *truediv*).

Equivalent to `dataframe / other`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *rtruediv*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *floordiv*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other
: [scalar, sequence, Series, dict or DataFrame] Any single or multiple element data structure, or list-like object.

axis
: [{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns. (1 or 'columns'). For Series input, axis to match Series index on.

level
[int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value
[float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame
Result of the arithmetic operation.

See also:

`DataFrame.add`
Add DataFrames.

`DataFrame.sub`
Subtract DataFrames.

`DataFrame.mul`
Multiply DataFrames.

`DataFrame.div`
Divide DataFrames (float division).

`DataFrame.truediv`
Divide DataFrames (float division).

`DataFrame.floordiv`
Divide DataFrames (integer division).

`DataFrame.mod`
Calculate modulo (remainder after division).

`DataFrame.pow`
Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a dictionary by axis.

```
>>> df.mul({'angles': 0, 'degrees': 2})
            angles  degrees
circle           0      720
```

(continues on next page)

(continued from previous page)

```
triangle         0      360
rectangle        0      720
```

```
>>> df.mul({'circle': 0, 'triangle': 2, 'rectangle': 3}, axis='index')
           angles  degrees
circle          0        0
triangle        6      360
rectangle      12     1080
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
```

(continues on next page)

(continued from previous page)

```
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

AlloViz.AlloViz.Elements.Element.divide

Element.**divide**(*other*, *axis: Axis = 'columns'*, *level=None*, *fill_value=None*)

Get Floating division of dataframe and other, element-wise (binary operator *truediv*).

Equivalent to `dataframe / other`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *rtruediv*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *floordiv*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other
: [scalar, sequence, Series, dict or DataFrame] Any single or multiple element data structure, or list-like object.

axis
: [{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns. (1 or 'columns'). For Series input, axis to match Series index on.

level
: [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value
: [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame
: Result of the arithmetic operation.

See also:

`DataFrame.add`
: Add DataFrames.

`DataFrame.sub`
: Subtract DataFrames.

`DataFrame.mul`
: Multiply DataFrames.

`DataFrame.div`
: Divide DataFrames (float division).

`DataFrame.truediv`
: Divide DataFrames (float division).

DataFrame.floordiv
Divide DataFrames (integer division).

DataFrame.mod
Calculate modulo (remainder after division).

DataFrame.pow
Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a dictionary by axis.

```
>>> df.mul({'angles': 0, 'degrees': 2})
            angles  degrees
circle           0      720
triangle         0      360
rectangle        0      720
```

```
>>> df.mul({'circle': 0, 'triangle': 2, 'rectangle': 3}, axis='index')
            angles  degrees
circle           0        0
triangle         6      360
rectangle       12     1080
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
```

(continues on next page)

(continued from previous page)

```
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

AlloViz.AlloViz.Elements.Element.dot

`Element.dot`(*other: AnyArrayLike | DataFrame*) → DataFrame | Series

Compute the matrix multiplication between the DataFrame and other.

This method computes the matrix product between the DataFrame and the values of an other Series, DataFrame or a numpy array.

It can also be called using `self @ other`.

Parameters

other
[Series, DataFrame or array-like] The other object to compute the matrix product with.

Returns

Series or DataFrame
If other is a Series, return the matrix product between self and other as a Series. If other is a DataFrame or a numpy.array, return the matrix product of self and other in a DataFrame of a np.array.

See also:

`Series.dot`
Similar method for Series.

Notes

The dimensions of DataFrame and other must be compatible in order to compute the matrix multiplication. In addition, the column names of DataFrame and the index of other must contain the same values, as they will be aligned prior to the multiplication.

The dot method for Series computes the inner product, instead of the matrix product here.

Examples

Here we multiply a DataFrame with a Series.

```
>>> df = pd.DataFrame([[0, 1, -2, -1], [1, 1, 1, 1]])
>>> s = pd.Series([1, 1, 2, 1])
>>> df.dot(s)
0    -4
1     5
dtype: int64
```

Here we multiply a DataFrame with another DataFrame.

```
>>> other = pd.DataFrame([[0, 1], [1, 2], [-1, -1], [2, 0]])
>>> df.dot(other)
    0   1
0   1   4
1   2   2
```

Note that the dot method give the same result as @

```
>>> df @ other
    0   1
0   1   4
1   2   2
```

The dot method works also if other is an np.array.

```
>>> arr = np.array([[0, 1], [1, 2], [-1, -1], [2, 0]])
>>> df.dot(arr)
    0   1
0   1   4
1   2   2
```

Note how shuffling of the objects does not change the result.

```
>>> s2 = s.reindex([1, 0, 2, 3])
>>> df.dot(s2)
0    -4
1     5
dtype: int64
```

AlloViz.AlloViz.Elements.Element.drop

Element.**drop**(*labels: IndexLabel | None = None*, ***, *axis: Axis = 0*, *index: IndexLabel | None = None*, *columns: IndexLabel | None = None*, *level: Level | None = None*, *inplace: bool = False*, *errors: IgnoreRaise = 'raise'*) → DataFrame | None

Drop specified labels from rows or columns.

Remove rows or columns by specifying label names and corresponding axis, or by directly specifying index or column names. When using a multi-index, labels on different levels can be removed by specifying the level. See the user guide for more information about the now unused levels.

Parameters

labels
[single label or list-like] Index or column labels to drop. A tuple will be used as a single label and not treated as a list-like.

axis
[{0 or 'index', 1 or 'columns'}, default 0] Whether to drop labels from the index (0 or 'index') or columns (1 or 'columns').

index
[single label or list-like] Alternative to specifying axis (`labels, axis=0` is equivalent to `index=labels`).

columns
[single label or list-like] Alternative to specifying axis (`labels, axis=1` is equivalent to `columns=labels`).

level
[int or level name, optional] For MultiIndex, level from which the labels will be removed.

inplace
[bool, default False] If False, return a copy. Otherwise, do operation in place and return None.

errors
[{'ignore', 'raise'}, default 'raise'] If 'ignore', suppress error and only existing labels are dropped.

Returns

DataFrame or None
Returns DataFrame or None DataFrame with the specified index or column labels removed or None if inplace=True.

Raises

KeyError
If any of the labels is not found in the selected axis.

See also:

`DataFrame.loc`
Label-location based indexer for selection by label.

`DataFrame.dropna`
Return DataFrame with labels on given axis omitted where (all or any) data are missing.

DataFrame.drop_duplicates
Return DataFrame with duplicate rows removed, optionally only considering certain columns.

Series.drop
Return Series with specified index labels removed.

Examples

```
>>> df = pd.DataFrame(np.arange(12).reshape(3, 4),
...                   columns=['A', 'B', 'C', 'D'])
>>> df
   A  B   C   D
0  0  1   2   3
1  4  5   6   7
2  8  9  10  11
```

Drop columns

```
>>> df.drop(['B', 'C'], axis=1)
   A   D
0  0   3
1  4   7
2  8  11
```

```
>>> df.drop(columns=['B', 'C'])
   A   D
0  0   3
1  4   7
2  8  11
```

Drop a row by index

```
>>> df.drop([0, 1])
   A  B   C   D
2  8  9  10  11
```

Drop columns and/or rows of MultiIndex DataFrame

```
>>> midx = pd.MultiIndex(levels=[['llama', 'cow', 'falcon'],
...                              ['speed', 'weight', 'length']],
...                      codes=[[0, 0, 0, 1, 1, 1, 2, 2, 2],
...                             [0, 1, 2, 0, 1, 2, 0, 1, 2]])
>>> df = pd.DataFrame(index=midx, columns=['big', 'small'],
...                   data=[[45, 30], [200, 100], [1.5, 1], [30, 20],
...                         [250, 150], [1.5, 0.8], [320, 250],
...                         [1, 0.8], [0.3, 0.2]])
>>> df
                big     small
llama   speed   45.0    30.0
        weight  200.0   100.0
        length  1.5     1.0
cow     speed   30.0    20.0
        weight  250.0   150.0
```

(continues on next page)

(continued from previous page)

```
        length  1.5     0.8
falcon  speed   320.0   250.0
        weight  1.0     0.8
        length  0.3     0.2
```

Drop a specific index combination from the MultiIndex DataFrame, i.e., drop the combination `'falcon'` and `'weight'`, which deletes only the corresponding row

```
>>> df.drop(index=('falcon', 'weight'))
                big     small
llama   speed   45.0    30.0
        weight  200.0   100.0
        length  1.5     1.0
cow     speed   30.0    20.0
        weight  250.0   150.0
        length  1.5     0.8
falcon  speed   320.0   250.0
        length  0.3     0.2
```

```
>>> df.drop(index='cow', columns='small')
                big
llama   speed   45.0
        weight  200.0
        length  1.5
falcon  speed   320.0
        weight  1.0
        length  0.3
```

```
>>> df.drop(index='length', level=1)
                big     small
llama   speed   45.0    30.0
        weight  200.0   100.0
cow     speed   30.0    20.0
        weight  250.0   150.0
falcon  speed   320.0   250.0
        weight  1.0     0.8
```

AlloViz.AlloViz.Elements.Element.drop_duplicates

Element.**drop_duplicates**(*subset: Hashable | Sequence[Hashable] | None = None*, ***, *keep: DropKeep = 'first'*, *inplace: bool = False*, *ignore_index: bool = False*) → DataFrame | None

Return DataFrame with duplicate rows removed.

Considering certain columns is optional. Indexes, including time indexes are ignored.

Parameters

subset
[column label or sequence of labels, optional] Only consider certain columns for identifying duplicates, by default use all of the columns.

keep
[{'first', 'last', `False`}, default 'first'] Determines which duplicates (if any) to keep.

- 'first' : Drop duplicates except for the first occurrence.
- 'last' : Drop duplicates except for the last occurrence.
- `False` : Drop all duplicates.

inplace
[bool, default `False`] Whether to modify the DataFrame rather than creating a new one.

ignore_index
[bool, default `False`] If `True`, the resulting axis will be labeled 0, 1, …, n - 1.

Returns

DataFrame or None
DataFrame with duplicates removed or None if `inplace=True`.

See also:

`DataFrame.value_counts`
Count unique combinations of columns.

Examples

Consider dataset containing ramen rating.

```
>>> df = pd.DataFrame({
...     'brand': ['Yum Yum', 'Yum Yum', 'Indomie', 'Indomie', 'Indomie'],
...     'style': ['cup', 'cup', 'cup', 'pack', 'pack'],
...     'rating': [4, 4, 3.5, 15, 5]
... })
>>> df
    brand style  rating
0  Yum Yum   cup     4.0
1  Yum Yum   cup     4.0
2  Indomie   cup     3.5
3  Indomie  pack    15.0
4  Indomie  pack     5.0
```

By default, it removes duplicate rows based on all columns.

```
>>> df.drop_duplicates()
    brand style  rating
0  Yum Yum   cup     4.0
2  Indomie   cup     3.5
3  Indomie  pack    15.0
4  Indomie  pack     5.0
```

To remove duplicates on specific column(s), use `subset`.

```
>>> df.drop_duplicates(subset=['brand'])
    brand style  rating
0  Yum Yum   cup     4.0
2  Indomie   cup     3.5
```

To remove duplicates and keep last occurrences, use `keep`.

```
>>> df.drop_duplicates(subset=['brand', 'style'], keep='last')
    brand style  rating
1  Yum Yum   cup     4.0
2  Indomie   cup     3.5
4  Indomie  pack     5.0
```

AlloViz.AlloViz.Elements.Element.droplevel

Element.**droplevel**(*level: Hashable | Sequence[Hashable]*, *axis: int | Literal['index', 'columns', 'rows'] = 0*) → None

Return Series/DataFrame with requested index / column level(s) removed.

Parameters

level
[int, str, or list-like] If a string is given, must be the name of a level If list-like, elements must be names or positional indexes of levels.

axis
[{0 or 'index', 1 or 'columns'}, default 0] Axis along which the level(s) is removed:

- 0 or 'index': remove level(s) in column.
- 1 or 'columns': remove level(s) in row.

For *Series* this parameter is unused and defaults to 0.

Returns

Series/DataFrame
Series/DataFrame with requested index / column level(s) removed.

Examples

```
>>> df = pd.DataFrame([
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12]
... ]).set_index([0, 1]).rename_axis(['a', 'b'])
```

```
>>> df.columns = pd.MultiIndex.from_tuples([
...     ('c', 'e'), ('d', 'f')
... ], names=['level_1', 'level_2'])
```

```
>>> df
level_1   c   d
level_2   e   f
a b
1 2       3   4
5 6       7   8
9 10     11  12
```

```
>>> df.droplevel('a')
level_1   c   d
level_2   e   f
b
2         3   4
6         7   8
10       11  12
```

```
>>> df.droplevel('level_2', axis=1)
level_1   c   d
a b
1 2       3   4
5 6       7   8
9 10     11  12
```

AlloViz.AlloViz.Elements.Element.dropna

Element.**dropna**(*, *axis: Axis = 0*, *how: AnyAll | lib.NoDefault = _NoDefault.no_default*, *thresh: int | lib.NoDefault = _NoDefault.no_default*, *subset: IndexLabel | None = None*, *inplace: bool = False*, *ignore_index: bool = False*) → DataFrame | None

Remove missing values.

See the User Guide for more on which values are considered missing, and how to work with missing data.

Parameters

axis
[{0 or 'index', 1 or 'columns'}, default 0] Determine if rows or columns which contain missing values are removed.

- 0, or 'index' : Drop rows which contain missing values.
- 1, or 'columns' : Drop columns which contain missing value.

Only a single axis is allowed.

how
[{'any', 'all'}, default 'any'] Determine if row or column is removed from DataFrame, when we have at least one NA or all NA.

- 'any' : If any NA values are present, drop that row or column.
- 'all' : If all values are NA, drop that row or column.

thresh
[int, optional] Require that many non-NA values. Cannot be combined with how.

subset
[column label or sequence of labels, optional] Labels along other axis to consider, e.g. if you are dropping rows these would be a list of columns to include.

inplace
[bool, default False] Whether to modify the DataFrame rather than creating a new one.

ignore_index
[bool, default `False`] If `True`, the resulting axis will be labeled 0, 1, …, n - 1.

New in version 2.0.0.

Returns

DataFrame or None
DataFrame with NA entries dropped from it or None if `inplace=True`.

See also:

`DataFrame.isna`
Indicate missing values.

`DataFrame.notna`
Indicate existing (non-missing) values.

`DataFrame.fillna`
Replace missing values.

`Series.dropna`
Drop missing values.

`Index.dropna`
Drop missing indices.

Examples

```
>>> df = pd.DataFrame({"name": ['Alfred', 'Batman', 'Catwoman'],
...                    "toy": [np.nan, 'Batmobile', 'Bullwhip'],
...                    "born": [pd.NaT, pd.Timestamp("1940-04-25"),
...                             pd.NaT]})
>>> df
       name        toy       born
0    Alfred        NaN        NaT
1    Batman  Batmobile 1940-04-25
2  Catwoman   Bullwhip        NaT
```

Drop the rows where at least one element is missing.

```
>>> df.dropna()
     name        toy       born
1  Batman  Batmobile 1940-04-25
```

Drop the columns where at least one element is missing.

```
>>> df.dropna(axis='columns')
       name
0    Alfred
1    Batman
2  Catwoman
```

Drop the rows where all elements are missing.

```
>>> df.dropna(how='all')
       name        toy       born
0    Alfred        NaN        NaT
1    Batman  Batmobile 1940-04-25
2  Catwoman   Bullwhip        NaT
```

Keep only the rows with at least 2 non-NA values.

```
>>> df.dropna(thresh=2)
       name        toy       born
1    Batman  Batmobile 1940-04-25
2  Catwoman   Bullwhip        NaT
```

Define in which columns to look for missing values.

```
>>> df.dropna(subset=['name', 'toy'])
       name        toy       born
1    Batman  Batmobile 1940-04-25
2  Catwoman   Bullwhip        NaT
```

AlloViz.AlloViz.Elements.Element.duplicated

`Element.`**`duplicated`**(*subset: Hashable | Sequence[Hashable] | None = None*, *keep: DropKeep = 'first'*) → Series

Return boolean Series denoting duplicate rows.

Considering certain columns is optional.

Parameters

subset
[column label or sequence of labels, optional] Only consider certain columns for identifying duplicates, by default use all of the columns.

keep
[{'first', 'last', False}, default 'first'] Determines which duplicates (if any) to mark.

- `first` : Mark duplicates as `True` except for the first occurrence.
- `last` : Mark duplicates as `True` except for the last occurrence.
- False : Mark all duplicates as `True`.

Returns

Series
Boolean series for each duplicated rows.

See also:

`Index.duplicated`
Equivalent method on index.

`Series.duplicated`
Equivalent method on Series.

`Series.drop_duplicates`
Remove duplicate values from Series.

`DataFrame.drop_duplicates`
Remove duplicate values from DataFrame.

Examples

Consider dataset containing ramen rating.

```
>>> df = pd.DataFrame({
...     'brand': ['Yum Yum', 'Yum Yum', 'Indomie', 'Indomie', 'Indomie'],
...     'style': ['cup', 'cup', 'cup', 'pack', 'pack'],
...     'rating': [4, 4, 3.5, 15, 5]
... })
>>> df
    brand style  rating
0  Yum Yum   cup     4.0
1  Yum Yum   cup     4.0
2  Indomie   cup     3.5
3  Indomie  pack    15.0
4  Indomie  pack     5.0
```

By default, for each set of duplicated values, the first occurrence is set on False and all others on True.

```
>>> df.duplicated()
0    False
1     True
2    False
3    False
4    False
dtype: bool
```

By using ‘last’, the last occurrence of each set of duplicated values is set on False and all others on True.

```
>>> df.duplicated(keep='last')
0     True
1    False
2    False
3    False
4    False
dtype: bool
```

By setting `keep` on False, all duplicates are True.

```
>>> df.duplicated(keep=False)
0     True
1     True
2    False
3    False
4    False
dtype: bool
```

To find duplicates on specific column(s), use `subset`.

```
>>> df.duplicated(subset=['brand'])
0    False
1     True
2    False
3     True
```

(continues on next page)

(continued from previous page)

```
4     True
dtype: bool
```

AlloViz.AlloViz.Elements.Element.eq

Element.**eq**(*other*, *axis: Axis = 'columns'*, *level=None*)

Get Equal to of dataframe and other, element-wise (binary operator *eq*).

Among flexible wrappers (*eq*, *ne*, *le*, *lt*, *ge*, *gt*) to comparison operators.

Equivalent to ==, *!*=, <=, <, >=, > with support to choose axis (rows or columns) and level for comparison.

Parameters

other
: [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.

axis
: [{0 or 'index', 1 or 'columns'}, default 'columns'] Whether to compare by the index (0 or 'index') or columns (1 or 'columns').

level
: [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

Returns

DataFrame of bool
: Result of the comparison.

See also:

DataFrame.eq
: Compare DataFrames for equality elementwise.

DataFrame.ne
: Compare DataFrames for inequality elementwise.

DataFrame.le
: Compare DataFrames for less than inequality or equality elementwise.

DataFrame.lt
: Compare DataFrames for strictly less than inequality elementwise.

DataFrame.ge
: Compare DataFrames for greater than inequality or equality elementwise.

DataFrame.gt
: Compare DataFrames for strictly greater than inequality elementwise.

Notes

Mismatched indices will be unioned together. *NaN* values are considered different (i.e. *NaN* != *NaN*).

Examples

```
>>> df = pd.DataFrame({'cost': [250, 150, 100],
...                    'revenue': [100, 250, 300]},
...                   index=['A', 'B', 'C'])
>>> df
   cost  revenue
A   250      100
B   150      250
C   100      300
```

Comparison with a scalar, using either the operator or method:

```
>>> df == 100
    cost  revenue
A  False     True
B  False    False
C   True    False
```

```
>>> df.eq(100)
    cost  revenue
A  False     True
B  False    False
C   True    False
```

When *other* is a `Series`, the columns of a DataFrame are aligned with the index of *other* and broadcast:

```
>>> df != pd.Series([100, 250], index=["cost", "revenue"])
    cost  revenue
A   True     True
B   True    False
C  False     True
```

Use the method to control the broadcast axis:

```
>>> df.ne(pd.Series([100, 300], index=["A", "D"]), axis='index')
   cost  revenue
A  True    False
B  True     True
C  True     True
D  True     True
```

When comparing to an arbitrary sequence, the number of columns must match the number elements in *other*:

```
>>> df == [250, 100]
    cost  revenue
A   True     True
```

(continues on next page)

(continued from previous page)

```
B  False    False
C  False    False
```

Use the method to control the axis:

```
>>> df.eq([250, 250, 100], axis='index')
    cost  revenue
A   True    False
B  False     True
C   True    False
```

Compare to a DataFrame of different shape.

```
>>> other = pd.DataFrame({'revenue': [300, 250, 100, 150]},
...                      index=['A', 'B', 'C', 'D'])
>>> other
   revenue
A      300
B      250
C      100
D      150
```

```
>>> df.gt(other)
    cost  revenue
A  False    False
B  False    False
C  False     True
D  False    False
```

Compare to a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'cost': [250, 150, 100, 150, 300, 220],
...                              'revenue': [100, 250, 300, 200, 175, 225]},
...                             index=[['Q1', 'Q1', 'Q1', 'Q2', 'Q2', 'Q2'],
...                                    ['A', 'B', 'C', 'A', 'B', 'C']])
>>> df_multindex
      cost  revenue
Q1 A   250      100
   B   150      250
   C   100      300
Q2 A   150      200
   B   300      175
   C   220      225
```

```
>>> df.le(df_multindex, level=1)
       cost  revenue
Q1 A   True     True
   B   True     True
   C   True     True
Q2 A  False     True
   B   True    False
   C   True    False
```

AlloViz.AlloViz.Elements.Element.equals

`Element.equals`(*other: object*) → bool

Test whether two objects contain the same elements.

This function allows two Series or DataFrames to be compared against each other to see if they have the same shape and elements. NaNs in the same location are considered equal.

The row/column index do not need to have the same type, as long as the values are considered equal. Corresponding columns must be of the same dtype.

Parameters

other
[Series or DataFrame] The other Series or DataFrame to be compared with the first.

Returns

bool
True if all elements are the same in both objects, False otherwise.

See also:

`Series.eq`
Compare two Series objects of the same length and return a Series where each element is True if the element in each Series is equal, False otherwise.

`DataFrame.eq`
Compare two DataFrame objects of the same shape and return a DataFrame where each element is True if the respective element in each DataFrame is equal, False otherwise.

`testing.assert_series_equal`
Raises an AssertionError if left and right are not equal. Provides an easy interface to ignore inequality in dtypes, indexes and precision among others.

`testing.assert_frame_equal`
Like assert_series_equal, but targets DataFrames.

`numpy.array_equal`
Return True if two arrays have the same shape and elements, False otherwise.

Examples

```
>>> df = pd.DataFrame({1: [10], 2: [20]})
>>> df
    1   2
0  10  20
```

DataFrames df and exactly_equal have the same types and values for their elements and column labels, which will return True.

```
>>> exactly_equal = pd.DataFrame({1: [10], 2: [20]})
>>> exactly_equal
    1   2
0  10  20
>>> df.equals(exactly_equal)
True
```

DataFrames df and different_column_type have the same element types and values, but have different types for the column labels, which will still return True.

```
>>> different_column_type = pd.DataFrame({1.0: [10], 2.0: [20]})
>>> different_column_type
   1.0  2.0
0   10   20
>>> df.equals(different_column_type)
True
```

DataFrames df and different_data_type have different types for the same values for their elements, and will return False even though their column labels are the same values and types.

```
>>> different_data_type = pd.DataFrame({1: [10.0], 2: [20.0]})
>>> different_data_type
      1     2
0  10.0  20.0
>>> df.equals(different_data_type)
False
```

AlloViz.AlloViz.Elements.Element.eval

`Element.`**`eval`**(*expr: str*, *, *inplace: bool = False*, ***kwargs*) → Any | None

Evaluate a string describing operations on DataFrame columns.

Operates on columns only, not specific rows or elements. This allows *eval* to run arbitrary code, which can make you vulnerable to code injection if you pass user input to this function.

Parameters

expr
[str] The expression string to evaluate.

inplace
[bool, default False] If the expression contains an assignment, whether to perform the operation inplace and mutate the existing DataFrame. Otherwise, a new DataFrame is returned.

****kwargs**
See the documentation for *eval()* for complete details on the keyword arguments accepted by `query()`.

Returns

ndarray, scalar, pandas object, or None
The result of the evaluation or None if `inplace=True`.

See also:

`DataFrame.query`
Evaluates a boolean expression to query the columns of a frame.

`DataFrame.assign`
Can evaluate an expression or function to create new values for a column.

`eval`
Evaluate a Python expression as a string using various backends.

Notes

For more details see the API documentation for *eval()*. For detailed examples see enhancing performance with eval.

Examples

```
>>> df = pd.DataFrame({'A': range(1, 6), 'B': range(10, 0, -2)})
>>> df
   A   B
0  1  10
1  2   8
2  3   6
3  4   4
4  5   2
>>> df.eval('A + B')
0    11
1    10
2     9
3     8
4     7
dtype: int64
```

Assignment is allowed though by default the original DataFrame is not modified.

```
>>> df.eval('C = A + B')
   A   B   C
0  1  10  11
1  2   8  10
2  3   6   9
3  4   4   8
4  5   2   7
>>> df
   A   B
0  1  10
1  2   8
2  3   6
3  4   4
4  5   2
```

Multiple columns can be assigned to using multi-line expressions:

```
>>> df.eval(
...     '''
... C = A + B
... D = A - B
... '''
... )
   A   B   C  D
0  1  10  11 -9
1  2   8  10 -6
2  3   6   9 -3
```

(continues on next page)

(continued from previous page)

```
3  4   4   8  0
4  5   2   7  3
```

AlloViz.AlloViz.Elements.Element.ewm

Element.**ewm**(*com: float | None = None*, *span: float | None = None*, *halflife: float | TimedeltaConvertibleTypes | None = None*, *alpha: float | None = None*, *min_periods: int | None = 0*, *adjust: bool_t = True*, *ignore_na: bool_t = False*, *axis: Axis | lib.NoDefault = _NoDefault.no_default*, *times: np.ndarray | DataFrame | Series | None = None*, *method: Literal['single', 'table'] = 'single'*) → ExponentialMovingWindow

Provide exponentially weighted (EW) calculations.

Exactly one of `com`, `span`, `halflife`, or `alpha` must be provided if `times` is not provided. If `times` is provided, `halflife` and one of `com`, `span` or `alpha` may be provided.

Parameters

com
[float, optional] Specify decay in terms of center of mass

$\alpha = 1/(1 + com)$, for $com \geq 0$.

span
[float, optional] Specify decay in terms of span

$\alpha = 2/(span + 1)$, for $span \geq 1$.

halflife
[float, str, timedelta, optional] Specify decay in terms of half-life

$\alpha = 1 - \exp(-\ln(2)/halflife)$, for $halflife > 0$.

If `times` is specified, a timedelta convertible unit over which an observation decays to half its value. Only applicable to `mean()`, and halflife value will not apply to the other functions.

alpha
[float, optional] Specify smoothing factor α directly

$0 < \alpha \leq 1$.

min_periods
[int, default 0] Minimum number of observations in window required to have a value; otherwise, result is `np.nan`.

adjust
[bool, default True] Divide by decaying adjustment factor in beginning periods to account for imbalance in relative weightings (viewing EWMA as a moving average).

- When `adjust=True` (default), the EW function is calculated using weights $w_i = (1 - \alpha)^i$. For example, the EW moving average of the series $[x_0, x_1, ..., x_t]$ would be:

$$y_t = \frac{x_t + (1 - \alpha)x_{t-1} + (1 - \alpha)^2 x_{t-2} + ... + (1 - \alpha)^t x_0}{1 + (1 - \alpha) + (1 - \alpha)^2 + ... + (1 - \alpha)^t}$$

- When `adjust=False`, the exponentially weighted function is calculated recursively:

$$y_0 = x_0$$
$$y_t = (1 - \alpha)y_{t-1} + \alpha x_t,$$

ignore_na
[bool, default False] Ignore missing values when calculating weights.

- When `ignore_na=False` (default), weights are based on absolute positions. For example, the weights of x_0 and x_2 used in calculating the final weighted average of [x_0, None, x_2] are $(1-\alpha)^2$ and 1 if `adjust=True`, and $(1-\alpha)^2$ and α if `adjust=False`.
- When `ignore_na=True`, weights are based on relative positions. For example, the weights of x_0 and x_2 used in calculating the final weighted average of [x_0, None, x_2] are $1-\alpha$ and 1 if `adjust=True`, and $1-\alpha$ and α if `adjust=False`.

axis
[{0, 1}, default 0] If `0` or `'index'`, calculate across the rows.

If `1` or `'columns'`, calculate across the columns.

For *Series* this parameter is unused and defaults to 0.

times
[np.ndarray, Series, default None] Only applicable to `mean()`.

Times corresponding to the observations. Must be monotonically increasing and `datetime64[ns]` dtype.

If 1-D array like, a sequence with the same shape as the observations.

method
[str {'single', 'table'}, default 'single'] New in version 1.4.0.

Execute the rolling operation per single column or row (`'single'`) or over the entire object (`'table'`).

This argument is only implemented when specifying `engine='numba'` in the method call.

Only applicable to `mean()`

Returns

pandas.api.typing.ExponentialMovingWindow

See also:

rolling
Provides rolling window calculations.

expanding
Provides expanding transformations.

Notes

See Windowing Operations for further usage details and examples.

Examples

```
>>> df = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]})
>>> df
     B
0  0.0
1  1.0
2  2.0
3  NaN
4  4.0
```

```
>>> df.ewm(com=0.5).mean()
          B
0  0.000000
1  0.750000
2  1.615385
3  1.615385
4  3.670213
>>> df.ewm(alpha=2 / 3).mean()
          B
0  0.000000
1  0.750000
2  1.615385
3  1.615385
4  3.670213
```

adjust

```
>>> df.ewm(com=0.5, adjust=True).mean()
          B
0  0.000000
1  0.750000
2  1.615385
3  1.615385
4  3.670213
>>> df.ewm(com=0.5, adjust=False).mean()
          B
0  0.000000
1  0.666667
2  1.555556
3  1.555556
4  3.650794
```

ignore_na

```
>>> df.ewm(com=0.5, ignore_na=True).mean()
          B
0  0.000000
```

(continues on next page)

(continued from previous page)

```
1  0.750000
2  1.615385
3  1.615385
4  3.225000
>>> df.ewm(com=0.5, ignore_na=False).mean()
          B
0  0.000000
1  0.750000
2  1.615385
3  1.615385
4  3.670213
```

times

Exponentially weighted mean with weights calculated with a timedelta `halflife` relative to `times`.

```
>>> times = ['2020-01-01', '2020-01-03', '2020-01-10', '2020-01-15', '2020-01-
↪17']
>>> df.ewm(halflife='4 days', times=pd.DatetimeIndex(times)).mean()
          B
0  0.000000
1  0.585786
2  1.523889
3  1.523889
4  3.233686
```

AlloViz.AlloViz.Elements.Element.expanding

`Element.`**`expanding`**(*min_periods: int = 1*, *axis: int | Literal['index', 'columns', 'rows'] | Literal[_NoDefault.no_default] = _NoDefault.no_default*, *method: Literal['single', 'table'] = 'single'*) → Expanding

Provide expanding window calculations.

Parameters

min_periods
: [int, default 1] Minimum number of observations in window required to have a value; otherwise, result is `np.nan`.

axis
: [int or str, default 0] If `0` or `'index'`, roll across the rows.

  If `1` or `'columns'`, roll across the columns.

  For *Series* this parameter is unused and defaults to 0.

method
: [str {'single', 'table'}, default 'single'] Execute the rolling operation per single column or row (`'single'`) or over the entire object (`'table'`).

  This argument is only implemented when specifying `engine='numba'` in the method call.

  New in version 1.3.0.

Returns

pandas.api.typing.Expanding

See also:

rolling
Provides rolling window calculations.

ewm
Provides exponential weighted functions.

Notes

See Windowing Operations for further usage details and examples.

Examples

```
>>> df = pd.DataFrame({"B": [0, 1, 2, np.nan, 4]})
>>> df
     B
0  0.0
1  1.0
2  2.0
3  NaN
4  4.0
```

min_periods

Expanding sum with 1 vs 3 observations needed to calculate a value.

```
>>> df.expanding(1).sum()
     B
0  0.0
1  1.0
2  3.0
3  3.0
4  7.0
>>> df.expanding(3).sum()
     B
0  NaN
1  NaN
2  3.0
3  3.0
4  7.0
```

AlloViz.AlloViz.Elements.Element.explode

Element.**explode**(*column: IndexLabel*, *ignore_index: bool = False*) → DataFrame

Transform each element of a list-like to a row, replicating index values.

Parameters

column
[IndexLabel] Column(s) to explode. For multiple columns, specify a non-empty list with each element be str or tuple, and all specified columns their list-like data on same row of the frame must have matching length.

New in version 1.3.0: Multi-column explode

ignore_index
[bool, default False] If True, the resulting index will be labeled 0, 1, …, n - 1.

Returns

DataFrame
Exploded lists to rows of the subset columns; index will be duplicated for these rows.

Raises

ValueError

- If columns of the frame are not unique.
- If specified columns to explode is empty list.
- If specified columns to explode have not matching count of elements rowwise in the frame.

See also:

DataFrame.unstack
Pivot a level of the (necessarily hierarchical) index labels.

DataFrame.melt
Unpivot a DataFrame from wide format to long format.

Series.explode
Explode a DataFrame from list-like columns to long format.

Notes

This routine will explode list-likes including lists, tuples, sets, Series, and np.ndarray. The result dtype of the subset rows will be object. Scalars will be returned unchanged, and empty list-likes will result in a np.nan for that row. In addition, the ordering of rows in the output will be non-deterministic when exploding sets.

Reference the user guide for more examples.

Examples

```
>>> df = pd.DataFrame({'A': [[0, 1, 2], 'foo', [], [3, 4]],
...                    'B': 1,
...                    'C': [['a', 'b', 'c'], np.nan, [], ['d', 'e']]})
>>> df
           A  B          C
0  [0, 1, 2]  1  [a, b, c]
1        foo  1        NaN
2         []  1         []
3     [3, 4]  1     [d, e]
```

Single-column explode.

```
>>> df.explode('A')
     A  B          C
0    0  1  [a, b, c]
0    1  1  [a, b, c]
0    2  1  [a, b, c]
1  foo  1        NaN
2  NaN  1         []
3    3  1     [d, e]
3    4  1     [d, e]
```

Multi-column explode.

```
>>> df.explode(list('AC'))
     A  B    C
0    0  1    a
0    1  1    b
0    2  1    c
1  foo  1  NaN
2  NaN  1  NaN
3    3  1    d
3    4  1    e
```

AlloViz.AlloViz.Elements.Element.ffill

Element.**ffill**(*, *axis: None | Axis = None*, *inplace: bool_t = False*, *limit: None | int = None*, *downcast: dict | None | lib.NoDefault = _NoDefault.no_default*) → Self | None

Fill NA/NaN values by propagating the last valid observation to next valid.

Parameters

axis
[{0 or 'index'} for Series, {0 or 'index', 1 or 'columns'} for DataFrame] Axis along which to fill missing values. For *Series* this parameter is unused and defaults to 0.

inplace
[bool, default False] If True, fill in-place. Note: this will modify any other views on this object (e.g., a no-copy slice for a column in a DataFrame).

limit
[int, default None] If method is specified, this is the maximum number of consecu-

tive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled. Must be greater than 0 if not None.

downcast
[dict, default is None] A dict of item->dtype of what to downcast if possible, or the string 'infer' which will try to downcast to an appropriate equal type (e.g. float64 to int64 if possible).

Returns

Series/DataFrame or None
Object with missing values filled or None if `inplace=True`.

Examples

```
>>> df = pd.DataFrame([[np.nan, 2, np.nan, 0],
...                    [3, 4, np.nan, 1],
...                    [np.nan, np.nan, np.nan, np.nan],
...                    [np.nan, 3, np.nan, 4]],
...                   columns=list("ABCD"))
>>> df
     A    B   C    D
0  NaN  2.0 NaN  0.0
1  3.0  4.0 NaN  1.0
2  NaN  NaN NaN  NaN
3  NaN  3.0 NaN  4.0
```

```
>>> df.ffill()
     A    B   C    D
0  NaN  2.0 NaN  0.0
1  3.0  4.0 NaN  1.0
2  3.0  4.0 NaN  1.0
3  3.0  3.0 NaN  4.0
```

```
>>> ser = pd.Series([1, np.nan, 2, 3])
>>> ser.ffill()
0   1.0
1   1.0
2   2.0
3   3.0
dtype: float64
```

AlloViz.AlloViz.Elements.Element.fillna

Element.**fillna**(*value: Hashable | Mapping | Series | DataFrame | None = None*, ***, *method: FillnaOptions | None = None*, *axis: Axis | None = None*, *inplace: bool_t = False*, *limit: int | None = None*, *downcast: dict | None | lib.NoDefault = _NoDefault.no_default*) → Self | None

Fill NA/NaN values using the specified method.

Parameters

value
[scalar, dict, Series, or DataFrame] Value to use to fill holes (e.g. 0), alternately a dict/Series/DataFrame of values specifying which value to use for each index (for a Series) or column (for a DataFrame). Values not in the dict/Series/DataFrame will not be filled. This value cannot be a list.

method
[{'backfill', 'bfill', 'ffill', None}, default None] Method to use for filling holes in reindexed Series:

- ffill: propagate last valid observation forward to next valid.
- backfill / bfill: use next valid observation to fill gap.

Deprecated since version 2.1.0: Use ffill or bfill instead.

axis
[{0 or 'index'} for Series, {0 or 'index', 1 or 'columns'} for DataFrame] Axis along which to fill missing values. For *Series* this parameter is unused and defaults to 0.

inplace
[bool, default False] If True, fill in-place. Note: this will modify any other views on this object (e.g., a no-copy slice for a column in a DataFrame).

limit
[int, default None] If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled. Must be greater than 0 if not None.

downcast
[dict, default is None] A dict of item->dtype of what to downcast if possible, or the string 'infer' which will try to downcast to an appropriate equal type (e.g. float64 to int64 if possible).

Returns

Series/DataFrame or None
Object with missing values filled or None if `inplace=True`.

See also:

ffill
Fill values by propagating the last valid observation to next valid.

bfill
Fill values by using the next valid observation to fill the gap.

interpolate
Fill NaN values using interpolation.

reindex
Conform object to new index.

asfreq
Convert TimeSeries to specified frequency.

Examples

```
>>> df = pd.DataFrame([[np.nan, 2, np.nan, 0],
...                    [3, 4, np.nan, 1],
...                    [np.nan, np.nan, np.nan, np.nan],
...                    [np.nan, 3, np.nan, 4]],
...                   columns=list("ABCD"))
>>> df
     A    B   C    D
0  NaN  2.0 NaN  0.0
1  3.0  4.0 NaN  1.0
2  NaN  NaN NaN  NaN
3  NaN  3.0 NaN  4.0
```

Replace all NaN elements with 0s.

```
>>> df.fillna(0)
     A    B    C    D
0  0.0  2.0  0.0  0.0
1  3.0  4.0  0.0  1.0
2  0.0  0.0  0.0  0.0
3  0.0  3.0  0.0  4.0
```

Replace all NaN elements in column 'A', 'B', 'C', and 'D', with 0, 1, 2, and 3 respectively.

```
>>> values = {"A": 0, "B": 1, "C": 2, "D": 3}
>>> df.fillna(value=values)
     A    B    C    D
0  0.0  2.0  2.0  0.0
1  3.0  4.0  2.0  1.0
2  0.0  1.0  2.0  3.0
3  0.0  3.0  2.0  4.0
```

Only replace the first NaN element.

```
>>> df.fillna(value=values, limit=1)
     A    B    C    D
0  0.0  2.0  2.0  0.0
1  3.0  4.0  NaN  1.0
2  NaN  1.0  NaN  3.0
3  NaN  3.0  NaN  4.0
```

When filling using a DataFrame, replacement happens along the same column names and same indices

```
>>> df2 = pd.DataFrame(np.zeros((4, 4)), columns=list("ABCE"))
>>> df.fillna(df2)
     A    B    C    D
```

(continues on next page)

(continued from previous page)

```
0  0.0  2.0  0.0  0.0
1  3.0  4.0  0.0  1.0
2  0.0  0.0  0.0  NaN
3  0.0  3.0  0.0  4.0
```

Note that column D is not affected since it is not present in df2.

AlloViz.AlloViz.Elements.Element.filter

Element.**filter**(*items=None*, *like: str | None = None*, *regex: str | None = None*, *axis: int | Literal['index', 'columns', 'rows'] | None = None*) → None

Subset the dataframe rows or columns according to the specified index labels.

Note that this routine does not filter a dataframe on its contents. The filter is applied to the labels of the index.

Parameters

items
[list-like] Keep labels from axis which are in items.

like
[str] Keep labels from axis for which "like in label == True".

regex
[str (regular expression)] Keep labels from axis for which re.search(regex, label) == True.

axis
[{0 or 'index', 1 or 'columns', None}, default None] The axis to filter on, expressed either as an index (int) or axis name (str). By default this is the info axis, 'columns' for DataFrame. For *Series* this parameter is unused and defaults to *None*.

Returns

same type as input object

See also:

`DataFrame.loc`
Access a group of rows and columns by label(s) or a boolean array.

Notes

The `items`, `like`, and `regex` parameters are enforced to be mutually exclusive.

`axis` defaults to the info axis that is used when indexing with `[]`.

Examples

```
>>> df = pd.DataFrame(np.array(([1, 2, 3], [4, 5, 6])),
...                   index=['mouse', 'rabbit'],
...                   columns=['one', 'two', 'three'])
>>> df
        one  two  three
mouse     1    2      3
rabbit    4    5      6
```

```
>>> # select columns by name
>>> df.filter(items=['one', 'three'])
         one  three
mouse     1      3
rabbit    4      6
```

```
>>> # select columns by regular expression
>>> df.filter(regex='e$', axis=1)
         one  three
mouse     1      3
rabbit    4      6
```

```
>>> # select rows containing 'bbi'
>>> df.filter(like='bbi', axis=0)
         one  two  three
rabbit    4    5      6
```

AlloViz.AlloViz.Elements.Element.first

`Element.`**`first`**(*offset*) → None

Select initial periods of time series data based on a date offset.

For a DataFrame with a sorted DatetimeIndex, this function can select the first few rows based on a date offset.

Parameters

offset
[str, DateOffset or dateutil.relativedelta] The offset length of the data that will be selected. For instance, '1M' will display all the rows having their index within the first month.

Returns

Series or DataFrame
A subset of the caller.

Raises

TypeError
If the index is not a `DatetimeIndex`

See also:

last
Select final periods of time series based on a date offset.

at_time
Select values at a particular time of the day.

between_time
Select values between particular times of the day.

Examples

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='2D')
>>> ts = pd.DataFrame({'A': [1, 2, 3, 4]}, index=i)
>>> ts
            A
2018-04-09  1
2018-04-11  2
2018-04-13  3
2018-04-15  4
```

Get the rows for the first 3 days:

```
>>> ts.first('3D')
            A
2018-04-09  1
2018-04-11  2
```

Notice the data for 3 first calendar days were returned, not the first 3 days observed in the dataset, and therefore data for 2018-04-13 was not returned.

AlloViz.AlloViz.Elements.Element.first_valid_index

Element.**first_valid_index**() → Hashable | None

Return index for first non-NA value or None, if no non-NA value is found.

Returns

type of index

Notes

If all elements are non-NA/null, returns None. Also returns None for empty Series/DataFrame.

Examples

For Series:

```
>>> s = pd.Series([None, 3, 4])
>>> s.first_valid_index()
1
>>> s.last_valid_index()
2
```

For DataFrame:

```
>>> df = pd.DataFrame({'A': [None, None, 2], 'B': [None, 3, 4]})
>>> df
     A      B
0  NaN    NaN
1  NaN    3.0
2  2.0    4.0
>>> df.first_valid_index()
1
>>> df.last_valid_index()
2
```

AlloViz.AlloViz.Elements.Element.floordiv

`Element.floordiv`(*other*, *axis: Axis = 'columns'*, *level=None*, *fill_value=None*)

Get Integer division of dataframe and other, element-wise (binary operator *floordiv*).

Equivalent to `dataframe // other`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *rfloordiv*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *floordiv*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other
: [scalar, sequence, Series, dict or DataFrame] Any single or multiple element data structure, or list-like object.

axis
: [{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns. (1 or 'columns'). For Series input, axis to match Series index on.

level
: [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value
: [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame
Result of the arithmetic operation.

See also:

DataFrame.add
Add DataFrames.

DataFrame.sub
Subtract DataFrames.

DataFrame.mul
Multiply DataFrames.

DataFrame.div
Divide DataFrames (float division).

DataFrame.truediv
Divide DataFrames (float division).

DataFrame.floordiv
Divide DataFrames (integer division).

DataFrame.mod
Calculate modulo (remainder after division).

DataFrame.pow
Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
```

(continues on next page)

(continued from previous page)

```
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a dictionary by axis.

```
>>> df.mul({'angles': 0, 'degrees': 2})
           angles  degrees
circle          0      720
triangle        0      360
rectangle       0      720
```

```
>>> df.mul({'circle': 0, 'triangle': 2, 'rectangle': 3}, axis='index')
           angles  degrees
circle          0        0
triangle        6      360
rectangle      12     1080
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

AlloViz.AlloViz.Elements.Element.from_dict

classmethod Element.**from_dict**(*data: dict*, *orient: FromDictOrient = 'columns'*, *dtype: Dtype | None = None*, *columns: Axes | None = None*) → DataFrame

Construct DataFrame from dict of array-like or dicts.

Creates DataFrame object from dictionary by columns or by index allowing dtype specification.

Parameters

data
[dict] Of the form {field : array-like} or {field : dict}.

orient
[{'columns', 'index', 'tight'}, default 'columns'] The "orientation" of the data. If the keys of the passed dict should be the columns of the resulting DataFrame, pass 'columns' (default). Otherwise if the keys should be rows, pass 'index'. If 'tight', assume a dict with keys ['index', 'columns', 'data', 'index_names', 'column_names'].

New in version 1.4.0: 'tight' as an allowed value for the `orient` argument

dtype
[dtype, default None] Data type to force after DataFrame construction, otherwise infer.

columns
[list, default None] Column labels to use when `orient='index'`. Raises a ValueError if used with `orient='columns'` or `orient='tight'`.

Returns

DataFrame

See also:

`DataFrame.from_records`
DataFrame from structured ndarray, sequence of tuples or dicts, or DataFrame.

`DataFrame`
DataFrame object creation using constructor.

`DataFrame.to_dict`
Convert the DataFrame to a dictionary.

Examples

By default the keys of the dict become the DataFrame columns:

```
>>> data = {'col_1': [3, 2, 1, 0], 'col_2': ['a', 'b', 'c', 'd']}
>>> pd.DataFrame.from_dict(data)
   col_1 col_2
0      3     a
1      2     b
2      1     c
3      0     d
```

Specify `orient='index'` to create the DataFrame using dictionary keys as rows:

```
>>> data = {'row_1': [3, 2, 1, 0], 'row_2': ['a', 'b', 'c', 'd']}
>>> pd.DataFrame.from_dict(data, orient='index')
       0  1  2  3
row_1  3  2  1  0
row_2  a  b  c  d
```

When using the 'index' orientation, the column names can be specified manually:

```
>>> pd.DataFrame.from_dict(data, orient='index',
...                        columns=['A', 'B', 'C', 'D'])
       A  B  C  D
row_1  3  2  1  0
row_2  a  b  c  d
```

Specify `orient='tight'` to create the DataFrame using a 'tight' format:

```
>>> data = {'index': [('a', 'b'), ('a', 'c')],
...         'columns': [('x', 1), ('y', 2)],
...         'data': [[1, 3], [2, 4]],
...         'index_names': ['n1', 'n2'],
...         'column_names': ['z1', 'z2']}
>>> pd.DataFrame.from_dict(data, orient='tight')
z1     x  y
z2     1  2
n1 n2
a  b   1  3
   c   2  4
```

AlloViz.AlloViz.Elements.Element.from_records

classmethod Element.**from_records**(*data*, *index=None*, *exclude=None*, *columns=None*, *coerce_float: bool = False*, *nrows: None | int = None*) → DataFrame

Convert structured or record ndarray to DataFrame.

Creates a DataFrame object from a structured ndarray, sequence of tuples or dicts, or DataFrame.

Parameters

data
: [structured ndarray, sequence of tuples or dicts, or DataFrame] Structured input data.

  Deprecated since version 2.1.0: Passing a DataFrame is deprecated.

index
: [str, list of fields, array-like] Field of array to use as the index, alternately a specific set of input labels to use.

exclude
: [sequence, default None] Columns or fields to exclude.

columns
: [sequence, default None] Column names to use. If the passed data do not have names associated with them, this argument provides names for the columns. Otherwise this argument indicates the order of the columns in the result (any names not found in the data will become all-NA columns).

coerce_float
[bool, default False] Attempt to convert values of non-string, non-numeric objects (like decimal.Decimal) to floating point, useful for SQL result sets.

nrows
[int, default None] Number of rows to read if data is an iterator.

Returns

DataFrame

See also:

`DataFrame.from_dict`
DataFrame from dict of array-like or dicts.

`DataFrame`
DataFrame object creation using constructor.

Examples

Data can be provided as a structured ndarray:

```
>>> data = np.array([(3, 'a'), (2, 'b'), (1, 'c'), (0, 'd')],
...                 dtype=[('col_1', 'i4'), ('col_2', 'U1')])
>>> pd.DataFrame.from_records(data)
   col_1 col_2
0      3     a
1      2     b
2      1     c
3      0     d
```

Data can be provided as a list of dicts:

```
>>> data = [{'col_1': 3, 'col_2': 'a'},
...         {'col_1': 2, 'col_2': 'b'},
...         {'col_1': 1, 'col_2': 'c'},
...         {'col_1': 0, 'col_2': 'd'}]
>>> pd.DataFrame.from_records(data)
   col_1 col_2
0      3     a
1      2     b
2      1     c
3      0     d
```

Data can be provided as a list of tuples with corresponding columns:

```
>>> data = [(3, 'a'), (2, 'b'), (1, 'c'), (0, 'd')]
>>> pd.DataFrame.from_records(data, columns=['col_1', 'col_2'])
   col_1 col_2
0      3     a
1      2     b
2      1     c
3      0     d
```

AlloViz.AlloViz.Elements.Element.ge

`Element.ge`(*other*, *axis: Axis = 'columns'*, *level=None*)

Get Greater than or equal to of dataframe and other, element-wise (binary operator *ge*).

Among flexible wrappers (*eq*, *ne*, *le*, *lt*, *ge*, *gt*) to comparison operators.

Equivalent to ==, *!*=, <=, <, >=, > with support to choose axis (rows or columns) and level for comparison.

Parameters

other
[scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.

axis
[{0 or 'index', 1 or 'columns'}, default 'columns'] Whether to compare by the index (0 or 'index') or columns (1 or 'columns').

level
[int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

Returns

DataFrame of bool
Result of the comparison.

See also:

`DataFrame.eq`
Compare DataFrames for equality elementwise.

`DataFrame.ne`
Compare DataFrames for inequality elementwise.

`DataFrame.le`
Compare DataFrames for less than inequality or equality elementwise.

`DataFrame.lt`
Compare DataFrames for strictly less than inequality elementwise.

`DataFrame.ge`
Compare DataFrames for greater than inequality or equality elementwise.

`DataFrame.gt`
Compare DataFrames for strictly greater than inequality elementwise.

Notes

Mismatched indices will be unioned together. *NaN* values are considered different (i.e. *NaN* != *NaN*).

Examples

```
>>> df = pd.DataFrame({'cost': [250, 150, 100],
...                    'revenue': [100, 250, 300]},
...                   index=['A', 'B', 'C'])
>>> df
   cost  revenue
A   250      100
B   150      250
C   100      300
```

Comparison with a scalar, using either the operator or method:

```
>>> df == 100
    cost  revenue
A  False     True
B  False    False
C   True    False
```

```
>>> df.eq(100)
    cost  revenue
A  False     True
B  False    False
C   True    False
```

When *other* is a `Series`, the columns of a DataFrame are aligned with the index of *other* and broadcast:

```
>>> df != pd.Series([100, 250], index=["cost", "revenue"])
    cost  revenue
A   True     True
B   True    False
C  False     True
```

Use the method to control the broadcast axis:

```
>>> df.ne(pd.Series([100, 300], index=["A", "D"]), axis='index')
   cost  revenue
A  True    False
B  True     True
C  True     True
D  True     True
```

When comparing to an arbitrary sequence, the number of columns must match the number elements in *other*:

```
>>> df == [250, 100]
    cost  revenue
A   True     True
B  False    False
C  False    False
```

Use the method to control the axis:

```
>>> df.eq([250, 250, 100], axis='index')
    cost  revenue
A   True    False
B  False     True
C   True    False
```

Compare to a DataFrame of different shape.

```
>>> other = pd.DataFrame({'revenue': [300, 250, 100, 150]},
...                      index=['A', 'B', 'C', 'D'])
>>> other
   revenue
A      300
B      250
C      100
D      150
```

```
>>> df.gt(other)
    cost  revenue
A  False    False
B  False    False
C  False     True
D  False    False
```

Compare to a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'cost': [250, 150, 100, 150, 300, 220],
...                              'revenue': [100, 250, 300, 200, 175, 225]},
...                             index=[['Q1', 'Q1', 'Q1', 'Q2', 'Q2', 'Q2'],
...                                    ['A', 'B', 'C', 'A', 'B', 'C']])
>>> df_multindex
      cost  revenue
Q1 A   250      100
   B   150      250
   C   100      300
Q2 A   150      200
   B   300      175
   C   220      225
```

```
>>> df.le(df_multindex, level=1)
       cost  revenue
Q1 A   True     True
   B   True     True
   C   True     True
Q2 A  False     True
   B   True    False
   C   True    False
```

AlloViz.AlloViz.Elements.Element.get

Element.**get**(*key*, *default=None*)

Get item from object for given key (ex: DataFrame column).

Returns default value if not found.

Parameters

key
[object]

Returns

same type as items contained in object

Examples

```
>>> df = pd.DataFrame(
...     [
...         [24.3, 75.7, "high"],
...         [31, 87.8, "high"],
...         [22, 71.6, "medium"],
...         [35, 95, "medium"],
...     ],
...     columns=["temp_celsius", "temp_fahrenheit", "windspeed"],
...     index=pd.date_range(start="2014-02-12", end="2014-02-15", freq="D"),
... )
```

```
>>> df
            temp_celsius  temp_fahrenheit windspeed
2014-02-12          24.3             75.7      high
2014-02-13          31.0             87.8      high
2014-02-14          22.0             71.6    medium
2014-02-15          35.0             95.0    medium
```

```
>>> df.get(["temp_celsius", "windspeed"])
            temp_celsius windspeed
2014-02-12          24.3      high
2014-02-13          31.0      high
2014-02-14          22.0    medium
2014-02-15          35.0    medium
```

```
>>> ser = df['windspeed']
>>> ser.get('2014-02-13')
'high'
```

If the key isn't found, the default value will be used.

```
>>> df.get(["temp_celsius", "temp_kelvin"], default="default_value")
'default_value'
```

```
>>> ser.get('2014-02-10', '[unknown]')
'[unknown]'
```

AlloViz.AlloViz.Elements.Element.groupby

Element.**groupby**(*by=None*, *axis: Axis | lib.NoDefault = _NoDefault.no_default*, *level: IndexLabel | None = None*, *as_index: bool = True*, *sort: bool = True*, *group_keys: bool = True*, *observed: bool | lib.NoDefault = _NoDefault.no_default*, *dropna: bool = True*) → DataFrameGroupBy

Group DataFrame using a mapper or by a Series of columns.

A groupby operation involves some combination of splitting the object, applying a function, and combining the results. This can be used to group large amounts of data and compute operations on these groups.

Parameters

by
[mapping, function, label, pd.Grouper or list of such] Used to determine the groups for the groupby. If `by` is a function, it's called on each value of the object's index. If a dict or Series is passed, the Series or dict VALUES will be used to determine the groups (the Series' values are first aligned; see `.align()` method). If a list or ndarray of length equal to the selected axis is passed (see the groupby user guide), the values are used as-is to determine the groups. A label or list of labels may be passed to group by the columns in `self`. Notice that a tuple is interpreted as a (single) key.

axis
[{0 or 'index', 1 or 'columns'}, default 0] Split along rows (0) or columns (1). For *Series* this parameter is unused and defaults to 0.

level
[int, level name, or sequence of such, default None] If the axis is a MultiIndex (hierarchical), group by a particular level or levels. Do not specify both `by` and `level`.

as_index
[bool, default True] Return object with group labels as the index. Only relevant for DataFrame input. as_index=False is effectively "SQL-style" grouped output. This argument has no effect on filtrations (see the filtrations in the user guide), such as `head()`, `tail()`, `nth()` and in transformations (see the transformations in the user guide).

sort
[bool, default True] Sort group keys. Get better performance by turning this off. Note this does not influence the order of observations within each group. Groupby preserves the order of rows within each group. If False, the groups will appear in the same order as they did in the original DataFrame. This argument has no effect on filtrations (see the filtrations in the user guide), such as `head()`, `tail()`, `nth()` and in transformations (see the transformations in the user guide).

Changed in version 2.0.0: Specifying `sort=False` with an ordered categorical grouper will no longer sort the values.

group_keys
[bool, default True] When calling apply and the `by` argument produces a like-indexed (i.e. a transform) result, add group keys to index to identify pieces. By default group keys are not included when the result's index (and column) labels match the inputs, and are included otherwise.

Changed in version 1.5.0: Warns that `group_keys` will no longer be ignored when the result from `apply` is a like-indexed Series or DataFrame. Specify `group_keys` explicitly to include the group keys or not.

Changed in version 2.0.0: `group_keys` now defaults to `True`.

observed
[bool, default False] This only applies if any of the groupers are Categoricals. If True: only show observed values for categorical groupers. If False: show all values for categorical groupers.

Deprecated since version 2.1.0: The default value will change to True in a future version of pandas.

dropna
[bool, default True] If True, and if group keys contain NA values, NA values together with row/column will be dropped. If False, NA values will also be treated as the key in groups.

Returns

pandas.api.typing.DataFrameGroupBy
Returns a groupby object that contains information about the groups.

See also:

resample
Convenience method for frequency conversion and resampling of time series.

Notes

See the user guide for more detailed usage and examples, including splitting an object into groups, iterating through groups, selecting a group, aggregation, and more.

Examples

```
>>> df = pd.DataFrame({'Animal': ['Falcon', 'Falcon',
...                               'Parrot', 'Parrot'],
...                    'Max Speed': [380., 370., 24., 26.]})
>>> df
   Animal  Max Speed
0  Falcon      380.0
1  Falcon      370.0
2  Parrot       24.0
3  Parrot       26.0
>>> df.groupby(['Animal']).mean()
        Max Speed
Animal
Falcon      375.0
Parrot       25.0
```

Hierarchical Indexes

We can groupby different levels of a hierarchical index using the *level* parameter:

```
>>> arrays = [['Falcon', 'Falcon', 'Parrot', 'Parrot'],
...           ['Captive', 'Wild', 'Captive', 'Wild']]
>>> index = pd.MultiIndex.from_arrays(arrays, names=('Animal', 'Type'))
>>> df = pd.DataFrame({'Max Speed': [390., 350., 30., 20.]},
```

(continues on next page)

(continued from previous page)

```
...                     index=index)
>>> df
                Max Speed
Animal Type
Falcon Captive      390.0
       Wild         350.0
Parrot Captive       30.0
       Wild          20.0
>>> df.groupby(level=0).mean()
        Max Speed
Animal
Falcon      370.0
Parrot       25.0
>>> df.groupby(level="Type").mean()
         Max Speed
Type
Captive      210.0
Wild         185.0
```

We can also choose to include NA in group keys or not by setting *dropna* parameter, the default setting is *True*.

```
>>> l = [[1, 2, 3], [1, None, 4], [2, 1, 3], [1, 2, 2]]
>>> df = pd.DataFrame(l, columns=["a", "b", "c"])
```

```
>>> df.groupby(by=["b"]).sum()
    a   c
b
1.0 2   3
2.0 2   5
```

```
>>> df.groupby(by=["b"], dropna=False).sum()
    a   c
b
1.0 2   3
2.0 2   5
NaN 1   4
```

```
>>> l = [["a", 12, 12], [None, 12.3, 33.], ["b", 12.3, 123], ["a", 1, 1]]
>>> df = pd.DataFrame(l, columns=["a", "b", "c"])
```

```
>>> df.groupby(by="a").sum()
    b     c
a
a   13.0   13.0
b   12.3  123.0
```

```
>>> df.groupby(by="a", dropna=False).sum()
    b     c
a
a   13.0   13.0
```

(continues on next page)

(continued from previous page)

```
b   12.3  123.0
NaN 12.3   33.0
```

When using `.apply()`, use `group_keys` to include or exclude the group keys. The `group_keys` argument defaults to `True` (include).

```
>>> df = pd.DataFrame({'Animal': ['Falcon', 'Falcon',
...                               'Parrot', 'Parrot'],
...                    'Max Speed': [380., 370., 24., 26.]})
>>> df.groupby("Animal", group_keys=True).apply(lambda x: x)
          Animal  Max Speed
Animal
Falcon 0  Falcon      380.0
       1  Falcon      370.0
Parrot 2  Parrot       24.0
       3  Parrot       26.0
```

```
>>> df.groupby("Animal", group_keys=False).apply(lambda x: x)
   Animal  Max Speed
0  Falcon      380.0
1  Falcon      370.0
2  Parrot       24.0
3  Parrot       26.0
```

AlloViz.AlloViz.Elements.Element.gt

`Element.gt`(*other*, *axis: Axis = 'columns'*, *level=None*)

Get Greater than of dataframe and other, element-wise (binary operator *gt*).

Among flexible wrappers (*eq*, *ne*, *le*, *lt*, *ge*, *gt*) to comparison operators.

Equivalent to ==, *!*=, <=, <, >=, > with support to choose axis (rows or columns) and level for comparison.

Parameters

other
: [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.

axis
: [{0 or 'index', 1 or 'columns'}, default 'columns'] Whether to compare by the index (0 or 'index') or columns (1 or 'columns').

level
: [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

Returns

DataFrame of bool
: Result of the comparison.

See also:

DataFrame.eq
Compare DataFrames for equality elementwise.

DataFrame.ne
Compare DataFrames for inequality elementwise.

DataFrame.le
Compare DataFrames for less than inequality or equality elementwise.

DataFrame.lt
Compare DataFrames for strictly less than inequality elementwise.

DataFrame.ge
Compare DataFrames for greater than inequality or equality elementwise.

DataFrame.gt
Compare DataFrames for strictly greater than inequality elementwise.

Notes

Mismatched indices will be unioned together. *NaN* values are considered different (i.e. *NaN* != *NaN*).

Examples

```
>>> df = pd.DataFrame({'cost': [250, 150, 100],
...                    'revenue': [100, 250, 300]},
...                   index=['A', 'B', 'C'])
>>> df
   cost  revenue
A   250      100
B   150      250
C   100      300
```

Comparison with a scalar, using either the operator or method:

```
>>> df == 100
    cost  revenue
A  False     True
B  False    False
C   True    False
```

```
>>> df.eq(100)
    cost  revenue
A  False     True
B  False    False
C   True    False
```

When *other* is a `Series`, the columns of a DataFrame are aligned with the index of *other* and broadcast:

```
>>> df != pd.Series([100, 250], index=["cost", "revenue"])
    cost  revenue
A   True     True
B   True    False
C  False     True
```

Use the method to control the broadcast axis:

```
>>> df.ne(pd.Series([100, 300], index=["A", "D"]), axis='index')
   cost  revenue
A  True    False
B  True     True
C  True     True
D  True     True
```

When comparing to an arbitrary sequence, the number of columns must match the number elements in *other*:

```
>>> df == [250, 100]
    cost  revenue
A   True     True
B  False    False
C  False    False
```

Use the method to control the axis:

```
>>> df.eq([250, 250, 100], axis='index')
    cost  revenue
A   True    False
B  False     True
C   True    False
```

Compare to a DataFrame of different shape.

```
>>> other = pd.DataFrame({'revenue': [300, 250, 100, 150]},
...                      index=['A', 'B', 'C', 'D'])
>>> other
   revenue
A      300
B      250
C      100
D      150
```

```
>>> df.gt(other)
    cost  revenue
A  False    False
B  False    False
C  False     True
D  False    False
```

Compare to a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'cost': [250, 150, 100, 150, 300, 220],
...                              'revenue': [100, 250, 300, 200, 175, 225]},
...                             index=[['Q1', 'Q1', 'Q1', 'Q2', 'Q2', 'Q2'],
...                                    ['A', 'B', 'C', 'A', 'B', 'C']])
>>> df_multindex
      cost  revenue
Q1 A   250      100
   B   150      250
```

(continues on next page)

(continued from previous page)

```
   C   100      300
Q2 A   150      200
   B   300      175
   C   220      225
```

```
>>> df.le(df_multindex, level=1)
       cost  revenue
Q1 A   True     True
   B   True     True
   C   True     True
Q2 A  False     True
   B   True    False
   C   True    False
```

AlloViz.AlloViz.Elements.Element.head

Element.**head**(*n: int = 5*) → None

Return the first *n* rows.

This function returns the first *n* rows for the object based on position. It is useful for quickly testing if your object has the right type of data in it.

For negative values of *n*, this function returns all rows except the last |*n*| rows, equivalent to `df[:n]`.

If n is larger than the number of rows, this function returns all rows.

Parameters

n
[int, default 5] Number of rows to select.

Returns

same type as caller
The first *n* rows of the caller object.

See also:

DataFrame.tail
Returns the last *n* rows.

Examples

```
>>> df = pd.DataFrame({'animal': ['alligator', 'bee', 'falcon', 'lion',
...                    'monkey', 'parrot', 'shark', 'whale', 'zebra']})
>>> df
      animal
0  alligator
1        bee
2     falcon
3       lion
4     monkey
5     parrot
```

(continues on next page)

(continued from previous page)

```
6      shark
7      whale
8      zebra
```

Viewing the first 5 lines

```
>>> df.head()
      animal
0  alligator
1        bee
2     falcon
3       lion
4     monkey
```

Viewing the first *n* lines (three in this case)

```
>>> df.head(3)
      animal
0  alligator
1        bee
2     falcon
```

For negative values of *n*

```
>>> df.head(-3)
      animal
0  alligator
1        bee
2     falcon
3       lion
4     monkey
5     parrot
```

AlloViz.AlloViz.Elements.Element.hist

`Element.`**`hist`**(*column: IndexLabel | None = None*, *by=None*, *grid: bool = True*, *xlabelsize: int | None = None*, *xrot: float | None = None*, *ylabelsize: int | None = None*, *yrot: float | None = None*, *ax=None*, *sharex: bool = False*, *sharey: bool = False*, *figsize: tuple[int, int] | None = None*, *layout: tuple[int, int] | None = None*, *bins: int | Sequence[int] = 10*, *backend: str | None = None*, *legend: bool = False*, ***kwargs*)

Make a histogram of the DataFrame's columns.

A histogram is a representation of the distribution of data. This function calls `matplotlib.pyplot.hist()`, on each series in the DataFrame, resulting in one histogram per column.

Parameters

data
[DataFrame] The pandas object holding the data.

column
[str or sequence, optional] If passed, will be used to limit data to a subset of columns.

by
[object, optional] If passed, then used to form histograms for separate groups.

grid
[bool, default True] Whether to show axis grid lines.

xlabelsize
[int, default None] If specified changes the x-axis label size.

xrot
[float, default None] Rotation of x axis labels. For example, a value of 90 displays the x labels rotated 90 degrees clockwise.

ylabelsize
[int, default None] If specified changes the y-axis label size.

yrot
[float, default None] Rotation of y axis labels. For example, a value of 90 displays the y labels rotated 90 degrees clockwise.

ax
[Matplotlib axes object, default None] The axes to plot the histogram on.

sharex
[bool, default True if ax is None else False] In case subplots=True, share x axis and set some x axis labels to invisible; defaults to True if ax is None otherwise False if an ax is passed in. Note that passing in both an ax and sharex=True will alter all x axis labels for all subplots in a figure.

sharey
[bool, default False] In case subplots=True, share y axis and set some y axis labels to invisible.

figsize
[tuple, optional] The size in inches of the figure to create. Uses the value in *matplotlib.rcParams* by default.

layout
[tuple, optional] Tuple of (rows, columns) for the layout of the histograms.

bins
[int or sequence, default 10] Number of histogram bins to be used. If an integer is given, bins + 1 bin edges are calculated and returned. If bins is a sequence, gives bin edges, including left edge of first bin and right edge of last bin. In this case, bins is returned unmodified.

backend
[str, default None] Backend to use instead of the backend specified in the option `plotting.backend`. For instance, 'matplotlib'. Alternatively, to specify the `plotting.backend` for the whole session, set `pd.options.plotting.backend`.

legend
[bool, default False] Whether to show the legend.

****kwargs**
All other plotting keyword arguments to be passed to `matplotlib.pyplot.hist()`.

Returns

matplotlib.AxesSubplot or numpy.ndarray of them

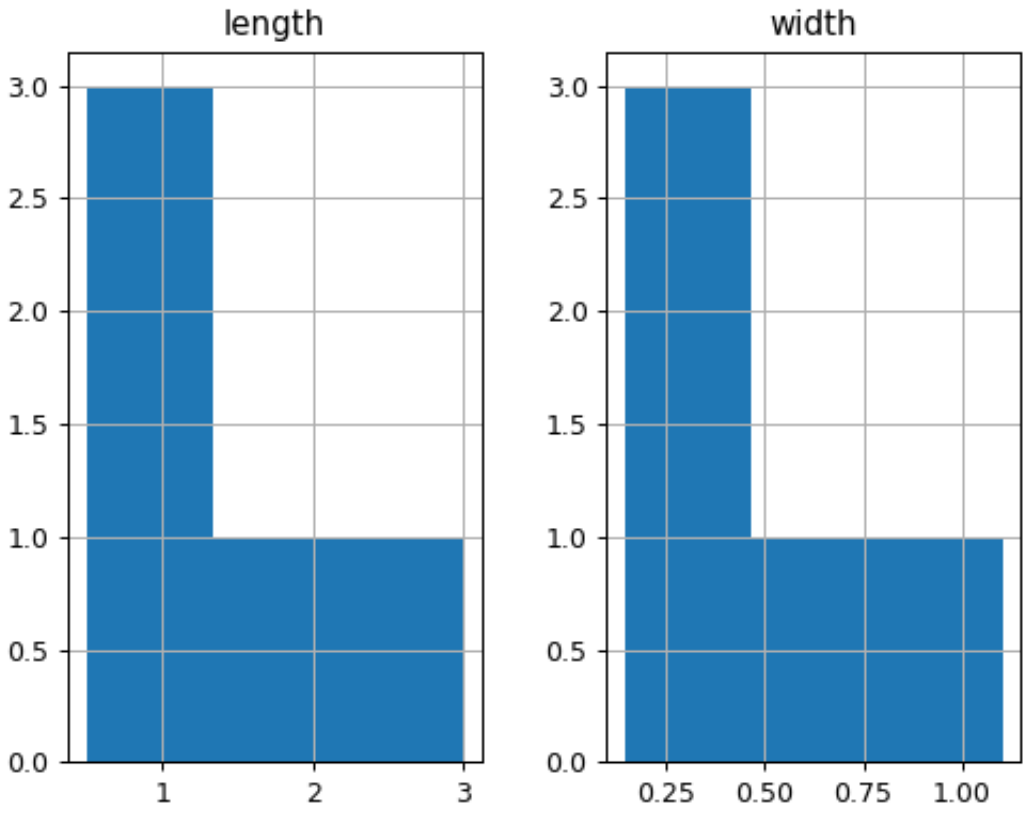
See also:

`matplotlib.pyplot.hist`
Plot a histogram using matplotlib.

Examples

This example draws a histogram based on the length and width of some animals, displayed in three bins

```
>>> df = pd.DataFrame({
...     'length': [1.5, 0.5, 1.2, 0.9, 3],
...     'width': [0.7, 0.2, 0.15, 0.2, 1.1]
...     }, index=['pig', 'rabbit', 'duck', 'chicken', 'horse'])
>>> hist = df.hist(bins=3)
```



AlloViz.AlloViz.Elements.Element.idxmax

`Element.idxmax`(*axis: Axis = 0*, *skipna: bool = True*, *numeric_only: bool = False*) → Series

Return index of first occurrence of maximum over requested axis.

NA/null values are excluded.

Parameters

axis
[{0 or 'index', 1 or 'columns'}, default 0] The axis to use. 0 or 'index' for row-wise, 1 or 'columns' for column-wise.

skipna
[bool, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

numeric_only
[bool, default False] Include only *float*, *int* or *boolean* data.

New in version 1.5.0.

Returns

Series
Indexes of maxima along the specified axis.

Raises

ValueError

- If the row/column is empty

See also:

`Series.idxmax`
Return index of the maximum element.

Notes

This method is the DataFrame version of `ndarray.argmax`.

Examples

Consider a dataset containing food consumption in Argentina.

```
>>> df = pd.DataFrame({'consumption': [10.51, 103.11, 55.48],
...                    'co2_emissions': [37.2, 19.66, 1712]},
...                   index=['Pork', 'Wheat Products', 'Beef'])
```

```
>>> df
                consumption  co2_emissions
Pork                  10.51         37.20
Wheat Products       103.11         19.66
Beef                  55.48       1712.00
```

By default, it returns the index for the maximum value in each column.

```
>>> df.idxmax()
consumption     Wheat Products
co2_emissions             Beef
dtype: object
```

To return the index for the maximum value in each row, use `axis="columns"`.

```
>>> df.idxmax(axis="columns")
Pork              co2_emissions
Wheat Products     consumption
Beef              co2_emissions
dtype: object
```

AlloViz.AlloViz.Elements.Element.idxmin

`Element.`**`idxmin`**(*axis: Axis = 0*, *skipna: bool = True*, *numeric_only: bool = False*) → Series

Return index of first occurrence of minimum over requested axis.

NA/null values are excluded.

Parameters

axis
[{0 or 'index', 1 or 'columns'}, default 0] The axis to use. 0 or 'index' for row-wise, 1 or 'columns' for column-wise.

skipna
[bool, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

numeric_only
[bool, default False] Include only *float*, *int* or *boolean* data.

New in version 1.5.0.

Returns

Series
Indexes of minima along the specified axis.

Raises

ValueError

- If the row/column is empty

See also:

`Series.idxmin`
Return index of the minimum element.

Notes

This method is the DataFrame version of `ndarray.argmin`.

Examples

Consider a dataset containing food consumption in Argentina.

```
>>> df = pd.DataFrame({'consumption': [10.51, 103.11, 55.48],
...                    'co2_emissions': [37.2, 19.66, 1712]},
...                   index=['Pork', 'Wheat Products', 'Beef'])
```

```
>>> df
                consumption  co2_emissions
Pork                  10.51         37.20
Wheat Products       103.11         19.66
Beef                  55.48       1712.00
```

By default, it returns the index for the minimum value in each column.

```
>>> df.idxmin()
consumption                Pork
co2_emissions    Wheat Products
dtype: object
```

To return the index for the minimum value in each row, use `axis="columns"`.

```
>>> df.idxmin(axis="columns")
Pork                consumption
Wheat Products    co2_emissions
Beef                consumption
dtype: object
```

AlloViz.AlloViz.Elements.Element.infer_objects

`Element.`**`infer_objects`**(*copy: bool | None = None*) → None

Attempt to infer better dtypes for object columns.

Attempts soft conversion of object-dtyped columns, leaving non-object and unconvertible columns unchanged. The inference rules are the same as during normal Series/DataFrame construction.

Parameters

copy
[bool, default True] Whether to make a copy for non-object or non-inferable columns or Series.

Returns

same type as input object

See also:

`to_datetime`
Convert argument to datetime.

to_timedelta
Convert argument to timedelta.

to_numeric
Convert argument to numeric type.

convert_dtypes
Convert argument to best possible dtype.

Examples

```
>>> df = pd.DataFrame({"A": ["a", 1, 2, 3]})
>>> df = df.iloc[1:]
>>> df
   A
1  1
2  2
3  3
```

```
>>> df.dtypes
A    object
dtype: object
```

```
>>> df.infer_objects().dtypes
A    int64
dtype: object
```

AlloViz.AlloViz.Elements.Element.info

Element.**info**(*verbose: bool | None = None*, *buf: WriteBuffer[str] | None = None*, *max_cols: int | None = None*, *memory_usage: bool | str | None = None*, *show_counts: bool | None = None*) → None

Print a concise summary of a DataFrame.

This method prints information about a DataFrame including the index dtype and columns, non-null values and memory usage.

Parameters

verbose
[bool, optional] Whether to print the full summary. By default, the setting in `pandas.options.display.max_info_columns` is followed.

buf
[writable buffer, defaults to sys.stdout] Where to send the output. By default, the output is printed to sys.stdout. Pass a writable buffer if you need to further process the output.

max_cols
[int, optional] When to switch from the verbose to the truncated output. If the DataFrame has more than *max_cols* columns, the truncated output is used. By default, the setting in `pandas.options.display.max_info_columns` is used.

memory_usage
[bool, str, optional] Specifies whether total memory usage of the DataFrame el-

ements (including the index) should be displayed. By default, this follows the `pandas.options.display.memory_usage` setting.

True always show memory usage. False never shows memory usage. A value of 'deep' is equivalent to "True with deep introspection". Memory usage is shown in human-readable units (base-2 representation). Without deep introspection a memory estimation is made based in column dtype and number of rows assuming values consume the same memory amount for corresponding dtypes. With deep memory introspection, a real memory usage calculation is performed at the cost of computational resources. See the Frequently Asked Questions for more details.

show_counts
[bool, optional] Whether to show the non-null counts. By default, this is shown only if the DataFrame is smaller than `pandas.options.display.max_info_rows` and `pandas.options.display.max_info_columns`. A value of True always shows the counts, and False never shows the counts.

Returns

None
This method prints a summary of a DataFrame and returns None.

See also:

`DataFrame.describe`
Generate descriptive statistics of DataFrame columns.

`DataFrame.memory_usage`
Memory usage of DataFrame columns.

Examples

```
>>> int_values = [1, 2, 3, 4, 5]
>>> text_values = ['alpha', 'beta', 'gamma', 'delta', 'epsilon']
>>> float_values = [0.0, 0.25, 0.5, 0.75, 1.0]
>>> df = pd.DataFrame({"int_col": int_values, "text_col": text_values,
...                    "float_col": float_values})
>>> df
    int_col text_col  float_col
0        1    alpha       0.00
1        2     beta       0.25
2        3    gamma       0.50
3        4    delta       0.75
4        5  epsilon       1.00
```

Prints information of all columns:

```
>>> df.info(verbose=True)
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 3 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   int_col    5 non-null      int64
 1   text_col   5 non-null      object
```

(continues on next page)

(continued from previous page)

```
 2   float_col  5 non-null      float64
dtypes: float64(1), int64(1), object(1)
memory usage: 248.0+ bytes
```

Prints a summary of columns count and its dtypes but not per column information:

```
>>> df.info(verbose=False)
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Columns: 3 entries, int_col to float_col
dtypes: float64(1), int64(1), object(1)
memory usage: 248.0+ bytes
```

Pipe output of DataFrame.info to buffer instead of sys.stdout, get buffer content and writes to a text file:

```
>>> import io
>>> buffer = io.StringIO()
>>> df.info(buf=buffer)
>>> s = buffer.getvalue()
>>> with open("df_info.txt", "w",
...           encoding="utf-8") as f:
...     f.write(s)
260
```

The *memory_usage* parameter allows deep introspection mode, specially useful for big DataFrames and fine-tune memory optimization:

```
>>> random_strings_array = np.random.choice(['a', 'b', 'c'], 10 ** 6)
>>> df = pd.DataFrame({
...     'column_1': np.random.choice(['a', 'b', 'c'], 10 ** 6),
...     'column_2': np.random.choice(['a', 'b', 'c'], 10 ** 6),
...     'column_3': np.random.choice(['a', 'b', 'c'], 10 ** 6)
... })
>>> df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000000 entries, 0 to 999999
Data columns (total 3 columns):
 #   Column    Non-Null Count    Dtype
---  ------    --------------    -----
 0   column_1  1000000 non-null  object
 1   column_2  1000000 non-null  object
 2   column_3  1000000 non-null  object
dtypes: object(3)
memory usage: 22.9+ MB
```

```
>>> df.info(memory_usage='deep')
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000000 entries, 0 to 999999
Data columns (total 3 columns):
 #   Column    Non-Null Count    Dtype
---  ------    --------------    -----
 0   column_1  1000000 non-null  object
```

(continues on next page)

(continued from previous page)

```
 1   column_2  1000000 non-null  object
 2   column_3  1000000 non-null  object
dtypes: object(3)
memory usage: 165.9 MB
```

AlloViz.AlloViz.Elements.Element.insert

Element.**insert**(*loc: int*, *column: Hashable*, *value: Scalar | AnyArrayLike*, *allow_duplicates: bool | lib.NoDefault = _NoDefault.no_default*) → None

Insert column into DataFrame at specified location.

Raises a ValueError if *column* is already contained in the DataFrame, unless *allow_duplicates* is set to True.

Parameters

loc
[int] Insertion index. Must verify 0 <= loc <= len(columns).

column
[str, number, or hashable object] Label of the inserted column.

value
[Scalar, Series, or array-like]

allow_duplicates
[bool, optional, default lib.no_default]

See also:

Index.insert
Insert new item by index.

Examples

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df
   col1  col2
0     1     3
1     2     4
>>> df.insert(1, "newcol", [99, 99])
>>> df
   col1  newcol  col2
0     1      99     3
1     2      99     4
>>> df.insert(0, "col1", [100, 100], allow_duplicates=True)
>>> df
   col1  col1  newcol  col2
0   100     1      99     3
1   100     2      99     4
```

Notice that pandas uses index alignment in case of *value* from type *Series*:

```
>>> df.insert(0, "col0", pd.Series([5, 6], index=[1, 2]))
>>> df
   col0  col1  col1  newcol  col2
0   NaN   100     1      99     3
1   5.0   100     2      99     4
```

AlloViz.AlloViz.Elements.Element.interpolate

Element.**interpolate**(*method: InterpolateOptions = 'linear'*, ***, *axis: Axis = 0*, *limit: int | None = None*, *inplace: bool_t = False*, *limit_direction: Literal['forward', 'backward', 'both'] | None = None*, *limit_area: Literal['inside', 'outside'] | None = None*, *downcast: Literal['infer'] | None | lib.NoDefault = _NoDefault.no_default*, ***kwargs*) → Self | None

Fill NaN values using an interpolation method.

Please note that only `method='linear'` is supported for DataFrame/Series with a MultiIndex.

Parameters

method
[str, default 'linear'] Interpolation technique to use. One of:

- 'linear': Ignore the index and treat the values as equally spaced. This is the only method supported on MultiIndexes.
- 'time': Works on daily and higher resolution data to interpolate given length of interval.
- 'index', 'values': use the actual numerical values of the index.
- 'pad': Fill in NaNs using existing values.
- 'nearest', 'zero', 'slinear', 'quadratic', 'cubic', 'barycentric', 'polynomial': Passed to *scipy.interpolate.interp1d*, whereas 'spline' is passed to *scipy.interpolate.UnivariateSpline*. These methods use the numerical values of the index. Both 'polynomial' and 'spline' require that you also specify an *order* (int), e.g. `df.interpolate(method='polynomial', order=5)`. Note that, *slinear* method in Pandas refers to the Scipy first order *spline* instead of Pandas first order *spline*.
- 'krogh', 'piecewise_polynomial', 'spline', 'pchip', 'akima', 'cubicspline': Wrappers around the SciPy interpolation methods of similar names. See *Notes*.
- 'from_derivatives': Refers to *scipy.interpolate.BPoly.from_derivatives*.

axis
[{{0 or 'index', 1 or 'columns', None}}, default None] Axis to interpolate along. For *Series* this parameter is unused and defaults to 0.

limit
[int, optional] Maximum number of consecutive NaNs to fill. Must be greater than 0.

inplace
[bool, default False] Update the data in place if possible.

limit_direction
[{{'forward', 'backward', 'both'}}, Optional] Consecutive NaNs will be filled in this direction.

If limit is specified:

- If 'method' is 'pad' or 'ffill', 'limit_direction' must be 'forward'.
- If 'method' is 'backfill' or 'bfill', 'limit_direction' must be 'backwards'.

If 'limit' is not specified:

- If 'method' is 'backfill' or 'bfill', the default is 'backward'
- else the default is 'forward'

raises ValueError if *limit_direction* is 'forward' or 'both' and
method is 'backfill' or 'bfill'.

raises ValueError if *limit_direction* is 'backward' or 'both' and
method is 'pad' or 'ffill'.

limit_area
[{{*None*, 'inside', 'outside'}}, default None] If limit is specified, consecutive NaNs will be filled with this restriction.

- `None`: No fill restriction.
- 'inside': Only fill NaNs surrounded by valid values (interpolate).
- 'outside': Only fill NaNs outside valid values (extrapolate).

downcast
[optional, 'infer' or None, defaults to None] Downcast dtypes if possible.

Deprecated since version 2.1.0.

``**kwargs``
[optional] Keyword arguments to pass on to the interpolating function.

Returns

Series or DataFrame or None
Returns the same object type as the caller, interpolated at some or all `NaN` values or None if `inplace=True`.

See also:

`fillna`
Fill missing values using different methods.

`scipy.interpolate.Akima1DInterpolator`
Piecewise cubic polynomials (Akima interpolator).

`scipy.interpolate.BPoly.from_derivatives`
Piecewise polynomial in the Bernstein basis.

`scipy.interpolate.interp1d`
Interpolate a 1-D function.

`scipy.interpolate.KroghInterpolator`
Interpolate polynomial (Krogh interpolator).

`scipy.interpolate.PchipInterpolator`
PCHIP 1-d monotonic cubic interpolation.

`scipy.interpolate.CubicSpline`
Cubic spline data interpolator.

Notes

The 'krogh', 'piecewise_polynomial', 'spline', 'pchip' and 'akima' methods are wrappers around the respective SciPy implementations of similar names. These use the actual numerical values of the index. For more information on their behavior, see the SciPy documentation.

Examples

Filling in `NaN` in a `Series` via linear interpolation.

```
>>> s = pd.Series([0, 1, np.nan, 3])
>>> s
0    0.0
1    1.0
2    NaN
3    3.0
dtype: float64
>>> s.interpolate()
0    0.0
1    1.0
2    2.0
3    3.0
dtype: float64
```

Filling in `NaN` in a Series via polynomial interpolation or splines: Both 'polynomial' and 'spline' methods require that you also specify an `order` (int).

```
>>> s = pd.Series([0, 2, np.nan, 8])
>>> s.interpolate(method='polynomial', order=2)
0    0.000000
1    2.000000
2    4.666667
3    8.000000
dtype: float64
```

Fill the DataFrame forward (that is, going down) along each column using linear interpolation.

Note how the last entry in column 'a' is interpolated differently, because there is no entry after it to use for interpolation. Note how the first entry in column 'b' remains `NaN`, because there is no entry before it to use for interpolation.

```
>>> df = pd.DataFrame([(0.0, np.nan, -1.0, 1.0),
...                    (np.nan, 2.0, np.nan, np.nan),
...                    (2.0, 3.0, np.nan, 9.0),
...                    (np.nan, 4.0, -4.0, 16.0)],
...                   columns=list('abcd'))
>>> df
     a    b    c     d
0  0.0  NaN -1.0   1.0
1  NaN  2.0  NaN   NaN
2  2.0  3.0  NaN   9.0
3  NaN  4.0 -4.0  16.0
>>> df.interpolate(method='linear', limit_direction='forward', axis=0)
     a    b    c     d
```

(continues on next page)

(continued from previous page)

```
0  0.0  NaN -1.0   1.0
1  1.0  2.0 -2.0   5.0
2  2.0  3.0 -3.0   9.0
3  2.0  4.0 -4.0  16.0
```

Using polynomial interpolation.

```
>>> df['d'].interpolate(method='polynomial', order=2)
0     1.0
1     4.0
2     9.0
3    16.0
Name: d, dtype: float64
```

AlloViz.AlloViz.Elements.Element.isetitem

Element.**isetitem**(*loc*, *value*) → None

Set the given value in the column with position *loc*.

This is a positional analogue to `__setitem__`.

Parameters

loc
[int or sequence of ints] Index position for the column.

value
[scalar or arraylike] Value(s) for the column.

Notes

`frame.isetitem(loc, value)` is an in-place method as it will modify the DataFrame in place (not returning a new object). In contrast to `frame.iloc[:, i] = value` which will try to update the existing values in place, `frame.isetitem(loc, value)` will not update the values of the column itself in place, it will instead insert a new array.

In cases where `frame.columns` is unique, this is equivalent to `frame[frame.columns[i]] = value`.

AlloViz.AlloViz.Elements.Element.isin

Element.**isin**(*values: Series | DataFrame | Sequence | Mapping*) → DataFrame

Whether each element in the DataFrame is contained in values.

Parameters

values
[iterable, Series, DataFrame or dict] The result will only be true at a location if all the labels match. If *values* is a Series, that's the index. If *values* is a dict, the keys must be the column names, which must match. If *values* is a DataFrame, then both the index and column labels must match.

Returns

DataFrame
DataFrame of booleans showing whether each element in the DataFrame is contained in values.

See also:

`DataFrame.eq`
Equality test for DataFrame.

`Series.isin`
Equivalent method on Series.

`Series.str.contains`
Test if pattern or regex is contained within a string of a Series or Index.

Examples

```
>>> df = pd.DataFrame({'num_legs': [2, 4], 'num_wings': [2, 0]},
...                   index=['falcon', 'dog'])
>>> df
        num_legs  num_wings
falcon         2          2
dog            4          0
```

When `values` is a list check whether every value in the DataFrame is present in the list (which animals have 0 or 2 legs or wings)

```
>>> df.isin([0, 2])
        num_legs  num_wings
falcon      True       True
dog        False       True
```

To check if `values` is *not* in the DataFrame, use the ~ operator:

```
>>> ~df.isin([0, 2])
        num_legs  num_wings
falcon     False      False
dog         True      False
```

When `values` is a dict, we can pass values to check for each column separately:

```
>>> df.isin({'num_wings': [0, 3]})
        num_legs  num_wings
falcon     False      False
dog        False       True
```

When `values` is a Series or DataFrame the index and column must match. Note that 'falcon' does not match based on the number of legs in other.

```
>>> other = pd.DataFrame({'num_legs': [8, 3], 'num_wings': [0, 2]},
...                      index=['spider', 'falcon'])
>>> df.isin(other)
        num_legs  num_wings
falcon     False       True
dog        False      False
```

AlloViz.AlloViz.Elements.Element.isna

Element.**isna**() → DataFrame

Detect missing values.

Return a boolean same-sized object indicating if the values are NA. NA values, such as None or `numpy.NaN`, gets mapped to True values. Everything else gets mapped to False values. Characters such as empty strings `''` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`).

Returns

DataFrame
Mask of bool values for each element in DataFrame that indicates whether an element is an NA value.

See also:

DataFrame.isnull
Alias of isna.

DataFrame.notna
Boolean inverse of isna.

DataFrame.dropna
Omit axes labels with missing values.

isna
Top-level isna.

Examples

Show which entries in a DataFrame are NA.

```
>>> df = pd.DataFrame(dict(age=[5, 6, np.nan],
...                        born=[pd.NaT, pd.Timestamp('1939-05-27'),
...                              pd.Timestamp('1940-04-25')],
...                        name=['Alfred', 'Batman', ''],
...                        toy=[None, 'Batmobile', 'Joker']))
>>> df
   age       born    name        toy
0  5.0        NaT  Alfred       None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25              Joker
```

```
>>> df.isna()
     age   born   name    toy
0  False   True  False   True
1  False  False  False  False
2   True  False  False  False
```

Show which entries in a Series are NA.

```
>>> ser = pd.Series([5, 6, np.nan])
>>> ser
```

(continues on next page)

(continued from previous page)

```
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.isna()
0    False
1    False
2     True
dtype: bool
```

AlloViz.AlloViz.Elements.Element.isnull

`Element.isnull()` → DataFrame

DataFrame.isnull is an alias for DataFrame.isna.

Detect missing values.

Return a boolean same-sized object indicating if the values are NA. NA values, such as None or `numpy.NaN`, gets mapped to True values. Everything else gets mapped to False values. Characters such as empty strings `''` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`).

Returns

DataFrame
Mask of bool values for each element in DataFrame that indicates whether an element is an NA value.

See also:

`DataFrame.isnull`
Alias of isna.

`DataFrame.notna`
Boolean inverse of isna.

`DataFrame.dropna`
Omit axes labels with missing values.

`isna`
Top-level isna.

Examples

Show which entries in a DataFrame are NA.

```
>>> df = pd.DataFrame(dict(age=[5, 6, np.nan],
...                        born=[pd.NaT, pd.Timestamp('1939-05-27'),
...                              pd.Timestamp('1940-04-25')],
...                        name=['Alfred', 'Batman', ''],
...                        toy=[None, 'Batmobile', 'Joker']))
>>> df
```

(continues on next page)

(continued from previous page)

```
   age       born    name        toy
0  5.0        NaT  Alfred       None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25              Joker
```

```
>>> df.isna()
     age   born   name    toy
0  False   True  False   True
1  False  False  False  False
2   True  False  False  False
```

Show which entries in a Series are NA.

```
>>> ser = pd.Series([5, 6, np.nan])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.isna()
0    False
1    False
2     True
dtype: bool
```

AlloViz.AlloViz.Elements.Element.items

`Element.items()` → Iterable[tuple[Hashable, Series]]

Iterate over (column name, Series) pairs.

Iterates over the DataFrame columns, returning a tuple with the column name and the content as a Series.

Yields

label
[object] The column names for the DataFrame being iterated over.

content
[Series] The column entries belonging to each label, as a Series.

See also:

`DataFrame.iterrows`
Iterate over DataFrame rows as (index, Series) pairs.

`DataFrame.itertuples`
Iterate over DataFrame rows as namedtuples of the values.

Examples

```
>>> df = pd.DataFrame({'species': ['bear', 'bear', 'marsupial'],
...                    'population': [1864, 22000, 80000]},
...                    index=['panda', 'polar', 'koala'])
>>> df
        species   population
panda   bear      1864
polar   bear      22000
koala   marsupial 80000
>>> for label, content in df.items():
...     print(f'label: {label}')
...     print(f'content: {content}', sep='\n')
...
label: species
content:
panda         bear
polar         bear
koala    marsupial
Name: species, dtype: object
label: population
content:
panda     1864
polar    22000
koala    80000
Name: population, dtype: int64
```

AlloViz.AlloViz.Elements.Element.iterrows

Element.**iterrows**() → Iterable[tuple[Hashable, Series]]

Iterate over DataFrame rows as (index, Series) pairs.

Yields

index
[label or tuple of label] The index of the row. A tuple for a *MultiIndex*.

data
[Series] The data of the row as a Series.

See also:

DataFrame.itertuples
Iterate over DataFrame rows as namedtuples of the values.

DataFrame.items
Iterate over (column name, Series) pairs.

Notes

1. Because `iterrows` returns a Series for each row, it does **not** preserve dtypes across the rows (dtypes are preserved across columns for DataFrames).

   To preserve dtypes while iterating over the rows, it is better to use *`itertuples()`* which returns namedtuples of the values and which is generally faster than `iterrows`.

2. You should **never modify** something you are iterating over. This is not guaranteed to work in all cases. Depending on the data types, the iterator returns a copy and not a view, and writing to it will have no effect.

Examples

```
>>> df = pd.DataFrame([[1, 1.5]], columns=['int', 'float'])
>>> row = next(df.iterrows())[1]
>>> row
int      1.0
float    1.5
Name: 0, dtype: float64
>>> print(row['int'].dtype)
float64
>>> print(df['int'].dtype)
int64
```

AlloViz.AlloViz.Elements.Element.itertuples

`Element.itertuples`(*index: bool = True*, *name: str | None = 'Pandas'*) → Iterable[tuple[Any, ...]]

Iterate over DataFrame rows as namedtuples.

Parameters

index
: [bool, default True] If True, return the index as the first element of the tuple.

name
: [str or None, default "Pandas"] The name of the returned namedtuples or None to return regular tuples.

Returns

iterator
: An object to iterate over namedtuples for each row in the DataFrame with the first field possibly being the index and following fields being the column values.

See also:

`DataFrame.iterrows`
: Iterate over DataFrame rows as (index, Series) pairs.

`DataFrame.items`
: Iterate over (column name, Series) pairs.

Notes

The column names will be renamed to positional names if they are invalid Python identifiers, repeated, or start with an underscore.

Examples

```
>>> df = pd.DataFrame({'num_legs': [4, 2], 'num_wings': [0, 2]},
...                   index=['dog', 'hawk'])
>>> df
      num_legs  num_wings
dog          4          0
hawk         2          2
>>> for row in df.itertuples():
...     print(row)
...
Pandas(Index='dog', num_legs=4, num_wings=0)
Pandas(Index='hawk', num_legs=2, num_wings=2)
```

By setting the *index* parameter to False we can remove the index as the first element of the tuple:

```
>>> for row in df.itertuples(index=False):
...     print(row)
...
Pandas(num_legs=4, num_wings=0)
Pandas(num_legs=2, num_wings=2)
```

With the *name* parameter set we set a custom name for the yielded namedtuples:

```
>>> for row in df.itertuples(name='Animal'):
...     print(row)
...
Animal(Index='dog', num_legs=4, num_wings=0)
Animal(Index='hawk', num_legs=2, num_wings=2)
```

AlloViz.AlloViz.Elements.Element.join

Element.**join**(*other: DataFrame | Series | Iterable[DataFrame | Series]*, *on: IndexLabel | None = None*, *how: MergeHow = 'left'*, *lsuffix: str = ''*, *rsuffix: str = ''*, *sort: bool = False*, *validate: JoinValidate | None = None*) → DataFrame

Join columns of another DataFrame.

Join columns with *other* DataFrame either on index or on a key column. Efficiently join multiple DataFrame objects by index at once by passing a list.

Parameters

other
[DataFrame, Series, or a list containing any combination of them] Index should be similar to one of the columns in this one. If a Series is passed, its name attribute must be set, and that will be used as the column name in the resulting joined DataFrame.

on
: [str, list of str, or array-like, optional] Column or index level name(s) in the caller to join on the index in *other*, otherwise joins index-on-index. If multiple values given, the *other* DataFrame must have a MultiIndex. Can pass an array as the join key if it is not already contained in the calling DataFrame. Like an Excel VLOOKUP operation.

how
: [{'left', 'right', 'outer', 'inner', 'cross'}, default 'left'] How to handle the operation of the two objects.

- left: use calling frame's index (or column if on is specified)
- right: use *other*'s index.
- outer: form union of calling frame's index (or column if on is specified) with *other*'s index, and sort it lexicographically.
- inner: form intersection of calling frame's index (or column if on is specified) with *other*'s index, preserving the order of the calling's one.
- cross: creates the cartesian product from both frames, preserves the order of the left keys.

  New in version 1.2.0.

lsuffix
: [str, default ''] Suffix to use from left frame's overlapping columns.

rsuffix
: [str, default ''] Suffix to use from right frame's overlapping columns.

sort
: [bool, default False] Order result DataFrame lexicographically by the join key. If False, the order of the join key depends on the join type (how keyword).

validate
: [str, optional] If specified, checks if join is of specified type.

- "one_to_one" or "1:1": check if join keys are unique in both left and right datasets.
- "one_to_many" or "1:m": check if join keys are unique in left dataset.
- "many_to_one" or "m:1": check if join keys are unique in right dataset.
- "many_to_many" or "m:m": allowed, but does not result in checks.

  New in version 1.5.0.

Returns

DataFrame
: A dataframe containing columns from both the caller and *other*.

See also:

`DataFrame.merge`
: For column(s)-on-column(s) operations.

Notes

Parameters *on*, *lsuffix*, and *rsuffix* are not supported when passing a list of *DataFrame* objects.

Examples

```
>>> df = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3', 'K4', 'K5'],
...                    'A': ['A0', 'A1', 'A2', 'A3', 'A4', 'A5']})
```

```
>>> df
  key   A
0  K0  A0
1  K1  A1
2  K2  A2
3  K3  A3
4  K4  A4
5  K5  A5
```

```
>>> other = pd.DataFrame({'key': ['K0', 'K1', 'K2'],
...                       'B': ['B0', 'B1', 'B2']})
```

```
>>> other
  key   B
0  K0  B0
1  K1  B1
2  K2  B2
```

Join DataFrames using their indexes.

```
>>> df.join(other, lsuffix='_caller', rsuffix='_other')
  key_caller   A key_other    B
0         K0  A0        K0   B0
1         K1  A1        K1   B1
2         K2  A2        K2   B2
3         K3  A3       NaN  NaN
4         K4  A4       NaN  NaN
5         K5  A5       NaN  NaN
```

If we want to join using the key columns, we need to set key to be the index in both *df* and *other*. The joined DataFrame will have key as its index.

```
>>> df.set_index('key').join(other.set_index('key'))
      A    B
key
K0   A0   B0
K1   A1   B1
K2   A2   B2
K3   A3  NaN
K4   A4  NaN
K5   A5  NaN
```

Another option to join using the key columns is to use the *on* parameter. DataFrame.join always uses

other's index but we can use any column in *df*. This method preserves the original DataFrame's index in the result.

```
>>> df.join(other.set_index('key'), on='key')
  key   A    B
0  K0  A0   B0
1  K1  A1   B1
2  K2  A2   B2
3  K3  A3  NaN
4  K4  A4  NaN
5  K5  A5  NaN
```

Using non-unique key values shows how they are matched.

```
>>> df = pd.DataFrame({'key': ['K0', 'K1', 'K1', 'K3', 'K0', 'K1'],
...                    'A': ['A0', 'A1', 'A2', 'A3', 'A4', 'A5']})
```

```
>>> df
  key   A
0  K0  A0
1  K1  A1
2  K1  A2
3  K3  A3
4  K0  A4
5  K1  A5
```

```
>>> df.join(other.set_index('key'), on='key', validate='m:1')
  key   A    B
0  K0  A0   B0
1  K1  A1   B1
2  K1  A2   B1
3  K3  A3  NaN
4  K0  A4   B0
5  K1  A5   B1
```

AlloViz.AlloViz.Elements.Element.keys

`Element.`**`keys`**`()` → Index

Get the 'info axis' (see Indexing for more).

This is index for Series, columns for DataFrame.

Returns

Index
Info axis.

Examples

```
>>> d = pd.DataFrame(data={'A': [1, 2, 3], 'B': [0, 4, 8]},
...                  index=['a', 'b', 'c'])
>>> d
   A  B
a  1  0
b  2  4
c  3  8
>>> d.keys()
Index(['A', 'B'], dtype='object')
```

AlloViz.AlloViz.Elements.Element.kurt

Element.**kurt**(*axis: Axis | None = 0*, *skipna: bool = True*, *numeric_only: bool = False*, ***kwargs*)

Return unbiased kurtosis over requested axis.

Kurtosis obtained using Fisher's definition of kurtosis (kurtosis of normal == 0.0). Normalized by N-1.

Parameters

axis
[{index (0), columns (1)}] Axis for the function to be applied on. For *Series* this parameter is unused and defaults to 0.

For DataFrames, specifying `axis=None` will apply the aggregation across both axes.

New in version 2.0.0.

skipna
[bool, default True] Exclude NA/null values when computing the result.

numeric_only
[bool, default False] Include only float, int, boolean columns. Not implemented for Series.

****kwargs**
Additional keyword arguments to be passed to the function.

Returns

Series or scalar

Examples

```
>>> s = pd.Series([1, 2, 2, 3], index=['cat', 'dog', 'dog', 'mouse'])
>>> s
cat    1
dog    2
dog    2
mouse  3
dtype: int64
>>> s.kurt()
1.5
```

With a DataFrame

```
>>> df = pd.DataFrame({'a': [1, 2, 2, 3], 'b': [3, 4, 4, 4]},
...                   index=['cat', 'dog', 'dog', 'mouse'])
>>> df
       a   b
  cat  1   3
  dog  2   4
  dog  2   4
mouse  3   4
>>> df.kurt()
a   1.5
b   4.0
dtype: float64
```

With axis=None

```
>>> df.kurt(axis=None).round(6)
-0.988693
```

Using axis=1

```
>>> df = pd.DataFrame({'a': [1, 2], 'b': [3, 4], 'c': [3, 4], 'd': [1, 2]},
...                   index=['cat', 'dog'])
>>> df.kurt(axis=1)
cat   -6.0
dog   -6.0
dtype: float64
```

AlloViz.AlloViz.Elements.Element.kurtosis

`Element.`**`kurtosis`**(*axis: Axis | None = 0*, *skipna: bool = True*, *numeric_only: bool = False*, ***kwargs*)

Return unbiased kurtosis over requested axis.

Kurtosis obtained using Fisher's definition of kurtosis (kurtosis of normal == 0.0). Normalized by N-1.

Parameters

axis
: [{index (0), columns (1)}] Axis for the function to be applied on. For *Series* this parameter is unused and defaults to 0.

  For DataFrames, specifying `axis=None` will apply the aggregation across both axes.

  New in version 2.0.0.

skipna
: [bool, default True] Exclude NA/null values when computing the result.

numeric_only
: [bool, default False] Include only float, int, boolean columns. Not implemented for Series.

****kwargs**
: Additional keyword arguments to be passed to the function.

Returns

Series or scalar

Examples

```
>>> s = pd.Series([1, 2, 2, 3], index=['cat', 'dog', 'dog', 'mouse'])
>>> s
cat    1
dog    2
dog    2
mouse  3
dtype: int64
>>> s.kurt()
1.5
```

With a DataFrame

```
>>> df = pd.DataFrame({'a': [1, 2, 2, 3], 'b': [3, 4, 4, 4]},
...                   index=['cat', 'dog', 'dog', 'mouse'])
>>> df
       a   b
  cat  1   3
  dog  2   4
  dog  2   4
mouse  3   4
>>> df.kurt()
a   1.5
b   4.0
dtype: float64
```

With axis=None

```
>>> df.kurt(axis=None).round(6)
-0.988693
```

Using axis=1

```
>>> df = pd.DataFrame({'a': [1, 2], 'b': [3, 4], 'c': [3, 4], 'd': [1, 2]},
...                   index=['cat', 'dog'])
>>> df.kurt(axis=1)
cat   -6.0
dog   -6.0
dtype: float64
```

AlloViz.AlloViz.Elements.Element.last

`Element.last`(*offset*) → None

Select final periods of time series data based on a date offset.

For a DataFrame with a sorted DatetimeIndex, this function selects the last few rows based on a date offset.

Parameters

offset
[str, DateOffset, dateutil.relativedelta] The offset length of the data that will be selected. For instance, '3D' will display all the rows having their index within the last 3 days.

Returns

Series or DataFrame
A subset of the caller.

Raises

TypeError
If the index is not a `DatetimeIndex`

See also:

first
Select initial periods of time series based on a date offset.

at_time
Select values at a particular time of the day.

between_time
Select values between particular times of the day.

Notes

Deprecated since version 2.1.0: Please create a mask and filter using *.loc* instead

Examples

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='2D')
>>> ts = pd.DataFrame({'A': [1, 2, 3, 4]}, index=i)
>>> ts
            A
2018-04-09  1
2018-04-11  2
2018-04-13  3
2018-04-15  4
```

Get the rows for the last 3 days:

```
>>> ts.last('3D')
            A
2018-04-13  3
2018-04-15  4
```

Notice the data for 3 last calendar days were returned, not the last 3 observed days in the dataset, and therefore data for 2018-04-11 was not returned.

AlloViz.AlloViz.Elements.Element.last_valid_index

Element.**last_valid_index**() → Hashable | None

Return index for last non-NA value or None, if no non-NA value is found.

Returns

type of index

Notes

If all elements are non-NA/null, returns None. Also returns None for empty Series/DataFrame.

Examples

For Series:

```
>>> s = pd.Series([None, 3, 4])
>>> s.first_valid_index()
1
>>> s.last_valid_index()
2
```

For DataFrame:

```
>>> df = pd.DataFrame({'A': [None, None, 2], 'B': [None, 3, 4]})
>>> df
     A      B
0  NaN    NaN
1  NaN    3.0
2  2.0    4.0
>>> df.first_valid_index()
1
>>> df.last_valid_index()
2
```

AlloViz.AlloViz.Elements.Element.le

Element.**le**(*other*, *axis: Axis = 'columns'*, *level=None*)

Get Less than or equal to of dataframe and other, element-wise (binary operator *le*).

Among flexible wrappers (*eq*, *ne*, *le*, *lt*, *ge*, *gt*) to comparison operators.

Equivalent to ==, *!*=, <=, <, >=, > with support to choose axis (rows or columns) and level for comparison.

Parameters

other
[scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.

axis
[{0 or 'index', 1 or 'columns'}, default 'columns'] Whether to compare by the index (0 or 'index') or columns (1 or 'columns').

level
[int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

Returns

DataFrame of bool
Result of the comparison.

See also:

`DataFrame.eq`
Compare DataFrames for equality elementwise.

`DataFrame.ne`
Compare DataFrames for inequality elementwise.

`DataFrame.le`
Compare DataFrames for less than inequality or equality elementwise.

`DataFrame.lt`
Compare DataFrames for strictly less than inequality elementwise.

`DataFrame.ge`
Compare DataFrames for greater than inequality or equality elementwise.

`DataFrame.gt`
Compare DataFrames for strictly greater than inequality elementwise.

Notes

Mismatched indices will be unioned together. *NaN* values are considered different (i.e. *NaN* != *NaN*).

Examples

```
>>> df = pd.DataFrame({'cost': [250, 150, 100],
...                    'revenue': [100, 250, 300]},
...                   index=['A', 'B', 'C'])
>>> df
   cost  revenue
A   250      100
B   150      250
C   100      300
```

Comparison with a scalar, using either the operator or method:

```
>>> df == 100
    cost  revenue
A  False     True
```

(continues on next page)

(continued from previous page)

```
B  False    False
C   True    False
```

```
>>> df.eq(100)
    cost  revenue
A  False     True
B  False    False
C   True    False
```

When *other* is a `Series`, the columns of a DataFrame are aligned with the index of *other* and broadcast:

```
>>> df != pd.Series([100, 250], index=["cost", "revenue"])
    cost  revenue
A   True     True
B   True    False
C  False     True
```

Use the method to control the broadcast axis:

```
>>> df.ne(pd.Series([100, 300], index=["A", "D"]), axis='index')
   cost  revenue
A  True    False
B  True     True
C  True     True
D  True     True
```

When comparing to an arbitrary sequence, the number of columns must match the number elements in *other*:

```
>>> df == [250, 100]
    cost  revenue
A   True     True
B  False    False
C  False    False
```

Use the method to control the axis:

```
>>> df.eq([250, 250, 100], axis='index')
    cost  revenue
A   True    False
B  False     True
C   True    False
```

Compare to a DataFrame of different shape.

```
>>> other = pd.DataFrame({'revenue': [300, 250, 100, 150]},
...                      index=['A', 'B', 'C', 'D'])
>>> other
   revenue
A      300
B      250
C      100
D      150
```

```
>>> df.gt(other)
    cost  revenue
A  False    False
B  False    False
C  False     True
D  False    False
```

Compare to a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'cost': [250, 150, 100, 150, 300, 220],
...                              'revenue': [100, 250, 300, 200, 175, 225]},
...                             index=[['Q1', 'Q1', 'Q1', 'Q2', 'Q2', 'Q2'],
...                                    ['A', 'B', 'C', 'A', 'B', 'C']])
>>> df_multindex
      cost  revenue
Q1 A   250      100
   B   150      250
   C   100      300
Q2 A   150      200
   B   300      175
   C   220      225
```

```
>>> df.le(df_multindex, level=1)
       cost  revenue
Q1 A   True     True
   B   True     True
   C   True     True
Q2 A  False     True
   B   True    False
   C   True    False
```

AlloViz.AlloViz.Elements.Element.lt

`Element.lt`(*other*, *axis: Axis = 'columns'*, *level=None*)

Get Less than of dataframe and other, element-wise (binary operator *lt*).

Among flexible wrappers (*eq*, *ne*, *le*, *lt*, *ge*, *gt*) to comparison operators.

Equivalent to ==, *!*=, <=, <, >=, > with support to choose axis (rows or columns) and level for comparison.

Parameters

other
: [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.

axis
: [{0 or 'index', 1 or 'columns'}, default 'columns'] Whether to compare by the index (0 or 'index') or columns (1 or 'columns').

level
: [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

Returns

DataFrame of bool
Result of the comparison.

See also:

DataFrame.eq
Compare DataFrames for equality elementwise.

DataFrame.ne
Compare DataFrames for inequality elementwise.

DataFrame.le
Compare DataFrames for less than inequality or equality elementwise.

DataFrame.lt
Compare DataFrames for strictly less than inequality elementwise.

DataFrame.ge
Compare DataFrames for greater than inequality or equality elementwise.

DataFrame.gt
Compare DataFrames for strictly greater than inequality elementwise.

Notes

Mismatched indices will be unioned together. *NaN* values are considered different (i.e. *NaN* != *NaN*).

Examples

```
>>> df = pd.DataFrame({'cost': [250, 150, 100],
...                    'revenue': [100, 250, 300]},
...                   index=['A', 'B', 'C'])
>>> df
   cost  revenue
A   250      100
B   150      250
C   100      300
```

Comparison with a scalar, using either the operator or method:

```
>>> df == 100
    cost  revenue
A  False     True
B  False    False
C   True    False
```

```
>>> df.eq(100)
    cost  revenue
A  False     True
B  False    False
C   True    False
```

When *other* is a `Series`, the columns of a DataFrame are aligned with the index of *other* and broadcast:

```
>>> df != pd.Series([100, 250], index=["cost", "revenue"])
    cost  revenue
A   True     True
B   True    False
C  False     True
```

Use the method to control the broadcast axis:

```
>>> df.ne(pd.Series([100, 300], index=["A", "D"]), axis='index')
   cost  revenue
A  True    False
B  True     True
C  True     True
D  True     True
```

When comparing to an arbitrary sequence, the number of columns must match the number elements in *other*:

```
>>> df == [250, 100]
    cost  revenue
A   True     True
B  False    False
C  False    False
```

Use the method to control the axis:

```
>>> df.eq([250, 250, 100], axis='index')
    cost  revenue
A   True    False
B  False     True
C   True    False
```

Compare to a DataFrame of different shape.

```
>>> other = pd.DataFrame({'revenue': [300, 250, 100, 150]},
...                      index=['A', 'B', 'C', 'D'])
>>> other
   revenue
A      300
B      250
C      100
D      150
```

```
>>> df.gt(other)
    cost  revenue
A  False    False
B  False    False
C  False     True
D  False    False
```

Compare to a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'cost': [250, 150, 100, 150, 300, 220],
...                              'revenue': [100, 250, 300, 200, 175, 225]},
...                             index=[['Q1', 'Q1', 'Q1', 'Q2', 'Q2', 'Q2'],
...                                    ['A', 'B', 'C', 'A', 'B', 'C']])
>>> df_multindex
      cost  revenue
Q1 A   250      100
   B   150      250
   C   100      300
Q2 A   150      200
   B   300      175
   C   220      225
```

```
>>> df.le(df_multindex, level=1)
       cost  revenue
Q1 A   True     True
   B   True     True
   C   True     True
Q2 A  False     True
   B   True    False
   C   True    False
```

AlloViz.AlloViz.Elements.Element.map

`Element.map`(*func: PythonFuncType*, *na_action: str | None = None*, ***kwargs*) → DataFrame

Apply a function to a Dataframe elementwise.

New in version 2.1.0: DataFrame.applymap was deprecated and renamed to DataFrame.map.

This method applies a function that accepts and returns a scalar to every element of a DataFrame.

Parameters

func
[callable] Python function, returns a single value from a single value.

na_action
[{None, 'ignore'}, default None] If 'ignore', propagate NaN values, without passing them to func.

****kwargs**
Additional keyword arguments to pass as keywords arguments to *func*.

Returns

DataFrame
Transformed DataFrame.

See also:

`DataFrame.apply`
Apply a function along input axis of DataFrame.

`DataFrame.replace`
Replace values given in *to_replace* with *value*.

Series.map
Apply a function elementwise on a Series.

Examples

```
>>> df = pd.DataFrame([[1, 2.12], [3.356, 4.567]])
>>> df
       0      1
0  1.000  2.120
1  3.356  4.567
```

```
>>> df.map(lambda x: len(str(x)))
   0  1
0  3  4
1  5  5
```

Like Series.map, NA values can be ignored:

```
>>> df_copy = df.copy()
>>> df_copy.iloc[0, 0] = pd.NA
>>> df_copy.map(lambda x: len(str(x)), na_action='ignore')
     0  1
0  NaN  4
1  5.0  5
```

Note that a vectorized version of *func* often exists, which will be much faster. You could square each number elementwise.

```
>>> df.map(lambda x: x**2)
           0          1
0   1.000000   4.494400
1  11.262736  20.857489
```

But it's better to avoid map in that case.

```
>>> df ** 2
           0          1
0   1.000000   4.494400
1  11.262736  20.857489
```

AlloViz.AlloViz.Elements.Element.mask

Element.**mask**(*cond*, *other=_NoDefault.no_default*, ***, *inplace: bool_t = False*, *axis: Axis | None = None*, *level: Level | None = None*) → Self | None

Replace values where the condition is True.

Parameters

cond
[bool Series/DataFrame, array-like, or callable] Where *cond* is False, keep the original value. Where True, replace with corresponding value from *other*. If *cond* is

callable, it is computed on the Series/DataFrame and should return boolean Series/DataFrame or array. The callable must not change input Series/DataFrame (though pandas doesn't check it).

other
[scalar, Series/DataFrame, or callable] Entries where *cond* is True are replaced with corresponding value from *other*. If other is callable, it is computed on the Series/DataFrame and should return scalar or Series/DataFrame. The callable must not change input Series/DataFrame (though pandas doesn't check it). If not specified, entries will be filled with the corresponding NULL value (`np.nan` for numpy dtypes, `pd.NA` for extension dtypes).

inplace
[bool, default False] Whether to perform the operation in place on the data.

axis
[int, default None] Alignment axis if needed. For *Series* this parameter is unused and defaults to 0.

level
[int, default None] Alignment level if needed.

Returns

Same type as caller or None if `inplace=True`.

See also:

`DataFrame.where()`
Return an object of same shape as self.

Notes

The mask method is an application of the if-then idiom. For each element in the calling DataFrame, if `cond` is `False` the element is used; otherwise the corresponding element from the DataFrame `other` is used. If the axis of `other` does not align with axis of `cond` Series/DataFrame, the misaligned index positions will be filled with True.

The signature for `DataFrame.where()` differs from `numpy.where()`. Roughly `df1.where(m, df2)` is equivalent to `np.where(m, df1, df2)`.

For further details and examples see the `mask` documentation in indexing.

The dtype of the object takes precedence. The fill value is casted to the object's dtype, if this can be done losslessly.

Examples

```
>>> s = pd.Series(range(5))
>>> s.where(s > 0)
0    NaN
1    1.0
2    2.0
3    3.0
4    4.0
dtype: float64
```

(continues on next page)

(continued from previous page)

```
>>> s.mask(s > 0)
0    0.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

```
>>> s = pd.Series(range(5))
>>> t = pd.Series([True, False])
>>> s.where(t, 99)
0     0
1    99
2    99
3    99
4    99
dtype: int64
>>> s.mask(t, 99)
0    99
1     1
2    99
3    99
4    99
dtype: int64
```

```
>>> s.where(s > 1, 10)
0    10
1    10
2    2
3    3
4    4
dtype: int64
>>> s.mask(s > 1, 10)
0     0
1     1
2    10
3    10
4    10
dtype: int64
```

```
>>> df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])
>>> df
   A  B
0  0  1
1  2  3
2  4  5
3  6  7
4  8  9
>>> m = df % 3 == 0
>>> df.where(m, -df)
   A  B
```

(continues on next page)

(continued from previous page)

```
0  0 -1
1 -2  3
2 -4 -5
3  6 -7
4 -8  9
>>> df.where(m, -df) == np.where(m, df, -df)
      A     B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True
>>> df.where(m, -df) == df.mask(~m, -df)
      A     B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True
```

AlloViz.AlloViz.Elements.Element.max

`Element.max`(*axis: Axis | None = 0*, *skipna: bool = True*, *numeric_only: bool = False*, ***kwargs*)

Return the maximum of the values over the requested axis.

If you want the *index* of the maximum, use `idxmax`. This is the equivalent of the `numpy.ndarray` method `argmax`.

Parameters

axis
[{index (0), columns (1)}] Axis for the function to be applied on. For *Series* this parameter is unused and defaults to 0.

For DataFrames, specifying `axis=None` will apply the aggregation across both axes.

New in version 2.0.0.

skipna
[bool, default True] Exclude NA/null values when computing the result.

numeric_only
[bool, default False] Include only float, int, boolean columns. Not implemented for Series.

****kwargs**
Additional keyword arguments to be passed to the function.

Returns

Series or scalar

See also:

`Series.sum`
Return the sum.

Series.min
Return the minimum.

Series.max
Return the maximum.

Series.idxmin
Return the index of the minimum.

Series.idxmax
Return the index of the maximum.

DataFrame.sum
Return the sum over the requested axis.

DataFrame.min
Return the minimum over the requested axis.

DataFrame.max
Return the maximum over the requested axis.

DataFrame.idxmin
Return the index of the minimum over the requested axis.

DataFrame.idxmax
Return the index of the maximum over the requested axis.

Examples

```
>>> idx = pd.MultiIndex.from_arrays([
...     ['warm', 'warm', 'cold', 'cold'],
...     ['dog', 'falcon', 'fish', 'spider']],
...     names=['blooded', 'animal'])
>>> s = pd.Series([4, 2, 0, 8], name='legs', index=idx)
>>> s
blooded  animal
warm     dog       4
         falcon    2
cold     fish      0
         spider    8
Name: legs, dtype: int64
```

```
>>> s.max()
8
```

AlloViz.AlloViz.Elements.Element.mean

Element.**mean**(*axis: Axis | None = 0*, *skipna: bool = True*, *numeric_only: bool = False*, ***kwargs*)

Return the mean of the values over the requested axis.

Parameters

axis
[{index (0), columns (1)}] Axis for the function to be applied on. For *Series* this parameter is unused and defaults to 0.

For DataFrames, specifying `axis=None` will apply the aggregation across both axes.

New in version 2.0.0.

skipna
[bool, default True] Exclude NA/null values when computing the result.

numeric_only
[bool, default False] Include only float, int, boolean columns. Not implemented for Series.

****kwargs**
Additional keyword arguments to be passed to the function.

Returns

Series or scalar

Examples

```
>>> s = pd.Series([1, 2, 3])
>>> s.mean()
2.0
```

With a DataFrame

```
>>> df = pd.DataFrame({'a': [1, 2], 'b': [2, 3]}, index=['tiger', 'zebra'])
>>> df
       a   b
tiger  1   2
zebra  2   3
>>> df.mean()
a   1.5
b   2.5
dtype: float64
```

Using axis=1

```
>>> df.mean(axis=1)
tiger   1.5
zebra   2.5
dtype: float64
```

In this case, *numeric_only* should be set to *True* to avoid getting an error.

```
>>> df = pd.DataFrame({'a': [1, 2], 'b': ['T', 'Z']},
...                   index=['tiger', 'zebra'])
>>> df.mean(numeric_only=True)
a   1.5
dtype: float64
```

AlloViz.AlloViz.Elements.Element.median

Element.**median**(*axis: Axis | None = 0*, *skipna: bool = True*, *numeric_only: bool = False*, ***kwargs*)

Return the median of the values over the requested axis.

Parameters

axis
[{index (0), columns (1)}] Axis for the function to be applied on. For *Series* this parameter is unused and defaults to 0.

For DataFrames, specifying `axis=None` will apply the aggregation across both axes.

New in version 2.0.0.

skipna
[bool, default True] Exclude NA/null values when computing the result.

numeric_only
[bool, default False] Include only float, int, boolean columns. Not implemented for Series.

****kwargs**
Additional keyword arguments to be passed to the function.

Returns

Series or scalar

Examples

```
>>> s = pd.Series([1, 2, 3])
>>> s.median()
2.0
```

With a DataFrame

```
>>> df = pd.DataFrame({'a': [1, 2], 'b': [2, 3]}, index=['tiger', 'zebra'])
>>> df
       a   b
tiger  1   2
zebra  2   3
>>> df.median()
a   1.5
b   2.5
dtype: float64
```

Using axis=1

```
>>> df.median(axis=1)
tiger   1.5
zebra   2.5
dtype: float64
```

In this case, *numeric_only* should be set to *True* to avoid getting an error.

```
>>> df = pd.DataFrame({'a': [1, 2], 'b': ['T', 'Z']},
...                   index=['tiger', 'zebra'])
>>> df.median(numeric_only=True)
a   1.5
dtype: float64
```

AlloViz.AlloViz.Elements.Element.melt

Element.**melt**(*id_vars=None*, *value_vars=None*, *var_name=None*, *value_name: Hashable = 'value'*, *col_level: Level | None = None*, *ignore_index: bool = True*) → DataFrame

Unpivot a DataFrame from wide to long format, optionally leaving identifiers set.

This function is useful to massage a DataFrame into a format where one or more columns are identifier variables (*id_vars*), while all other columns, considered measured variables (*value_vars*), are "unpivoted" to the row axis, leaving just two non-identifier columns, 'variable' and 'value'.

Parameters

id_vars
[tuple, list, or ndarray, optional] Column(s) to use as identifier variables.

value_vars
[tuple, list, or ndarray, optional] Column(s) to unpivot. If not specified, uses all columns that are not set as *id_vars*.

var_name
[scalar] Name to use for the 'variable' column. If None it uses `frame.columns.name` or 'variable'.

value_name
[scalar, default 'value'] Name to use for the 'value' column.

col_level
[int or str, optional] If columns are a MultiIndex then use this level to melt.

ignore_index
[bool, default True] If True, original index is ignored. If False, the original index is retained. Index labels will be repeated as necessary.

Returns

DataFrame
Unpivoted DataFrame.

See also:

melt
Identical method.

pivot_table
Create a spreadsheet-style pivot table as a DataFrame.

DataFrame.pivot
Return reshaped DataFrame organized by given index / column values.

DataFrame.explode
Explode a DataFrame from list-like columns to long format.

Notes

Reference the user guide for more examples.

Examples

```
>>> df = pd.DataFrame({'A': {0: 'a', 1: 'b', 2: 'c'},
...                    'B': {0: 1, 1: 3, 2: 5},
...                    'C': {0: 2, 1: 4, 2: 6}})
>>> df
   A  B  C
0  a  1  2
1  b  3  4
2  c  5  6
```

```
>>> df.melt(id_vars=['A'], value_vars=['B'])
   A variable  value
0  a        B      1
1  b        B      3
2  c        B      5
```

```
>>> df.melt(id_vars=['A'], value_vars=['B', 'C'])
   A variable  value
0  a        B      1
1  b        B      3
2  c        B      5
3  a        C      2
4  b        C      4
5  c        C      6
```

The names of 'variable' and 'value' columns can be customized:

```
>>> df.melt(id_vars=['A'], value_vars=['B'],
...         var_name='myVarname', value_name='myValname')
   A myVarname  myValname
0  a         B          1
1  b         B          3
2  c         B          5
```

Original index values can be kept around:

```
>>> df.melt(id_vars=['A'], value_vars=['B', 'C'], ignore_index=False)
   A variable  value
0  a        B      1
1  b        B      3
2  c        B      5
0  a        C      2
1  b        C      4
2  c        C      6
```

If you have multi-index columns:

```
>>> df.columns = [list('ABC'), list('DEF')]
>>> df
   A  B  C
   D  E  F
0  a  1  2
1  b  3  4
2  c  5  6
```

```
>>> df.melt(col_level=0, id_vars=['A'], value_vars=['B'])
   A variable  value
0  a        B      1
1  b        B      3
2  c        B      5
```

```
>>> df.melt(id_vars=[('A', 'D')], value_vars=[('B', 'E')])
  (A, D) variable_0 variable_1  value
0      a          B          E      1
1      b          B          E      3
2      c          B          E      5
```

AlloViz.AlloViz.Elements.Element.memory_usage

Element.**memory_usage**(*index: bool = True*, *deep: bool = False*) → Series

Return the memory usage of each column in bytes.

The memory usage can optionally include the contribution of the index and elements of *object* dtype.

This value is displayed in *DataFrame.info* by default. This can be suppressed by setting `pandas.options.display.memory_usage` to False.

Parameters

index
: [bool, default True] Specifies whether to include the memory usage of the DataFrame's index in returned Series. If `index=True`, the memory usage of the index is the first item in the output.

deep
: [bool, default False] If True, introspect the data deeply by interrogating *object* dtypes for system-level memory consumption, and include it in the returned values.

Returns

Series
: A Series whose index is the original column names and whose values is the memory usage of each column in bytes.

See also:

`numpy.ndarray.nbytes`
: Total bytes consumed by the elements of an ndarray.

`Series.memory_usage`
: Bytes consumed by a Series.

Categorical
: Memory-efficient array for string values with many repeated values.

DataFrame.info
: Concise summary of a DataFrame.

Notes

See the Frequently Asked Questions for more details.

Examples

```
>>> dtypes = ['int64', 'float64', 'complex128', 'object', 'bool']
>>> data = dict([(t, np.ones(shape=5000, dtype=int).astype(t))
...              for t in dtypes])
>>> df = pd.DataFrame(data)
>>> df.head()
   int64  float64            complex128  object  bool
0      1      1.0              1.0+0.0j       1  True
1      1      1.0              1.0+0.0j       1  True
2      1      1.0              1.0+0.0j       1  True
3      1      1.0              1.0+0.0j       1  True
4      1      1.0              1.0+0.0j       1  True
```

```
>>> df.memory_usage()
Index           128
int64         40000
float64       40000
complex128    80000
object        40000
bool           5000
dtype: int64
```

```
>>> df.memory_usage(index=False)
int64         40000
float64       40000
complex128    80000
object        40000
bool           5000
dtype: int64
```

The memory footprint of *object* dtype columns is ignored by default:

```
>>> df.memory_usage(deep=True)
Index           128
int64         40000
float64       40000
complex128    80000
object       180000
bool           5000
dtype: int64
```

Use a Categorical for efficient storage of an object-dtype column with many repeated values.

```
>>> df['object'].astype('category').memory_usage(deep=True)
5244
```

AlloViz.AlloViz.Elements.Element.merge

Element.**merge**(*right: DataFrame | Series*, *how: MergeHow = 'inner'*, *on: IndexLabel | None = None*, *left_on: IndexLabel | None = None*, *right_on: IndexLabel | None = None*, *left_index: bool = False*, *right_index: bool = False*, *sort: bool = False*, *suffixes: Suffixes = ('_x', '_y')*, *copy: bool | None = None*, *indicator: str | bool = False*, *validate: MergeValidate | None = None*) → DataFrame

Merge DataFrame or named Series objects with a database-style join.

A named Series object is treated as a DataFrame with a single named column.

The join is done on columns or indexes. If joining columns on columns, the DataFrame indexes *will be ignored*. Otherwise if joining indexes on indexes or indexes on a column or columns, the index will be passed on. When performing a cross merge, no column specifications to merge on are allowed.

> **Warning:** If both key columns contain rows where the key is a null value, those rows will be matched against each other. This is different from usual SQL join behaviour and can lead to unexpected results.

Parameters

right
[DataFrame or named Series] Object to merge with.

how
[{'left', 'right', 'outer', 'inner', 'cross'}, default 'inner'] Type of merge to be performed.

- left: use only keys from left frame, similar to a SQL left outer join; preserve key order.
- right: use only keys from right frame, similar to a SQL right outer join; preserve key order.
- outer: use union of keys from both frames, similar to a SQL full outer join; sort keys lexicographically.
- inner: use intersection of keys from both frames, similar to a SQL inner join; preserve the order of the left keys.
- cross: creates the cartesian product from both frames, preserves the order of the left keys.

  New in version 1.2.0.

on
[label or list] Column or index level names to join on. These must be found in both DataFrames. If *on* is None and not merging on indexes then this defaults to the intersection of the columns in both DataFrames.

left_on
[label or list, or array-like] Column or index level names to join on in the left

DataFrame. Can also be an array or list of arrays of the length of the left DataFrame. These arrays are treated as if they are columns.

right_on
[label or list, or array-like] Column or index level names to join on in the right DataFrame. Can also be an array or list of arrays of the length of the right DataFrame. These arrays are treated as if they are columns.

left_index
[bool, default False] Use the index from the left DataFrame as the join key(s). If it is a MultiIndex, the number of keys in the other DataFrame (either the index or a number of columns) must match the number of levels.

right_index
[bool, default False] Use the index from the right DataFrame as the join key. Same caveats as left_index.

sort
[bool, default False] Sort the join keys lexicographically in the result DataFrame. If False, the order of the join keys depends on the join type (how keyword).

suffixes
[list-like, default is ("_x", "_y")] A length-2 sequence where each element is optionally a string indicating the suffix to add to overlapping column names in *left* and *right* respectively. Pass a value of *None* instead of a string to indicate that the column name from *left* or *right* should be left as-is, with no suffix. At least one of the values must not be None.

copy
[bool, default True] If False, avoid copy if possible.

indicator
[bool or str, default False] If True, adds a column to the output DataFrame called "_merge" with information on the source of each row. The column can be given a different name by providing a string argument. The column will have a Categorical type with the value of "left_only" for observations whose merge key only appears in the left DataFrame, "right_only" for observations whose merge key only appears in the right DataFrame, and "both" if the observation's merge key is found in both DataFrames.

validate
[str, optional] If specified, checks if merge is of specified type.

- "one_to_one" or "1:1": check if merge keys are unique in both left and right datasets.
- "one_to_many" or "1:m": check if merge keys are unique in left dataset.
- "many_to_one" or "m:1": check if merge keys are unique in right dataset.
- "many_to_many" or "m:m": allowed, but does not result in checks.

Returns

DataFrame
A DataFrame of the two merged objects.

See also:

`merge_ordered`
Merge with optional filling/interpolation.

merge_asof
Merge on nearest keys.

DataFrame.join
Similar method using indices.

Examples

```
>>> df1 = pd.DataFrame({'lkey': ['foo', 'bar', 'baz', 'foo'],
...                     'value': [1, 2, 3, 5]})
>>> df2 = pd.DataFrame({'rkey': ['foo', 'bar', 'baz', 'foo'],
...                     'value': [5, 6, 7, 8]})
>>> df1
    lkey value
0   foo      1
1   bar      2
2   baz      3
3   foo      5
>>> df2
    rkey value
0   foo      5
1   bar      6
2   baz      7
3   foo      8
```

Merge df1 and df2 on the lkey and rkey columns. The value columns have the default suffixes, _x and _y, appended.

```
>>> df1.merge(df2, left_on='lkey', right_on='rkey')
  lkey  value_x rkey  value_y
0  foo        1  foo        5
1  foo        1  foo        8
2  foo        5  foo        5
3  foo        5  foo        8
4  bar        2  bar        6
5  baz        3  baz        7
```

Merge DataFrames df1 and df2 with specified left and right suffixes appended to any overlapping columns.

```
>>> df1.merge(df2, left_on='lkey', right_on='rkey',
...           suffixes=('_left', '_right'))
  lkey  value_left rkey  value_right
0  foo           1  foo            5
1  foo           1  foo            8
2  foo           5  foo            5
3  foo           5  foo            8
4  bar           2  bar            6
5  baz           3  baz            7
```

Merge DataFrames df1 and df2, but raise an exception if the DataFrames have any overlapping columns.

```
>>> df1.merge(df2, left_on='lkey', right_on='rkey', suffixes=(False, False))
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
ValueError: columns overlap but no suffix specified:
    Index(['value'], dtype='object')
```

```
>>> df1 = pd.DataFrame({'a': ['foo', 'bar'], 'b': [1, 2]})
>>> df2 = pd.DataFrame({'a': ['foo', 'baz'], 'c': [3, 4]})
>>> df1
      a  b
0   foo  1
1   bar  2
>>> df2
      a  c
0   foo  3
1   baz  4
```

```
>>> df1.merge(df2, how='inner', on='a')
      a  b  c
0   foo  1  3
```

```
>>> df1.merge(df2, how='left', on='a')
      a  b  c
0   foo  1  3.0
1   bar  2  NaN
```

```
>>> df1 = pd.DataFrame({'left': ['foo', 'bar']})
>>> df2 = pd.DataFrame({'right': [7, 8]})
>>> df1
    left
0   foo
1   bar
>>> df2
    right
0   7
1   8
```

```
>>> df1.merge(df2, how='cross')
   left  right
0   foo      7
1   foo      8
2   bar      7
3   bar      8
```

AlloViz.AlloViz.Elements.Element.min

Element.**min**(*axis: Axis | None = 0*, *skipna: bool = True*, *numeric_only: bool = False*, ***kwargs*)

Return the minimum of the values over the requested axis.

If you want the *index* of the minimum, use `idxmin`. This is the equivalent of the `numpy.ndarray` method `argmin`.

Parameters

axis
[{index (0), columns (1)}] Axis for the function to be applied on. For *Series* this parameter is unused and defaults to 0.

For DataFrames, specifying `axis=None` will apply the aggregation across both axes.

New in version 2.0.0.

skipna
[bool, default True] Exclude NA/null values when computing the result.

numeric_only
[bool, default False] Include only float, int, boolean columns. Not implemented for Series.

****kwargs**
Additional keyword arguments to be passed to the function.

Returns

Series or scalar

See also:

`Series.sum`
Return the sum.

`Series.min`
Return the minimum.

`Series.max`
Return the maximum.

`Series.idxmin`
Return the index of the minimum.

`Series.idxmax`
Return the index of the maximum.

`DataFrame.sum`
Return the sum over the requested axis.

`DataFrame.min`
Return the minimum over the requested axis.

`DataFrame.max`
Return the maximum over the requested axis.

`DataFrame.idxmin`
Return the index of the minimum over the requested axis.

DataFrame.idxmax
Return the index of the maximum over the requested axis.

Examples

```
>>> idx = pd.MultiIndex.from_arrays([
...     ['warm', 'warm', 'cold', 'cold'],
...     ['dog', 'falcon', 'fish', 'spider']],
...     names=['blooded', 'animal'])
>>> s = pd.Series([4, 2, 0, 8], name='legs', index=idx)
>>> s
blooded  animal
warm     dog       4
         falcon    2
cold     fish      0
         spider    8
Name: legs, dtype: int64
```

```
>>> s.min()
0
```

AlloViz.AlloViz.Elements.Element.mod

Element.**mod**(*other*, *axis: Axis = 'columns'*, *level=None*, *fill_value=None*)
Get Modulo of dataframe and other, element-wise (binary operator *mod*).

Equivalent to `dataframe % other`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *rmod*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *floordiv*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other
[scalar, sequence, Series, dict or DataFrame] Any single or multiple element data structure, or list-like object.

axis
[{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns. (1 or 'columns'). For Series input, axis to match Series index on.

level
[int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value
[float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame
Result of the arithmetic operation.

See also:

DataFrame.add
Add DataFrames.

DataFrame.sub
Subtract DataFrames.

DataFrame.mul
Multiply DataFrames.

DataFrame.div
Divide DataFrames (float division).

DataFrame.truediv
Divide DataFrames (float division).

DataFrame.floordiv
Divide DataFrames (integer division).

DataFrame.mod
Calculate modulo (remainder after division).

DataFrame.pow
Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...         axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a dictionary by axis.

```
>>> df.mul({'angles': 0, 'degrees': 2})
            angles  degrees
circle           0      720
triangle         0      360
rectangle        0      720
```

```
>>> df.mul({'circle': 0, 'triangle': 2, 'rectangle': 3}, axis='index')
            angles  degrees
circle           0        0
triangle         6      360
rectangle       12     1080
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
```

(continues on next page)

(continued from previous page)

```
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

AlloViz.AlloViz.Elements.Element.mode

Element.**mode**(*axis: Axis = 0*, *numeric_only: bool = False*, *dropna: bool = True*) → DataFrame

Get the mode(s) of each element along the selected axis.

The mode of a set of values is the value that appears most often. It can be multiple values.

Parameters

axis
[{0 or 'index', 1 or 'columns'}, default 0] The axis to iterate over while searching for the mode:

- 0 or 'index' : get mode of each column
- 1 or 'columns' : get mode of each row.

numeric_only
[bool, default False] If True, only apply to numeric columns.

dropna
[bool, default True] Don't consider counts of NaN/NaT.

Returns

DataFrame
The modes of each column or row.

See also:

Series.mode
Return the highest frequency value in a Series.

Series.value_counts
Return the counts of values in a Series.

Examples

```
>>> df = pd.DataFrame([('bird', 2, 2),
...                    ('mammal', 4, np.nan),
...                    ('arthropod', 8, 0),
...                    ('bird', 2, np.nan)],
...                   index=('falcon', 'horse', 'spider', 'ostrich'),
...                   columns=('species', 'legs', 'wings'))
>>> df
           species  legs  wings
falcon        bird     2    2.0
horse       mammal     4    NaN
spider   arthropod     8    0.0
ostrich       bird     2    NaN
```

By default, missing values are not considered, and the mode of wings are both 0 and 2. Because the resulting DataFrame has two rows, the second row of `species` and `legs` contains `NaN`.

```
>>> df.mode()
  species  legs  wings
0    bird   2.0    0.0
1     NaN   NaN    2.0
```

Setting `dropna=False` `NaN` values are considered and they can be the mode (like for wings).

```
>>> df.mode(dropna=False)
  species  legs  wings
0    bird     2    NaN
```

Setting `numeric_only=True`, only the mode of numeric columns is computed, and columns of other types are ignored.

```
>>> df.mode(numeric_only=True)
   legs  wings
0   2.0    0.0
1   NaN    2.0
```

To compute the mode over columns and not rows, use the axis parameter:

```
>>> df.mode(axis='columns', numeric_only=True)
           0    1
falcon   2.0  NaN
horse    4.0  NaN
spider   0.0  8.0
ostrich  2.0  NaN
```

AlloViz.AlloViz.Elements.Element.mul

`Element.`**`mul`**(*other*, *axis: Axis = 'columns'*, *level=None*, *fill_value=None*)

Get Multiplication of dataframe and other, element-wise (binary operator *mul*).

Equivalent to `dataframe * other`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *rmul*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *floordiv*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other
: [scalar, sequence, Series, dict or DataFrame] Any single or multiple element data structure, or list-like object.

axis
: [{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns. (1 or 'columns'). For Series input, axis to match Series index on.

level
: [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value
: [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame
Result of the arithmetic operation.

See also:

DataFrame.add
Add DataFrames.

DataFrame.sub
Subtract DataFrames.

DataFrame.mul
Multiply DataFrames.

DataFrame.div
Divide DataFrames (float division).

DataFrame.truediv
Divide DataFrames (float division).

DataFrame.floordiv
Divide DataFrames (integer division).

DataFrame.mod
Calculate modulo (remainder after division).

DataFrame.pow
Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
```

(continues on next page)

(continued from previous page)

```
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a dictionary by axis.

```
>>> df.mul({'angles': 0, 'degrees': 2})
           angles  degrees
circle          0      720
triangle        0      360
rectangle       0      720
```

```
>>> df.mul({'circle': 0, 'triangle': 2, 'rectangle': 3}, axis='index')
           angles  degrees
circle          0        0
triangle        6      360
rectangle      12     1080
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

AlloViz.AlloViz.Elements.Element.multiply

Element.**multiply**(*other*, *axis: Axis = 'columns'*, *level=None*, *fill_value=None*)

Get Multiplication of dataframe and other, element-wise (binary operator *mul*).

Equivalent to `dataframe * other`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *rmul*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *floordiv*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other
[scalar, sequence, Series, dict or DataFrame] Any single or multiple element data structure, or list-like object.

axis
[{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns. (1 or 'columns'). For Series input, axis to match Series index on.

level
[int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value
[float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame
Result of the arithmetic operation.

See also:

DataFrame.add
Add DataFrames.

DataFrame.sub
Subtract DataFrames.

DataFrame.mul
Multiply DataFrames.

DataFrame.div
Divide DataFrames (float division).

DataFrame.truediv
Divide DataFrames (float division).

DataFrame.floordiv
Divide DataFrames (integer division).

DataFrame.mod
Calculate modulo (remainder after division).

DataFrame.pow
Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...         axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a dictionary by axis.

```
>>> df.mul({'angles': 0, 'degrees': 2})
            angles  degrees
circle           0      720
triangle         0      360
rectangle        0      720
```

```
>>> df.mul({'circle': 0, 'triangle': 2, 'rectangle': 3}, axis='index')
            angles  degrees
circle           0        0
triangle         6      360
rectangle       12     1080
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

AlloViz.AlloViz.Elements.Element.ne

`Element.ne`(*other*, *axis: Axis = 'columns'*, *level=None*)

Get Not equal to of dataframe and other, element-wise (binary operator *ne*).

Among flexible wrappers (*eq*, *ne*, *le*, *lt*, *ge*, *gt*) to comparison operators.

Equivalent to ==, *!*=, <=, <, >=, > with support to choose axis (rows or columns) and level for comparison.

Parameters

other
[scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.

axis
[{0 or 'index', 1 or 'columns'}, default 'columns'] Whether to compare by the index (0 or 'index') or columns (1 or 'columns').

level
[int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

Returns

DataFrame of bool
Result of the comparison.

See also:

`DataFrame.eq`
Compare DataFrames for equality elementwise.

DataFrame.ne
Compare DataFrames for inequality elementwise.

DataFrame.le
Compare DataFrames for less than inequality or equality elementwise.

DataFrame.lt
Compare DataFrames for strictly less than inequality elementwise.

DataFrame.ge
Compare DataFrames for greater than inequality or equality elementwise.

DataFrame.gt
Compare DataFrames for strictly greater than inequality elementwise.

Notes

Mismatched indices will be unioned together. *NaN* values are considered different (i.e. *NaN* != *NaN*).

Examples

```
>>> df = pd.DataFrame({'cost': [250, 150, 100],
...                    'revenue': [100, 250, 300]},
...                   index=['A', 'B', 'C'])
>>> df
   cost  revenue
A   250      100
B   150      250
C   100      300
```

Comparison with a scalar, using either the operator or method:

```
>>> df == 100
    cost  revenue
A  False     True
B  False    False
C   True    False
```

```
>>> df.eq(100)
    cost  revenue
A  False     True
B  False    False
C   True    False
```

When *other* is a `Series`, the columns of a DataFrame are aligned with the index of *other* and broadcast:

```
>>> df != pd.Series([100, 250], index=["cost", "revenue"])
    cost  revenue
A   True     True
B   True    False
C  False     True
```

Use the method to control the broadcast axis:

```
>>> df.ne(pd.Series([100, 300], index=["A", "D"]), axis='index')
   cost  revenue
A  True    False
B  True     True
C  True     True
D  True     True
```

When comparing to an arbitrary sequence, the number of columns must match the number elements in *other*:

```
>>> df == [250, 100]
    cost  revenue
A   True     True
B  False    False
C  False    False
```

Use the method to control the axis:

```
>>> df.eq([250, 250, 100], axis='index')
    cost  revenue
A   True    False
B  False     True
C   True    False
```

Compare to a DataFrame of different shape.

```
>>> other = pd.DataFrame({'revenue': [300, 250, 100, 150]},
...                      index=['A', 'B', 'C', 'D'])
>>> other
   revenue
A      300
B      250
C      100
D      150
```

```
>>> df.gt(other)
    cost  revenue
A  False    False
B  False    False
C  False     True
D  False    False
```

Compare to a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'cost': [250, 150, 100, 150, 300, 220],
...                              'revenue': [100, 250, 300, 200, 175, 225]},
...                             index=[['Q1', 'Q1', 'Q1', 'Q2', 'Q2', 'Q2'],
...                                    ['A', 'B', 'C', 'A', 'B', 'C']])
>>> df_multindex
      cost  revenue
Q1 A   250      100
   B   150      250
   C   100      300
```

(continues on next page)

(continued from previous page)

```
Q2 A   150      200
   B   300      175
   C   220      225
```

```
>>> df.le(df_multindex, level=1)
       cost  revenue
Q1 A   True     True
   B   True     True
   C   True     True
Q2 A  False     True
   B   True    False
   C   True    False
```

AlloViz.AlloViz.Elements.Element.nlargest

`Element.`**`nlargest`**(*n: int*, *columns: IndexLabel*, *keep: NsmallestNlargestKeep = 'first'*) → DataFrame

Return the first *n* rows ordered by *columns* in descending order.

Return the first *n* rows with the largest values in *columns*, in descending order. The columns that are not specified are returned as well, but not used for ordering.

This method is equivalent to `df.sort_values(columns, ascending=False).head(n)`, but more performant.

Parameters

n
[int] Number of rows to return.

columns
[label or list of labels] Column label(s) to order by.

keep
[{'first', 'last', 'all'}, default 'first'] Where there are duplicate values:

- `first` : prioritize the first occurrence(s)
- `last` : prioritize the last occurrence(s)
- `all` : do not drop any duplicates, even it means selecting more than *n* items.

Returns

DataFrame
The first *n* rows ordered by the given columns in descending order.

See also:

`DataFrame.nsmallest`
Return the first *n* rows ordered by *columns* in ascending order.

`DataFrame.sort_values`
Sort DataFrame by the values.

`DataFrame.head`
Return the first *n* rows without re-ordering.

Notes

This function cannot be used with all column types. For example, when specifying columns with *object* or *category* dtypes, `TypeError` is raised.

Examples

```
>>> df = pd.DataFrame({'population': [59000000, 65000000, 434000,
...                                   434000, 434000, 337000, 11300,
...                                   11300, 11300],
...                    'GDP': [1937894, 2583560 , 12011, 4520, 12128,
...                            17036, 182, 38, 311],
...                    'alpha-2': ["IT", "FR", "MT", "MV", "BN",
...                                "IS", "NR", "TV", "AI"]},
...                   index=["Italy", "France", "Malta",
...                          "Maldives", "Brunei", "Iceland",
...                          "Nauru", "Tuvalu", "Anguilla"])
>>> df
          population      GDP alpha-2
Italy       59000000  1937894      IT
France      65000000  2583560      FR
Malta         434000    12011      MT
Maldives      434000     4520      MV
Brunei        434000    12128      BN
Iceland       337000    17036      IS
Nauru          11300      182      NR
Tuvalu         11300       38      TV
Anguilla       11300      311      AI
```

In the following example, we will use `nlargest` to select the three rows having the largest values in column "population".

```
>>> df.nlargest(3, 'population')
        population      GDP alpha-2
France    65000000  2583560      FR
Italy     59000000  1937894      IT
Malta       434000    12011      MT
```

When using `keep='last'`, ties are resolved in reverse order:

```
>>> df.nlargest(3, 'population', keep='last')
        population      GDP alpha-2
France    65000000  2583560      FR
Italy     59000000  1937894      IT
Brunei      434000    12128      BN
```

When using `keep='all'`, all duplicate items are maintained:

```
>>> df.nlargest(3, 'population', keep='all')
          population      GDP alpha-2
France      65000000  2583560      FR
Italy       59000000  1937894      IT
Malta         434000    12011      MT
```

(continues on next page)

(continued from previous page)

```
Maldives      434000     4520      MV
Brunei        434000    12128      BN
```

To order by the largest values in column "population" and then "GDP", we can specify multiple columns like in the next example.

```
>>> df.nlargest(3, ['population', 'GDP'])
        population      GDP alpha-2
France    65000000  2583560      FR
Italy     59000000  1937894      IT
Brunei      434000    12128      BN
```

AlloViz.AlloViz.Elements.Element.notna

Element.**notna**() → DataFrame

Detect existing (non-missing) values.

Return a boolean same-sized object indicating if the values are not NA. Non-missing values get mapped to True. Characters such as empty strings `''` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`). NA values, such as None or `numpy.NaN`, get mapped to False values.

Returns

DataFrame
Mask of bool values for each element in DataFrame that indicates whether an element is not an NA value.

See also:

DataFrame.notnull
Alias of notna.

DataFrame.isna
Boolean inverse of notna.

DataFrame.dropna
Omit axes labels with missing values.

notna
Top-level notna.

Examples

Show which entries in a DataFrame are not NA.

```
>>> df = pd.DataFrame(dict(age=[5, 6, np.nan],
...                        born=[pd.NaT, pd.Timestamp('1939-05-27'),
...                              pd.Timestamp('1940-04-25')],
...                        name=['Alfred', 'Batman', ''],
...                        toy=[None, 'Batmobile', 'Joker']))
>>> df
   age       born    name        toy
```

(continues on next page)

(continued from previous page)

```
0  5.0        NaT  Alfred       None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25              Joker
```

```
>>> df.notna()
     age   born  name    toy
0   True  False  True  False
1   True   True  True   True
2  False   True  True   True
```

Show which entries in a Series are not NA.

```
>>> ser = pd.Series([5, 6, np.nan])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.notna()
0     True
1     True
2    False
dtype: bool
```

AlloViz.AlloViz.Elements.Element.notnull

Element.**notnull**() → DataFrame

DataFrame.notnull is an alias for DataFrame.notna.

Detect existing (non-missing) values.

Return a boolean same-sized object indicating if the values are not NA. Non-missing values get mapped to True. Characters such as empty strings `''` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`). NA values, such as None or `numpy.NaN`, get mapped to False values.

Returns

DataFrame
Mask of bool values for each element in DataFrame that indicates whether an element is not an NA value.

See also:

DataFrame.notnull
Alias of notna.

DataFrame.isna
Boolean inverse of notna.

DataFrame.dropna
Omit axes labels with missing values.

notna
Top-level notna.

Examples

Show which entries in a DataFrame are not NA.

```
>>> df = pd.DataFrame(dict(age=[5, 6, np.nan],
...                        born=[pd.NaT, pd.Timestamp('1939-05-27'),
...                              pd.Timestamp('1940-04-25')],
...                        name=['Alfred', 'Batman', ''],
...                        toy=[None, 'Batmobile', 'Joker']))
>>> df
   age       born    name        toy
0  5.0        NaT  Alfred       None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25              Joker
```

```
>>> df.notna()
     age   born  name    toy
0   True  False  True  False
1   True   True  True   True
2  False   True  True   True
```

Show which entries in a Series are not NA.

```
>>> ser = pd.Series([5, 6, np.nan])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.notna()
0     True
1     True
2    False
dtype: bool
```

AlloViz.AlloViz.Elements.Element.nsmallest

`Element.`**`nsmallest`**(*n: int*, *columns: IndexLabel*, *keep: NsmallestNlargestKeep = 'first'*) → DataFrame

Return the first *n* rows ordered by *columns* in ascending order.

Return the first *n* rows with the smallest values in *columns*, in ascending order. The columns that are not specified are returned as well, but not used for ordering.

This method is equivalent to `df.sort_values(columns, ascending=True).head(n)`, but more performant.

Parameters

n
[int] Number of items to retrieve.

columns
[list or str] Column name or names to order by.

keep
[{'first', 'last', 'all'}, default 'first'] Where there are duplicate values:

- `first` : take the first occurrence.
- `last` : take the last occurrence.
- `all` : do not drop any duplicates, even it means selecting more than *n* items.

Returns

DataFrame

See also:

`DataFrame.nlargest`
Return the first *n* rows ordered by *columns* in descending order.

`DataFrame.sort_values`
Sort DataFrame by the values.

`DataFrame.head`
Return the first *n* rows without re-ordering.

Examples

```
>>> df = pd.DataFrame({'population': [59000000, 65000000, 434000,
...                                   434000, 434000, 337000, 337000,
...                                   11300, 11300],
...                    'GDP': [1937894, 2583560 , 12011, 4520, 12128,
...                            17036, 182, 38, 311],
...                    'alpha-2': ["IT", "FR", "MT", "MV", "BN",
...                                "IS", "NR", "TV", "AI"]},
...                   index=["Italy", "France", "Malta",
...                          "Maldives", "Brunei", "Iceland",
...                          "Nauru", "Tuvalu", "Anguilla"])
>>> df
          population      GDP alpha-2
Italy       59000000  1937894      IT
France      65000000  2583560      FR
Malta         434000    12011      MT
Maldives      434000     4520      MV
Brunei        434000    12128      BN
Iceland       337000    17036      IS
Nauru         337000      182      NR
Tuvalu         11300       38      TV
Anguilla       11300      311      AI
```

In the following example, we will use `nsmallest` to select the three rows having the smallest values in column "population".

```
>>> df.nsmallest(3, 'population')
          population    GDP alpha-2
Tuvalu         11300     38      TV
```

(continues on next page)

(continued from previous page)

```
Anguilla       11300    311      AI
Iceland       337000  17036      IS
```

When using `keep='last'`, ties are resolved in reverse order:

```
>>> df.nsmallest(3, 'population', keep='last')
          population  GDP alpha-2
Anguilla       11300  311      AI
Tuvalu         11300   38      TV
Nauru         337000  182      NR
```

When using `keep='all'`, all duplicate items are maintained:

```
>>> df.nsmallest(3, 'population', keep='all')
          population    GDP alpha-2
Tuvalu         11300     38      TV
Anguilla       11300    311      AI
Iceland       337000  17036      IS
Nauru         337000    182      NR
```

To order by the smallest values in column "population" and then "GDP", we can specify multiple columns like in the next example.

```
>>> df.nsmallest(3, ['population', 'GDP'])
          population  GDP alpha-2
Tuvalu         11300   38      TV
Anguilla       11300  311      AI
Nauru         337000  182      NR
```

AlloViz.AlloViz.Elements.Element.nunique

`Element.`**`nunique`**(*axis: Axis = 0*, *dropna: bool = True*) → Series

Count number of distinct elements in specified axis.

Return Series with number of distinct elements. Can ignore NaN values.

Parameters

axis
[{0 or 'index', 1 or 'columns'}, default 0] The axis to use. 0 or 'index' for row-wise, 1 or 'columns' for column-wise.

dropna
[bool, default True] Don't include NaN in the counts.

Returns

Series

See also:

`Series.nunique`
Method nunique for Series.

`DataFrame.count`
Count non-NA cells for each column or row.

Examples

```
>>> df = pd.DataFrame({'A': [4, 5, 6], 'B': [4, 1, 1]})
>>> df.nunique()
A    3
B    2
dtype: int64
```

```
>>> df.nunique(axis=1)
0    1
1    2
2    2
dtype: int64
```

AlloViz.AlloViz.Elements.Element.pad

Element.**pad**(*, *axis: None | Axis = None*, *inplace: bool_t = False*, *limit: None | int = None*, *downcast: dict | None | lib.NoDefault = _NoDefault.no_default*) → Self | None

Fill NA/NaN values by propagating the last valid observation to next valid.

Deprecated since version 2.0: Series/DataFrame.pad is deprecated. Use Series/DataFrame.ffill instead.

Returns

Series/DataFrame or None
Object with missing values filled or None if `inplace=True`.

Examples

Please see examples for `DataFrame.ffill()` or `Series.ffill()`.

AlloViz.AlloViz.Elements.Element.pct_change

Element.**pct_change**(*periods: int = 1*, *fill_method: Literal['backfill', 'bfill', 'ffill', 'pad'] | None | Literal[_NoDefault.no_default] = _NoDefault.no_default*, *limit: int | None | Literal[_NoDefault.no_default] = _NoDefault.no_default*, *freq=None*, ***kwargs*) → None

Fractional change between the current and a prior element.

Computes the fractional change from the immediately previous row by default. This is useful in comparing the fraction of change in a time series of elements.

Note: Despite the name of this method, it calculates fractional change (also known as per unit change or relative change) and not percentage change. If you need the percentage change, multiply these values by 100.

Parameters

periods
[int, default 1] Periods to shift for forming percent change.

fill_method
[{'backfill', 'bfill', 'pad', 'ffill', None}, default 'pad'] How to handle NAs **before** computing percent changes.

Deprecated since version 2.1.

limit
[int, default None] The number of consecutive NAs to fill before stopping.

Deprecated since version 2.1.

freq
[DateOffset, timedelta, or str, optional] Increment to use from time series API (e.g. 'M' or BDay()).

****kwargs**
Additional keyword arguments are passed into *DataFrame.shift* or *Series.shift*.

Returns

Series or DataFrame
The same type as the calling object.

See also:

`Series.diff`
Compute the difference of two elements in a Series.

`DataFrame.diff`
Compute the difference of two elements in a DataFrame.

`Series.shift`
Shift the index by some number of periods.

`DataFrame.shift`
Shift the index by some number of periods.

Examples

Series

```
>>> s = pd.Series([90, 91, 85])
>>> s
0    90
1    91
2    85
dtype: int64
```

```
>>> s.pct_change()
0         NaN
1    0.011111
2   -0.065934
dtype: float64
```

```
>>> s.pct_change(periods=2)
0         NaN
1         NaN
```

(continues on next page)

(continued from previous page)

```
2   -0.055556
dtype: float64
```

See the percentage change in a Series where filling NAs with last valid observation forward to next valid.

```
>>> s = pd.Series([90, 91, None, 85])
>>> s
0    90.0
1    91.0
2     NaN
3    85.0
dtype: float64
```

```
>>> s.ffill().pct_change()
0         NaN
1    0.011111
2    0.000000
3   -0.065934
dtype: float64
```

DataFrame

Percentage change in French franc, Deutsche Mark, and Italian lira from 1980-01-01 to 1980-03-01.

```
>>> df = pd.DataFrame({
...     'FR': [4.0405, 4.0963, 4.3149],
...     'GR': [1.7246, 1.7482, 1.8519],
...     'IT': [804.74, 810.01, 860.13]},
...     index=['1980-01-01', '1980-02-01', '1980-03-01'])
>>> df
                FR      GR      IT
1980-01-01  4.0405  1.7246  804.74
1980-02-01  4.0963  1.7482  810.01
1980-03-01  4.3149  1.8519  860.13
```

```
>>> df.pct_change()
                  FR        GR        IT
1980-01-01       NaN       NaN       NaN
1980-02-01  0.013810  0.013684  0.006549
1980-03-01  0.053365  0.059318  0.061876
```

Percentage of change in GOOG and APPL stock volume. Shows computing the percentage change between columns.

```
>>> df = pd.DataFrame({
...     '2016': [1769950, 30586265],
...     '2015': [1500923, 40912316],
...     '2014': [1371819, 41403351]},
...     index=['GOOG', 'APPL'])
>>> df
          2016      2015      2014
GOOG   1769950   1500923   1371819
APPL  30586265  40912316  41403351
```

```
>>> df.pct_change(axis='columns', periods=-1)
          2016      2015  2014
GOOG  0.179241  0.094112   NaN
APPL -0.252395 -0.011860   NaN
```

AlloViz.AlloViz.Elements.Element.pipe

Element.**pipe**(*func: Callable[[...], T] | tuple[Callable[[...], T], str]*, **args*, ***kwargs*) → T

Apply chainable functions that expect Series or DataFrames.

Parameters

func
: [function] Function to apply to the Series/DataFrame. `args`, and `kwargs` are passed into `func`. Alternatively a `(callable, data_keyword)` tuple where `data_keyword` is a string indicating the keyword of `callable` that expects the Series/DataFrame.

***args**
: [iterable, optional] Positional arguments passed into `func`.

****kwargs**
: [mapping, optional] A dictionary of keyword arguments passed into `func`.

Returns

the return type of `func`.

See also:

`DataFrame.apply`
: Apply a function along input axis of DataFrame.

`DataFrame.map`
: Apply a function elementwise on a whole DataFrame.

`Series.map`
: Apply a mapping correspondence on a `Series`.

Notes

Use `.pipe` when chaining together functions that expect Series, DataFrames or GroupBy objects.

Examples

Constructing a income DataFrame from a dictionary.

```
>>> data = [[8000, 1000], [9500, np.nan], [5000, 2000]]
>>> df = pd.DataFrame(data, columns=['Salary', 'Others'])
>>> df
   Salary  Others
0    8000  1000.0
1    9500     NaN
2    5000  2000.0
```

Functions that perform tax reductions on an income DataFrame.

```
>>> def subtract_federal_tax(df):
...     return df * 0.9
>>> def subtract_state_tax(df, rate):
...     return df * (1 - rate)
>>> def subtract_national_insurance(df, rate, rate_increase):
...     new_rate = rate + rate_increase
...     return df * (1 - new_rate)
```

Instead of writing

```
>>> subtract_national_insurance(
...     subtract_state_tax(subtract_federal_tax(df), rate=0.12),
...     rate=0.05,
...     rate_increase=0.02)
```

You can write

```
>>> (
...     df.pipe(subtract_federal_tax)
...     .pipe(subtract_state_tax, rate=0.12)
...     .pipe(subtract_national_insurance, rate=0.05, rate_increase=0.02)
... )
    Salary   Others
0  5892.48   736.56
1  6997.32      NaN
2  3682.80  1473.12
```

If you have a function that takes the data as (say) the second argument, pass a tuple indicating which keyword expects the data. For example, suppose `national_insurance` takes its data as `df` in the second argument:

```
>>> def subtract_national_insurance(rate, df, rate_increase):
...     new_rate = rate + rate_increase
...     return df * (1 - new_rate)
>>> (
...     df.pipe(subtract_federal_tax)
...     .pipe(subtract_state_tax, rate=0.12)
...     .pipe(
...         (subtract_national_insurance, 'df'),
...         rate=0.05,
...         rate_increase=0.02
...     )
... )
    Salary   Others
0  5892.48   736.56
1  6997.32      NaN
2  3682.80  1473.12
```

AlloViz.AlloViz.Elements.Element.pivot

Element.**pivot**(*, *columns*, *index=_NoDefault.no_default*, *values=_NoDefault.no_default*) → DataFrame

Return reshaped DataFrame organized by given index / column values.

Reshape data (produce a "pivot" table) based on column values. Uses unique values from specified *index* / *columns* to form axes of the resulting DataFrame. This function does not support data aggregation, multiple values will result in a MultiIndex in the columns. See the User Guide for more on reshaping.

Parameters

columns
[str or object or a list of str] Column to use to make new frame's columns.

index
[str or object or a list of str, optional] Column to use to make new frame's index. If not given, uses existing index.

values
[str, object or a list of the previous, optional] Column(s) to use for populating new frame's values. If not specified, all remaining columns will be used and the result will have hierarchically indexed columns.

Returns

DataFrame
Returns reshaped DataFrame.

Raises

ValueError:
When there are any *index*, *columns* combinations with multiple values. *DataFrame.pivot_table* when you need to aggregate.

See also:

`DataFrame.pivot_table`
Generalization of pivot that can handle duplicate values for one index/column pair.

`DataFrame.unstack`
Pivot based on the index values instead of a column.

`wide_to_long`
Wide panel to long format. Less flexible but more user-friendly than melt.

Notes

For finer-tuned control, see hierarchical indexing documentation along with the related stack/unstack methods.

Reference the user guide for more examples.

Examples

```
>>> df = pd.DataFrame({'foo': ['one', 'one', 'one', 'two', 'two',
...                            'two'],
...                    'bar': ['A', 'B', 'C', 'A', 'B', 'C'],
...                    'baz': [1, 2, 3, 4, 5, 6],
...                    'zoo': ['x', 'y', 'z', 'q', 'w', 't']})
>>> df
    foo   bar  baz  zoo
0   one   A    1    x
1   one   B    2    y
2   one   C    3    z
3   two   A    4    q
4   two   B    5    w
5   two   C    6    t
```

```
>>> df.pivot(index='foo', columns='bar', values='baz')
bar  A   B   C
foo
one  1   2   3
two  4   5   6
```

```
>>> df.pivot(index='foo', columns='bar')['baz']
bar  A   B   C
foo
one  1   2   3
two  4   5   6
```

```
>>> df.pivot(index='foo', columns='bar', values=['baz', 'zoo'])
      baz       zoo
bar   A  B  C   A  B  C
foo
one   1  2  3   x  y  z
two   4  5  6   q  w  t
```

You could also assign a list of column names or a list of index names.

```
>>> df = pd.DataFrame({
...        "lev1": [1, 1, 1, 2, 2, 2],
...        "lev2": [1, 1, 2, 1, 1, 2],
...        "lev3": [1, 2, 1, 2, 1, 2],
...        "lev4": [1, 2, 3, 4, 5, 6],
...        "values": [0, 1, 2, 3, 4, 5]})
>>> df
    lev1 lev2 lev3 lev4 values
0   1    1    1    1    0
1   1    1    2    2    1
2   1    2    1    3    2
3   2    1    2    4    3
4   2    1    1    5    4
5   2    2    2    6    5
```

```
>>> df.pivot(index="lev1", columns=["lev2", "lev3"], values="values")
lev2    1         2
lev3    1    2    1    2
lev1
1     0.0  1.0  2.0  NaN
2     4.0  3.0  NaN  5.0
```

```
>>> df.pivot(index=["lev1", "lev2"], columns=["lev3"], values="values")
      lev3    1    2
lev1  lev2
   1     1  0.0  1.0
         2  2.0  NaN
   2     1  4.0  3.0
         2  NaN  5.0
```

A ValueError is raised if there are any duplicates.

```
>>> df = pd.DataFrame({"foo": ['one', 'one', 'two', 'two'],
...                    "bar": ['A', 'A', 'B', 'C'],
...                    "baz": [1, 2, 3, 4]})
>>> df
   foo bar  baz
0  one   A    1
1  one   A    2
2  two   B    3
3  two   C    4
```

Notice that the first two rows are the same for our *index* and *columns* arguments.

```
>>> df.pivot(index='foo', columns='bar', values='baz')
Traceback (most recent call last):
   ...
ValueError: Index contains duplicate entries, cannot reshape
```

AlloViz.AlloViz.Elements.Element.pivot_table

Element.**pivot_table**(*values=None*, *index=None*, *columns=None*, *aggfunc: AggFuncType = 'mean'*, *fill_value=None*, *margins: bool = False*, *dropna: bool = True*, *margins_name: Level = 'All'*, *observed: bool = False*, *sort: bool = True*) → DataFrame

Create a spreadsheet-style pivot table as a DataFrame.

The levels in the pivot table will be stored in MultiIndex objects (hierarchical indexes) on the index and columns of the result DataFrame.

Parameters

values
[list-like or scalar, optional] Column or columns to aggregate.

index
[column, Grouper, array, or list of the previous] Keys to group by on the pivot table index. If a list is passed, it can contain any of the other types (except list). If an array is passed, it must be the same length as the data and will be used in the same manner as column values.

columns
[column, Grouper, array, or list of the previous] Keys to group by on the pivot table column. If a list is passed, it can contain any of the other types (except list). If an array is passed, it must be the same length as the data and will be used in the same manner as column values.

aggfunc
[function, list of functions, dict, default "mean"] If a list of functions is passed, the resulting pivot table will have hierarchical columns whose top level are the function names (inferred from the function objects themselves). If a dict is passed, the key is column to aggregate and the value is function or list of functions. If `margin=True`, aggfunc will be used to calculate the partial aggregates.

fill_value
[scalar, default None] Value to replace missing values with (in the resulting pivot table, after aggregation).

margins
[bool, default False] If `margins=True`, special `All` columns and rows will be added with partial group aggregates across the categories on the rows and columns.

dropna
[bool, default True] Do not include columns whose entries are all NaN. If True, rows with a NaN value in any column will be omitted before computing margins.

margins_name
[str, default 'All'] Name of the row / column that will contain the totals when margins is True.

observed
[bool, default False] This only applies if any of the groupers are Categoricals. If True: only show observed values for categorical groupers. If False: show all values for categorical groupers.

sort
[bool, default True] Specifies if the result should be sorted.

New in version 1.3.0.

Returns

DataFrame
An Excel style pivot table.

See also:

`DataFrame.pivot`
Pivot without aggregation that can handle non-numeric data.

`DataFrame.melt`
Unpivot a DataFrame from wide to long format, optionally leaving identifiers set.

`wide_to_long`
Wide panel to long format. Less flexible but more user-friendly than melt.

Notes

Reference the user guide for more examples.

Examples

```
>>> df = pd.DataFrame({"A": ["foo", "foo", "foo", "foo", "foo",
...                          "bar", "bar", "bar", "bar"],
...                    "B": ["one", "one", "one", "two", "two",
...                          "one", "one", "two", "two"],
...                    "C": ["small", "large", "large", "small",
...                          "small", "large", "small", "small",
...                          "large"],
...                    "D": [1, 2, 2, 3, 3, 4, 5, 6, 7],
...                    "E": [2, 4, 5, 5, 6, 6, 8, 9, 9]})
>>> df
     A    B      C  D  E
0  foo  one  small  1  2
1  foo  one  large  2  4
2  foo  one  large  2  5
3  foo  two  small  3  5
4  foo  two  small  3  6
5  bar  one  large  4  6
6  bar  one  small  5  8
7  bar  two  small  6  9
8  bar  two  large  7  9
```

This first example aggregates values by taking the sum.

```
>>> table = pd.pivot_table(df, values='D', index=['A', 'B'],
...                        columns=['C'], aggfunc="sum")
>>> table
C        large  small
A   B
bar one    4.0    5.0
    two    7.0    6.0
foo one    4.0    1.0
    two    NaN    6.0
```

We can also fill missing values using the *fill_value* parameter.

```
>>> table = pd.pivot_table(df, values='D', index=['A', 'B'],
...                        columns=['C'], aggfunc="sum", fill_value=0)
>>> table
C        large  small
A   B
bar one      4      5
    two      7      6
foo one      4      1
    two      0      6
```

The next example aggregates by taking the mean across multiple columns.

```
>>> table = pd.pivot_table(df, values=['D', 'E'], index=['A', 'C'],
...                        aggfunc={'D': "mean", 'E': "mean"})
>>> table
                D         E
A   C
bar large  5.500000  7.500000
    small  5.500000  8.500000
foo large  2.000000  4.500000
    small  2.333333  4.333333
```

We can also calculate multiple types of aggregations for any given value column.

```
>>> table = pd.pivot_table(df, values=['D', 'E'], index=['A', 'C'],
...                        aggfunc={'D': "mean",
...                                 'E': ["min", "max", "mean"]})
>>> table
                  D   E
               mean max      mean  min
A   C
bar large  5.500000   9  7.500000    6
    small  5.500000   9  8.500000    8
foo large  2.000000   5  4.500000    4
    small  2.333333   6  4.333333    2
```

AlloViz.AlloViz.Elements.Element.pop

Element.**pop**(*item: Hashable*) → Series

Return item and drop from frame. Raise KeyError if not found.

Parameters

item

[label] Label of column to be popped.

Returns

Series

Examples

```
>>> df = pd.DataFrame([('falcon', 'bird', 389.0),
...                    ('parrot', 'bird', 24.0),
...                    ('lion', 'mammal', 80.5),
...                    ('monkey', 'mammal', np.nan)],
...                   columns=('name', 'class', 'max_speed'))
>>> df
     name   class  max_speed
0  falcon    bird      389.0
1  parrot    bird       24.0
2    lion  mammal       80.5
3  monkey  mammal        NaN
```

```
>>> df.pop('class')
0      bird
1      bird
2    mammal
3    mammal
Name: class, dtype: object
```

```
>>> df
     name  max_speed
0  falcon      389.0
1  parrot       24.0
2    lion       80.5
3  monkey        NaN
```

AlloViz.AlloViz.Elements.Element.pow

Element.**pow**(*other*, *axis: Axis = 'columns'*, *level=None*, *fill_value=None*)

Get Exponential power of dataframe and other, element-wise (binary operator *pow*).

Equivalent to `dataframe ** other`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *rpow*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *floordiv*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other
: [scalar, sequence, Series, dict or DataFrame] Any single or multiple element data structure, or list-like object.

axis
: [{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns. (1 or 'columns'). For Series input, axis to match Series index on.

level
: [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value
: [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame
: Result of the arithmetic operation.

See also:

`DataFrame.add`
: Add DataFrames.

`DataFrame.sub`
: Subtract DataFrames.

DataFrame.mul
Multiply DataFrames.

DataFrame.div
Divide DataFrames (float division).

DataFrame.truediv
Divide DataFrames (float division).

DataFrame.floordiv
Divide DataFrames (integer division).

DataFrame.mod
Calculate modulo (remainder after division).

DataFrame.pow
Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a dictionary by axis.

```
>>> df.mul({'angles': 0, 'degrees': 2})
            angles  degrees
circle           0      720
triangle         0      360
rectangle        0      720
```

```
>>> df.mul({'circle': 0, 'triangle': 2, 'rectangle': 3}, axis='index')
            angles  degrees
circle           0        0
triangle         6      360
rectangle       12     1080
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

AlloViz.AlloViz.Elements.Element.prod

`Element.`**`prod`**(*axis: Axis | None = 0*, *skipna: bool = True*, *numeric_only: bool = False*, *min_count: int = 0*, ***kwargs*)

Return the product of the values over the requested axis.

Parameters

axis

[{index (0), columns (1)}] Axis for the function to be applied on. For *Series* this parameter is unused and defaults to 0.

For DataFrames, specifying `axis=None` will apply the aggregation across both axes.

New in version 2.0.0.

skipna
[bool, default True] Exclude NA/null values when computing the result.

numeric_only
[bool, default False] Include only float, int, boolean columns. Not implemented for Series.

min_count
[int, default 0] The required number of valid values to perform the operation. If fewer than `min_count` non-NA values are present the result will be NA.

****kwargs**
Additional keyword arguments to be passed to the function.

Returns

Series or scalar

See also:

`Series.sum`
Return the sum.

`Series.min`
Return the minimum.

`Series.max`
Return the maximum.

`Series.idxmin`
Return the index of the minimum.

`Series.idxmax`
Return the index of the maximum.

`DataFrame.sum`
Return the sum over the requested axis.

`DataFrame.min`
Return the minimum over the requested axis.

`DataFrame.max`
Return the maximum over the requested axis.

`DataFrame.idxmin`
Return the index of the minimum over the requested axis.

`DataFrame.idxmax`
Return the index of the maximum over the requested axis.

Examples

By default, the product of an empty or all-NA Series is 1

```
>>> pd.Series([], dtype="float64").prod()
1.0
```

This can be controlled with the `min_count` parameter

```
>>> pd.Series([], dtype="float64").prod(min_count=1)
nan
```

Thanks to the `skipna` parameter, `min_count` handles all-NA and empty series identically.

```
>>> pd.Series([np.nan]).prod()
1.0
```

```
>>> pd.Series([np.nan]).prod(min_count=1)
nan
```

AlloViz.AlloViz.Elements.Element.product

Element.**product**(*axis: Axis | None = 0*, *skipna: bool = True*, *numeric_only: bool = False*, *min_count: int = 0*, ***kwargs*)

Return the product of the values over the requested axis.

Parameters

axis
[{index (0), columns (1)}] Axis for the function to be applied on. For *Series* this parameter is unused and defaults to 0.

For DataFrames, specifying `axis=None` will apply the aggregation across both axes.

New in version 2.0.0.

skipna
[bool, default True] Exclude NA/null values when computing the result.

numeric_only
[bool, default False] Include only float, int, boolean columns. Not implemented for Series.

min_count
[int, default 0] The required number of valid values to perform the operation. If fewer than `min_count` non-NA values are present the result will be NA.

****kwargs**
Additional keyword arguments to be passed to the function.

Returns

Series or scalar

See also:

Series.sum
Return the sum.

Series.min
Return the minimum.

Series.max
Return the maximum.

Series.idxmin
Return the index of the minimum.

Series.idxmax
Return the index of the maximum.

DataFrame.sum
Return the sum over the requested axis.

DataFrame.min
Return the minimum over the requested axis.

DataFrame.max
Return the maximum over the requested axis.

DataFrame.idxmin
Return the index of the minimum over the requested axis.

DataFrame.idxmax
Return the index of the maximum over the requested axis.

Examples

By default, the product of an empty or all-NA Series is 1

```
>>> pd.Series([], dtype="float64").prod()
1.0
```

This can be controlled with the `min_count` parameter

```
>>> pd.Series([], dtype="float64").prod(min_count=1)
nan
```

Thanks to the `skipna` parameter, `min_count` handles all-NA and empty series identically.

```
>>> pd.Series([np.nan]).prod()
1.0
```

```
>>> pd.Series([np.nan]).prod(min_count=1)
nan
```

AlloViz.AlloViz.Elements.Element.quantile

`Element.`**`quantile`**(*q: float | AnyArrayLike | Sequence[float] = 0.5*, *axis: Axis = 0*, *numeric_only: bool = False*, *interpolation: QuantileInterpolation = 'linear'*, *method: Literal['single', 'table'] = 'single'*) → Series | DataFrame

Return values at the given quantile over requested axis.

Parameters

q
[float or array-like, default 0.5 (50% quantile)] Value between 0 <= q <= 1, the quantile(s) to compute.

axis
[{0 or 'index', 1 or 'columns'}, default 0] Equals 0 or 'index' for row-wise, 1 or 'columns' for column-wise.

numeric_only
[bool, default False] Include only *float*, *int* or *boolean* data.

Changed in version 2.0.0: The default value of `numeric_only` is now `False`.

interpolation
[{'linear', 'lower', 'higher', 'midpoint', 'nearest'}] This optional parameter specifies the interpolation method to use, when the desired quantile lies between two data points *i* and *j*:

- linear: *i* + (*j* - *i*) * *fraction*, where *fraction* is the fractional part of the index surrounded by *i* and *j*.
- lower: *i*.
- higher: *j*.
- nearest: *i* or *j* whichever is nearest.
- midpoint: (*i* + *j*) / 2.

method
[{'single', 'table'}, default 'single'] Whether to compute quantiles per-column ('single') or over all columns ('table'). When 'table', the only allowed interpolation methods are 'nearest', 'lower', and 'higher'.

Returns

Series or DataFrame

If q is an array, a DataFrame will be returned where the
index is q, the columns are the columns of self, and the values are the quantiles.

If q is a float, a Series will be returned where the
index is the columns of self and the values are the quantiles.

See also:

`core.window.rolling.Rolling.quantile`
Rolling quantile.

`numpy.percentile`
Numpy function to compute the percentile.

Examples

```
>>> df = pd.DataFrame(np.array([[1, 1], [2, 10], [3, 100], [4, 100]]),
...                   columns=['a', 'b'])
>>> df.quantile(.1)
a    1.3
b    3.7
Name: 0.1, dtype: float64
>>> df.quantile([.1, .5])
       a     b
0.1  1.3   3.7
0.5  2.5  55.0
```

Specifying *method='table'* will compute the quantile over all columns.

```
>>> df.quantile(.1, method="table", interpolation="nearest")
a    1
b    1
Name: 0.1, dtype: int64
```

(continues on next page)

(continued from previous page)

```
>>> df.quantile([.1, .5], method="table", interpolation="nearest")
     a    b
0.1  1    1
0.5  3  100
```

Specifying *numeric_only=False* will also compute the quantile of datetime and timedelta data.

```
>>> df = pd.DataFrame({'A': [1, 2],
...                    'B': [pd.Timestamp('2010'),
...                          pd.Timestamp('2011')],
...                    'C': [pd.Timedelta('1 days'),
...                          pd.Timedelta('2 days')]})
>>> df.quantile(0.5, numeric_only=False)
A                    1.5
B    2010-07-02 12:00:00
C        1 days 12:00:00
Name: 0.5, dtype: object
```

AlloViz.AlloViz.Elements.Element.query

`Element.query`(*expr: str*, *, *inplace: bool = False*, ***kwargs*) → DataFrame | None

Query the columns of a DataFrame with a boolean expression.

Parameters

expr
[str] The query string to evaluate.

You can refer to variables in the environment by prefixing them with an '@' character like `@a + b`.

You can refer to column names that are not valid Python variable names by surrounding them in backticks. Thus, column names containing spaces or punctuations (besides underscores) or starting with digits must be surrounded by backticks. (For example, a column named "Area (cm^2)" would be referenced as `` `Area (cm^2)` ``). Column names which are Python keywords (like "list", "for", "import", etc) cannot be used.

For example, if one of your columns is called `a a` and you want to sum it with `b`, your query should be `` `a a` + b ``.

inplace
[bool] Whether to modify the DataFrame rather than creating a new one.

****kwargs**
See the documentation for *eval()* for complete details on the keyword arguments accepted by `DataFrame.query()`.

Returns

DataFrame or None
DataFrame resulting from the provided query expression or None if `inplace=True`.

See also:

eval
: Evaluate a string describing operations on DataFrame columns.

DataFrame.eval
: Evaluate a string describing operations on DataFrame columns.

Notes

The result of the evaluation of this expression is first passed to `DataFrame.loc` and if that fails because of a multidimensional key (e.g., a DataFrame) then the result will be passed to `DataFrame.__getitem__()`.

This method uses the top-level *eval()* function to evaluate the passed query.

The `query()` method uses a slightly modified Python syntax by default. For example, the `&` and `|` (bitwise) operators have the precedence of their boolean cousins, `and` and `or`. This *is* syntactically valid Python, however the semantics are different.

You can change the semantics of the expression by passing the keyword argument `parser='python'`. This enforces the same semantics as evaluation in Python space. Likewise, you can pass `engine='python'` to evaluate an expression using Python itself as a backend. This is not recommended as it is inefficient compared to using `numexpr` as the engine.

The `DataFrame.index` and `DataFrame.columns` attributes of the `DataFrame` instance are placed in the query namespace by default, which allows you to treat both the index and columns of the frame as a column in the frame. The identifier `index` is used for the frame index; you can also use the name of the index to identify it in a query. Please note that Python keywords may not be used as identifiers.

For further details and examples see the `query` documentation in indexing.

Backtick quoted variables

Backtick quoted variables are parsed as literal Python code and are converted internally to a Python valid identifier. This can lead to the following problems.

During parsing a number of disallowed characters inside the backtick quoted string are replaced by strings that are allowed as a Python identifier. These characters include all operators in Python, the space character, the question mark, the exclamation mark, the dollar sign, and the euro sign. For other characters that fall outside the ASCII range (U+0001..U+007F) and those that are not further specified in PEP 3131, the query parser will raise an error. This excludes whitespace different than the space character, but also the hashtag (as it is used for comments) and the backtick itself (backtick can also not be escaped).

In a special case, quotes that make a pair around a backtick can confuse the parser. For example, `` `it's` > `that's` `` will raise an error, as it forms a quoted string (`'s > `that'`) with a backtick inside.

See also the Python documentation about lexical analysis (https://docs.python.org/3/reference/lexical_analysis.html) in combination with the source code in `pandas.core.computation.parsing`.

Examples

```
>>> df = pd.DataFrame({'A': range(1, 6),
...                    'B': range(10, 0, -2),
...                    'C C': range(10, 5, -1)})
>>> df
   A   B  C C
0  1  10   10
1  2   8    9
2  3   6    8
```

(continues on next page)

(continued from previous page)

```
3  4   4    7
4  5   2    6
>>> df.query('A > B')
   A  B  C C
4  5  2    6
```

The previous expression is equivalent to

```
>>> df[df.A > df.B]
   A  B  C C
4  5  2    6
```

For columns with spaces in their name, you can use backtick quoting.

```
>>> df.query('B == `C C`')
   A   B  C C
0  1  10   10
```

The previous expression is equivalent to

```
>>> df[df.B == df['C C']]
   A   B  C C
0  1  10   10
```

AlloViz.AlloViz.Elements.Element.radd

`Element.`**`radd`**(*other*, *axis: Axis = 'columns'*, *level=None*, *fill_value=None*)

Get Addition of dataframe and other, element-wise (binary operator *radd*).

Equivalent to `other + dataframe`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *add*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *floordiv*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other
: [scalar, sequence, Series, dict or DataFrame] Any single or multiple element data structure, or list-like object.

axis
: [{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns. (1 or 'columns'). For Series input, axis to match Series index on.

level
: [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value
: [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame
Result of the arithmetic operation.

See also:

DataFrame.add
Add DataFrames.

DataFrame.sub
Subtract DataFrames.

DataFrame.mul
Multiply DataFrames.

DataFrame.div
Divide DataFrames (float division).

DataFrame.truediv
Divide DataFrames (float division).

DataFrame.floordiv
Divide DataFrames (integer division).

DataFrame.mod
Calculate modulo (remainder after division).

DataFrame.pow
Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
```

(continues on next page)

(continued from previous page)

```
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a dictionary by axis.

```
>>> df.mul({'angles': 0, 'degrees': 2})
           angles  degrees
circle          0      720
triangle        0      360
rectangle       0      720
```

```
>>> df.mul({'circle': 0, 'triangle': 2, 'rectangle': 3}, axis='index')
           angles  degrees
circle          0        0
triangle        6      360
rectangle      12     1080
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

AlloViz.AlloViz.Elements.Element.rank

Element.**rank**(*axis: int | Literal['index', 'columns', 'rows'] = 0*, *method: Literal['average', 'min', 'max', 'first', 'dense'] = 'average'*, *numeric_only: bool = False*, *na_option: Literal['keep', 'top', 'bottom'] = 'keep'*, *ascending: bool = True*, *pct: bool = False*) → None

Compute numerical data ranks (1 through n) along axis.

By default, equal values are assigned a rank that is the average of the ranks of those values.

Parameters

axis
[{0 or 'index', 1 or 'columns'}, default 0] Index to direct ranking. For *Series* this parameter is unused and defaults to 0.

method
[{'average', 'min', 'max', 'first', 'dense'}, default 'average'] How to rank the group of records that have the same value (i.e. ties):

- average: average rank of the group
- min: lowest rank in the group
- max: highest rank in the group
- first: ranks assigned in order they appear in the array
- dense: like 'min', but rank always increases by 1 between groups.

numeric_only
[bool, default False] For DataFrame objects, rank only numeric columns if set to True.

Changed in version 2.0.0: The default value of `numeric_only` is now `False`.

na_option
[{'keep', 'top', 'bottom'}, default 'keep'] How to rank NaN values:

- keep: assign NaN rank to NaN values
- top: assign lowest rank to NaN values
- bottom: assign highest rank to NaN values

ascending
[bool, default True] Whether or not the elements should be ranked in ascending order.

pct
[bool, default False] Whether or not to display the returned rankings in percentile form.

Returns

same type as caller
Return a Series or DataFrame with data ranks as values.

See also:

`core.groupby.DataFrameGroupBy.rank`
Rank of values within each group.

`core.groupby.SeriesGroupBy.rank`
Rank of values within each group.

Examples

```
>>> df = pd.DataFrame(data={'Animal': ['cat', 'penguin', 'dog',
...                                    'spider', 'snake'],
...                         'Number_legs': [4, 2, 4, 8, np.nan]})
>>> df
    Animal  Number_legs
0      cat          4.0
1  penguin          2.0
2      dog          4.0
3   spider          8.0
4    snake          NaN
```

Ties are assigned the mean of the ranks (by default) for the group.

```
>>> s = pd.Series(range(5), index=list("abcde"))
>>> s["d"] = s["b"]
>>> s.rank()
a    1.0
b    2.5
c    4.0
d    2.5
e    5.0
dtype: float64
```

The following example shows how the method behaves with the above parameters:

- default_rank: this is the default behaviour obtained without using any parameter.
- max_rank: setting `method = 'max'` the records that have the same values are ranked using the highest rank (e.g.: since 'cat' and 'dog' are both in the 2nd and 3rd position, rank 3 is assigned.)
- NA_bottom: choosing `na_option = 'bottom'`, if there are records with NaN values they are placed at the bottom of the ranking.
- pct_rank: when setting `pct = True`, the ranking is expressed as percentile rank.

```
>>> df['default_rank'] = df['Number_legs'].rank()
>>> df['max_rank'] = df['Number_legs'].rank(method='max')
>>> df['NA_bottom'] = df['Number_legs'].rank(na_option='bottom')
>>> df['pct_rank'] = df['Number_legs'].rank(pct=True)
>>> df
    Animal  Number_legs  default_rank  max_rank  NA_bottom  pct_rank
0      cat          4.0           2.5       3.0        2.5     0.625
1  penguin          2.0           1.0       1.0        1.0     0.250
2      dog          4.0           2.5       3.0        2.5     0.625
3   spider          8.0           4.0       4.0        4.0     1.000
4    snake          NaN           NaN       NaN        5.0       NaN
```

AlloViz.AlloViz.Elements.Element.rdiv

Element.**rdiv**(*other*, *axis: Axis = 'columns'*, *level=None*, *fill_value=None*)

Get Floating division of dataframe and other, element-wise (binary operator *rtruediv*).

Equivalent to `other / dataframe`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *truediv*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *floordiv*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other
[scalar, sequence, Series, dict or DataFrame] Any single or multiple element data structure, or list-like object.

axis
[{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns. (1 or 'columns'). For Series input, axis to match Series index on.

level
[int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value
[float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame
Result of the arithmetic operation.

See also:

`DataFrame.add`
Add DataFrames.

`DataFrame.sub`
Subtract DataFrames.

`DataFrame.mul`
Multiply DataFrames.

`DataFrame.div`
Divide DataFrames (float division).

`DataFrame.truediv`
Divide DataFrames (float division).

`DataFrame.floordiv`
Divide DataFrames (integer division).

`DataFrame.mod`
Calculate modulo (remainder after division).

`DataFrame.pow`
Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...         axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a dictionary by axis.

```
>>> df.mul({'angles': 0, 'degrees': 2})
            angles  degrees
circle           0      720
triangle         0      360
rectangle        0      720
```

```
>>> df.mul({'circle': 0, 'triangle': 2, 'rectangle': 3}, axis='index')
            angles  degrees
circle           0        0
triangle         6      360
rectangle       12     1080
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

AlloViz.AlloViz.Elements.Element.reindex

Element.**reindex**(*labels=None*, ***, *index=None*, *columns=None*, *axis: Axis | None = None*, *method: ReindexMethod | None = None*, *copy: bool | None = None*, *level: Level | None = None*, *fill_value: Scalar | None = nan*, *limit: int | None = None*, *tolerance=None*) → DataFrame

Conform DataFrame to new index with optional filling logic.

Places NA/NaN in locations having no value in the previous index. A new object is produced unless the new index is equivalent to the current one and `copy=False`.

Parameters

labels
[array-like, optional] New labels / index to conform the axis specified by 'axis' to.

index
[array-like, optional] New labels for the index. Preferably an Index object to avoid duplicating data.

columns
[array-like, optional] New labels for the columns. Preferably an Index object to avoid duplicating data.

axis
[int or str, optional] Axis to target. Can be either the axis name ('index', 'columns') or number (0, 1).

method
[{None, 'backfill'/'bfill', 'pad'/'ffill', 'nearest'}] Method to use for filling holes in reindexed DataFrame. Please note: this is only applicable to DataFrames/Series with a monotonically increasing/decreasing index.

- None (default): don't fill gaps
- pad / ffill: Propagate last valid observation forward to next valid.
- backfill / bfill: Use next valid observation to fill gap.
- nearest: Use nearest valid observations to fill gap.

copy
[bool, default True] Return a new object, even if the passed indexes are the same.

level
[int or name] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value
[scalar, default np.nan] Value to use for missing values. Defaults to NaN, but can be any "compatible" value.

limit
[int, default None] Maximum number of consecutive elements to forward or backward fill.

tolerance
[optional] Maximum distance between original and new labels for inexact matches. The values of the index at the matching locations most satisfy the equation `abs(index[indexer] - target) <= tolerance`.

Tolerance may be a scalar value, which applies the same tolerance to all values, or list-like, which applies variable tolerance per element. List-like includes list, tuple, array, Series, and must be the same size as the index and its dtype must exactly match the index's type.

Returns

DataFrame with changed index.

See also:

`DataFrame.set_index`
Set row labels.

`DataFrame.reset_index`
Remove row labels or move them to new columns.

`DataFrame.reindex_like`
Change to same indices as other DataFrame.

Examples

`DataFrame.reindex` supports two calling conventions

- `(index=index_labels, columns=column_labels, ...)`
- `(labels, axis={'index', 'columns'}, ...)`

We *highly* recommend using keyword arguments to clarify your intent.

Create a dataframe with some fictional data.

```
>>> index = ['Firefox', 'Chrome', 'Safari', 'IE10', 'Konqueror']
>>> df = pd.DataFrame({'http_status': [200, 200, 404, 404, 301],
...                    'response_time': [0.04, 0.02, 0.07, 0.08, 1.0]},
...                    index=index)
>>> df
           http_status  response_time
Firefox            200           0.04
Chrome             200           0.02
Safari             404           0.07
IE10               404           0.08
Konqueror          301           1.00
```

Create a new index and reindex the dataframe. By default values in the new index that do not have corresponding records in the dataframe are assigned `NaN`.

```
>>> new_index = ['Safari', 'Iceweasel', 'Comodo Dragon', 'IE10',
...              'Chrome']
>>> df.reindex(new_index)
               http_status  response_time
Safari               404.0           0.07
Iceweasel              NaN            NaN
Comodo Dragon          NaN            NaN
IE10                 404.0           0.08
Chrome               200.0           0.02
```

We can fill in the missing values by passing a value to the keyword `fill_value`. Because the index is not monotonically increasing or decreasing, we cannot use arguments to the keyword `method` to fill the `NaN` values.

```
>>> df.reindex(new_index, fill_value=0)
               http_status  response_time
Safari                 404           0.07
Iceweasel                0           0.00
Comodo Dragon            0           0.00
IE10                   404           0.08
Chrome                 200           0.02
```

```
>>> df.reindex(new_index, fill_value='missing')
              http_status response_time
Safari                404          0.07
Iceweasel         missing       missing
Comodo Dragon     missing       missing
IE10                  404          0.08
Chrome                200          0.02
```

We can also reindex the columns.

```
>>> df.reindex(columns=['http_status', 'user_agent'])
           http_status  user_agent
Firefox            200         NaN
Chrome             200         NaN
Safari             404         NaN
IE10               404         NaN
```

(continues on next page)

(continued from previous page)

```
Konqueror          301         NaN
```

Or we can use "axis-style" keyword arguments

```
>>> df.reindex(['http_status', 'user_agent'], axis="columns")
           http_status  user_agent
Firefox            200         NaN
Chrome             200         NaN
Safari             404         NaN
IE10               404         NaN
Konqueror          301         NaN
```

To further illustrate the filling functionality in `reindex`, we will create a dataframe with a monotonically increasing index (for example, a sequence of dates).

```
>>> date_index = pd.date_range('1/1/2010', periods=6, freq='D')
>>> df2 = pd.DataFrame({"prices": [100, 101, np.nan, 100, 89, 88]},
...                    index=date_index)
>>> df2
            prices
2010-01-01   100.0
2010-01-02   101.0
2010-01-03     NaN
2010-01-04   100.0
2010-01-05    89.0
2010-01-06    88.0
```

Suppose we decide to expand the dataframe to cover a wider date range.

```
>>> date_index2 = pd.date_range('12/29/2009', periods=10, freq='D')
>>> df2.reindex(date_index2)
            prices
2009-12-29     NaN
2009-12-30     NaN
2009-12-31     NaN
2010-01-01   100.0
2010-01-02   101.0
2010-01-03     NaN
2010-01-04   100.0
2010-01-05    89.0
2010-01-06    88.0
2010-01-07     NaN
```

The index entries that did not have a value in the original data frame (for example, '2009-12-29') are by default filled with `NaN`. If desired, we can fill in the missing values using one of several options.

For example, to back-propagate the last valid value to fill the `NaN` values, pass `bfill` as an argument to the `method` keyword.

```
>>> df2.reindex(date_index2, method='bfill')
            prices
2009-12-29   100.0
2009-12-30   100.0
```

(continues on next page)

(continued from previous page)

```
2009-12-31   100.0
2010-01-01   100.0
2010-01-02   101.0
2010-01-03     NaN
2010-01-04   100.0
2010-01-05    89.0
2010-01-06    88.0
2010-01-07     NaN
```

Please note that the `NaN` value present in the original dataframe (at index value 2010-01-03) will not be filled by any of the value propagation schemes. This is because filling while reindexing does not look at dataframe values, but only compares the original and desired indexes. If you do want to fill in the `NaN` values present in the original dataframe, use the `fillna()` method.

See the user guide for more.

AlloViz.AlloViz.Elements.Element.reindex_like

`Element.reindex_like`(*other*, *method: Literal['backfill', 'bfill', 'pad', 'ffill', 'nearest'] | None = None*, *copy: bool | None = None*, *limit: None | int = None*, *tolerance=None*) → None

Return an object with matching indices as other object.

Conform the object to the same index on all axes. Optional filling logic, placing NaN in locations having no value in the previous index. A new object is produced unless the new index is equivalent to the current one and copy=False.

Parameters

other
[Object of the same data type] Its row and column indices are used to define the new indices of this object.

method
[{None, 'backfill'/'bfill', 'pad'/'ffill', 'nearest'}] Method to use for filling holes in reindexed DataFrame. Please note: this is only applicable to DataFrames/Series with a monotonically increasing/decreasing index.

- None (default): don't fill gaps
- pad / ffill: propagate last valid observation forward to next valid
- backfill / bfill: use next valid observation to fill gap
- nearest: use nearest valid observations to fill gap.

copy
[bool, default True] Return a new object, even if the passed indexes are the same.

limit
[int, default None] Maximum number of consecutive labels to fill for inexact matches.

tolerance
[optional] Maximum distance between original and new labels for inexact matches. The values of the index at the matching locations must satisfy the equation `abs(index[indexer] - target) <= tolerance`.

Tolerance may be a scalar value, which applies the same tolerance to all values, or list-like, which applies variable tolerance per element. List-like includes list, tuple, array, Series, and must be the same size as the index and its dtype must exactly match the index's type.

Returns

Series or DataFrame
Same type as caller, but with changed indices on each axis.

See also:

DataFrame.set_index
Set row labels.

DataFrame.reset_index
Remove row labels or move them to new columns.

DataFrame.reindex
Change to new indices or expand indices.

Notes

Same as calling `.reindex(index=other.index, columns=other.columns,...)`.

Examples

```
>>> df1 = pd.DataFrame([[24.3, 75.7, 'high'],
...                     [31, 87.8, 'high'],
...                     [22, 71.6, 'medium'],
...                     [35, 95, 'medium']],
...                    columns=['temp_celsius', 'temp_fahrenheit',
...                             'windspeed'],
...                    index=pd.date_range(start='2014-02-12',
...                                        end='2014-02-15', freq='D'))
```

```
>>> df1
            temp_celsius  temp_fahrenheit windspeed
2014-02-12          24.3             75.7      high
2014-02-13          31.0             87.8      high
2014-02-14          22.0             71.6    medium
2014-02-15          35.0             95.0    medium
```

```
>>> df2 = pd.DataFrame([[28, 'low'],
...                     [30, 'low'],
...                     [35.1, 'medium']],
...                    columns=['temp_celsius', 'windspeed'],
...                    index=pd.DatetimeIndex(['2014-02-12', '2014-02-13',
...                                            '2014-02-15']))
```

```
>>> df2
            temp_celsius windspeed
2014-02-12          28.0       low
```

(continues on next page)

(continued from previous page)

```
2014-02-13          30.0       low
2014-02-15          35.1    medium
```

```
>>> df2.reindex_like(df1)
            temp_celsius  temp_fahrenheit windspeed
2014-02-12          28.0              NaN       low
2014-02-13          30.0              NaN       low
2014-02-14           NaN              NaN       NaN
2014-02-15          35.1              NaN    medium
```

AlloViz.AlloViz.Elements.Element.rename

Element.**rename**(*mapper: Renamer | None = None*, ***, *index: Renamer | None = None*, *columns: Renamer | None = None*, *axis: Axis | None = None*, *copy: bool | None = None*, *inplace: bool = False*, *level: Level | None = None*, *errors: IgnoreRaise = 'ignore'*) → DataFrame | None

Rename columns or index labels.

Function / dict values must be unique (1-to-1). Labels not contained in a dict / Series will be left as-is. Extra labels listed don't throw an error.

See the user guide for more.

Parameters

mapper
[dict-like or function] Dict-like or function transformations to apply to that axis' values. Use either `mapper` and `axis` to specify the axis to target with `mapper`, or `index` and `columns`.

index
[dict-like or function] Alternative to specifying axis (`mapper, axis=0` is equivalent to `index=mapper`).

columns
[dict-like or function] Alternative to specifying axis (`mapper, axis=1` is equivalent to `columns=mapper`).

axis
[{0 or 'index', 1 or 'columns'}, default 0] Axis to target with `mapper`. Can be either the axis name ('index', 'columns') or number (0, 1). The default is 'index'.

copy
[bool, default True] Also copy underlying data.

inplace
[bool, default False] Whether to modify the DataFrame rather than creating a new one. If True then value of copy is ignored.

level
[int or level name, default None] In case of a MultiIndex, only rename labels in the specified level.

errors
[{'ignore', 'raise'}, default 'ignore'] If 'raise', raise a *KeyError* when a dict-like *mapper*, *index*, or *columns* contains labels that are not present in the Index being

transformed. If 'ignore', existing keys will be renamed and extra keys will be ignored.

Returns

DataFrame or None
DataFrame with the renamed axis labels or None if `inplace=True`.

Raises

KeyError
If any of the labels is not found in the selected axis and "errors='raise'".

See also:

`DataFrame.rename_axis`
Set the name of the axis.

Examples

`DataFrame.rename` supports two calling conventions

- `(index=index_mapper, columns=columns_mapper, ...)`
- `(mapper, axis={'index', 'columns'}, ...)`

We *highly* recommend using keyword arguments to clarify your intent.

Rename columns using a mapping:

```
>>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
>>> df.rename(columns={"A": "a", "B": "c"})
   a  c
0  1  4
1  2  5
2  3  6
```

Rename index using a mapping:

```
>>> df.rename(index={0: "x", 1: "y", 2: "z"})
   A  B
x  1  4
y  2  5
z  3  6
```

Cast index labels to a different type:

```
>>> df.index
RangeIndex(start=0, stop=3, step=1)
>>> df.rename(index=str).index
Index(['0', '1', '2'], dtype='object')
```

```
>>> df.rename(columns={"A": "a", "B": "b", "C": "c"}, errors="raise")
Traceback (most recent call last):
KeyError: ['C'] not found in axis
```

Using axis-style parameters:

```
>>> df.rename(str.lower, axis='columns')
   a  b
0  1  4
1  2  5
2  3  6
```

```
>>> df.rename({1: 2, 2: 4}, axis='index')
   A  B
0  1  4
2  2  5
4  3  6
```

AlloViz.AlloViz.Elements.Element.rename_axis

Element.**rename_axis**(*mapper: IndexLabel | lib.NoDefault = _NoDefault.no_default*, ***, *index=_NoDefault.no_default*, *columns=_NoDefault.no_default*, *axis: Axis = 0*, *copy: bool_t | None = None*, *inplace: bool_t = False*) → Self | None

Set the name of the axis for the index or columns.

Parameters

mapper
[scalar, list-like, optional] Value to set the axis name attribute.

index, columns
[scalar, list-like, dict-like or function, optional] A scalar, list-like, dict-like or functions transformations to apply to that axis' values. Note that the `columns` parameter is not allowed if the object is a Series. This parameter only apply for DataFrame type objects.

Use either `mapper` and `axis` to specify the axis to target with `mapper`, or `index` and/or `columns`.

axis
[{0 or 'index', 1 or 'columns'}, default 0] The axis to rename. For *Series* this parameter is unused and defaults to 0.

copy
[bool, default None] Also copy underlying data.

inplace
[bool, default False] Modifies the object directly, instead of creating a new Series or DataFrame.

Returns

Series, DataFrame, or None
The same type as the caller or None if `inplace=True`.

See also:

`Series.rename`
Alter Series index labels or name.

`DataFrame.rename`
Alter DataFrame index labels or name.

Index.rename
Set new names on index.

Notes

`DataFrame.rename_axis` supports two calling conventions

- `(index=index_mapper, columns=columns_mapper, ...)`
- `(mapper, axis={'index', 'columns'}, ...)`

The first calling convention will only modify the names of the index and/or the names of the Index object that is the columns. In this case, the parameter `copy` is ignored.

The second calling convention will modify the names of the corresponding index if mapper is a list or a scalar. However, if mapper is dict-like or a function, it will use the deprecated behavior of modifying the axis *labels*.

We *highly* recommend using keyword arguments to clarify your intent.

Examples

Series

```
>>> s = pd.Series(["dog", "cat", "monkey"])
>>> s
0       dog
1       cat
2    monkey
dtype: object
>>> s.rename_axis("animal")
animal
0    dog
1    cat
2    monkey
dtype: object
```

DataFrame

```
>>> df = pd.DataFrame({"num_legs": [4, 4, 2],
...                    "num_arms": [0, 0, 2]},
...                   ["dog", "cat", "monkey"])
>>> df
        num_legs  num_arms
dog            4         0
cat            4         0
monkey         2         2
>>> df = df.rename_axis("animal")
>>> df
        num_legs  num_arms
animal
dog            4         0
cat            4         0
monkey         2         2
>>> df = df.rename_axis("limbs", axis="columns")
```

(continues on next page)

(continued from previous page)

```
>>> df
limbs   num_legs  num_arms
animal
dog            4         0
cat            4         0
monkey         2         2
```

MultiIndex

```
>>> df.index = pd.MultiIndex.from_product([['mammal'],
...                                        ['dog', 'cat', 'monkey']],
...                                       names=['type', 'name'])
>>> df
limbs          num_legs  num_arms
type   name
mammal dog            4         0
       cat            4         0
       monkey         2         2
```

```
>>> df.rename_axis(index={'type': 'class'})
limbs          num_legs  num_arms
class  name
mammal dog            4         0
       cat            4         0
       monkey         2         2
```

```
>>> df.rename_axis(columns=str.upper)
LIMBS          num_legs  num_arms
type   name
mammal dog            4         0
       cat            4         0
       monkey         2         2
```

AlloViz.AlloViz.Elements.Element.reorder_levels

`Element.`**`reorder_levels`**(*order: Sequence[int | str]*, *axis: Axis = 0*) → DataFrame

Rearrange index levels using input order. May not drop or duplicate levels.

Parameters

order
[list of int or list of str] List representing new level order. Reference level by number (position) or by key (label).

axis
[{0 or 'index', 1 or 'columns'}, default 0] Where to reorder levels.

Returns

DataFrame

Examples

```
>>> data = {
...     "class": ["Mammals", "Mammals", "Reptiles"],
...     "diet": ["Omnivore", "Carnivore", "Carnivore"],
...     "species": ["Humans", "Dogs", "Snakes"],
... }
>>> df = pd.DataFrame(data, columns=["class", "diet", "species"])
>>> df = df.set_index(["class", "diet"])
>>> df
                                  species
class      diet
Mammals    Omnivore                Humans
           Carnivore                 Dogs
Reptiles   Carnivore               Snakes
```

Let's reorder the levels of the index:

```
>>> df.reorder_levels(["diet", "class"])
                                  species
diet      class
Omnivore  Mammals                 Humans
Carnivore Mammals                   Dogs
          Reptiles                Snakes
```

AlloViz.AlloViz.Elements.Element.replace

Element.**replace**(*to_replace=None*, *value=_NoDefault.no_default*, *, *inplace: bool_t = False*, *limit: int | None = None*, *regex: bool_t = False*, *method: Literal['pad', 'ffill', 'bfill'] | lib.NoDefault = _NoDefault.no_default*) → Self | None

Replace values given in *to_replace* with *value*.

Values of the Series/DataFrame are replaced with other values dynamically. This differs from updating with `.loc` or `.iloc`, which require you to specify a location to update with some value.

Parameters

to_replace
[str, regex, list, dict, Series, int, float, or None] How to find the values that will be replaced.

- numeric, str or regex:
  - numeric: numeric values equal to *to_replace* will be replaced with *value*
  - str: string exactly matching *to_replace* will be replaced with *value*
  - regex: regexs matching *to_replace* will be replaced with *value*
- list of str, regex, or numeric:
  - First, if *to_replace* and *value* are both lists, they **must** be the same length.
  - Second, if `regex=True` then all of the strings in **both** lists will be interpreted as regexs otherwise they will match directly. This doesn't matter much for *value* since there are only a few possible substitution regexes you can use.
  - str, regex and numeric rules apply as above.

- dict:
  - Dicts can be used to specify different replacement values for different existing values. For example, `{'a': 'b', 'y': 'z'}` replaces the value 'a' with 'b' and 'y' with 'z'. To use a dict in this way, the optional *value* parameter should not be given.
  - For a DataFrame a dict can specify that different values should be replaced in different columns. For example, `{'a': 1, 'b': 'z'}` looks for the value 1 in column 'a' and the value 'z' in column 'b' and replaces these values with whatever is specified in *value*. The *value* parameter should not be `None` in this case. You can treat this as a special case of passing two lists except that you are specifying the column to search in.
  - For a DataFrame nested dictionaries, e.g., `{'a': {'b': np.nan}}`, are read as follows: look in column 'a' for the value 'b' and replace it with NaN. The optional *value* parameter should not be specified to use a nested dict in this way. You can nest regular expressions as well. Note that column names (the top-level dictionary keys in a nested dictionary) **cannot** be regular expressions.
- None:
  - This means that the *regex* argument must be a string, compiled regular expression, or list, dict, ndarray or Series of such elements. If *value* is also `None` then this **must** be a nested dictionary or Series.

See the examples section for examples of each of these.

value
: [scalar, dict, list, str, regex, default None] Value to replace any values matching *to_replace* with. For a DataFrame a dict of values can be used to specify which value to use for each column (columns not in the dict will not be filled). Regular expressions, strings and lists or dicts of such objects are also allowed.

inplace
: [bool, default False] If True, performs operation inplace and returns None.

limit
: [int, default None] Maximum size gap to forward or backward fill.

  Deprecated since version 2.1.0.

regex
: [bool or same types as *to_replace*, default False] Whether to interpret *to_replace* and/or *value* as regular expressions. If this is `True` then *to_replace must* be a string. Alternatively, this could be a regular expression or a list, dict, or array of regular expressions in which case *to_replace* must be `None`.

method
: [{'pad', 'ffill', 'bfill'}] The method to use when for replacement, when *to_replace* is a scalar, list or tuple and *value* is `None`.

  Deprecated since version 2.1.0.

Returns

Series/DataFrame
: Object after replacement.

Raises

AssertionError

- If *regex* is not a `bool` and *to_replace* is not `None`.

TypeError

- If *to_replace* is not a scalar, array-like, `dict`, or `None`
- If *to_replace* is a `dict` and *value* is not a `list`, `dict`, `ndarray`, or `Series`
- If *to_replace* is `None` and *regex* is not compilable into a regular expression or is a list, dict, ndarray, or Series.
- When replacing multiple `bool` or `datetime64` objects and the arguments to *to_replace* does not match the type of the value being replaced

ValueError

- If a `list` or an `ndarray` is passed to *to_replace* and *value* but they are not the same length.

See also:

`Series.fillna`
Fill NA values.

`DataFrame.fillna`
Fill NA values.

`Series.where`
Replace values based on boolean condition.

`DataFrame.where`
Replace values based on boolean condition.

`DataFrame.map`
Apply a function to a Dataframe elementwise.

`Series.map`
Map values of Series according to an input mapping or function.

`Series.str.replace`
Simple string replacement.

Notes

- Regex substitution is performed under the hood with `re.sub`. The rules for substitution for `re.sub` are the same.
- Regular expressions will only substitute on strings, meaning you cannot provide, for example, a regular expression matching floating point numbers and expect the columns in your frame that have a numeric dtype to be matched. However, if those floating point numbers *are* strings, then you can do this.
- This method has *a lot* of options. You are encouraged to experiment and play with this method to gain intuition about how it works.
- When dict is used as the *to_replace* value, it is like key(s) in the dict are the to_replace part and value(s) in the dict are the value parameter.

Examples

Scalar `to_replace` and `value`

```
>>> s = pd.Series([1, 2, 3, 4, 5])
>>> s.replace(1, 5)
0    5
1    2
2    3
3    4
4    5
dtype: int64
```

```
>>> df = pd.DataFrame({'A': [0, 1, 2, 3, 4],
...                    'B': [5, 6, 7, 8, 9],
...                    'C': ['a', 'b', 'c', 'd', 'e']})
>>> df.replace(0, 5)
    A  B  C
0  5  5  a
1  1  6  b
2  2  7  c
3  3  8  d
4  4  9  e
```

List-like `to_replace`

```
>>> df.replace([0, 1, 2, 3], 4)
    A  B  C
0  4  5  a
1  4  6  b
2  4  7  c
3  4  8  d
4  4  9  e
```

```
>>> df.replace([0, 1, 2, 3], [4, 3, 2, 1])
    A  B  C
0  4  5  a
1  3  6  b
2  2  7  c
3  1  8  d
4  4  9  e
```

```
>>> s.replace([1, 2], method='bfill')
0    3
1    3
2    3
3    4
4    5
dtype: int64
```

dict-like `to_replace`

```
>>> df.replace({0: 10, 1: 100})
        A  B  C
0   10  5  a
1  100  6  b
2    2  7  c
3    3  8  d
4    4  9  e
```

```
>>> df.replace({'A': 0, 'B': 5}, 100)
        A    B  C
0  100  100  a
1    1    6  b
2    2    7  c
3    3    8  d
4    4    9  e
```

```
>>> df.replace({'A': {0: 100, 4: 400}})
        A  B  C
0  100  5  a
1    1  6  b
2    2  7  c
3    3  8  d
4  400  9  e
```

Regular expression `to_replace`

```
>>> df = pd.DataFrame({'A': ['bat', 'foo', 'bait'],
...                    'B': ['abc', 'bar', 'xyz']})
>>> df.replace(to_replace=r'^ba.$', value='new', regex=True)
        A    B
0   new  abc
1   foo  new
2  bait  xyz
```

```
>>> df.replace({'A': r'^ba.$'}, {'A': 'new'}, regex=True)
        A    B
0   new  abc
1   foo  bar
2  bait  xyz
```

```
>>> df.replace(regex=r'^ba.$', value='new')
        A    B
0   new  abc
1   foo  new
2  bait  xyz
```

```
>>> df.replace(regex={r'^ba.$': 'new', 'foo': 'xyz'})
        A    B
0   new  abc
1   xyz  new
2  bait  xyz
```

```
>>> df.replace(regex=[r'^ba.$', 'foo'], value='new')
        A    B
0   new  abc
1   new  new
2  bait  xyz
```

Compare the behavior of `s.replace({'a': None})` and `s.replace('a', None)` to understand the peculiarities of the *to_replace* parameter:

```
>>> s = pd.Series([10, 'a', 'a', 'b', 'a'])
```

When one uses a dict as the *to_replace* value, it is like the value(s) in the dict are equal to the *value* parameter. `s.replace({'a': None})` is equivalent to `s.replace(to_replace={'a': None}, value=None, method=None)`:

```
>>> s.replace({'a': None})
0      10
1    None
2    None
3       b
4    None
dtype: object
```

When `value` is not explicitly passed and *to_replace* is a scalar, list or tuple, *replace* uses the method parameter (default 'pad') to do the replacement. So this is why the 'a' values are being replaced by 10 in rows 1 and 2 and 'b' in row 4 in this case.

```
>>> s.replace('a')
0    10
1    10
2    10
3     b
4     b
dtype: object
```

Deprecated since version 2.1.0: The 'method' parameter and padding behavior are deprecated.

On the other hand, if `None` is explicitly passed for `value`, it will be respected:

```
>>> s.replace('a', None)
0      10
1    None
2    None
3       b
4    None
dtype: object
```

Changed in version 1.4.0: Previously the explicit `None` was silently ignored.

AlloViz.AlloViz.Elements.Element.resample

Element.**resample**(*rule*, *axis: Axis | lib.NoDefault = _NoDefault.no_default*, *closed: Literal['right', 'left'] | None = None*, *label: Literal['right', 'left'] | None = None*, *convention: Literal['start', 'end', 's', 'e'] = 'start'*, *kind: Literal['timestamp', 'period'] | None = None*, *on: Level | None = None*, *level: Level | None = None*, *origin: str | TimestampConvertibleTypes = 'start_day'*, *offset: TimedeltaConvertibleTypes | None = None*, *group_keys: bool_t = False*) → Resampler

Resample time-series data.

Convenience method for frequency conversion and resampling of time series. The object must have a datetime-like index (*DatetimeIndex*, *PeriodIndex*, or *TimedeltaIndex*), or the caller must pass the label of a datetime-like series/index to the `on`/`level` keyword parameter.

Parameters

rule
[DateOffset, Timedelta or str] The offset string or object representing target conversion.

axis
[{0 or 'index', 1 or 'columns'}, default 0] Which axis to use for up- or down-sampling. For *Series* this parameter is unused and defaults to 0. Must be *DatetimeIndex*, *TimedeltaIndex* or *PeriodIndex*.

Deprecated since version 2.0.0: Use frame.T.resample(...) instead.

closed
[{'right', 'left'}, default None] Which side of bin interval is closed. The default is 'left' for all frequency offsets except for 'M', 'A', 'Q', 'BM', 'BA', 'BQ', and 'W' which all have a default of 'right'.

label
[{'right', 'left'}, default None] Which bin edge label to label bucket with. The default is 'left' for all frequency offsets except for 'M', 'A', 'Q', 'BM', 'BA', 'BQ', and 'W' which all have a default of 'right'.

convention
[{'start', 'end', 's', 'e'}, default 'start'] For *PeriodIndex* only, controls whether to use the start or end of *rule*.

kind
[{'timestamp', 'period'}, optional, default None] Pass 'timestamp' to convert the resulting index to a *DateTimeIndex* or 'period' to convert it to a *PeriodIndex*. By default the input representation is retained.

on
[str, optional] For a DataFrame, column to use instead of index for resampling. Column must be datetime-like.

level
[str or int, optional] For a MultiIndex, level (name or number) to use for resampling. *level* must be datetime-like.

origin
[Timestamp or str, default 'start_day'] The timestamp on which to adjust the grouping. The timezone of origin must match the timezone of the index. If string, must be one of the following:

- 'epoch': *origin* is 1970-01-01

- 'start': *origin* is the first value of the timeseries
- 'start_day': *origin* is the first day at midnight of the timeseries
- 'end': *origin* is the last value of the timeseries
- 'end_day': *origin* is the ceiling midnight of the last day

New in version 1.3.0.

offset
[Timedelta or str, default is None] An offset timedelta added to the origin.

group_keys
[bool, default False] Whether to include the group keys in the result index when using `.apply()` on the resampled object.

New in version 1.5.0: Not specifying `group_keys` will retain values-dependent behavior from pandas 1.4 and earlier (see pandas 1.5.0 Release notes for examples).

Changed in version 2.0.0: `group_keys` now defaults to `False`.

Returns

pandas.api.typing.Resampler
`Resampler` object.

See also:

`Series.resample`
Resample a Series.

`DataFrame.resample`
Resample a DataFrame.

`groupby`
Group Series/DataFrame by mapping, function, label, or list of labels.

`asfreq`
Reindex a Series/DataFrame with the given frequency without grouping.

Notes

See the user guide for more.

To learn more about the offset strings, please see this link.

Examples

Start by creating a series with 9 one minute timestamps.

```
>>> index = pd.date_range('1/1/2000', periods=9, freq='T')
>>> series = pd.Series(range(9), index=index)
>>> series
2000-01-01 00:00:00    0
2000-01-01 00:01:00    1
2000-01-01 00:02:00    2
2000-01-01 00:03:00    3
2000-01-01 00:04:00    4
```

(continues on next page)

(continued from previous page)

```
2000-01-01 00:05:00    5
2000-01-01 00:06:00    6
2000-01-01 00:07:00    7
2000-01-01 00:08:00    8
Freq: T, dtype: int64
```

Downsample the series into 3 minute bins and sum the values of the timestamps falling into a bin.

```
>>> series.resample('3T').sum()
2000-01-01 00:00:00     3
2000-01-01 00:03:00    12
2000-01-01 00:06:00    21
Freq: 3T, dtype: int64
```

Downsample the series into 3 minute bins as above, but label each bin using the right edge instead of the left. Please note that the value in the bucket used as the label is not included in the bucket, which it labels. For example, in the original series the bucket `2000-01-01 00:03:00` contains the value 3, but the summed value in the resampled bucket with the label `2000-01-01 00:03:00` does not include 3 (if it did, the summed value would be 6, not 3). To include this value close the right side of the bin interval as illustrated in the example below this one.

```
>>> series.resample('3T', label='right').sum()
2000-01-01 00:03:00     3
2000-01-01 00:06:00    12
2000-01-01 00:09:00    21
Freq: 3T, dtype: int64
```

Downsample the series into 3 minute bins as above, but close the right side of the bin interval.

```
>>> series.resample('3T', label='right', closed='right').sum()
2000-01-01 00:00:00     0
2000-01-01 00:03:00     6
2000-01-01 00:06:00    15
2000-01-01 00:09:00    15
Freq: 3T, dtype: int64
```

Upsample the series into 30 second bins.

```
>>> series.resample('30S').asfreq()[0:5]   # Select first 5 rows
2000-01-01 00:00:00   0.0
2000-01-01 00:00:30   NaN
2000-01-01 00:01:00   1.0
2000-01-01 00:01:30   NaN
2000-01-01 00:02:00   2.0
Freq: 30S, dtype: float64
```

Upsample the series into 30 second bins and fill the `NaN` values using the `ffill` method.

```
>>> series.resample('30S').ffill()[0:5]
2000-01-01 00:00:00    0
2000-01-01 00:00:30    0
2000-01-01 00:01:00    1
2000-01-01 00:01:30    1
```

(continues on next page)

(continued from previous page)

```
2000-01-01 00:02:00    2
Freq: 30S, dtype: int64
```

Upsample the series into 30 second bins and fill the `NaN` values using the `bfill` method.

```
>>> series.resample('30S').bfill()[0:5]
2000-01-01 00:00:00    0
2000-01-01 00:00:30    1
2000-01-01 00:01:00    1
2000-01-01 00:01:30    2
2000-01-01 00:02:00    2
Freq: 30S, dtype: int64
```

Pass a custom function via `apply`

```
>>> def custom_resampler(arraylike):
...     return np.sum(arraylike) + 5
...
>>> series.resample('3T').apply(custom_resampler)
2000-01-01 00:00:00     8
2000-01-01 00:03:00    17
2000-01-01 00:06:00    26
Freq: 3T, dtype: int64
```

For a Series with a PeriodIndex, the keyword *convention* can be used to control whether to use the start or end of *rule*.

Resample a year by quarter using 'start' *convention*. Values are assigned to the first quarter of the period.

```
>>> s = pd.Series([1, 2], index=pd.period_range('2012-01-01',
...                                             freq='A',
...                                             periods=2))
>>> s
2012    1
2013    2
Freq: A-DEC, dtype: int64
>>> s.resample('Q', convention='start').asfreq()
2012Q1    1.0
2012Q2    NaN
2012Q3    NaN
2012Q4    NaN
2013Q1    2.0
2013Q2    NaN
2013Q3    NaN
2013Q4    NaN
Freq: Q-DEC, dtype: float64
```

Resample quarters by month using 'end' *convention*. Values are assigned to the last month of the period.

```
>>> q = pd.Series([1, 2, 3, 4], index=pd.period_range('2018-01-01',
...                                                   freq='Q',
...                                                   periods=4))
>>> q
```

(continues on next page)

(continued from previous page)

```
2018Q1    1
2018Q2    2
2018Q3    3
2018Q4    4
Freq: Q-DEC, dtype: int64
>>> q.resample('M', convention='end').asfreq()
2018-03    1.0
2018-04    NaN
2018-05    NaN
2018-06    2.0
2018-07    NaN
2018-08    NaN
2018-09    3.0
2018-10    NaN
2018-11    NaN
2018-12    4.0
Freq: M, dtype: float64
```

For DataFrame objects, the keyword *on* can be used to specify the column instead of the index for resampling.

```
>>> d = {'price': [10, 11, 9, 13, 14, 18, 17, 19],
...      'volume': [50, 60, 40, 100, 50, 100, 40, 50]}
>>> df = pd.DataFrame(d)
>>> df['week_starting'] = pd.date_range('01/01/2018',
...                                     periods=8,
...                                     freq='W')
>>> df
   price  volume week_starting
0     10      50    2018-01-07
1     11      60    2018-01-14
2      9      40    2018-01-21
3     13     100    2018-01-28
4     14      50    2018-02-04
5     18     100    2018-02-11
6     17      40    2018-02-18
7     19      50    2018-02-25
>>> df.resample('M', on='week_starting').mean()
               price  volume
week_starting
2018-01-31     10.75    62.5
2018-02-28     17.00    60.0
```

For a DataFrame with MultiIndex, the keyword *level* can be used to specify on which level the resampling needs to take place.

```
>>> days = pd.date_range('1/1/2000', periods=4, freq='D')
>>> d2 = {'price': [10, 11, 9, 13, 14, 18, 17, 19],
...       'volume': [50, 60, 40, 100, 50, 100, 40, 50]}
>>> df2 = pd.DataFrame(
...     d2,
...     index=pd.MultiIndex.from_product(
```

(continues on next page)

(continued from previous page)

```
...         [days, ['morning', 'afternoon']]
...     )
... )
>>> df2
                     price  volume
2000-01-01 morning      10      50
           afternoon    11      60
2000-01-02 morning       9      40
           afternoon    13     100
2000-01-03 morning      14      50
           afternoon    18     100
2000-01-04 morning      17      40
           afternoon    19      50
>>> df2.resample('D', level=0).sum()
            price  volume
2000-01-01     21     110
2000-01-02     22     140
2000-01-03     32     150
2000-01-04     36      90
```

If you want to adjust the start of the bins based on a fixed timestamp:

```
>>> start, end = '2000-10-01 23:30:00', '2000-10-02 00:30:00'
>>> rng = pd.date_range(start, end, freq='7min')
>>> ts = pd.Series(np.arange(len(rng)) * 3, index=rng)
>>> ts
2000-10-01 23:30:00     0
2000-10-01 23:37:00     3
2000-10-01 23:44:00     6
2000-10-01 23:51:00     9
2000-10-01 23:58:00    12
2000-10-02 00:05:00    15
2000-10-02 00:12:00    18
2000-10-02 00:19:00    21
2000-10-02 00:26:00    24
Freq: 7T, dtype: int64
```

```
>>> ts.resample('17min').sum()
2000-10-01 23:14:00     0
2000-10-01 23:31:00     9
2000-10-01 23:48:00    21
2000-10-02 00:05:00    54
2000-10-02 00:22:00    24
Freq: 17T, dtype: int64
```

```
>>> ts.resample('17min', origin='epoch').sum()
2000-10-01 23:18:00     0
2000-10-01 23:35:00    18
2000-10-01 23:52:00    27
2000-10-02 00:09:00    39
2000-10-02 00:26:00    24
Freq: 17T, dtype: int64
```

```
>>> ts.resample('17W', origin='2000-01-01').sum()
2000-01-02      0
2000-04-30      0
2000-08-27      0
2000-12-24    108
Freq: 17W-SUN, dtype: int64
```

If you want to adjust the start of the bins with an *offset* Timedelta, the two following lines are equivalent:

```
>>> ts.resample('17min', origin='start').sum()
2000-10-01 23:30:00     9
2000-10-01 23:47:00    21
2000-10-02 00:04:00    54
2000-10-02 00:21:00    24
Freq: 17T, dtype: int64
```

```
>>> ts.resample('17min', offset='23h30min').sum()
2000-10-01 23:30:00     9
2000-10-01 23:47:00    21
2000-10-02 00:04:00    54
2000-10-02 00:21:00    24
Freq: 17T, dtype: int64
```

If you want to take the largest Timestamp as the end of the bins:

```
>>> ts.resample('17min', origin='end').sum()
2000-10-01 23:35:00     0
2000-10-01 23:52:00    18
2000-10-02 00:09:00    27
2000-10-02 00:26:00    63
Freq: 17T, dtype: int64
```

In contrast with the *start_day*, you can use *end_day* to take the ceiling midnight of the largest Timestamp as the end of the bins and drop the bins not containing data:

```
>>> ts.resample('17min', origin='end_day').sum()
2000-10-01 23:38:00     3
2000-10-01 23:55:00    15
2000-10-02 00:12:00    45
2000-10-02 00:29:00    45
Freq: 17T, dtype: int64
```

AlloViz.AlloViz.Elements.Element.reset_index

Element.**reset_index**(*level: IndexLabel | None = None*, ***, *drop: bool = False*, *inplace: bool = False*, *col_level: Hashable = 0*, *col_fill: Hashable = ''*, *allow_duplicates: bool | lib.NoDefault = _NoDefault.no_default*, *names: Hashable | Sequence[Hashable] | None = None*) → DataFrame | None

Reset the index, or a level of it.

Reset the index of the DataFrame, and use the default one instead. If the DataFrame has a MultiIndex, this method can remove one or more levels.

Parameters

level
[int, str, tuple, or list, default None] Only remove the given levels from the index. Removes all levels by default.

drop
[bool, default False] Do not try to insert index into dataframe columns. This resets the index to the default integer index.

inplace
[bool, default False] Whether to modify the DataFrame rather than creating a new one.

col_level
[int or str, default 0] If the columns have multiple levels, determines which level the labels are inserted into. By default it is inserted into the first level.

col_fill
[object, default ‘’] If the columns have multiple levels, determines how the other levels are named. If None then the index name is repeated.

allow_duplicates
[bool, optional, default lib.no_default] Allow duplicate column labels to be created.

New in version 1.5.0.

names
[int, str or 1-dimensional list, default None] Using the given string, rename the DataFrame column which contains the index data. If the DataFrame has a MultiIndex, this has to be a list or tuple with length equal to the number of levels.

New in version 1.5.0.

Returns

DataFrame or None
DataFrame with the new index or None if `inplace=True`.

See also:

`DataFrame.set_index`
Opposite of reset_index.

`DataFrame.reindex`
Change to new indices or expand indices.

`DataFrame.reindex_like`
Change to same indices as other DataFrame.

Examples

```
>>> df = pd.DataFrame([('bird', 389.0),
...                    ('bird', 24.0),
...                    ('mammal', 80.5),
...                    ('mammal', np.nan)],
...                   index=['falcon', 'parrot', 'lion', 'monkey'],
...                   columns=('class', 'max_speed'))
>>> df
```

(continues on next page)

(continued from previous page)

```
         class  max_speed
falcon    bird      389.0
parrot    bird       24.0
lion    mammal       80.5
monkey  mammal        NaN
```

When we reset the index, the old index is added as a column, and a new sequential index is used:

```
>>> df.reset_index()
    index   class  max_speed
0  falcon    bird      389.0
1  parrot    bird       24.0
2    lion  mammal       80.5
3  monkey  mammal        NaN
```

We can use the *drop* parameter to avoid the old index being added as a column:

```
>>> df.reset_index(drop=True)
    class  max_speed
0    bird      389.0
1    bird       24.0
2  mammal       80.5
3  mammal        NaN
```

You can also use *reset_index* with *MultiIndex*.

```
>>> index = pd.MultiIndex.from_tuples([('bird', 'falcon'),
...                                    ('bird', 'parrot'),
...                                    ('mammal', 'lion'),
...                                    ('mammal', 'monkey')],
...                                   names=['class', 'name'])
>>> columns = pd.MultiIndex.from_tuples([('speed', 'max'),
...                                      ('species', 'type')])
>>> df = pd.DataFrame([(389.0, 'fly'),
...                    (24.0, 'fly'),
...                    (80.5, 'run'),
...                    (np.nan, 'jump')],
...                   index=index,
...                   columns=columns)
>>> df
               speed species
                 max    type
class  name
bird   falcon  389.0     fly
       parrot   24.0     fly
mammal lion     80.5     run
       monkey    NaN    jump
```

Using the *names* parameter, choose a name for the index column:

```
>>> df.reset_index(names=['classes', 'names'])
  classes   names  speed species
                     max    type
```

(continues on next page)

(continued from previous page)

```
0    bird  falcon  389.0     fly
1    bird  parrot   24.0     fly
2  mammal    lion   80.5     run
3  mammal  monkey    NaN    jump
```

If the index has multiple levels, we can reset a subset of them:

```
>>> df.reset_index(level='class')
         class  speed species
                  max    type
name
falcon    bird  389.0     fly
parrot    bird   24.0     fly
lion    mammal   80.5     run
monkey  mammal    NaN    jump
```

If we are not dropping the index, by default, it is placed in the top level. We can place it in another level:

```
>>> df.reset_index(level='class', col_level=1)
                speed species
         class    max    type
name
falcon    bird  389.0     fly
parrot    bird   24.0     fly
lion    mammal   80.5     run
monkey  mammal    NaN    jump
```

When the index is inserted under another level, we can specify under which one with the parameter *col_fill*:

```
>>> df.reset_index(level='class', col_level=1, col_fill='species')
              species  speed species
                class    max    type
name
falcon           bird  389.0     fly
parrot           bird   24.0     fly
lion           mammal   80.5     run
monkey         mammal    NaN    jump
```

If we specify a nonexistent level for *col_fill*, it is created:

```
>>> df.reset_index(level='class', col_level=1, col_fill='genus')
               genus  speed species
               class    max    type
name
falcon           bird  389.0     fly
parrot           bird   24.0     fly
lion           mammal   80.5     run
monkey         mammal    NaN    jump
```

AlloViz.AlloViz.Elements.Element.rfloordiv

Element.**rfloordiv**(*other*, *axis: Axis = 'columns'*, *level=None*, *fill_value=None*)

Get Integer division of dataframe and other, element-wise (binary operator *rfloordiv*).

Equivalent to `other // dataframe`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *floordiv*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *floordiv*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other
[scalar, sequence, Series, dict or DataFrame] Any single or multiple element data structure, or list-like object.

axis
[{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns. (1 or 'columns'). For Series input, axis to match Series index on.

level
[int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value
[float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame
Result of the arithmetic operation.

See also:

DataFrame.add
Add DataFrames.

DataFrame.sub
Subtract DataFrames.

DataFrame.mul
Multiply DataFrames.

DataFrame.div
Divide DataFrames (float division).

DataFrame.truediv
Divide DataFrames (float division).

DataFrame.floordiv
Divide DataFrames (integer division).

DataFrame.mod
Calculate modulo (remainder after division).

DataFrame.pow
Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...         axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a dictionary by axis.

```
>>> df.mul({'angles': 0, 'degrees': 2})
            angles  degrees
circle           0      720
triangle         0      360
rectangle        0      720
```

```
>>> df.mul({'circle': 0, 'triangle': 2, 'rectangle': 3}, axis='index')
            angles  degrees
circle           0        0
triangle         6      360
rectangle       12     1080
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

AlloViz.AlloViz.Elements.Element.rmod

`Element.`**`rmod`**(*other*, *axis: Axis = 'columns'*, *level=None*, *fill_value=None*)

Get Modulo of dataframe and other, element-wise (binary operator *rmod*).

Equivalent to `other % dataframe`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *mod*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *floordiv*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other
: [scalar, sequence, Series, dict or DataFrame] Any single or multiple element data structure, or list-like object.

axis
: [{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns. (1 or 'columns'). For Series input, axis to match Series index on.

level
: [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value
: [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame
Result of the arithmetic operation.

See also:

DataFrame.add
Add DataFrames.

DataFrame.sub
Subtract DataFrames.

DataFrame.mul
Multiply DataFrames.

DataFrame.div
Divide DataFrames (float division).

DataFrame.truediv
Divide DataFrames (float division).

DataFrame.floordiv
Divide DataFrames (integer division).

DataFrame.mod
Calculate modulo (remainder after division).

DataFrame.pow
Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
```

(continues on next page)

(continued from previous page)

```
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a dictionary by axis.

```
>>> df.mul({'angles': 0, 'degrees': 2})
           angles  degrees
circle          0      720
triangle        0      360
rectangle       0      720
```

```
>>> df.mul({'circle': 0, 'triangle': 2, 'rectangle': 3}, axis='index')
           angles  degrees
circle          0        0
triangle        6      360
rectangle      12     1080
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

AlloViz.AlloViz.Elements.Element.rmul

Element.**rmul**(*other*, *axis: Axis = 'columns'*, *level=None*, *fill_value=None*)

Get Multiplication of dataframe and other, element-wise (binary operator *rmul*).

Equivalent to `other * dataframe`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *mul*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *floordiv*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other
[scalar, sequence, Series, dict or DataFrame] Any single or multiple element data structure, or list-like object.

axis
[{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns. (1 or 'columns'). For Series input, axis to match Series index on.

level
[int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value
[float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame
Result of the arithmetic operation.

See also:

DataFrame.add
Add DataFrames.

DataFrame.sub
Subtract DataFrames.

DataFrame.mul
Multiply DataFrames.

DataFrame.div
Divide DataFrames (float division).

DataFrame.truediv
Divide DataFrames (float division).

DataFrame.floordiv
Divide DataFrames (integer division).

DataFrame.mod
Calculate modulo (remainder after division).

DataFrame.pow
Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...         axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a dictionary by axis.

```
>>> df.mul({'angles': 0, 'degrees': 2})
            angles  degrees
circle           0      720
triangle         0      360
rectangle        0      720
```

```
>>> df.mul({'circle': 0, 'triangle': 2, 'rectangle': 3}, axis='index')
            angles  degrees
circle           0        0
triangle         6      360
rectangle       12     1080
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

AlloViz.AlloViz.Elements.Element.rolling

Element.**rolling**(*window: int | dt.timedelta | str | BaseOffset | BaseIndexer*, *min_periods: int | None = None*, *center: bool_t = False*, *win_type: str | None = None*, *on: str | None = None*, *axis: Axis | lib.NoDefault = _NoDefault.no_default*, *closed: IntervalClosedType | None = None*, *step: int | None = None*, *method: str = 'single'*) → Window | Rolling

Provide rolling window calculations.

Parameters

window

[int, timedelta, str, offset, or BaseIndexer subclass] Size of the moving window.

If an integer, the fixed number of observations used for each window.

If a timedelta, str, or offset, the time period of each window. Each window will be a variable sized based on the observations included in the time-period. This is only valid for datetimelike indexes. To learn more about the offsets & frequency strings, please see this link.

If a BaseIndexer subclass, the window boundaries based on the defined `get_window_bounds` method. Additional rolling keyword arguments, namely `min_periods`, `center`, `closed` and `step` will be passed to `get_window_bounds`.

min_periods

[int, default None] Minimum number of observations in window required to have a value; otherwise, result is `np.nan`.

For a window that is specified by an offset, `min_periods` will default to 1.

For a window that is specified by an integer, `min_periods` will default to the size of the window.

center
[bool, default False] If False, set the window labels as the right edge of the window index.

If True, set the window labels as the center of the window index.

win_type
[str, default None] If `None`, all points are evenly weighted.

If a string, it must be a valid scipy.signal window function.

Certain Scipy window types require additional parameters to be passed in the aggregation function. The additional parameters must match the keywords specified in the Scipy window type method signature.

on
[str, optional] For a DataFrame, a column label or Index level on which to calculate the rolling window, rather than the DataFrame's index.

Provided integer column is ignored and excluded from result since an integer index is not used to calculate the rolling window.

axis
[int or str, default 0] If `0` or `'index'`, roll across the rows.

If `1` or `'columns'`, roll across the columns.

For *Series* this parameter is unused and defaults to 0.

closed
[str, default None] If `'right'`, the first point in the window is excluded from calculations.

If `'left'`, the last point in the window is excluded from calculations.

If `'both'`, the no points in the window are excluded from calculations.

If `'neither'`, the first and last points in the window are excluded from calculations.

Default `None` (`'right'`).

Changed in version 1.2.0: The closed parameter with fixed windows is now supported.

step
[int, default None] New in version 1.5.0.

Evaluate the window at every `step` result, equivalent to slicing as `[::step]`. `window` must be an integer. Using a step argument other than None or 1 will produce a result with a different shape than the input.

method
[str {'single', 'table'}, default 'single'] New in version 1.3.0.

Execute the rolling operation per single column or row (`'single'`) or over the entire object (`'table'`).

This argument is only implemented when specifying `engine='numba'` in the method call.

Returns

pandas.api.typing.Window or pandas.api.typing.Rolling
An instance of Window is returned if `win_type` is passed. Otherwise, an instance of Rolling is returned.

See also:

expanding
: Provides expanding transformations.

ewm
: Provides exponential weighted functions.

Notes

See Windowing Operations for further usage details and examples.

Examples

```
>>> df = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]})
>>> df
     B
0  0.0
1  1.0
2  2.0
3  NaN
4  4.0
```

window

Rolling sum with a window length of 2 observations.

```
>>> df.rolling(2).sum()
     B
0  NaN
1  1.0
2  3.0
3  NaN
4  NaN
```

Rolling sum with a window span of 2 seconds.

```
>>> df_time = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]},
...                        index=[pd.Timestamp('20130101 09:00:00'),
...                               pd.Timestamp('20130101 09:00:02'),
...                               pd.Timestamp('20130101 09:00:03'),
...                               pd.Timestamp('20130101 09:00:05'),
...                               pd.Timestamp('20130101 09:00:06')])
```

```
>>> df_time
                       B
2013-01-01 09:00:00  0.0
2013-01-01 09:00:02  1.0
2013-01-01 09:00:03  2.0
2013-01-01 09:00:05  NaN
2013-01-01 09:00:06  4.0
```

```
>>> df_time.rolling('2s').sum()
                       B
2013-01-01 09:00:00  0.0
2013-01-01 09:00:02  1.0
2013-01-01 09:00:03  3.0
2013-01-01 09:00:05  NaN
2013-01-01 09:00:06  4.0
```

Rolling sum with forward looking windows with 2 observations.

```
>>> indexer = pd.api.indexers.FixedForwardWindowIndexer(window_size=2)
>>> df.rolling(window=indexer, min_periods=1).sum()
     B
0  1.0
1  3.0
2  2.0
3  4.0
4  4.0
```

min_periods

Rolling sum with a window length of 2 observations, but only needs a minimum of 1 observation to calculate a value.

```
>>> df.rolling(2, min_periods=1).sum()
     B
0  0.0
1  1.0
2  3.0
3  2.0
4  4.0
```

center

Rolling sum with the result assigned to the center of the window index.

```
>>> df.rolling(3, min_periods=1, center=True).sum()
     B
0  1.0
1  3.0
2  3.0
3  6.0
4  4.0
```

```
>>> df.rolling(3, min_periods=1, center=False).sum()
     B
0  0.0
1  1.0
2  3.0
3  3.0
4  6.0
```

step

Rolling sum with a window length of 2 observations, minimum of 1 observation to calculate a value, and

a step of 2.

```
>>> df.rolling(2, min_periods=1, step=2).sum()
     B
0  0.0
2  3.0
4  4.0
```

win_type

Rolling sum with a window length of 2, using the Scipy `'gaussian'` window type. `std` is required in the aggregation function.

```
>>> df.rolling(2, win_type='gaussian').sum(std=3)
          B
0       NaN
1  0.986207
2  2.958621
3       NaN
4       NaN
```

on

Rolling sum with a window length of 2 days.

```
>>> df = pd.DataFrame({
...     'A': [pd.to_datetime('2020-01-01'),
...           pd.to_datetime('2020-01-01'),
...           pd.to_datetime('2020-01-02'),],
...     'B': [1, 2, 3], },
...     index=pd.date_range('2020', periods=3))
```

```
>>> df
                   A  B
2020-01-01 2020-01-01  1
2020-01-02 2020-01-01  2
2020-01-03 2020-01-02  3
```

```
>>> df.rolling('2D', on='A').sum()
                   A    B
2020-01-01 2020-01-01  1.0
2020-01-02 2020-01-01  3.0
2020-01-03 2020-01-02  6.0
```

AlloViz.AlloViz.Elements.Element.round

`Element.`**round**(*decimals: int | dict[IndexLabel, int] | Series = 0*, **args*, ***kwargs*) → DataFrame

Round a DataFrame to a variable number of decimal places.

Parameters

decimals

[int, dict, Series] Number of decimal places to round each column to. If an int is given, round each column to the same number of places. Otherwise dict and

Series round to variable numbers of places. Column names should be in the keys if *decimals* is a dict-like, or in the index if *decimals* is a Series. Any columns not included in *decimals* will be left as is. Elements of *decimals* which are not columns of the input will be ignored.

***args**
Additional keywords have no effect but might be accepted for compatibility with numpy.

****kwargs**
Additional keywords have no effect but might be accepted for compatibility with numpy.

Returns

DataFrame
A DataFrame with the affected columns rounded to the specified number of decimal places.

See also:

`numpy.around`
Round a numpy array to the given number of decimals.

`Series.round`
Round a Series to the given number of decimals.

Examples

```
>>> df = pd.DataFrame([(.21, .32), (.01, .67), (.66, .03), (.21, .18)],
...                   columns=['dogs', 'cats'])
>>> df
    dogs  cats
0  0.21  0.32
1  0.01  0.67
2  0.66  0.03
3  0.21  0.18
```

By providing an integer each column is rounded to the same number of decimal places

```
>>> df.round(1)
    dogs  cats
0   0.2   0.3
1   0.0   0.7
2   0.7   0.0
3   0.2   0.2
```

With a dict, the number of places for specific columns can be specified with the column names as key and the number of decimal places as value

```
>>> df.round({'dogs': 1, 'cats': 0})
    dogs  cats
0   0.2   0.0
1   0.0   1.0
2   0.7   0.0
3   0.2   0.0
```

Using a Series, the number of places for specific columns can be specified with the column names as index and the number of decimal places as value

```
>>> decimals = pd.Series([0, 1], index=['cats', 'dogs'])
>>> df.round(decimals)
    dogs  cats
0   0.2   0.0
1   0.0   1.0
2   0.7   0.0
3   0.2   0.0
```

AlloViz.AlloViz.Elements.Element.rpow

Element.**rpow**(*other*, *axis: Axis = 'columns'*, *level=None*, *fill_value=None*)

Get Exponential power of dataframe and other, element-wise (binary operator *rpow*).

Equivalent to `other ** dataframe`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *pow*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *floordiv*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other
: [scalar, sequence, Series, dict or DataFrame] Any single or multiple element data structure, or list-like object.

axis
: [{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns. (1 or 'columns'). For Series input, axis to match Series index on.

level
: [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value
: [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame
: Result of the arithmetic operation.

See also:

`DataFrame.add`
: Add DataFrames.

`DataFrame.sub`
: Subtract DataFrames.

`DataFrame.mul`
: Multiply DataFrames.

DataFrame.div
Divide DataFrames (float division).

DataFrame.truediv
Divide DataFrames (float division).

DataFrame.floordiv
Divide DataFrames (integer division).

DataFrame.mod
Calculate modulo (remainder after division).

DataFrame.pow
Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a dictionary by axis.

```
>>> df.mul({'angles': 0, 'degrees': 2})
            angles  degrees
circle           0      720
triangle         0      360
rectangle        0      720
```

```
>>> df.mul({'circle': 0, 'triangle': 2, 'rectangle': 3}, axis='index')
            angles  degrees
circle           0        0
triangle         6      360
rectangle       12     1080
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

AlloViz.AlloViz.Elements.Element.rsub

`Element.`**`rsub`**(*other*, *axis: Axis = 'columns'*, *level=None*, *fill_value=None*)

Get Subtraction of dataframe and other, element-wise (binary operator *rsub*).

Equivalent to `other - dataframe`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *sub*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *floordiv*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other

[scalar, sequence, Series, dict or DataFrame] Any single or multiple element data structure, or list-like object.

axis
[{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns. (1 or 'columns'). For Series input, axis to match Series index on.

level
[int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value
[float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame
Result of the arithmetic operation.

See also:

`DataFrame.add`
Add DataFrames.

`DataFrame.sub`
Subtract DataFrames.

`DataFrame.mul`
Multiply DataFrames.

`DataFrame.div`
Divide DataFrames (float division).

`DataFrame.truediv`
Divide DataFrames (float division).

`DataFrame.floordiv`
Divide DataFrames (integer division).

`DataFrame.mod`
Calculate modulo (remainder after division).

`DataFrame.pow`
Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a dictionary by axis.

```
>>> df.mul({'angles': 0, 'degrees': 2})
           angles  degrees
circle          0      720
triangle        0      360
rectangle       0      720
```

```
>>> df.mul({'circle': 0, 'triangle': 2, 'rectangle': 3}, axis='index')
           angles  degrees
circle          0        0
triangle        6      360
rectangle      12     1080
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
```

(continues on next page)

(continued from previous page)

```
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

AlloViz.AlloViz.Elements.Element.rtruediv

Element.**rtruediv**(*other*, *axis: Axis = 'columns'*, *level=None*, *fill_value=None*)

Get Floating division of dataframe and other, element-wise (binary operator *rtruediv*).

Equivalent to `other / dataframe`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *truediv*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *floordiv*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other
: [scalar, sequence, Series, dict or DataFrame] Any single or multiple element data structure, or list-like object.

axis
: [{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns. (1 or 'columns'). For Series input, axis to match Series index on.

level
: [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value
: [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame
: Result of the arithmetic operation.

See also:

DataFrame.add
: Add DataFrames.

DataFrame.sub
Subtract DataFrames.

DataFrame.mul
Multiply DataFrames.

DataFrame.div
Divide DataFrames (float division).

DataFrame.truediv
Divide DataFrames (float division).

DataFrame.floordiv
Divide DataFrames (integer division).

DataFrame.mod
Calculate modulo (remainder after division).

DataFrame.pow
Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
```

(continues on next page)

(continued from previous page)

```
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a dictionary by axis.

```
>>> df.mul({'angles': 0, 'degrees': 2})
            angles  degrees
circle           0      720
triangle         0      360
rectangle        0      720
```

```
>>> df.mul({'circle': 0, 'triangle': 2, 'rectangle': 3}, axis='index')
            angles  degrees
circle           0        0
triangle         6      360
rectangle       12     1080
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
```

(continues on next page)

(continued from previous page)

```
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

AlloViz.AlloViz.Elements.Element.sample

Element.**sample**(*n: None | int = None*, *frac: float | None = None*, *replace: bool = False*, *weights=None*, *random_state: int | ndarray | Generator | BitGenerator | RandomState | None = None*, *axis: int | Literal['index', 'columns', 'rows'] | None = None*, *ignore_index: bool = False*) → None

Return a random sample of items from an axis of object.

You can use *random_state* for reproducibility.

Parameters

n
[int, optional] Number of items from axis to return. Cannot be used with *frac*. Default = 1 if *frac* = None.

frac
[float, optional] Fraction of axis items to return. Cannot be used with *n*.

replace
[bool, default False] Allow or disallow sampling of the same row more than once.

weights
[str or ndarray-like, optional] Default 'None' results in equal probability weighting. If passed a Series, will align with target object on index. Index values in weights not found in sampled object will be ignored and index values in sampled object not in weights will be assigned weights of zero. If called on a DataFrame, will accept the name of a column when axis = 0. Unless weights are a Series, weights must be same length as axis being sampled. If weights do not sum to 1, they will be normalized to sum to 1. Missing values in the weights column will be treated as zero. Infinite values not allowed.

random_state
[int, array-like, BitGenerator, np.random.RandomState, np.random.Generator, optional] If int, array-like, or BitGenerator, seed for random number generator. If np.random.RandomState or np.random.Generator, use as given.

Changed in version 1.4.0: np.random.Generator objects now accepted

axis
[{0 or 'index', 1 or 'columns', None}, default None] Axis to sample. Accepts axis number or name. Default is stat axis for given data type. For *Series* this parameter is unused and defaults to *None*.

ignore_index
[bool, default False] If True, the resulting index will be labeled 0, 1, …, n - 1.

New in version 1.3.0.

Returns

Series or DataFrame
A new object of same type as caller containing *n* items randomly sampled from the caller object.

See also:

`DataFrameGroupBy.sample`
Generates random samples from each group of a DataFrame object.

`SeriesGroupBy.sample`
Generates random samples from each group of a Series object.

`numpy.random.choice`
Generates a random sample from a given 1-D numpy array.

Notes

If *frac* > 1, *replacement* should be set to *True*.

Examples

```
>>> df = pd.DataFrame({'num_legs': [2, 4, 8, 0],
...                    'num_wings': [2, 0, 0, 0],
...                    'num_specimen_seen': [10, 2, 1, 8]},
...                   index=['falcon', 'dog', 'spider', 'fish'])
>>> df
        num_legs  num_wings  num_specimen_seen
falcon         2          2                 10
dog            4          0                  2
spider         8          0                  1
fish           0          0                  8
```

Extract 3 random elements from the `Series` `df['num_legs']`: Note that we use *random_state* to ensure the reproducibility of the examples.

```
>>> df['num_legs'].sample(n=3, random_state=1)
fish      0
spider    8
falcon    2
Name: num_legs, dtype: int64
```

A random 50% sample of the `DataFrame` with replacement:

```
>>> df.sample(frac=0.5, replace=True, random_state=1)
      num_legs  num_wings  num_specimen_seen
dog          4          0                  2
fish         0          0                  8
```

An upsample sample of the `DataFrame` with replacement: Note that *replace* parameter has to be *True* for *frac* parameter > 1.

```
>>> df.sample(frac=2, replace=True, random_state=1)
        num_legs  num_wings  num_specimen_seen
dog            4          0                  2
fish           0          0                  8
falcon         2          2                 10
falcon         2          2                 10
fish           0          0                  8
dog            4          0                  2
fish           0          0                  8
dog            4          0                  2
```

Using a DataFrame column as weights. Rows with larger value in the *num_specimen_seen* column are more likely to be sampled.

```
>>> df.sample(n=2, weights='num_specimen_seen', random_state=1)
        num_legs  num_wings  num_specimen_seen
```

(continues on next page)

(continued from previous page)

```
falcon         2          2                 10
fish           0          0                  8
```

AlloViz.AlloViz.Elements.Element.select_dtypes

Element.**select_dtypes**(*include=None*, *exclude=None*) → Self

Return a subset of the DataFrame's columns based on the column dtypes.

Parameters

include, exclude
[scalar or list-like] A selection of dtypes or strings to be included/excluded. At least one of these parameters must be supplied.

Returns

DataFrame
The subset of the frame including the dtypes in `include` and excluding the dtypes in `exclude`.

Raises

ValueError

- If both of `include` and `exclude` are empty
- If `include` and `exclude` have overlapping elements
- If any kind of string dtype is passed in.

See also:

`DataFrame.dtypes`
Return Series with the data type of each column.

Notes

- To select all *numeric* types, use `np.number` or `'number'`
- To select strings you must use the `object` dtype, but note that this will return *all* object dtype columns
- See the numpy dtype hierarchy
- To select datetimes, use `np.datetime64`, `'datetime'` or `'datetime64'`
- To select timedeltas, use `np.timedelta64`, `'timedelta'` or `'timedelta64'`
- To select Pandas categorical dtypes, use `'category'`
- To select Pandas datetimetz dtypes, use `'datetimetz'` or `'datetime64[ns, tz]'`

Examples

```
>>> df = pd.DataFrame({'a': [1, 2] * 3,
...                    'b': [True, False] * 3,
...                    'c': [1.0, 2.0] * 3})
>>> df
        a      b  c
0       1   True  1.0
1       2  False  2.0
2       1   True  1.0
3       2  False  2.0
4       1   True  1.0
5       2  False  2.0
```

```
>>> df.select_dtypes(include='bool')
   b
0  True
1  False
2  True
3  False
4  True
5  False
```

```
>>> df.select_dtypes(include=['float64'])
   c
0  1.0
1  2.0
2  1.0
3  2.0
4  1.0
5  2.0
```

```
>>> df.select_dtypes(exclude=['int64'])
       b    c
0   True  1.0
1  False  2.0
2   True  1.0
3  False  2.0
4   True  1.0
5  False  2.0
```

AlloViz.AlloViz.Elements.Element.sem

Element.**sem**(*axis: Axis | None = 0*, *skipna: bool = True*, *ddof: int = 1*, *numeric_only: bool = False*, ***kwargs*)

Return unbiased standard error of the mean over requested axis.

Normalized by N-1 by default. This can be changed using the ddof argument

Parameters

axis

[{index (0), columns (1)}] For *Series* this parameter is unused and defaults to 0.

skipna
: [bool, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

ddof
: [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is N - ddof, where N represents the number of elements.

numeric_only
: [bool, default False] Include only float, int, boolean columns. Not implemented for Series.

Returns

Series or DataFrame (if level specified)

Examples

```
>>> s = pd.Series([1, 2, 3])
>>> s.sem().round(6)
0.57735
```

With a DataFrame

```
>>> df = pd.DataFrame({'a': [1, 2], 'b': [2, 3]}, index=['tiger', 'zebra'])
>>> df
       a   b
tiger  1   2
zebra  2   3
>>> df.sem()
a   0.5
b   0.5
dtype: float64
```

Using axis=1

```
>>> df.sem(axis=1)
tiger   0.5
zebra   0.5
dtype: float64
```

In this case, *numeric_only* should be set to *True* to avoid getting an error.

```
>>> df = pd.DataFrame({'a': [1, 2], 'b': ['T', 'Z']},
...                   index=['tiger', 'zebra'])
>>> df.sem(numeric_only=True)
a   0.5
dtype: float64
```

AlloViz.AlloViz.Elements.Element.set_axis

`Element.set_axis`(*labels*, *, *axis: Axis = 0*, *copy: bool | None = None*) → DataFrame

Assign desired index to given axis.

Indexes for column or row labels can be changed by assigning a list-like or Index.

Parameters

labels
[list-like, Index] The values for the new index.

axis
[{0 or 'index', 1 or 'columns'}, default 0] The axis to update. The value 0 identifies the rows. For *Series* this parameter is unused and defaults to 0.

copy
[bool, default True] Whether to make a copy of the underlying data.

New in version 1.5.0.

Returns

DataFrame
An object of type DataFrame.

See also:

`DataFrame.rename_axis`
Alter the name of the index or columns.

Examples

```
>>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
```

Change the row labels.

```
>>> df.set_axis(['a', 'b', 'c'], axis='index')
   A  B
a  1  4
b  2  5
c  3  6
```

Change the column labels.

```
>>> df.set_axis(['I', 'II'], axis='columns')
   I  II
0  1   4
1  2   5
2  3   6
```

AlloViz.AlloViz.Elements.Element.set_flags

`Element.set_flags`(*, *copy: bool = False*, *allows_duplicate_labels: bool | None = None*) → None

Return a new object with updated flags.

Parameters

copy
[bool, default False] Specify if a copy of the object should be made.

allows_duplicate_labels
[bool, optional] Whether the returned object allows duplicate labels.

Returns

Series or DataFrame
The same type as the caller.

See also:

`DataFrame.attrs`
Global metadata applying to this dataset.

`DataFrame.flags`
Global flags applying to this object.

Notes

This method returns a new object that's a view on the same data as the input. Mutating the input or the output values will be reflected in the other.

This method is intended to be used in method chains.

"Flags" differ from "metadata". Flags reflect properties of the pandas object (the Series or DataFrame). Metadata refer to properties of the dataset, and should be stored in `DataFrame.attrs`.

Examples

```
>>> df = pd.DataFrame({"A": [1, 2]})
>>> df.flags.allows_duplicate_labels
True
>>> df2 = df.set_flags(allows_duplicate_labels=False)
>>> df2.flags.allows_duplicate_labels
False
```

AlloViz.AlloViz.Elements.Element.set_index

`Element.set_index`(*keys*, *, *drop: bool = True*, *append: bool = False*, *inplace: bool = False*, *verify_integrity: bool = False*) → DataFrame | None

Set the DataFrame index using existing columns.

Set the DataFrame index (row labels) using one or more existing columns or arrays (of the correct length). The index can replace the existing index or expand on it.

Parameters

keys
[label or array-like or list of labels/arrays] This parameter can be either a single column key, a single array of the same length as the calling DataFrame, or a list containing an arbitrary combination of column keys and arrays. Here, "array" encompasses `Series`, `Index`, `np.ndarray`, and instances of `Iterator`.

drop
[bool, default True] Delete columns to be used as the new index.

append
[bool, default False] Whether to append columns to existing index.

inplace
[bool, default False] Whether to modify the DataFrame rather than creating a new one.

verify_integrity
[bool, default False] Check the new index for duplicates. Otherwise defer the check until necessary. Setting to False will improve the performance of this method.

Returns

DataFrame or None
Changed row labels or None if `inplace=True`.

See also:

`DataFrame.reset_index`
Opposite of set_index.

`DataFrame.reindex`
Change to new indices or expand indices.

`DataFrame.reindex_like`
Change to same indices as other DataFrame.

Examples

```
>>> df = pd.DataFrame({'month': [1, 4, 7, 10],
...                    'year': [2012, 2014, 2013, 2014],
...                    'sale': [55, 40, 84, 31]})
>>> df
   month  year  sale
0      1  2012    55
1      4  2014    40
2      7  2013    84
3     10  2014    31
```

Set the index to become the 'month' column:

```
>>> df.set_index('month')
       year  sale
month
1      2012    55
4      2014    40
7      2013    84
10     2014    31
```

Create a MultiIndex using columns 'year' and 'month':

```
>>> df.set_index(['year', 'month'])
            sale
year  month
2012  1     55
2014  4     40
2013  7     84
2014  10    31
```

Create a MultiIndex using an Index and a column:

```
>>> df.set_index([pd.Index([1, 2, 3, 4]), 'year'])
         month  sale
   year
1  2012  1      55
2  2014  4      40
3  2013  7      84
4  2014  10     31
```

Create a MultiIndex using two Series:

```
>>> s = pd.Series([1, 2, 3, 4])
>>> df.set_index([s, s**2])
      month  year  sale
1 1       1  2012    55
2 4       4  2014    40
3 9       7  2013    84
4 16     10  2014    31
```

AlloViz.AlloViz.Elements.Element.shift

Element.**shift**(*periods: int | Sequence[int] = 1*, *freq: Frequency | None = None*, *axis: Axis = 0*, *fill_value: Hashable = _NoDefault.no_default*, *suffix: str | None = None*) → DataFrame

Shift index by desired number of periods with an optional time *freq*.

When *freq* is not passed, shift the index without realigning the data. If *freq* is passed (in this case, the index must be date or datetime, or it will raise a *NotImplementedError*), the index will be increased using the periods and the *freq*. *freq* can be inferred when specified as "infer" as long as either freq or inferred_freq attribute is set in the index.

Parameters

periods
: [int or Sequence] Number of periods to shift. Can be positive or negative. If an iterable of ints, the data will be shifted once by each int. This is equivalent to shifting by one value at a time and concatenating all resulting frames. The resulting columns will have the shift suffixed to their column names. For multiple periods, axis must not be 1.

freq
: [DateOffset, tseries.offsets, timedelta, or str, optional] Offset to use from the tseries module or time rule (e.g. 'EOM'). If *freq* is specified then the index values are shifted but the data is not realigned. That is, use *freq* if you would like to extend the index when shifting and preserve the original data. If *freq* is specified as "infer"

then it will be inferred from the freq or inferred_freq attributes of the index. If neither of those attributes exist, a ValueError is thrown.

axis
[{0 or 'index', 1 or 'columns', None}, default None] Shift direction. For *Series* this parameter is unused and defaults to 0.

fill_value
[object, optional] The scalar value to use for newly introduced missing values. the default depends on the dtype of *self*. For numeric data, `np.nan` is used. For datetime, timedelta, or period data, etc. `NaT` is used. For extension dtypes, `self.dtype.na_value` is used.

suffix
[str, optional] If str and periods is an iterable, this is added after the column name and before the shift value for each shifted column name.

Returns

DataFrame
Copy of input object, shifted.

See also:

`Index.shift`
Shift values of Index.

`DatetimeIndex.shift`
Shift values of DatetimeIndex.

`PeriodIndex.shift`
Shift values of PeriodIndex.

Examples

```
>>> df = pd.DataFrame({"Col1": [10, 20, 15, 30, 45],
...                    "Col2": [13, 23, 18, 33, 48],
...                    "Col3": [17, 27, 22, 37, 52]},
...                   index=pd.date_range("2020-01-01", "2020-01-05"))
>>> df
            Col1  Col2  Col3
2020-01-01    10    13    17
2020-01-02    20    23    27
2020-01-03    15    18    22
2020-01-04    30    33    37
2020-01-05    45    48    52
```

```
>>> df.shift(periods=3)
            Col1  Col2  Col3
2020-01-01   NaN   NaN   NaN
2020-01-02   NaN   NaN   NaN
2020-01-03   NaN   NaN   NaN
2020-01-04  10.0  13.0  17.0
2020-01-05  20.0  23.0  27.0
```

```
>>> df.shift(periods=1, axis="columns")
            Col1  Col2  Col3
2020-01-01   NaN    10    13
2020-01-02   NaN    20    23
2020-01-03   NaN    15    18
2020-01-04   NaN    30    33
2020-01-05   NaN    45    48
```

```
>>> df.shift(periods=3, fill_value=0)
            Col1  Col2  Col3
2020-01-01     0     0     0
2020-01-02     0     0     0
2020-01-03     0     0     0
2020-01-04    10    13    17
2020-01-05    20    23    27
```

```
>>> df.shift(periods=3, freq="D")
            Col1  Col2  Col3
2020-01-04    10    13    17
2020-01-05    20    23    27
2020-01-06    15    18    22
2020-01-07    30    33    37
2020-01-08    45    48    52
```

```
>>> df.shift(periods=3, freq="infer")
            Col1  Col2  Col3
2020-01-04    10    13    17
2020-01-05    20    23    27
2020-01-06    15    18    22
2020-01-07    30    33    37
2020-01-08    45    48    52
```

```
>>> df['Col1'].shift(periods=[0, 1, 2])
            Col1_0  Col1_1  Col1_2
2020-01-01      10     NaN     NaN
2020-01-02      20    10.0     NaN
2020-01-03      15    20.0    10.0
2020-01-04      30    15.0    20.0
2020-01-05      45    30.0    15.0
```

AlloViz.AlloViz.Elements.Element.skew

Element.**skew**(*axis: Axis | None = 0*, *skipna: bool = True*, *numeric_only: bool = False*, ***kwargs*)

Return unbiased skew over requested axis.

Normalized by N-1.

Parameters

axis

[{index (0), columns (1)}] Axis for the function to be applied on. For *Series* this parameter is unused and defaults to 0.

For DataFrames, specifying `axis=None` will apply the aggregation across both axes.

New in version 2.0.0.

skipna
[bool, default True] Exclude NA/null values when computing the result.

numeric_only
[bool, default False] Include only float, int, boolean columns. Not implemented for Series.

****kwargs**
Additional keyword arguments to be passed to the function.

Returns

Series or scalar

Examples

```
>>> s = pd.Series([1, 2, 3])
>>> s.skew()
0.0
```

With a DataFrame

```
>>> df = pd.DataFrame({'a': [1, 2, 3], 'b': [2, 3, 4], 'c': [1, 3, 5]},
...                   index=['tiger', 'zebra', 'cow'])
>>> df
        a   b   c
tiger   1   2   1
zebra   2   3   3
cow     3   4   5
>>> df.skew()
a   0.0
b   0.0
c   0.0
dtype: float64
```

Using axis=1

```
>>> df.skew(axis=1)
tiger   1.732051
zebra  -1.732051
cow     0.000000
dtype: float64
```

In this case, *numeric_only* should be set to *True* to avoid getting an error.

```
>>> df = pd.DataFrame({'a': [1, 2, 3], 'b': ['T', 'Z', 'X']},
...                   index=['tiger', 'zebra', 'cow'])
>>> df.skew(numeric_only=True)
a   0.0
dtype: float64
```

AlloViz.AlloViz.Elements.Element.sort_index

Element.**sort_index**(*, *axis: Axis = 0*, *level: IndexLabel | None = None*, *ascending: bool | Sequence[bool] = True*, *inplace: bool = False*, *kind: SortKind = 'quicksort'*, *na_position: NaPosition = 'last'*, *sort_remaining: bool = True*, *ignore_index: bool = False*, *key: IndexKeyFunc | None = None*) → DataFrame | None

Sort object by labels (along an axis).

Returns a new DataFrame sorted by label if *inplace* argument is `False`, otherwise updates the original DataFrame and returns None.

Parameters

axis
[{0 or 'index', 1 or 'columns'}, default 0] The axis along which to sort. The value 0 identifies the rows, and 1 identifies the columns.

level
[int or level name or list of ints or list of level names] If not None, sort on values in specified index level(s).

ascending
[bool or list-like of bools, default True] Sort ascending vs. descending. When the index is a MultiIndex the sort direction can be controlled for each level individually.

inplace
[bool, default False] Whether to modify the DataFrame rather than creating a new one.

kind
[{'quicksort', 'mergesort', 'heapsort', 'stable'}, default 'quicksort'] Choice of sorting algorithm. See also `numpy.sort()` for more information. *mergesort* and *stable* are the only stable algorithms. For DataFrames, this option is only applied when sorting on a single column or label.

na_position
[{'first', 'last'}, default 'last'] Puts NaNs at the beginning if *first*; *last* puts NaNs at the end. Not implemented for MultiIndex.

sort_remaining
[bool, default True] If True and sorting by level and index is multilevel, sort by other levels too (in order) after sorting by specified level.

ignore_index
[bool, default False] If True, the resulting axis will be labeled 0, 1, …, n - 1.

key
[callable, optional] If not None, apply the key function to the index values before sorting. This is similar to the *key* argument in the builtin `sorted()` function, with the notable difference that this *key* function should be *vectorized*. It should expect an `Index` and return an `Index` of the same shape. For MultiIndex inputs, the key is applied *per level*.

Returns

DataFrame or None
The original DataFrame sorted by the labels or None if `inplace=True`.

See also:

Series.sort_index
Sort Series by the index.

DataFrame.sort_values
Sort DataFrame by the value.

Series.sort_values
Sort Series by the value.

Examples

```
>>> df = pd.DataFrame([1, 2, 3, 4, 5], index=[100, 29, 234, 1, 150],
...                   columns=['A'])
>>> df.sort_index()
     A
1    4
29   2
100  1
150  5
234  3
```

By default, it sorts in ascending order, to sort in descending order, use `ascending=False`

```
>>> df.sort_index(ascending=False)
     A
234  3
150  5
100  1
29   2
1    4
```

A key function can be specified which is applied to the index before sorting. For a `MultiIndex` this is applied to each level separately.

```
>>> df = pd.DataFrame({"a": [1, 2, 3, 4]}, index=['A', 'b', 'C', 'd'])
>>> df.sort_index(key=lambda x: x.str.lower())
   a
A  1
b  2
C  3
d  4
```

AlloViz.AlloViz.Elements.Element.sort_values

Element.**sort_values**(*by: IndexLabel*, ***, *axis: Axis = 0*, *ascending: bool | list[bool] | tuple[bool, ...] = True*, *inplace: bool = False*, *kind: SortKind = 'quicksort'*, *na_position: str = 'last'*, *ignore_index: bool = False*, *key: ValueKeyFunc | None = None*) → DataFrame | None

Sort by the values along either axis.

Parameters

by
[str or list of str] Name or list of names to sort by.

- if *axis* is 0 or *'index'* then *by* may contain index levels and/or column labels.
- if *axis* is 1 or *'columns'* then *by* may contain column levels and/or index labels.

axis
["{0 or 'index', 1 or 'columns'}", default 0] Axis to be sorted.

ascending
[bool or list of bool, default True] Sort ascending vs. descending. Specify list for multiple sort orders. If this is a list of bools, must match the length of the by.

inplace
[bool, default False] If True, perform operation in-place.

kind
[{'quicksort', 'mergesort', 'heapsort', 'stable'}, default 'quicksort'] Choice of sorting algorithm. See also `numpy.sort()` for more information. *mergesort* and *stable* are the only stable algorithms. For DataFrames, this option is only applied when sorting on a single column or label.

na_position
[{'first', 'last'}, default 'last'] Puts NaNs at the beginning if *first*; *last* puts NaNs at the end.

ignore_index
[bool, default False] If True, the resulting axis will be labeled 0, 1, . . . , n - 1.

key
[callable, optional] Apply the key function to the values before sorting. This is similar to the *key* argument in the builtin `sorted()` function, with the notable difference that this *key* function should be *vectorized*. It should expect a `Series` and return a Series with the same shape as the input. It will be applied to each column in *by* independently.

Returns

DataFrame or None
DataFrame with sorted values or None if `inplace=True`.

See also:

`DataFrame.sort_index`
Sort a DataFrame by the index.

`Series.sort_values`
Similar method for a Series.

Examples

```
>>> df = pd.DataFrame({
...     'col1': ['A', 'A', 'B', np.nan, 'D', 'C'],
...     'col2': [2, 1, 9, 8, 7, 4],
...     'col3': [0, 1, 9, 4, 2, 3],
...     'col4': ['a', 'B', 'c', 'D', 'e', 'F']
... })
>>> df
  col1  col2  col3 col4
0    A     2     0    a
```

(continues on next page)

(continued from previous page)

```
1    A     1     1    B
2    B     9     9    c
3  NaN     8     4    D
4    D     7     2    e
5    C     4     3    F
```

Sort by col1

```
>>> df.sort_values(by=['col1'])
  col1  col2  col3 col4
0    A     2     0    a
1    A     1     1    B
2    B     9     9    c
5    C     4     3    F
4    D     7     2    e
3  NaN     8     4    D
```

Sort by multiple columns

```
>>> df.sort_values(by=['col1', 'col2'])
  col1  col2  col3 col4
1    A     1     1    B
0    A     2     0    a
2    B     9     9    c
5    C     4     3    F
4    D     7     2    e
3  NaN     8     4    D
```

Sort Descending

```
>>> df.sort_values(by='col1', ascending=False)
  col1  col2  col3 col4
4    D     7     2    e
5    C     4     3    F
2    B     9     9    c
0    A     2     0    a
1    A     1     1    B
3  NaN     8     4    D
```

Putting NAs first

```
>>> df.sort_values(by='col1', ascending=False, na_position='first')
  col1  col2  col3 col4
3  NaN     8     4    D
4    D     7     2    e
5    C     4     3    F
2    B     9     9    c
0    A     2     0    a
1    A     1     1    B
```

Sorting with a key function

```
>>> df.sort_values(by='col4', key=lambda col: col.str.lower())
   col1  col2  col3 col4
0    A     2     0    a
1    A     1     1    B
2    B     9     9    c
3  NaN     8     4    D
4    D     7     2    e
5    C     4     3    F
```

Natural sort with the key argument, using the *natsort <https://github.com/SethMMorton/natsort>* package.

```
>>> df = pd.DataFrame({
...    "time": ['0hr', '128hr', '72hr', '48hr', '96hr'],
...    "value": [10, 20, 30, 40, 50]
... })
>>> df
    time  value
0    0hr     10
1  128hr     20
2   72hr     30
3   48hr     40
4   96hr     50
>>> from natsort import index_natsorted
>>> df.sort_values(
...     by="time",
...     key=lambda x: np.argsort(index_natsorted(df["time"]))
... )
    time  value
0    0hr     10
3   48hr     40
2   72hr     30
4   96hr     50
1  128hr     20
```

AlloViz.AlloViz.Elements.Element.squeeze

`Element.`**`squeeze`**(*axis: int | Literal['index', 'columns', 'rows'] | None = None*)

Squeeze 1 dimensional axis objects into scalars.

Series or DataFrames with a single element are squeezed to a scalar. DataFrames with a single column or a single row are squeezed to a Series. Otherwise the object is unchanged.

This method is most useful when you don't know if your object is a Series or DataFrame, but you do know it has just a single column. In that case you can safely call *squeeze* to ensure you have a Series.

Parameters

axis
[{0 or 'index', 1 or 'columns', None}, default None] A specific axis to squeeze. By default, all length-1 axes are squeezed. For *Series* this parameter is unused and defaults to *None*.

Returns

DataFrame, Series, or scalar
The projection after squeezing *axis* or all the axes.

See also:

`Series.iloc`
Integer-location based indexing for selecting scalars.

`DataFrame.iloc`
Integer-location based indexing for selecting Series.

`Series.to_frame`
Inverse of DataFrame.squeeze for a single-column DataFrame.

Examples

```
>>> primes = pd.Series([2, 3, 5, 7])
```

Slicing might produce a Series with a single value:

```
>>> even_primes = primes[primes % 2 == 0]
>>> even_primes
0    2
dtype: int64
```

```
>>> even_primes.squeeze()
2
```

Squeezing objects with more than one value in every axis does nothing:

```
>>> odd_primes = primes[primes % 2 == 1]
>>> odd_primes
1    3
2    5
3    7
dtype: int64
```

```
>>> odd_primes.squeeze()
1    3
2    5
3    7
dtype: int64
```

Squeezing is even more effective when used with DataFrames.

```
>>> df = pd.DataFrame([[1, 2], [3, 4]], columns=['a', 'b'])
>>> df
   a  b
0  1  2
1  3  4
```

Slicing a single column will produce a DataFrame with the columns having only one value:

```
>>> df_a = df[['a']]
>>> df_a
   a
0  1
1  3
```

So the columns can be squeezed down, resulting in a Series:

```
>>> df_a.squeeze('columns')
0    1
1    3
Name: a, dtype: int64
```

Slicing a single row from a single column will produce a single scalar DataFrame:

```
>>> df_0a = df.loc[df.index < 1, ['a']]
>>> df_0a
   a
0  1
```

Squeezing the rows produces a single scalar Series:

```
>>> df_0a.squeeze('rows')
a    1
Name: 0, dtype: int64
```

Squeezing all axes will project directly into a scalar:

```
>>> df_0a.squeeze()
1
```

AlloViz.AlloViz.Elements.Element.stack

`Element.stack`(*level: IndexLabel = -1*, *dropna: bool | lib.NoDefault = _NoDefault.no_default*, *sort: bool | lib.NoDefault = _NoDefault.no_default*, *future_stack: bool = False*)

Stack the prescribed level(s) from columns to index.

Return a reshaped DataFrame or Series having a multi-level index with one or more new inner-most levels compared to the current DataFrame. The new inner-most levels are created by pivoting the columns of the current dataframe:

- if the columns have a single level, the output is a Series;
- if the columns have multiple levels, the new index level(s) is (are) taken from the prescribed level(s) and the output is a DataFrame.

Parameters

level
[int, str, list, default -1] Level(s) to stack from the column axis onto the index axis, defined as one index or label, or a list of indices or labels.

dropna
[bool, default True] Whether to drop rows in the resulting Frame/Series with missing values. Stacking a column level onto the index axis can create combinations of

index and column values that are missing from the original dataframe. See Examples section.

sort
[bool, default True] Whether to sort the levels of the resulting MultiIndex.

future_stack
[bool, default False] Whether to use the new implementation that will replace the current implementation in pandas 3.0. When True, dropna and sort have no impact on the result and must remain unspecified. See pandas 2.1.0 Release notes for more details.

Returns

DataFrame or Series
Stacked dataframe or series.

See also:

`DataFrame.unstack`
Unstack prescribed level(s) from index axis onto column axis.

`DataFrame.pivot`
Reshape dataframe from long format to wide format.

`DataFrame.pivot_table`
Create a spreadsheet-style pivot table as a DataFrame.

Notes

The function is named by analogy with a collection of books being reorganized from being side by side on a horizontal position (the columns of the dataframe) to being stacked vertically on top of each other (in the index of the dataframe).

Reference the user guide for more examples.

Examples

Single level columns

```
>>> df_single_level_cols = pd.DataFrame([[0, 1], [2, 3]],
...                                     index=['cat', 'dog'],
...                                     columns=['weight', 'height'])
```

Stacking a dataframe with a single level column axis returns a Series:

```
>>> df_single_level_cols
     weight height
cat       0      1
dog       2      3
>>> df_single_level_cols.stack(future_stack=True)
cat  weight    0
     height    1
dog  weight    2
     height    3
dtype: int64
```

Multi level columns: simple case

```
>>> multicol1 = pd.MultiIndex.from_tuples([('weight', 'kg'),
...                                        ('weight', 'pounds')])
>>> df_multi_level_cols1 = pd.DataFrame([[1, 2], [2, 4]],
...                                     index=['cat', 'dog'],
...                                     columns=multicol1)
```

Stacking a dataframe with a multi-level column axis:

```
>>> df_multi_level_cols1
     weight
         kg    pounds
cat       1         2
dog       2         4
>>> df_multi_level_cols1.stack(future_stack=True)
            weight
cat kg           1
    pounds       2
dog kg           2
    pounds       4
```

Missing values

```
>>> multicol2 = pd.MultiIndex.from_tuples([('weight', 'kg'),
...                                        ('height', 'm')])
>>> df_multi_level_cols2 = pd.DataFrame([[1.0, 2.0], [3.0, 4.0]],
...                                     index=['cat', 'dog'],
...                                     columns=multicol2)
```

It is common to have missing values when stacking a dataframe with multi-level columns, as the stacked dataframe typically has more values than the original dataframe. Missing values are filled with NaNs:

```
>>> df_multi_level_cols2
    weight height
        kg      m
cat    1.0    2.0
dog    3.0    4.0
>>> df_multi_level_cols2.stack(future_stack=True)
        weight  height
cat kg     1.0     NaN
    m      NaN     2.0
dog kg     3.0     NaN
    m      NaN     4.0
```

Prescribing the level(s) to be stacked

The first parameter controls which level or levels are stacked:

```
>>> df_multi_level_cols2.stack(0, future_stack=True)
             kg    m
cat weight  1.0  NaN
    height  NaN  2.0
dog weight  3.0  NaN
    height  NaN  4.0
```

(continues on next page)

(continued from previous page)

```
>>> df_multi_level_cols2.stack([0, 1], future_stack=True)
cat  weight  kg    1.0
     height  m     2.0
dog  weight  kg    3.0
     height  m     4.0
dtype: float64
```

Dropping missing values

```
>>> df_multi_level_cols3 = pd.DataFrame([[None, 1.0], [2.0, 3.0]],
...                                     index=['cat', 'dog'],
...                                     columns=multicol2)
```

Note that rows where all values are missing are dropped by default but this behaviour can be controlled via the dropna keyword parameter:

```
>>> df_multi_level_cols3
    weight height
        kg      m
cat    NaN    1.0
dog    2.0    3.0
>>> df_multi_level_cols3.stack(dropna=False)
        weight  height
cat kg     NaN     NaN
    m      NaN     1.0
dog kg     2.0     NaN
    m      NaN     3.0
>>> df_multi_level_cols3.stack(dropna=True)
        weight  height
cat m      NaN     1.0
dog kg     2.0     NaN
    m      NaN     3.0
```

AlloViz.AlloViz.Elements.Element.std

`Element.std`(*axis: Axis | None = 0*, *skipna: bool = True*, *ddof: int = 1*, *numeric_only: bool = False*, ***kwargs*)

Return sample standard deviation over requested axis.

Normalized by N-1 by default. This can be changed using the ddof argument.

Parameters

axis
[{index (0), columns (1)}] For *Series* this parameter is unused and defaults to 0.

skipna
[bool, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

ddof
[int, default 1] Delta Degrees of Freedom. The divisor used in calculations is N - ddof, where N represents the number of elements.

numeric_only
[bool, default False] Include only float, int, boolean columns. Not implemented for Series.

Returns

Series or DataFrame (if level specified)

Notes

To have the same behaviour as *numpy.std*, use *ddof=0* (instead of the default *ddof=1*)

Examples

```
>>> df = pd.DataFrame({'person_id': [0, 1, 2, 3],
...                    'age': [21, 25, 62, 43],
...                    'height': [1.61, 1.87, 1.49, 2.01]}
...                   ).set_index('person_id')
>>> df
           age  height
person_id
0           21    1.61
1           25    1.87
2           62    1.49
3           43    2.01
```

The standard deviation of the columns can be found as follows:

```
>>> df.std()
age       18.786076
height     0.237417
dtype: float64
```

Alternatively, *ddof=0* can be set to normalize by N instead of N-1:

```
>>> df.std(ddof=0)
age       16.269219
height     0.205609
dtype: float64
```

AlloViz.AlloViz.Elements.Element.sub

`Element.`**`sub`**(*other*, *axis: Axis = 'columns'*, *level=None*, *fill_value=None*)

Get Subtraction of dataframe and other, element-wise (binary operator *sub*).

Equivalent to `dataframe - other`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *rsub*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *floordiv*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other
[scalar, sequence, Series, dict or DataFrame] Any single or multiple element data structure, or list-like object.

axis
[{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns. (1 or 'columns'). For Series input, axis to match Series index on.

level
[int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value
[float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame
Result of the arithmetic operation.

See also:

`DataFrame.add`
Add DataFrames.

`DataFrame.sub`
Subtract DataFrames.

`DataFrame.mul`
Multiply DataFrames.

`DataFrame.div`
Divide DataFrames (float division).

`DataFrame.truediv`
Divide DataFrames (float division).

`DataFrame.floordiv`
Divide DataFrames (integer division).

`DataFrame.mod`
Calculate modulo (remainder after division).

`DataFrame.pow`
Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a dictionary by axis.

```
>>> df.mul({'angles': 0, 'degrees': 2})
           angles  degrees
circle          0      720
triangle        0      360
rectangle       0      720
```

```
>>> df.mul({'circle': 0, 'triangle': 2, 'rectangle': 3}, axis='index')
           angles  degrees
circle          0        0
triangle        6      360
rectangle      12     1080
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
```

(continues on next page)

(continued from previous page)

```
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

AlloViz.AlloViz.Elements.Element.subtract

Element.**subtract**(*other*, *axis: Axis = 'columns'*, *level=None*, *fill_value=None*)

Get Subtraction of dataframe and other, element-wise (binary operator *sub*).

Equivalent to `dataframe - other`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *rsub*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *floordiv*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other
: [scalar, sequence, Series, dict or DataFrame] Any single or multiple element data structure, or list-like object.

axis
: [{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns. (1 or 'columns'). For Series input, axis to match Series index on.

level
: [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value
: [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame
: Result of the arithmetic operation.

See also:

DataFrame.add
: Add DataFrames.

DataFrame.sub
Subtract DataFrames.

DataFrame.mul
Multiply DataFrames.

DataFrame.div
Divide DataFrames (float division).

DataFrame.truediv
Divide DataFrames (float division).

DataFrame.floordiv
Divide DataFrames (integer division).

DataFrame.mod
Calculate modulo (remainder after division).

DataFrame.pow
Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
```

(continues on next page)

(continued from previous page)

```
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a dictionary by axis.

```
>>> df.mul({'angles': 0, 'degrees': 2})
            angles  degrees
circle           0      720
triangle         0      360
rectangle        0      720
```

```
>>> df.mul({'circle': 0, 'triangle': 2, 'rectangle': 3}, axis='index')
            angles  degrees
circle           0        0
triangle         6      360
rectangle       12     1080
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
```

(continues on next page)

(continued from previous page)

```
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

AlloViz.AlloViz.Elements.Element.sum

Element.**sum**(*axis: Axis | None = 0*, *skipna: bool = True*, *numeric_only: bool = False*, *min_count: int = 0*, ***kwargs*)

Return the sum of the values over the requested axis.

This is equivalent to the method `numpy.sum`.

Parameters

axis

[{index (0), columns (1)}] Axis for the function to be applied on. For *Series* this

parameter is unused and defaults to 0.

For DataFrames, specifying `axis=None` will apply the aggregation across both axes.

New in version 2.0.0.

skipna
[bool, default True] Exclude NA/null values when computing the result.

numeric_only
[bool, default False] Include only float, int, boolean columns. Not implemented for Series.

min_count
[int, default 0] The required number of valid values to perform the operation. If fewer than `min_count` non-NA values are present the result will be NA.

****kwargs**
Additional keyword arguments to be passed to the function.

Returns

Series or scalar

See also:

`Series.sum`
Return the sum.

`Series.min`
Return the minimum.

`Series.max`
Return the maximum.

`Series.idxmin`
Return the index of the minimum.

`Series.idxmax`
Return the index of the maximum.

`DataFrame.sum`
Return the sum over the requested axis.

`DataFrame.min`
Return the minimum over the requested axis.

`DataFrame.max`
Return the maximum over the requested axis.

`DataFrame.idxmin`
Return the index of the minimum over the requested axis.

`DataFrame.idxmax`
Return the index of the maximum over the requested axis.

Examples

```
>>> idx = pd.MultiIndex.from_arrays([
...     ['warm', 'warm', 'cold', 'cold'],
...     ['dog', 'falcon', 'fish', 'spider']],
...     names=['blooded', 'animal'])
>>> s = pd.Series([4, 2, 0, 8], name='legs', index=idx)
>>> s
blooded  animal
warm     dog       4
         falcon    2
cold     fish      0
         spider    8
Name: legs, dtype: int64
```

```
>>> s.sum()
14
```

By default, the sum of an empty or all-NA Series is `0`.

```
>>> pd.Series([], dtype="float64").sum()  # min_count=0 is the default
0.0
```

This can be controlled with the `min_count` parameter. For example, if you'd like the sum of an empty series to be NaN, pass `min_count=1`.

```
>>> pd.Series([], dtype="float64").sum(min_count=1)
nan
```

Thanks to the `skipna` parameter, `min_count` handles all-NA and empty series identically.

```
>>> pd.Series([np.nan]).sum()
0.0
```

```
>>> pd.Series([np.nan]).sum(min_count=1)
nan
```

AlloViz.AlloViz.Elements.Element.swapaxes

Element.**swapaxes**(*axis1: int | Literal['index', 'columns', 'rows']*, *axis2: int | Literal['index', 'columns', 'rows']*, *copy: bool | None = None*) → None

Interchange axes and swap values axes appropriately.

Deprecated since version 2.1.0: `swapaxes` is deprecated and will be removed. Please use `transpose` instead.

Returns

same as input

Examples

Please see examples for `DataFrame.transpose()`.

AlloViz.AlloViz.Elements.Element.swaplevel

`Element.`**`swaplevel`**(*i: Axis = -2*, *j: Axis = -1*, *axis: Axis = 0*) → DataFrame

Swap levels i and j in a `MultiIndex`.

Default is to swap the two innermost levels of the index.

Parameters

i, j
: [int or str] Levels of the indices to be swapped. Can pass level name as string.

axis
: [{0 or 'index', 1 or 'columns'}, default 0] The axis to swap levels on. 0 or 'index' for row-wise, 1 or 'columns' for column-wise.

Returns

DataFrame
: DataFrame with levels swapped in MultiIndex.

Examples

```
>>> df = pd.DataFrame(
...     {"Grade": ["A", "B", "A", "C"]},
...     index=[
...         ["Final exam", "Final exam", "Coursework", "Coursework"],
...         ["History", "Geography", "History", "Geography"],
...         ["January", "February", "March", "April"],
...     ],
... )
>>> df
                                Grade
Final exam  History     January      A
            Geography   February     B
Coursework  History     March        A
            Geography   April        C
```

In the following example, we will swap the levels of the indices. Here, we will swap the levels column-wise, but levels can be swapped row-wise in a similar manner. Note that column-wise is the default behaviour. By not supplying any arguments for i and j, we swap the last and second to last indices.

```
>>> df.swaplevel()
                                Grade
Final exam  January     History      A
            February    Geography    B
Coursework  March       History      A
            April       Geography    C
```

By supplying one argument, we can choose which index to swap the last index with. We can for example swap the first index with the last one as follows.

```
>>> df.swaplevel(0)
                                  Grade
January     History     Final exam      A
February    Geography   Final exam      B
March       History     Coursework      A
April       Geography   Coursework      C
```

We can also define explicitly which indices we want to swap by supplying values for both i and j. Here, we for example swap the first and second indices.

```
>>> df.swaplevel(0, 1)
                                  Grade
History     Final exam  January         A
Geography   Final exam  February        B
History     Coursework  March           A
Geography   Coursework  April           C
```

AlloViz.AlloViz.Elements.Element.tail

`Element.tail`(*n: int = 5*) → None

Return the last *n* rows.

This function returns last *n* rows from the object based on position. It is useful for quickly verifying data, for example, after sorting or appending rows.

For negative values of *n*, this function returns all rows except the first |*n*| rows, equivalent to `df[|n|:]`.

If n is larger than the number of rows, this function returns all rows.

Parameters

n
[int, default 5] Number of rows to select.

Returns

type of caller
The last *n* rows of the caller object.

See also:

`DataFrame.head`
The first *n* rows of the caller object.

Examples

```
>>> df = pd.DataFrame({'animal': ['alligator', 'bee', 'falcon', 'lion',
...                    'monkey', 'parrot', 'shark', 'whale', 'zebra']})
>>> df
      animal
0  alligator
1        bee
2     falcon
3       lion
```

(continues on next page)

(continued from previous page)

```
4     monkey
5     parrot
6      shark
7      whale
8      zebra
```

Viewing the last 5 lines

```
>>> df.tail()
   animal
4  monkey
5  parrot
6   shark
7   whale
8   zebra
```

Viewing the last *n* lines (three in this case)

```
>>> df.tail(3)
  animal
6  shark
7  whale
8  zebra
```

For negative values of *n*

```
>>> df.tail(-3)
   animal
3    lion
4  monkey
5  parrot
6   shark
7   whale
8   zebra
```

AlloViz.AlloViz.Elements.Element.take

`Element.take`(*indices*, *axis: int | Literal['index', 'columns', 'rows'] = 0*, ***kwargs*) → None

Return the elements in the given *positional* indices along an axis.

This means that we are not indexing according to actual values in the index attribute of the object. We are indexing according to the actual position of the element in the object.

Parameters

indices
[array-like] An array of ints indicating which positions to take.

axis
[{0 or 'index', 1 or 'columns', None}, default 0] The axis on which to select elements. `0` means that we are selecting rows, `1` means that we are selecting columns. For *Series* this parameter is unused and defaults to 0.

****kwargs**
For compatibility with `numpy.take()`. Has no effect on the output.

Returns

same type as caller
An array-like containing the elements taken from the object.

See also:

`DataFrame.loc`
Select a subset of a DataFrame by labels.

`DataFrame.iloc`
Select a subset of a DataFrame by positions.

`numpy.take`
Take elements from an array along an axis.

Examples

```
>>> df = pd.DataFrame([('falcon', 'bird', 389.0),
...                    ('parrot', 'bird', 24.0),
...                    ('lion', 'mammal', 80.5),
...                    ('monkey', 'mammal', np.nan)],
...                   columns=['name', 'class', 'max_speed'],
...                   index=[0, 2, 3, 1])
>>> df
     name   class  max_speed
0  falcon    bird      389.0
2  parrot    bird       24.0
3    lion  mammal       80.5
1  monkey  mammal        NaN
```

Take elements at positions 0 and 3 along the axis 0 (default).

Note how the actual indices selected (0 and 1) do not correspond to our selected indices 0 and 3. That's because we are selecting the 0th and 3rd rows, not rows whose indices equal 0 and 3.

```
>>> df.take([0, 3])
     name   class  max_speed
0  falcon    bird      389.0
1  monkey  mammal        NaN
```

Take elements at indices 1 and 2 along the axis 1 (column selection).

```
>>> df.take([1, 2], axis=1)
    class  max_speed
0    bird      389.0
2    bird       24.0
3  mammal       80.5
1  mammal        NaN
```

We may take elements using negative integers for positive indices, starting from the end of the object, just like with Python lists.

```
>>> df.take([-1, -2])
     name   class  max_speed
1  monkey  mammal        NaN
3    lion  mammal       80.5
```

AlloViz.AlloViz.Elements.Element.to_clipboard

Element.**to_clipboard**(*excel: bool = True*, *sep: str | None = None*, ***kwargs*) → None

Copy object to the system clipboard.

Write a text representation of object to the system clipboard. This can be pasted into Excel, for example.

Parameters

excel
[bool, default True] Produce output in a csv format for easy pasting into excel.

- True, use the provided separator for csv pasting.
- False, write a string representation of the object to the clipboard.

sep
[str, default `'\t'`] Field delimiter.

****kwargs**
These parameters will be passed to DataFrame.to_csv.

See also:

DataFrame.to_csv
Write a DataFrame to a comma-separated values (csv) file.

read_clipboard
Read text from clipboard and pass to read_csv.

Notes

Requirements for your platform.

- Linux : *xclip*, or *xsel* (with *PyQt4* modules)
- Windows : none
- macOS : none

This method uses the processes developed for the package *pyperclip*. A solution to render any output string format is given in the examples.

Examples

Copy the contents of a DataFrame to the clipboard.

```
>>> df = pd.DataFrame([[1, 2, 3], [4, 5, 6]], columns=['A', 'B', 'C'])
```

```
>>> df.to_clipboard(sep=',')
... # Wrote the following to the system clipboard:
... # ,A,B,C
... # 0,1,2,3
... # 1,4,5,6
```

We can omit the index by passing the keyword *index* and setting it to false.

```
>>> df.to_clipboard(sep=',', index=False)
... # Wrote the following to the system clipboard:
... # A,B,C
... # 1,2,3
... # 4,5,6
```

Using the original *pyperclip* package for any string output format.

```
import pyperclip
html = df.style.to_html()
pyperclip.copy(html)
```

AlloViz.AlloViz.Elements.Element.to_csv

Element.**to_csv**(*path_or_buf: FilePath | WriteBuffer[bytes] | WriteBuffer[str] | None = None*, *sep: str = ','*, *na_rep: str = ''*, *float_format: str | Callable | None = None*, *columns: Sequence[Hashable] | None = None*, *header: bool_t | list[str] = True*, *index: bool_t = True*, *index_label: IndexLabel | None = None*, *mode: str = 'w'*, *encoding: str | None = None*, *compression: CompressionOptions = 'infer'*, *quoting: int | None = None*, *quotechar: str = '"'*, *lineterminator: str | None = None*, *chunksize: int | None = None*, *date_format: str | None = None*, *doublequote: bool_t = True*, *escapechar: str | None = None*, *decimal: str = '.'*, *errors: OpenFileErrors = 'strict'*, *storage_options: StorageOptions | None = None*) → str | None

Write object to a comma-separated values (csv) file.

Parameters

path_or_buf
: [str, path object, file-like object, or None, default None] String, path object (implementing os.PathLike[str]), or file-like object implementing a write() function. If None, the result is returned as a string. If a non-binary file object is passed, it should be opened with *newline=''*, disabling universal newlines. If a binary file object is passed, *mode* might need to contain a *'b'*.

  Changed in version 1.2.0: Support for binary file objects was introduced.

sep
: [str, default ','] String of length 1. Field delimiter for the output file.

na_rep
: [str, default ''] Missing data representation.

float_format
[str, Callable, default None] Format string for floating point numbers. If a Callable is given, it takes precedence over other numeric formatting parameters, like decimal.

columns
[sequence, optional] Columns to write.

header
[bool or list of str, default True] Write out the column names. If a list of strings is given it is assumed to be aliases for the column names.

index
[bool, default True] Write row names (index).

index_label
[str or sequence, or False, default None] Column label for index column(s) if desired. If None is given, and *header* and *index* are True, then the index names are used. A sequence should be given if the object uses MultiIndex. If False do not print fields for index names. Use index_label=False for easier importing in R.

mode
[{'w', 'x', 'a'}, default 'w'] Forwarded to either *open(mode=)* or *fsspec.open(mode=)* to control the file opening. Typical values include:

- 'w', truncate the file first.
- 'x', exclusive creation, failing if the file already exists.
- 'a', append to the end of file if it exists.

encoding
[str, optional] A string representing the encoding to use in the output file, defaults to 'utf-8'. *encoding* is not supported if *path_or_buf* is a non-binary file object.

compression
[str or dict, default 'infer'] For on-the-fly compression of the output data. If 'infer' and 'path_or_buf' is path-like, then detect compression from the following extensions: '.gz', '.bz2', '.zip', '.xz', '.zst', '.tar', '.tar.gz', '.tar.xz' or '.tar.bz2' (otherwise no compression). Set to `None` for no compression. Can also be a dict with key `'method'` set to one of {`'zip'`, `'gzip'`, `'bz2'`, `'zstd'`, `'xz'`, `'tar'`} and other key-value pairs are forwarded to `zipfile.ZipFile`, `gzip.GzipFile`, `bz2.BZ2File`, `zstandard.ZstdCompressor`, `lzma.LZMAFile` or `tarfile.TarFile`, respectively. As an example, the following could be passed for faster compression and to create a reproducible gzip archive: `compression={'method': 'gzip', 'compresslevel': 1, 'mtime': 1}`.

New in version 1.5.0: Added support for *.tar* files.

May be a dict with key 'method' as compression mode and other entries as additional compression options if compression mode is 'zip'.

Passing compression options as keys in dict is supported for compression modes 'gzip', 'bz2', 'zstd', and 'zip'.

Changed in version 1.2.0: Compression is supported for binary file objects.

Changed in version 1.2.0: Previous versions forwarded dict entries for 'gzip' to *gzip.open* instead of *gzip.GzipFile* which prevented setting *mtime*.

quoting
[optional constant from csv module] Defaults to csv.QUOTE_MINIMAL. If

you have set a *float_format* then floats are converted to strings and thus csv.QUOTE_NONNUMERIC will treat them as non-numeric.

quotechar
[str, default '"'] String of length 1. Character used to quote fields.

lineterminator
[str, optional] The newline character or character sequence to use in the output file. Defaults to *os.linesep*, which depends on the OS in which this method is called ('\n' for linux, '\r\n' for Windows, i.e.).

Changed in version 1.5.0: Previously was line_terminator, changed for consistency with read_csv and the standard library 'csv' module.

chunksize
[int or None] Rows to write at a time.

date_format
[str, default None] Format string for datetime objects.

doublequote
[bool, default True] Control quoting of *quotechar* inside a field.

escapechar
[str, default None] String of length 1. Character used to escape *sep* and *quotechar* when appropriate.

decimal
[str, default '.'] Character recognized as decimal separator. E.g. use ',' for European data.

errors
[str, default 'strict'] Specifies how encoding and decoding errors are to be handled. See the errors argument for `open()` for a full list of options.

storage_options
[dict, optional] Extra options that make sense for a particular storage connection, e.g. host, port, username, password, etc. For HTTP(S) URLs the key-value pairs are forwarded to `urllib.request.Request` as header options. For other URLs (e.g. starting with "s3://", and "gcs://") the key-value pairs are forwarded to `fsspec.open`. Please see `fsspec` and `urllib` for more details, and for more examples on storage options refer here.

New in version 1.2.0.

Returns

None or str
If path_or_buf is None, returns the resulting csv format as a string. Otherwise returns None.

See also:

`read_csv`
Load a CSV file into a DataFrame.

`to_excel`
Write DataFrame to an Excel file.

Examples

```
>>> df = pd.DataFrame({'name': ['Raphael', 'Donatello'],
...                    'mask': ['red', 'purple'],
...                    'weapon': ['sai', 'bo staff']})
>>> df.to_csv(index=False)
'name,mask,weapon\nRaphael,red,sai\nDonatello,purple,bo staff\n'
```

Create 'out.zip' containing 'out.csv'

```
>>> compression_opts = dict(method='zip',
...                         archive_name='out.csv')
>>> df.to_csv('out.zip', index=False,
...           compression=compression_opts)
```

To write a csv file to a new folder or nested folder you will first need to create it using either Pathlib or os:

```
>>> from pathlib import Path
>>> filepath = Path('folder/subfolder/out.csv')
>>> filepath.parent.mkdir(parents=True, exist_ok=True)
>>> df.to_csv(filepath)
```

```
>>> import os
>>> os.makedirs('folder/subfolder', exist_ok=True)
>>> df.to_csv('folder/subfolder/out.csv')
```

AlloViz.AlloViz.Elements.Element.to_dict

Element.**to_dict**(*orient: ~typing.Literal['dict', 'list', 'series', 'split', 'tight', 'records', 'index'] = 'dict'*, *into: type[dict] = <class 'dict'>*, *index: bool = True*) → dict | list[dict]

Convert the DataFrame to a dictionary.

The type of the key-value pairs can be customized with the parameters (see below).

Parameters

orient

[str {'dict', 'list', 'series', 'split', 'tight', 'records', 'index'}] Determines the type of the values of the dictionary.

- 'dict' (default) : dict like {column -> {index -> value}}
- 'list' : dict like {column -> [values]}
- 'series' : dict like {column -> Series(values)}
- 'split' : dict like {'index' -> [index], 'columns' -> [columns], 'data' -> [values]}
- 'tight' : dict like {'index' -> [index], 'columns' -> [columns], 'data' -> [values], 'index_names' -> [index.names], 'column_names' -> [column.names]}
- 'records' : list like [{column -> value}, … , {column -> value}]
- 'index' : dict like {index -> {column -> value}}

New in version 1.4.0: 'tight' as an allowed value for the `orient` argument

into
: [class, default dict] The collections.abc.Mapping subclass used for all Mappings in the return value. Can be the actual class or an empty instance of the mapping type you want. If you want a collections.defaultdict, you must pass it initialized.

index
: [bool, default True] Whether to include the index item (and index_names item if *orient* is 'tight') in the returned dictionary. Can only be `False` when *orient* is 'split' or 'tight'.

  New in version 2.0.0.

Returns

dict, list or collections.abc.Mapping
: Return a collections.abc.Mapping object representing the DataFrame. The resulting transformation depends on the *orient* parameter.

See also:

`DataFrame.from_dict`
: Create a DataFrame from a dictionary.

`DataFrame.to_json`
: Convert a DataFrame to JSON format.

Examples

```
>>> df = pd.DataFrame({'col1': [1, 2],
...                    'col2': [0.5, 0.75]},
...                   index=['row1', 'row2'])
>>> df
      col1  col2
row1     1  0.50
row2     2  0.75
>>> df.to_dict()
{'col1': {'row1': 1, 'row2': 2}, 'col2': {'row1': 0.5, 'row2': 0.75}}
```

You can specify the return orientation.

```
>>> df.to_dict('series')
{'col1': row1    1
         row2    2
Name: col1, dtype: int64,
'col2': row1    0.50
        row2    0.75
Name: col2, dtype: float64}
```

```
>>> df.to_dict('split')
{'index': ['row1', 'row2'], 'columns': ['col1', 'col2'],
 'data': [[1, 0.5], [2, 0.75]]}
```

```
>>> df.to_dict('records')
[{'col1': 1, 'col2': 0.5}, {'col1': 2, 'col2': 0.75}]
```

```
>>> df.to_dict('index')
{'row1': {'col1': 1, 'col2': 0.5}, 'row2': {'col1': 2, 'col2': 0.75}}
```

```
>>> df.to_dict('tight')
{'index': ['row1', 'row2'], 'columns': ['col1', 'col2'],
 'data': [[1, 0.5], [2, 0.75]], 'index_names': [None], 'column_names': [None]}
```

You can also specify the mapping type.

```
>>> from collections import OrderedDict, defaultdict
>>> df.to_dict(into=OrderedDict)
OrderedDict([('col1', OrderedDict([('row1', 1), ('row2', 2)])),
             ('col2', OrderedDict([('row1', 0.5), ('row2', 0.75)]))])
```

If you want a *defaultdict*, you need to initialize it:

```
>>> dd = defaultdict(list)
>>> df.to_dict('records', into=dd)
[defaultdict(<class 'list'>, {'col1': 1, 'col2': 0.5}),
 defaultdict(<class 'list'>, {'col1': 2, 'col2': 0.75})]
```

AlloViz.AlloViz.Elements.Element.to_excel

Element.**to_excel**(*excel_writer: FilePath | WriteExcelBuffer | ExcelWriter*, *sheet_name: str = 'Sheet1'*, *na_rep: str = ''*, *float_format: str | None = None*, *columns: Sequence[Hashable] | None = None*, *header: Sequence[Hashable] | bool_t = True*, *index: bool_t = True*, *index_label: IndexLabel | None = None*, *startrow: int = 0*, *startcol: int = 0*, *engine: Literal['openpyxl', 'xlsxwriter'] | None = None*, *merge_cells: bool_t = True*, *inf_rep: str = 'inf'*, *freeze_panes: tuple[int, int] | None = None*, *storage_options: StorageOptions | None = None*, *engine_kwargs: dict[str, Any] | None = None*) → None

Write object to an Excel sheet.

To write a single object to an Excel .xlsx file it is only necessary to specify a target file name. To write to multiple sheets it is necessary to create an *ExcelWriter* object with a target file name, and specify a sheet in the file to write to.

Multiple sheets may be written to by specifying unique *sheet_name*. With all data written to the file it is necessary to save the changes. Note that creating an *ExcelWriter* object with a file name that already exists will result in the contents of the existing file being erased.

Parameters

excel_writer
[path-like, file-like, or ExcelWriter object] File path or existing ExcelWriter.

sheet_name
[str, default 'Sheet1'] Name of sheet which will contain DataFrame.

na_rep
[str, default ''] Missing data representation.

float_format
[str, optional] Format string for floating point numbers. For example `float_format="%.2f"` will format 0.1234 to 0.12.

columns
[sequence or list of str, optional] Columns to write.

header
[bool or list of str, default True] Write out the column names. If a list of string is given it is assumed to be aliases for the column names.

index
[bool, default True] Write row names (index).

index_label
[str or sequence, optional] Column label for index column(s) if desired. If not specified, and *header* and *index* are True, then the index names are used. A sequence should be given if the DataFrame uses MultiIndex.

startrow
[int, default 0] Upper left cell row to dump data frame.

startcol
[int, default 0] Upper left cell column to dump data frame.

engine
[str, optional] Write engine to use, 'openpyxl' or 'xlsxwriter'. You can also set this via the options `io.excel.xlsx.writer` or `io.excel.xlsm.writer`.

merge_cells
[bool, default True] Write MultiIndex and Hierarchical Rows as merged cells.

inf_rep
[str, default 'inf'] Representation for infinity (there is no native representation for infinity in Excel).

freeze_panes
[tuple of int (length 2), optional] Specifies the one-based bottommost row and rightmost column that is to be frozen.

storage_options
[dict, optional] Extra options that make sense for a particular storage connection, e.g. host, port, username, password, etc. For HTTP(S) URLs the key-value pairs are forwarded to `urllib.request.Request` as header options. For other URLs (e.g. starting with "s3://", and "gcs://") the key-value pairs are forwarded to `fsspec.open`. Please see `fsspec` and `urllib` for more details, and for more examples on storage options refer here.

New in version 1.2.0.

engine_kwargs
[dict, optional] Arbitrary keyword arguments passed to excel engine.

See also:

to_csv
Write DataFrame to a comma-separated values (csv) file.

`ExcelWriter`
Class for writing DataFrame objects into excel sheets.

`read_excel`
Read an Excel file into a pandas DataFrame.

`read_csv`
Read a comma-separated values (csv) file into DataFrame.

io.formats.style.Styler.to_excel
Add styles to Excel sheet.

Notes

For compatibility with `to_csv()`, to_excel serializes lists and dicts to strings before writing.

Once a workbook has been saved it is not possible to write further data without rewriting the whole workbook.

Examples

Create, write to and save a workbook:

```
>>> df1 = pd.DataFrame([['a', 'b'], ['c', 'd']],
...                    index=['row 1', 'row 2'],
...                    columns=['col 1', 'col 2'])
>>> df1.to_excel("output.xlsx")
```

To specify the sheet name:

```
>>> df1.to_excel("output.xlsx",
...              sheet_name='Sheet_name_1')
```

If you wish to write to more than one sheet in the workbook, it is necessary to specify an ExcelWriter object:

```
>>> df2 = df1.copy()
>>> with pd.ExcelWriter('output.xlsx') as writer:
...     df1.to_excel(writer, sheet_name='Sheet_name_1')
...     df2.to_excel(writer, sheet_name='Sheet_name_2')
```

ExcelWriter can also be used to append to an existing Excel file:

```
>>> with pd.ExcelWriter('output.xlsx',
...                     mode='a') as writer:
...     df1.to_excel(writer, sheet_name='Sheet_name_3')
```

To set the library that is used to write the Excel file, you can pass the *engine* keyword (the default engine is automatically chosen depending on the file extension):

```
>>> df1.to_excel('output1.xlsx', engine='xlsxwriter')
```

AlloViz.AlloViz.Elements.Element.to_feather

`Element.`**to_feather**(*path: FilePath | WriteBuffer[bytes]*, ***kwargs*) → None

Write a DataFrame to the binary Feather format.

Parameters

path
[str, path object, file-like object] String, path object (implementing `os.PathLike[str]`), or file-like object implementing a binary `write()` function. If

a string or a path, it will be used as Root Directory path when writing a partitioned dataset.

****kwargs**
Additional keywords passed to `pyarrow.feather.write_feather()`. This includes the *compression*, *compression_level*, *chunksize* and *version* keywords.

Notes

This function writes the dataframe as a feather file. Requires a default index. For saving the DataFrame with your custom index use a method that supports custom indices e.g. *to_parquet*.

Examples

```
>>> df = pd.DataFrame([[1, 2, 3], [4, 5, 6]])
>>> df.to_feather("file.feather")
```

AlloViz.AlloViz.Elements.Element.to_gbq

`Element.to_gbq`(*destination_table: str*, *project_id: str | None = None*, *chunksize: int | None = None*, *reauth: bool = False*, *if_exists: ToGbqIfexist = 'fail'*, *auth_local_webserver: bool = True*, *table_schema: list[dict[str, str]] | None = None*, *location: str | None = None*, *progress_bar: bool = True*, *credentials=None*) → None

Write a DataFrame to a Google BigQuery table.

This function requires the pandas-gbq package.

See the How to authenticate with Google BigQuery guide for authentication instructions.

Parameters

destination_table
[str] Name of table to be written, in the form `dataset.tablename`.

project_id
[str, optional] Google BigQuery Account project ID. Optional when available from the environment.

chunksize
[int, optional] Number of rows to be inserted in each chunk from the dataframe. Set to `None` to load the whole dataframe at once.

reauth
[bool, default False] Force Google BigQuery to re-authenticate the user. This is useful if multiple accounts are used.

if_exists
[str, default 'fail'] Behavior when the destination table exists. Value can be one of:

`'fail'`
If table exists raise pandas_gbq.gbq.TableCreationError.

`'replace'`
If table exists, drop it, recreate it, and insert data.

`'append'`
If table exists, insert data. Create if does not exist.

auth_local_webserver
: [bool, default True] Use the local webserver flow instead of the console flow when getting user credentials.

New in version 0.2.0 of pandas-gbq.

Changed in version 1.5.0: Default value is changed to `True`. Google has deprecated the `auth_local_webserver = False` "out of band" (copy-paste) flow.

table_schema
: [list of dicts, optional] List of BigQuery table fields to which according DataFrame columns conform to, e.g. `[{'name': 'col1', 'type': 'STRING'},...]`. If schema is not provided, it will be generated according to dtypes of DataFrame columns. See BigQuery API documentation on available names of a field.

New in version 0.3.1 of pandas-gbq.

location
: [str, optional] Location where the load job should run. See the BigQuery locations documentation for a list of available locations. The location must match that of the target dataset.

New in version 0.5.0 of pandas-gbq.

progress_bar
: [bool, default True] Use the library *tqdm* to show the progress bar for the upload, chunk by chunk.

New in version 0.5.0 of pandas-gbq.

credentials
: [google.auth.credentials.Credentials, optional] Credentials for accessing Google APIs. Use this parameter to override default credentials, such as to use Compute Engine `google.auth.compute_engine.Credentials` or Service Account `google.oauth2.service_account.Credentials` directly.

New in version 0.8.0 of pandas-gbq.

See also:

`pandas_gbq.to_gbq`
: This function in the pandas-gbq library.

`read_gbq`
: Read a DataFrame from Google BigQuery.

Examples

Example taken from Google BigQuery documentation

```
>>> project_id = "my-project"
>>> table_id = 'my_dataset.my_table'
>>> df = pd.DataFrame({
...                    "my_string": ["a", "b", "c"],
...                    "my_int64": [1, 2, 3],
...                    "my_float64": [4.0, 5.0, 6.0],
...                    "my_bool1": [True, False, True],
...                    "my_bool2": [False, True, False],
...                    "my_dates": pd.date_range("now", periods=3),
```

(continues on next page)

(continued from previous page)

```
...                     }
...                     )
```

```
>>> df.to_gbq(table_id, project_id=project_id)
```

AlloViz.AlloViz.Elements.Element.to_hdf

Element.**to_hdf**(*path_or_buf: FilePath | HDFStore*, *key: str*, *mode: Literal['a', 'w', 'r+'] = 'a'*, *complevel: int | None = None*, *complib: Literal['zlib', 'lzo', 'bzip2', 'blosc'] | None = None*, *append: bool_t = False*, *format: Literal['fixed', 'table'] | None = None*, *index: bool_t = True*, *min_itemsize: int | dict[str, int] | None = None*, *nan_rep=None*, *dropna: bool_t | None = None*, *data_columns: Literal[True] | list[str] | None = None*, *errors: OpenFileErrors = 'strict'*, *encoding: str = 'UTF-8'*) → None

Write the contained data to an HDF5 file using HDFStore.

Hierarchical Data Format (HDF) is self-describing, allowing an application to interpret the structure and contents of a file with no outside information. One HDF file can hold a mix of related objects which can be accessed as a group or as individual objects.

In order to add another DataFrame or Series to an existing HDF file please use append mode and a different a key.

Warning: One can store a subclass of `DataFrame` or `Series` to HDF5, but the type of the subclass is lost upon storing.

For more information see the user guide.

Parameters

path_or_buf
[str or pandas.HDFStore] File path or HDFStore object.

key
[str] Identifier for the group in the store.

mode
[{'a', 'w', 'r+'}, default 'a'] Mode to open file:

- 'w': write, a new file is created (an existing file with the same name would be deleted).
- 'a': append, an existing file is opened for reading and writing, and if the file does not exist it is created.
- 'r+': similar to 'a', but the file must already exist.

complevel
[{0-9}, default None] Specifies a compression level for data. A value of 0 or None disables compression.

complib
[{'zlib', 'lzo', 'bzip2', 'blosc'}, default 'zlib'] Specifies the compression library to be used. These additional compressors for Blosc are supported (default if no compressor specified: 'blosc:blosclz'): {'blosc:blosclz', 'blosc:lz4', 'blosc:lz4hc',

'blosc:snappy', 'blosc:zlib', 'blosc:zstd'}. Specifying a compression library which is not available issues a ValueError.

append
[bool, default False] For Table formats, append the input data to the existing.

format
[{'fixed', 'table', None}, default 'fixed'] Possible values:

- 'fixed': Fixed format. Fast writing/reading. Not-appendable, nor searchable.
- 'table': Table format. Write as a PyTables Table structure which may perform worse but allow more flexible operations like searching / selecting subsets of the data.
- If None, pd.get_option('io.hdf.default_format') is checked, followed by fallback to "fixed".

index
[bool, default True] Write DataFrame index as a column.

min_itemsize
[dict or int, optional] Map column names to minimum string sizes for columns.

nan_rep
[Any, optional] How to represent null values as str. Not allowed with append=True.

dropna
[bool, default False, optional] Remove missing values.

data_columns
[list of columns or True, optional] List of columns to create as indexed data columns for on-disk queries, or True to use all columns. By default only the axes of the object are indexed. See Query via data columns. for more information. Applicable only to format='table'.

errors
[str, default 'strict'] Specifies how encoding and decoding errors are to be handled. See the errors argument for `open()` for a full list of options.

encoding
[str, default "UTF-8"]

See also:

`read_hdf`
Read from HDF file.

`DataFrame.to_orc`
Write a DataFrame to the binary orc format.

`DataFrame.to_parquet`
Write a DataFrame to the binary parquet format.

`DataFrame.to_sql`
Write to a SQL table.

`DataFrame.to_feather`
Write out feather-format for DataFrames.

`DataFrame.to_csv`
Write out to a csv file.

Examples

```
>>> df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]},
...                   index=['a', 'b', 'c'])
>>> df.to_hdf('data.h5', key='df', mode='w')
```

We can add another object to the same file:

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s.to_hdf('data.h5', key='s')
```

Reading from HDF file:

```
>>> pd.read_hdf('data.h5', 'df')
A  B
a  1  4
b  2  5
c  3  6
>>> pd.read_hdf('data.h5', 's')
0    1
1    2
2    3
3    4
dtype: int64
```

AlloViz.AlloViz.Elements.Element.to_html

Element.**to_html**(*buf: FilePath | WriteBuffer[str] | None = None*, *columns: Axes | None = None*, *col_space: ColspaceArgType | None = None*, *header: bool = True*, *index: bool = True*, *na_rep: str = 'NaN'*, *formatters: FormattersType | None = None*, *float_format: FloatFormatType | None = None*, *sparsify: bool | None = None*, *index_names: bool = True*, *justify: str | None = None*, *max_rows: int | None = None*, *max_cols: int | None = None*, *show_dimensions: bool | str = False*, *decimal: str = '.'*, *bold_rows: bool = True*, *classes: str | list | tuple | None = None*, *escape: bool = True*, *notebook: bool = False*, *border: int | bool | None = None*, *table_id: str | None = None*, *render_links: bool = False*, *encoding: str | None = None*) → str | None

Render a DataFrame as an HTML table.

Parameters

buf
: [str, Path or StringIO-like, optional, default None] Buffer to write to. If None, the output is returned as a string.

columns
: [array-like, optional, default None] The subset of columns to write. Writes all columns by default.

col_space
: [str or int, list or dict of int or str, optional] The minimum width of each column in CSS length units. An int is assumed to be px units..

header
: [bool, optional] Whether to print column labels, default True.

index
[bool, optional, default True] Whether to print index (row) labels.

na_rep
[str, optional, default 'NaN'] String representation of `NaN` to use.

formatters
[list, tuple or dict of one-param. functions, optional] Formatter functions to apply to columns' elements by position or name. The result of each function must be a unicode string. List/tuple must be of length equal to the number of columns.

float_format
[one-parameter function, optional, default None] Formatter function to apply to columns' elements if they are floats. This function must return a unicode string and will be applied only to the non-`NaN` elements, with `NaN` being handled by `na_rep`.

Changed in version 1.2.0.

sparsify
[bool, optional, default True] Set to False for a DataFrame with a hierarchical index to print every multiindex key at each row.

index_names
[bool, optional, default True] Prints the names of the indexes.

justify
[str, default None] How to justify the column labels. If None uses the option from the print configuration (controlled by set_option), 'right' out of the box. Valid values are

- left
- right
- center
- justify
- justify-all
- start
- end
- inherit
- match-parent
- initial
- unset.

max_rows
[int, optional] Maximum number of rows to display in the console.

max_cols
[int, optional] Maximum number of columns to display in the console.

show_dimensions
[bool, default False] Display DataFrame dimensions (number of rows by number of columns).

decimal
[str, default '.'] Character recognized as decimal separator, e.g. ',' in Europe.

bold_rows
[bool, default True] Make the row labels bold in the output.

classes
[str or list or tuple, default None] CSS class(es) to apply to the resulting html table.

escape
[bool, default True] Convert the characters <, >, and & to HTML-safe sequences.

notebook
[{True, False}, default False] Whether the generated HTML is for IPython Notebook.

border
[int] A `border=border` attribute is included in the opening *<table>* tag. Default `pd.options.display.html.border`.

table_id
[str, optional] A css id is included in the opening *<table>* tag if specified.

render_links
[bool, default False] Convert URLs to HTML links.

encoding
[str, default "utf-8"] Set character encoding.

Returns

str or None
If buf is None, returns the result as a string. Otherwise returns None.

See also:

to_string
Convert DataFrame to a string.

Examples

```
>>> df = pd.DataFrame(data={'col1': [1, 2], 'col2': [4, 3]})
>>> html_string = '''<table border="1" class="dataframe">
...   <thead>
...     <tr style="text-align: right;">
...       <th></th>
...       <th>col1</th>
...       <th>col2</th>
...     </tr>
...   </thead>
...   <tbody>
...     <tr>
...       <th>0</th>
...       <td>1</td>
...       <td>4</td>
...     </tr>
...     <tr>
...       <th>1</th>
...       <td>2</td>
...       <td>3</td>
```

(continues on next page)

(continued from previous page)

```
...     </tr>
...   </tbody>
... </table>'''
>>> assert html_string == df.to_html()
```

AlloViz.AlloViz.Elements.Element.to_json

Element.**to_json**(*path_or_buf: FilePath | WriteBuffer[bytes] | WriteBuffer[str] | None = None*, *orient: Literal['split', 'records', 'index', 'table', 'columns', 'values'] | None = None*, *date_format: str | None = None*, *double_precision: int = 10*, *force_ascii: bool_t = True*, *date_unit: TimeUnit = 'ms'*, *default_handler: Callable[[Any], JSONSerializable] | None = None*, *lines: bool_t = False*, *compression: CompressionOptions = 'infer'*, *index: bool_t | None = None*, *indent: int | None = None*, *storage_options: StorageOptions | None = None*, *mode: Literal['a', 'w'] = 'w'*) → str | None

Convert the object to a JSON string.

Note NaN's and None will be converted to null and datetime objects will be converted to UNIX timestamps.

Parameters

path_or_buf
[str, path object, file-like object, or None, default None] String, path object (implementing os.PathLike[str]), or file-like object implementing a write() function. If None, the result is returned as a string.

orient
[str] Indication of expected JSON string format.

- Series:
  - default is 'index'
  - allowed values are: {'split', 'records', 'index', 'table'}.
- DataFrame:
  - default is 'columns'
  - allowed values are: {'split', 'records', 'index', 'columns', 'values', 'table'}.
- The format of the JSON string:
  - 'split' : dict like {'index' -> [index], 'columns' -> [columns], 'data' -> [values]}
  - 'records' : list like [{column -> value}, … , {column -> value}]
  - 'index' : dict like {index -> {column -> value}}
  - 'columns' : dict like {column -> {index -> value}}
  - 'values' : just the values array
  - 'table' : dict like {'schema': {schema}, 'data': {data}}

  Describing the data, where data component is like `orient='records'`.

date_format
[{None, 'epoch', 'iso'}] Type of date conversion. 'epoch' = epoch milliseconds,

'iso' = ISO8601. The default depends on the *orient*. For `orient='table'`, the default is 'iso'. For all other orients, the default is 'epoch'.

double_precision
[int, default 10] The number of decimal places to use when encoding floating point values. The possible maximal value is 15. Passing double_precision greater than 15 will raise a ValueError.

force_ascii
[bool, default True] Force encoded string to be ASCII.

date_unit
[str, default 'ms' (milliseconds)] The time unit to encode to, governs timestamp and ISO8601 precision. One of 's', 'ms', 'us', 'ns' for second, millisecond, microsecond, and nanosecond respectively.

default_handler
[callable, default None] Handler to call if object cannot otherwise be converted to a suitable format for JSON. Should receive a single argument which is the object to convert and return a serialisable object.

lines
[bool, default False] If 'orient' is 'records' write out line-delimited json format. Will throw ValueError if incorrect 'orient' since others are not list-like.

compression
[str or dict, default 'infer'] For on-the-fly compression of the output data. If 'infer' and 'path_or_buf' is path-like, then detect compression from the following extensions: '.gz', '.bz2', '.zip', '.xz', '.zst', '.tar', '.tar.gz', '.tar.xz' or '.tar.bz2' (otherwise no compression). Set to `None` for no compression. Can also be a dict with key `'method'` set to one of {`'zip'`, `'gzip'`, `'bz2'`, `'zstd'`, `'xz'`, `'tar'`} and other key-value pairs are forwarded to `zipfile.ZipFile`, `gzip.GzipFile`, `bz2.BZ2File`, `zstandard.ZstdCompressor`, `lzma.LZMAFile` or `tarfile.TarFile`, respectively. As an example, the following could be passed for faster compression and to create a reproducible gzip archive: `compression={'method': 'gzip', 'compresslevel': 1, 'mtime': 1}`.

New in version 1.5.0: Added support for *.tar* files.

Changed in version 1.4.0: Zstandard support.

index
[bool or None, default None] The index is only used when 'orient' is 'split', 'index', 'column', or 'table'. Of these, 'index' and 'column' do not support *index=False*.

indent
[int, optional] Length of whitespace used to indent each record.

storage_options
[dict, optional] Extra options that make sense for a particular storage connection, e.g. host, port, username, password, etc. For HTTP(S) URLs the key-value pairs are forwarded to `urllib.request.Request` as header options. For other URLs (e.g. starting with "s3://", and "gcs://") the key-value pairs are forwarded to `fsspec.open`. Please see `fsspec` and `urllib` for more details, and for more examples on storage options refer here.

New in version 1.2.0.

mode
[str, default 'w' (writing)] Specify the IO mode for output when supplying a

path_or_buf. Accepted args are 'w' (writing) and 'a' (append) only. mode='a' is only supported when lines is True and orient is 'records'.

Returns

None or str
If path_or_buf is None, returns the resulting json format as a string. Otherwise returns None.

See also:

read_json
Convert a JSON string to pandas object.

Notes

The behavior of `indent=0` varies from the stdlib, which does not indent the output but does insert newlines. Currently, `indent=0` and the default `indent=None` are equivalent in pandas, though this may change in a future release.

`orient='table'` contains a 'pandas_version' field under 'schema'. This stores the version of *pandas* used in the latest revision of the schema.

Examples

```
>>> from json import loads, dumps
>>> df = pd.DataFrame(
...     [["a", "b"], ["c", "d"]],
...     index=["row 1", "row 2"],
...     columns=["col 1", "col 2"],
... )
```

```
>>> result = df.to_json(orient="split")
>>> parsed = loads(result)
>>> dumps(parsed, indent=4)
{
    "columns": [
        "col 1",
        "col 2"
    ],
    "index": [
        "row 1",
        "row 2"
    ],
    "data": [
        [
            "a",
            "b"
        ],
        [
            "c",
            "d"
        ]
```

(continues on next page)

(continued from previous page)

```
    ]
}
```

Encoding/decoding a Dataframe using `'records'` formatted JSON. Note that index labels are not preserved with this encoding.

```
>>> result = df.to_json(orient="records")
>>> parsed = loads(result)
>>> dumps(parsed, indent=4)
[
    {
        "col 1": "a",
        "col 2": "b"
    },
    {
        "col 1": "c",
        "col 2": "d"
    }
]
```

Encoding/decoding a Dataframe using `'index'` formatted JSON:

```
>>> result = df.to_json(orient="index")
>>> parsed = loads(result)
>>> dumps(parsed, indent=4)
{
    "row 1": {
        "col 1": "a",
        "col 2": "b"
    },
    "row 2": {
        "col 1": "c",
        "col 2": "d"
    }
}
```

Encoding/decoding a Dataframe using `'columns'` formatted JSON:

```
>>> result = df.to_json(orient="columns")
>>> parsed = loads(result)
>>> dumps(parsed, indent=4)
{
    "col 1": {
        "row 1": "a",
        "row 2": "c"
    },
    "col 2": {
        "row 1": "b",
        "row 2": "d"
    }
}
```

Encoding/decoding a Dataframe using `'values'` formatted JSON:

```
>>> result = df.to_json(orient="values")
>>> parsed = loads(result)
>>> dumps(parsed, indent=4)
[
    [
        "a",
        "b"
    ],
    [
        "c",
        "d"
    ]
]
```

Encoding with Table Schema:

```
>>> result = df.to_json(orient="table")
>>> parsed = loads(result)
>>> dumps(parsed, indent=4)
{
    "schema": {
        "fields": [
            {
                "name": "index",
                "type": "string"
            },
            {
                "name": "col 1",
                "type": "string"
            },
            {
                "name": "col 2",
                "type": "string"
            }
        ],
        "primaryKey": [
            "index"
        ],
        "pandas_version": "1.4.0"
    },
    "data": [
        {
            "index": "row 1",
            "col 1": "a",
            "col 2": "b"
        },
        {
            "index": "row 2",
            "col 1": "c",
            "col 2": "d"
        }
    ]
}
```

AlloViz.AlloViz.Elements.Element.to_latex

Element.**to_latex**(*buf: FilePath | WriteBuffer[str] | None = None*, *columns: Sequence[Hashable] | None = None*, *header: bool_t | list[str] = True*, *index: bool_t = True*, *na_rep: str = 'NaN'*, *formatters: FormattersType | None = None*, *float_format: FloatFormatType | None = None*, *sparsify: bool_t | None = None*, *index_names: bool_t = True*, *bold_rows: bool_t = False*, *column_format: str | None = None*, *longtable: bool_t | None = None*, *escape: bool_t | None = None*, *encoding: str | None = None*, *decimal: str = '.'*, *multicolumn: bool_t | None = None*, *multicolumn_format: str | None = None*, *multirow: bool_t | None = None*, *caption: str | tuple[str, str] | None = None*, *label: str | None = None*, *position: str | None = None*) → str | None

Render object to a LaTeX tabular, longtable, or nested table.

Requires `\usepackage{{booktabs}}`. The output can be copy/pasted into a main LaTeX document or read from an external file with `\input{{table.tex}}`.

Changed in version 1.2.0: Added position argument, changed meaning of caption argument.

Changed in version 2.0.0: Refactored to use the Styler implementation via jinja2 templating.

Parameters

buf
[str, Path or StringIO-like, optional, default None] Buffer to write to. If None, the output is returned as a string.

columns
[list of label, optional] The subset of columns to write. Writes all columns by default.

header
[bool or list of str, default True] Write out the column names. If a list of strings is given, it is assumed to be aliases for the column names.

index
[bool, default True] Write row names (index).

na_rep
[str, default 'NaN'] Missing data representation.

formatters
[list of functions or dict of {{str: function}}, optional] Formatter functions to apply to columns' elements by position or name. The result of each function must be a unicode string. List must be of length equal to the number of columns.

float_format
[one-parameter function or str, optional, default None] Formatter for floating point numbers. For example `float_format="%.2f"` and `float_format="{{:0.2f}}".format` will both result in 0.1234 being formatted as 0.12.

sparsify
[bool, optional] Set to False for a DataFrame with a hierarchical index to print every multiindex key at each row. By default, the value will be read from the config module.

index_names
[bool, default True] Prints the names of the indexes.

bold_rows
[bool, default False] Make the row labels bold in the output.

column_format
[str, optional] The columns format as specified in LaTeX table format e.g. 'rcl' for 3 columns. By default, 'l' will be used for all columns except columns of numbers, which default to 'r'.

longtable
[bool, optional] Use a longtable environment instead of tabular. Requires adding a usepackage{{longtable}} to your LaTeX preamble. By default, the value will be read from the pandas config module, and set to *True* if the option `styler.latex.environment` is *"longtable"*.

Changed in version 2.0.0: The pandas option affecting this argument has changed.

escape
[bool, optional] By default, the value will be read from the pandas config module and set to *True* if the option `styler.format.escape` is *"latex"*. When set to False prevents from escaping latex special characters in column names.

Changed in version 2.0.0: The pandas option affecting this argument has changed, as has the default value to *False*.

encoding
[str, optional] A string representing the encoding to use in the output file, defaults to 'utf-8'.

decimal
[str, default '.'] Character recognized as decimal separator, e.g. ',' in Europe.

multicolumn
[bool, default True] Use multicolumn to enhance MultiIndex columns. The default will be read from the config module, and is set as the option `styler.sparse.columns`.

Changed in version 2.0.0: The pandas option affecting this argument has changed.

multicolumn_format
[str, default 'r'] The alignment for multicolumns, similar to *column_format* The default will be read from the config module, and is set as the option `styler.latex.multicol_align`.

Changed in version 2.0.0: The pandas option affecting this argument has changed, as has the default value to "r".

multirow
[bool, default True] Use multirow to enhance MultiIndex rows. Requires adding a usepackage{{multirow}} to your LaTeX preamble. Will print centered labels (instead of top-aligned) across the contained rows, separating groups via clines. The default will be read from the pandas config module, and is set as the option `styler.sparse.index`.

Changed in version 2.0.0: The pandas option affecting this argument has changed, as has the default value to *True*.

caption
[str or tuple, optional] Tuple (full_caption, short_caption), which results in \ `caption[short_caption]{{full_caption}}`; if a single string is passed, no short caption will be set.

Changed in version 1.2.0: Optionally allow caption to be a tuple `(full_caption, short_caption)`.

label
[str, optional] The LaTeX label to be placed inside `\label{{}}` in the output. This is used with `\ref{{}}` in the main `.tex` file.

position
[str, optional] The LaTeX positional argument for tables, to be placed after `\begin{{}}` in the output.

New in version 1.2.0.

Returns

str or None
If buf is None, returns the result as a string. Otherwise returns None.

See also:

`io.formats.style.Styler.to_latex`
Render a DataFrame to LaTeX with conditional formatting.

`DataFrame.to_string`
Render a DataFrame to a console-friendly tabular output.

`DataFrame.to_html`
Render a DataFrame as an HTML table.

Notes

As of v2.0.0 this method has changed to use the Styler implementation as part of `Styler.to_latex()` via `jinja2` templating. This means that `jinja2` is a requirement, and needs to be installed, for this method to function. It is advised that users switch to using Styler, since that implementation is more frequently updated and contains much more flexibility with the output.

Examples

Convert a general DataFrame to LaTeX with formatting:

```
>>> df = pd.DataFrame(dict(name=['Raphael', 'Donatello'],
...                        age=[26, 45],
...                        height=[181.23, 177.65]))
>>> print(df.to_latex(index=False,
...                   formatters={"name": str.upper},
...                   float_format="{:.1f}".format,
... ))
\begin{tabular}{lrr}
\toprule
name & age & height \\
\midrule
RAPHAEL & 26 & 181.2 \\
DONATELLO & 45 & 177.7 \\
\bottomrule
\end{tabular}
```

AlloViz.AlloViz.Elements.Element.to_markdown

Element.**to_markdown**(*buf: FilePath | WriteBuffer[str] | None = None*, *mode: str = 'wt'*, *index: bool = True*, *storage_options: StorageOptions | None = None*, ***kwargs*) → str | None

Print DataFrame in Markdown-friendly format.

Parameters

buf
[str, Path or StringIO-like, optional, default None] Buffer to write to. If None, the output is returned as a string.

mode
[str, optional] Mode in which file is opened, "wt" by default.

index
[bool, optional, default True] Add index (row) labels.

storage_options
[dict, optional] Extra options that make sense for a particular storage connection, e.g. host, port, username, password, etc. For HTTP(S) URLs the key-value pairs are forwarded to `urllib.request.Request` as header options. For other URLs (e.g. starting with "s3://", and "gcs://") the key-value pairs are forwarded to `fsspec.open`. Please see `fsspec` and `urllib` for more details, and for more examples on storage options refer here.

New in version 1.2.0.

****kwargs**
These parameters will be passed to tabulate.

Returns

str
DataFrame in Markdown-friendly format.

Notes

Requires the tabulate package.

Examples

```
>>> df = pd.DataFrame(
...     data={"animal_1": ["elk", "pig"], "animal_2": ["dog", "quetzal"]}
... )
>>> print(df.to_markdown())
|    | animal_1   | animal_2   |
|---:|:-----------|:-----------|
|  0 | elk        | dog        |
|  1 | pig        | quetzal    |
```

Output markdown with a tabulate option.

```
>>> print(df.to_markdown(tablefmt="grid"))
+----+------------+------------+
```

(continues on next page)

(continued from previous page)

```
|    | animal_1   | animal_2   |
+====+============+============+
|  0 | elk        | dog        |
+----+------------+------------+
|  1 | pig        | quetzal    |
+----+------------+------------+
```

AlloViz.AlloViz.Elements.Element.to_numpy

Element.**to_numpy**(*dtype: npt.DTypeLike | None = None*, *copy: bool = False*, *na_value: object = _NoDefault.no_default*) → np.ndarray

Convert the DataFrame to a NumPy array.

By default, the dtype of the returned array will be the common NumPy dtype of all types in the DataFrame. For example, if the dtypes are `float16` and `float32`, the results dtype will be `float32`. This may require copying data and coercing values, which may be expensive.

Parameters

dtype
: [str or numpy.dtype, optional] The dtype to pass to `numpy.asarray()`.

copy
: [bool, default False] Whether to ensure that the returned value is not a view on another array. Note that `copy=False` does not *ensure* that `to_numpy()` is no-copy. Rather, `copy=True` ensure that a copy is made, even if not strictly necessary.

na_value
: [Any, optional] The value to use for missing values. The default value depends on *dtype* and the dtypes of the DataFrame columns.

Returns

numpy.ndarray

See also:

`Series.to_numpy`
: Similar method for Series.

Examples

```
>>> pd.DataFrame({"A": [1, 2], "B": [3, 4]}).to_numpy()
array([[1, 3],
       [2, 4]])
```

With heterogeneous data, the lowest common type will have to be used.

```
>>> df = pd.DataFrame({"A": [1, 2], "B": [3.0, 4.5]})
>>> df.to_numpy()
array([[1. , 3. ],
       [2. , 4.5]])
```

For a mix of numeric and non-numeric types, the output array will have object dtype.

```
>>> df['C'] = pd.date_range('2000', periods=2)
>>> df.to_numpy()
array([[1, 3.0, Timestamp('2000-01-01 00:00:00')],
       [2, 4.5, Timestamp('2000-01-02 00:00:00')]], dtype=object)
```

AlloViz.AlloViz.Elements.Element.to_orc

Element.**to_orc**(*path: FilePath | WriteBuffer[bytes] | None = None*, ***, *engine: Literal['pyarrow'] = 'pyarrow'*, *index: bool | None = None*, *engine_kwargs: dict[str, Any] | None = None*) → bytes | None

Write a DataFrame to the ORC format.

New in version 1.5.0.

Parameters

path
[str, file-like object or None, default None] If a string, it will be used as Root Directory path when writing a partitioned dataset. By file-like object, we refer to objects with a write() method, such as a file handle (e.g. via builtin open function). If path is None, a bytes object is returned.

engine
[{'pyarrow'}, default 'pyarrow'] ORC library to use. Pyarrow must be >= 7.0.0.

index
[bool, optional] If `True`, include the dataframe's index(es) in the file output. If `False`, they will not be written to the file. If `None`, similar to `infer` the dataframe's index(es) will be saved. However, instead of being saved as values, the RangeIndex will be stored as a range in the metadata so it doesn't require much space and is faster. Other indexes will be included as columns in the file output.

engine_kwargs
[dict[str, Any] or None, default None] Additional keyword arguments passed to `pyarrow.orc.write_table()`.

Returns

bytes if no path argument is provided else None

Raises

NotImplementedError
Dtype of one or more columns is category, unsigned integers, interval, period or sparse.

ValueError
engine is not pyarrow.

See also:

`read_orc`
Read a ORC file.

`DataFrame.to_parquet`
Write a parquet file.

`DataFrame.to_csv`
Write a csv file.

DataFrame.to_sql
Write to a sql table.

DataFrame.to_hdf
Write to hdf.

Notes

- Before using this function you should read the user guide about ORC and install optional dependencies.
- This function requires pyarrow library.
- For supported dtypes please refer to supported ORC features in Arrow.
- Currently timezones in datetime columns are not preserved when a dataframe is converted into ORC files.

Examples

```
>>> df = pd.DataFrame(data={'col1': [1, 2], 'col2': [4, 3]})
>>> df.to_orc('df.orc')
>>> pd.read_orc('df.orc')
   col1  col2
0     1     4
1     2     3
```

If you want to get a buffer to the orc content you can write it to io.BytesIO

```
>>> import io
>>> b = io.BytesIO(df.to_orc())
>>> b.seek(0)
0
>>> content = b.read()
```

AlloViz.AlloViz.Elements.Element.to_parquet

Element.**to_parquet**(*path: FilePath | WriteBuffer[bytes] | None = None*, *engine: Literal['auto', 'pyarrow', 'fastparquet'] = 'auto'*, *compression: str | None = 'snappy'*, *index: bool | None = None*, *partition_cols: list[str] | None = None*, *storage_options: StorageOptions | None = None*, ***kwargs*) → bytes | None

Write a DataFrame to the binary parquet format.

This function writes the dataframe as a parquet file. You can choose different parquet backends, and have the option of compression. See the user guide for more details.

Parameters

path
[str, path object, file-like object, or None, default None] String, path object (implementing `os.PathLike[str]`), or file-like object implementing a binary `write()` function. If None, the result is returned as bytes. If a string or path, it will be used as Root Directory path when writing a partitioned dataset.

Changed in version 1.2.0.

Previously this was “fname”

engine
[{‘auto’, ‘pyarrow’, ‘fastparquet’}, default ‘auto’] Parquet library to use. If ‘auto’, then the option `io.parquet.engine` is used. The default `io.parquet.engine` behavior is to try ‘pyarrow’, falling back to ‘fastparquet’ if ‘pyarrow’ is unavailable.

compression
[str or None, default ‘snappy’] Name of the compression to use. Use `None` for no compression. Supported options: ‘snappy’, ‘gzip’, ‘brotli’, ‘lz4’, ‘zstd’.

index
[bool, default None] If `True`, include the dataframe’s index(es) in the file output. If `False`, they will not be written to the file. If `None`, similar to `True` the dataframe’s index(es) will be saved. However, instead of being saved as values, the RangeIndex will be stored as a range in the metadata so it doesn’t require much space and is faster. Other indexes will be included as columns in the file output.

partition_cols
[list, optional, default None] Column names by which to partition the dataset. Columns are partitioned in the order they are given. Must be None if path is not a string.

storage_options
[dict, optional] Extra options that make sense for a particular storage connection, e.g. host, port, username, password, etc. For HTTP(S) URLs the key-value pairs are forwarded to `urllib.request.Request` as header options. For other URLs (e.g. starting with “s3://”, and “gcs://”) the key-value pairs are forwarded to `fsspec.open`. Please see `fsspec` and `urllib` for more details, and for more examples on storage options refer here.

New in version 1.2.0.

****kwargs**
Additional arguments passed to the parquet library. See pandas io for more details.

Returns

bytes if no path argument is provided else None

See also:

`read_parquet`
Read a parquet file.

`DataFrame.to_orc`
Write an orc file.

`DataFrame.to_csv`
Write a csv file.

`DataFrame.to_sql`
Write to a sql table.

`DataFrame.to_hdf`
Write to hdf.

Notes

This function requires either the fastparquet or pyarrow library.

Examples

```
>>> df = pd.DataFrame(data={'col1': [1, 2], 'col2': [3, 4]})
>>> df.to_parquet('df.parquet.gzip',
...               compression='gzip')
>>> pd.read_parquet('df.parquet.gzip')
   col1  col2
0     1     3
1     2     4
```

If you want to get a buffer to the parquet content you can use a io.BytesIO object, as long as you don't use partition_cols, which creates multiple files.

```
>>> import io
>>> f = io.BytesIO()
>>> df.to_parquet(f)
>>> f.seek(0)
0
>>> content = f.read()
```

AlloViz.AlloViz.Elements.Element.to_period

Element.**to_period**(*freq: Frequency | None = None*, *axis: Axis = 0*, *copy: bool | None = None*) → DataFrame

Convert DataFrame from DatetimeIndex to PeriodIndex.

Convert DataFrame from DatetimeIndex to PeriodIndex with desired frequency (inferred from index if not passed).

Parameters

freq
[str, default] Frequency of the PeriodIndex.

axis
[{0 or 'index', 1 or 'columns'}, default 0] The axis to convert (the index by default).

copy
[bool, default True] If False then underlying input data is not copied.

Returns

DataFrame
The DataFrame has a PeriodIndex.

Examples

```
>>> idx = pd.to_datetime(
...     [
...         "2001-03-31 00:00:00",
...         "2002-05-31 00:00:00",
...         "2003-08-31 00:00:00",
...     ]
... )
```

```
>>> idx
DatetimeIndex(['2001-03-31', '2002-05-31', '2003-08-31'],
dtype='datetime64[ns]', freq=None)
```

```
>>> idx.to_period("M")
PeriodIndex(['2001-03', '2002-05', '2003-08'], dtype='period[M]')
```

For the yearly frequency

```
>>> idx.to_period("Y")
PeriodIndex(['2001', '2002', '2003'], dtype='period[A-DEC]')
```

AlloViz.AlloViz.Elements.Element.to_pickle

Element.**to_pickle**(*path: FilePath | WriteBuffer[bytes]*, *compression: CompressionOptions = 'infer'*, *protocol: int = 5*, *storage_options: StorageOptions | None = None*) → None

Pickle (serialize) object to file.

Parameters

path
[str, path object, or file-like object] String, path object (implementing `os.PathLike[str]`), or file-like object implementing a binary `write()` function. File path where the pickled object will be stored.

compression
[str or dict, default 'infer'] For on-the-fly compression of the output data. If 'infer' and 'path' is path-like, then detect compression from the following extensions: '.gz', '.bz2', '.zip', '.xz', '.zst', '.tar', '.tar.gz', '.tar.xz' or '.tar.bz2' (otherwise no compression). Set to `None` for no compression. Can also be a dict with key `'method'` set to one of {`'zip'`, `'gzip'`, `'bz2'`, `'zstd'`, `'xz'`, `'tar'`} and other key-value pairs are forwarded to `zipfile.ZipFile`, `gzip.GzipFile`, `bz2.BZ2File`, `zstandard.ZstdCompressor`, `lzma.LZMAFile` or `tarfile.TarFile`, respectively. As an example, the following could be passed for faster compression and to create a reproducible gzip archive: `compression={'method': 'gzip', 'compresslevel': 1, 'mtime': 1}`.

New in version 1.5.0: Added support for *.tar* files.

protocol
[int] Int which indicates which protocol should be used by the pickler, default HIGHEST_PROTOCOL (see [1] paragraph 12.1.2). The possible values are 0, 1, 2, 3, 4, 5. A negative value for the protocol parameter is equivalent to setting its value to HIGHEST_PROTOCOL.

storage_options
[dict, optional] Extra options that make sense for a particular storage connection, e.g. host, port, username, password, etc. For HTTP(S) URLs the key-value pairs are forwarded to `urllib.request.Request` as header options. For other URLs (e.g. starting with "s3://", and "gcs://") the key-value pairs are forwarded to `fsspec.open`. Please see `fsspec` and `urllib` for more details, and for more examples on storage options refer here.

New in version 1.2.0.

See also:

`read_pickle`
Load pickled pandas object (or any object) from file.

`DataFrame.to_hdf`
Write DataFrame to an HDF5 file.

`DataFrame.to_sql`
Write DataFrame to a SQL database.

`DataFrame.to_parquet`
Write a DataFrame to the binary parquet format.

Examples

```
>>> original_df = pd.DataFrame({"foo": range(5), "bar": range(5, 10)})
>>> original_df
   foo  bar
0    0    5
1    1    6
2    2    7
3    3    8
4    4    9
>>> original_df.to_pickle("./dummy.pkl")
```

```
>>> unpickled_df = pd.read_pickle("./dummy.pkl")
>>> unpickled_df
   foo  bar
0    0    5
1    1    6
2    2    7
3    3    8
4    4    9
```

AlloViz.AlloViz.Elements.Element.to_records

`Element.to_records`(*index: bool = True*, *column_dtypes=None*, *index_dtypes=None*) → recarray

Convert DataFrame to a NumPy record array.

Index will be included as the first field of the record array if requested.

Parameters

index
: [bool, default True] Include index in resulting record array, stored in 'index' field or using the index label, if set.

column_dtypes
: [str, type, dict, default None] If a string or type, the data type to store all columns. If a dictionary, a mapping of column names and indices (zero-indexed) to specific data types.

index_dtypes
: [str, type, dict, default None] If a string or type, the data type to store all index levels. If a dictionary, a mapping of index level names and indices (zero-indexed) to specific data types.

  This mapping is applied only if *index=True*.

Returns

numpy.recarray
: NumPy ndarray with the DataFrame labels as fields and each row of the DataFrame as entries.

See also:

`DataFrame.from_records`
: Convert structured or record ndarray to DataFrame.

`numpy.recarray`
: An ndarray that allows field access using attributes, analogous to typed columns in a spreadsheet.

Examples

```
>>> df = pd.DataFrame({'A': [1, 2], 'B': [0.5, 0.75]},
...                   index=['a', 'b'])
>>> df
   A     B
a  1  0.50
b  2  0.75
>>> df.to_records()
rec.array([('a', 1, 0.5 ), ('b', 2, 0.75)],
          dtype=[('index', 'O'), ('A', '<i8'), ('B', '<f8')])
```

If the DataFrame index has no label then the recarray field name is set to 'index'. If the index has a label then this is used as the field name:

```
>>> df.index = df.index.rename("I")
>>> df.to_records()
```

(continues on next page)

(continued from previous page)

```
rec.array([('a', 1, 0.5 ), ('b', 2, 0.75)],
          dtype=[('I', 'O'), ('A', '<i8'), ('B', '<f8')])
```

The index can be excluded from the record array:

```
>>> df.to_records(index=False)
rec.array([(1, 0.5 ), (2, 0.75)],
          dtype=[('A', '<i8'), ('B', '<f8')])
```

Data types can be specified for the columns:

```
>>> df.to_records(column_dtypes={"A": "int32"})
rec.array([('a', 1, 0.5 ), ('b', 2, 0.75)],
          dtype=[('I', 'O'), ('A', '<i4'), ('B', '<f8')])
```

As well as for the index:

```
>>> df.to_records(index_dtypes="<S2")
rec.array([(b'a', 1, 0.5 ), (b'b', 2, 0.75)],
          dtype=[('I', 'S2'), ('A', '<i8'), ('B', '<f8')])
```

```
>>> index_dtypes = f"<S{df.index.str.len().max()}"
>>> df.to_records(index_dtypes=index_dtypes)
rec.array([(b'a', 1, 0.5 ), (b'b', 2, 0.75)],
          dtype=[('I', 'S1'), ('A', '<i8'), ('B', '<f8')])
```

AlloViz.AlloViz.Elements.Element.to_sql

Element.**to_sql**(*name: str*, *con*, ***, *schema: str | None = None*, *if_exists: Literal['fail', 'replace', 'append'] = 'fail'*, *index: bool = True*, *index_label: Hashable | Sequence[Hashable] | None = None*, *chunksize: int | None = None*, *dtype: ExtensionDtype | str | dtype | Type[str | complex | bool | object] | dict[Hashable, ExtensionDtype | str | dtype | Type[str | complex | bool | object]] | None = None*, *method: Literal['multi'] | Callable | None = None*) → int | None

Write records stored in a DataFrame to a SQL database.

Databases supported by SQLAlchemy [1] are supported. Tables can be newly created, appended to, or overwritten.

Parameters

name
: [str] Name of SQL table.

con
: [sqlalchemy.engine.(Engine or Connection) or sqlite3.Connection] Using SQLAlchemy makes it possible to use any DB supported by that library. Legacy support is provided for sqlite3.Connection objects. The user is responsible for engine disposal and connection closure for the SQLAlchemy connectable. See here. If passing a sqlalchemy.engine.Connection which is already in a transaction, the transaction will not be committed. If passing a sqlite3.Connection, it will not be possible to roll back the record insertion.

schema
[str, optional] Specify the schema (if database flavor supports this). If None, use default schema.

if_exists
[{'fail', 'replace', 'append'}, default 'fail'] How to behave if the table already exists.

- fail: Raise a ValueError.
- replace: Drop the table before inserting new values.
- append: Insert new values to the existing table.

index
[bool, default True] Write DataFrame index as a column. Uses *index_label* as the column name in the table.

index_label
[str or sequence, default None] Column label for index column(s). If None is given (default) and *index* is True, then the index names are used. A sequence should be given if the DataFrame uses MultiIndex.

chunksize
[int, optional] Specify the number of rows in each batch to be written at a time. By default, all rows will be written at once.

dtype
[dict or scalar, optional] Specifying the datatype for columns. If a dictionary is used, the keys should be the column names and the values should be the SQLAlchemy types or strings for the sqlite3 legacy mode. If a scalar is provided, it will be applied to all columns.

method
[{None, 'multi', callable}, optional] Controls the SQL insertion clause used:

- None : Uses standard SQL `INSERT` clause (one per row).
- 'multi': Pass multiple values in a single `INSERT` clause.
- callable with signature `(pd_table, conn, keys, data_iter)`.

Details and a sample callable implementation can be found in the section insert method.

Returns

None or int
Number of rows affected by to_sql. None is returned if the callable passed into `method` does not return an integer number of rows.

The number of returned rows affected is the sum of the `rowcount` attribute of `sqlite3.Cursor` or SQLAlchemy connectable which may not reflect the exact number of written rows as stipulated in the sqlite3 or SQLAlchemy.

New in version 1.4.0.

Raises

ValueError
When the table already exists and *if_exists* is 'fail' (the default).

See also:

read_sql
Read a DataFrame from a table.

Notes

Timezone aware datetime columns will be written as `Timestamp with timezone` type with SQLAlchemy if supported by the database. Otherwise, the datetimes will be stored as timezone unaware timestamps local to the original timezone.

References

[1], [2]

Examples

Create an in-memory SQLite database.

```
>>> from sqlalchemy import create_engine
>>> engine = create_engine('sqlite://', echo=False)
```

Create a table from scratch with 3 rows.

```
>>> df = pd.DataFrame({'name' : ['User 1', 'User 2', 'User 3']})
>>> df
     name
0  User 1
1  User 2
2  User 3
```

```
>>> df.to_sql(name='users', con=engine)
3
>>> from sqlalchemy import text
>>> with engine.connect() as conn:
...    conn.execute(text("SELECT * FROM users")).fetchall()
[(0, 'User 1'), (1, 'User 2'), (2, 'User 3')]
```

An *sqlalchemy.engine.Connection* can also be passed to *con*:

```
>>> with engine.begin() as connection:
...     df1 = pd.DataFrame({'name' : ['User 4', 'User 5']})
...     df1.to_sql(name='users', con=connection, if_exists='append')
2
```

This is allowed to support operations that require that the same DBAPI connection is used for the entire operation.

```
>>> df2 = pd.DataFrame({'name' : ['User 6', 'User 7']})
>>> df2.to_sql(name='users', con=engine, if_exists='append')
2
>>> with engine.connect() as conn:
...    conn.execute(text("SELECT * FROM users")).fetchall()
[(0, 'User 1'), (1, 'User 2'), (2, 'User 3'),
```

(continues on next page)

(continued from previous page)

```
 (0, 'User 4'), (1, 'User 5'), (0, 'User 6'),
 (1, 'User 7')]
```

Overwrite the table with just `df2`.

```
>>> df2.to_sql(name='users', con=engine, if_exists='replace',
...            index_label='id')
2
>>> with engine.connect() as conn:
...     conn.execute(text("SELECT * FROM users")).fetchall()
[(0, 'User 6'), (1, 'User 7')]
```

Use `method` to define a callable insertion method to do nothing if there's a primary key conflict on a table in a PostgreSQL database.

```
>>> from sqlalchemy.dialects.postgresql import insert
>>> def insert_on_conflict_nothing(table, conn, keys, data_iter):
...     # "a" is the primary key in "conflict_table"
...     data = [dict(zip(keys, row)) for row in data_iter]
...     stmt = insert(table.table).values(data).on_conflict_do_nothing(index_
↪elements=["a"])
...     result = conn.execute(stmt)
...     return result.rowcount
>>> df_conflict.to_sql(name="conflict_table", con=conn, if_exists="append",␣
↪method=insert_on_conflict_nothing)
0
```

For MySQL, a callable to update columns `b` and `c` if there's a conflict on a primary key.

```
>>> from sqlalchemy.dialects.mysql import insert
>>> def insert_on_conflict_update(table, conn, keys, data_iter):
...     # update columns "b" and "c" on primary key conflict
...     data = [dict(zip(keys, row)) for row in data_iter]
...     stmt = (
...         insert(table.table)
...         .values(data)
...     )
...     stmt = stmt.on_duplicate_key_update(b=stmt.inserted.b, c=stmt.inserted.
↪c)
...     result = conn.execute(stmt)
...     return result.rowcount
>>> df_conflict.to_sql(name="conflict_table", con=conn, if_exists="append",␣
↪method=insert_on_conflict_update)
2
```

Specify the dtype (especially useful for integers with missing values). Notice that while pandas is forced to store the data as floating point, the database supports nullable integers. When fetching the data with Python, we get back integer scalars.

```
>>> df = pd.DataFrame({"A": [1, None, 2]})
>>> df
     A
0  1.0
```

(continues on next page)

(continued from previous page)

```
1  NaN
2  2.0
```

```
>>> from sqlalchemy.types import Integer
>>> df.to_sql(name='integers', con=engine, index=False,
...           dtype={"A": Integer()})
3
```

```
>>> with engine.connect() as conn:
...   conn.execute(text("SELECT * FROM integers")).fetchall()
[(1,), (None,), (2,)]
```

AlloViz.AlloViz.Elements.Element.to_stata

Element.**to_stata**(*path: FilePath | WriteBuffer[bytes]*, ***, *convert_dates: dict[Hashable, str] | None = None*, *write_index: bool = True*, *byteorder: ToStataByteorder | None = None*, *time_stamp: datetime.datetime | None = None*, *data_label: str | None = None*, *variable_labels: dict[Hashable, str] | None = None*, *version: int | None = 114*, *convert_strl: Sequence[Hashable] | None = None*, *compression: CompressionOptions = 'infer'*, *storage_options: StorageOptions | None = None*, *value_labels: dict[Hashable, dict[float, str]] | None = None*) → None

Export DataFrame object to Stata dta format.

Writes the DataFrame to a Stata dataset file. “dta” files contain a Stata dataset.

Parameters

path
: [str, path object, or buffer] String, path object (implementing `os.PathLike[str]`), or file-like object implementing a binary `write()` function.

convert_dates
: [dict] Dictionary mapping columns containing datetime types to stata internal format to use when writing the dates. Options are ‘tc’, ‘td’, ‘tm’, ‘tw’, ‘th’, ‘tq’, ‘ty’. Column can be either an integer or a name. Datetime columns that do not have a conversion type specified will be converted to ‘tc’. Raises NotImplementedError if a datetime column has timezone information.

write_index
: [bool] Write the index to Stata dataset.

byteorder
: [str] Can be “>”, “<”, “little”, or “big”. default is *sys.byteorder*.

time_stamp
: [datetime] A datetime to use as file creation date. Default is the current time.

data_label
: [str, optional] A label for the data set. Must be 80 characters or smaller.

variable_labels
: [dict] Dictionary containing columns as keys and variable labels as values. Each label must be 80 characters or smaller.

version

[{114, 117, 118, 119, None}, default 114] Version to use in the output dta file. Set to None to let pandas decide between 118 or 119 formats depending on the number of columns in the frame. Version 114 can be read by Stata 10 and later. Version 117 can be read by Stata 13 or later. Version 118 is supported in Stata 14 and later. Version 119 is supported in Stata 15 and later. Version 114 limits string variables to 244 characters or fewer while versions 117 and later allow strings with lengths up to 2,000,000 characters. Versions 118 and 119 support Unicode characters, and version 119 supports more than 32,767 variables.

Version 119 should usually only be used when the number of variables exceeds the capacity of dta format 118. Exporting smaller datasets in format 119 may have unintended consequences, and, as of November 2020, Stata SE cannot read version 119 files.

convert_strl

[list, optional] List of column names to convert to string columns to Stata StrL format. Only available if version is 117. Storing strings in the StrL format can produce smaller dta files if strings have more than 8 characters and values are repeated.

compression

[str or dict, default 'infer'] For on-the-fly compression of the output data. If 'infer' and 'path' is path-like, then detect compression from the following extensions: '.gz', '.bz2', '.zip', '.xz', '.zst', '.tar', '.tar.gz', '.tar.xz' or '.tar.bz2' (otherwise no compression). Set to `None` for no compression. Can also be a dict with key `'method'` set to one of {`'zip'`, `'gzip'`, `'bz2'`, `'zstd'`, `'xz'`, `'tar'`} and other key-value pairs are forwarded to `zipfile.ZipFile`, `gzip.GzipFile`, `bz2.BZ2File`, `zstandard.ZstdCompressor`, `lzma.LZMAFile` or `tarfile.TarFile`, respectively. As an example, the following could be passed for faster compression and to create a reproducible gzip archive: `compression={'method': 'gzip', 'compresslevel': 1, 'mtime': 1}`.

New in version 1.5.0: Added support for *.tar* files.

Changed in version 1.4.0: Zstandard support.

storage_options

[dict, optional] Extra options that make sense for a particular storage connection, e.g. host, port, username, password, etc. For HTTP(S) URLs the key-value pairs are forwarded to `urllib.request.Request` as header options. For other URLs (e.g. starting with "s3://", and "gcs://") the key-value pairs are forwarded to `fsspec.open`. Please see `fsspec` and `urllib` for more details, and for more examples on storage options refer here.

New in version 1.2.0.

value_labels

[dict of dicts] Dictionary containing columns as keys and dictionaries of column value to labels as values. Labels for a single variable must be 32,000 characters or smaller.

New in version 1.4.0.

Raises

NotImplementedError

- If datetimes contain timezone information
- Column dtype is not representable in Stata

ValueError

- Columns listed in convert_dates are neither datetime64[ns] or datetime.datetime
- Column listed in convert_dates is not in DataFrame
- Categorical label contains more than 32,000 characters

See also:

read_stata
Import Stata data files.

io.stata.StataWriter
Low-level writer for Stata data files.

io.stata.StataWriter117
Low-level writer for version 117 files.

Examples

```
>>> df = pd.DataFrame({'animal': ['falcon', 'parrot', 'falcon',
...                               'parrot'],
...                    'speed': [350, 18, 361, 15]})
>>> df.to_stata('animals.dta')
```

AlloViz.AlloViz.Elements.Element.to_string

Element.**to_string**(*buf: FilePath | WriteBuffer[str] | None = None*, *columns: Axes | None = None*, *col_space: int | list[int] | dict[Hashable, int] | None = None*, *header: bool | list[str] = True*, *index: bool = True*, *na_rep: str = 'NaN'*, *formatters: fmt.FormattersType | None = None*, *float_format: fmt.FloatFormatType | None = None*, *sparsify: bool | None = None*, *index_names: bool = True*, *justify: str | None = None*, *max_rows: int | None = None*, *max_cols: int | None = None*, *show_dimensions: bool = False*, *decimal: str = '.'*, *line_width: int | None = None*, *min_rows: int | None = None*, *max_colwidth: int | None = None*, *encoding: str | None = None*) → str | None

Render a DataFrame to a console-friendly tabular output.

Parameters

buf
[str, Path or StringIO-like, optional, default None] Buffer to write to. If None, the output is returned as a string.

columns
[array-like, optional, default None] The subset of columns to write. Writes all columns by default.

col_space
[int, list or dict of int, optional] The minimum width of each column. If a list of ints is given every integers corresponds with one column. If a dict is given, the key references the column, while the value defines the space to use..

header
[bool or list of str, optional] Write out the column names. If a list of columns is given, it is assumed to be aliases for the column names.

index
[bool, optional, default True] Whether to print index (row) labels.

na_rep
[str, optional, default 'NaN'] String representation of `NaN` to use.

formatters
[list, tuple or dict of one-param. functions, optional] Formatter functions to apply to columns' elements by position or name. The result of each function must be a unicode string. List/tuple must be of length equal to the number of columns.

float_format
[one-parameter function, optional, default None] Formatter function to apply to columns' elements if they are floats. This function must return a unicode string and will be applied only to the non-`NaN` elements, with `NaN` being handled by `na_rep`.

Changed in version 1.2.0.

sparsify
[bool, optional, default True] Set to False for a DataFrame with a hierarchical index to print every multiindex key at each row.

index_names
[bool, optional, default True] Prints the names of the indexes.

justify
[str, default None] How to justify the column labels. If None uses the option from the print configuration (controlled by set_option), 'right' out of the box. Valid values are

- left
- right
- center
- justify
- justify-all
- start
- end
- inherit
- match-parent
- initial
- unset.

max_rows
[int, optional] Maximum number of rows to display in the console.

max_cols
[int, optional] Maximum number of columns to display in the console.

show_dimensions
[bool, default False] Display DataFrame dimensions (number of rows by number of columns).

decimal
[str, default '.'] Character recognized as decimal separator, e.g. ',' in Europe.

line_width
[int, optional] Width to wrap a line in characters.

min_rows
[int, optional] The number of rows to display in the console in a truncated repr (when number of rows is above *max_rows*).

max_colwidth
[int, optional] Max width to truncate each column in characters. By default, no limit.

encoding
[str, default "utf-8"] Set character encoding.

Returns

str or None
If buf is None, returns the result as a string. Otherwise returns None.

See also:

to_html
Convert DataFrame to HTML.

Examples

```
>>> d = {'col1': [1, 2, 3], 'col2': [4, 5, 6]}
>>> df = pd.DataFrame(d)
>>> print(df.to_string())
   col1  col2
0     1     4
1     2     5
2     3     6
```

AlloViz.AlloViz.Elements.Element.to_timestamp

`Element.to_timestamp`(*freq: Frequency | None = None*, *how: ToTimestampHow = 'start'*, *axis: Axis = 0*, *copy: bool | None = None*) → DataFrame

Cast to DatetimeIndex of timestamps, at *beginning* of period.

Parameters

freq
[str, default frequency of PeriodIndex] Desired frequency.

how
[{'s', 'e', 'start', 'end'}] Convention for converting period to timestamp; start of period vs. end.

axis
[{0 or 'index', 1 or 'columns'}, default 0] The axis to convert (the index by default).

copy
[bool, default True] If False then underlying input data is not copied.

Returns

DataFrame
The DataFrame has a DatetimeIndex.

Examples

```
>>> idx = pd.PeriodIndex(['2023', '2024'], freq='Y')
>>> d = {'col1': [1, 2], 'col2': [3, 4]}
>>> df1 = pd.DataFrame(data=d, index=idx)
>>> df1
      col1   col2
2023     1      3
2024     2      4
```

The resulting timestamps will be at the beginning of the year in this case

```
>>> df1 = df1.to_timestamp()
>>> df1
            col1   col2
2023-01-01     1      3
2024-01-01     2      4
>>> df1.index
DatetimeIndex(['2023-01-01', '2024-01-01'], dtype='datetime64[ns]', freq=None)
```

Using *freq* which is the offset that the Timestamps will have

```
>>> df2 = pd.DataFrame(data=d, index=idx)
>>> df2 = df2.to_timestamp(freq='M')
>>> df2
            col1   col2
2023-01-31     1      3
2024-01-31     2      4
>>> df2.index
DatetimeIndex(['2023-01-31', '2024-01-31'], dtype='datetime64[ns]', freq=None)
```

AlloViz.AlloViz.Elements.Element.to_xarray

`Element.`**`to_xarray`**`()`

Return an xarray object from the pandas object.

Returns

xarray.DataArray or xarray.Dataset
Data in the pandas structure converted to Dataset if the object is a DataFrame, or a DataArray if the object is a Series.

See also:

`DataFrame.to_hdf`
Write DataFrame to an HDF5 file.

`DataFrame.to_parquet`
Write a DataFrame to the binary parquet format.

Notes

See the xarray docs

Examples

```
>>> df = pd.DataFrame([('falcon', 'bird', 389.0, 2),
...                    ('parrot', 'bird', 24.0, 2),
...                    ('lion', 'mammal', 80.5, 4),
...                    ('monkey', 'mammal', np.nan, 4)],
...                   columns=['name', 'class', 'max_speed',
...                            'num_legs'])
>>> df
     name   class  max_speed  num_legs
0  falcon    bird      389.0         2
1  parrot    bird       24.0         2
2    lion  mammal       80.5         4
3  monkey  mammal        NaN         4
```

```
>>> df.to_xarray()
<xarray.Dataset>
Dimensions:    (index: 4)
Coordinates:
  * index      (index) int64 0 1 2 3
Data variables:
    name       (index) object 'falcon' 'parrot' 'lion' 'monkey'
    class      (index) object 'bird' 'bird' 'mammal' 'mammal'
    max_speed  (index) float64 389.0 24.0 80.5 nan
    num_legs   (index) int64 2 2 4 4
```

```
>>> df['max_speed'].to_xarray()
<xarray.DataArray 'max_speed' (index: 4)>
array([389. ,  24. ,  80.5,   nan])
Coordinates:
  * index    (index) int64 0 1 2 3
```

```
>>> dates = pd.to_datetime(['2018-01-01', '2018-01-01',
...                         '2018-01-02', '2018-01-02'])
>>> df_multiindex = pd.DataFrame({'date': dates,
...                               'animal': ['falcon', 'parrot',
...                                          'falcon', 'parrot'],
...                               'speed': [350, 18, 361, 15]})
>>> df_multiindex = df_multiindex.set_index(['date', 'animal'])
```

```
>>> df_multiindex
                   speed
date       animal
2018-01-01 falcon    350
           parrot     18
2018-01-02 falcon    361
           parrot     15
```

```
>>> df_multiindex.to_xarray()
<xarray.Dataset>
Dimensions:  (date: 2, animal: 2)
Coordinates:
  * date     (date) datetime64[ns] 2018-01-01 2018-01-02
  * animal   (animal) object 'falcon' 'parrot'
Data variables:
    speed    (date, animal) int64 350 18 361 15
```

AlloViz.AlloViz.Elements.Element.to_xml

Element.**to_xml**(*path_or_buffer: FilePath | WriteBuffer[bytes] | WriteBuffer[str] | None = None*, *index: bool = True*, *root_name: str | None = 'data'*, *row_name: str | None = 'row'*, *na_rep: str | None = None*, *attr_cols: list[str] | None = None*, *elem_cols: list[str] | None = None*, *namespaces: dict[str | None, str] | None = None*, *prefix: str | None = None*, *encoding: str = 'utf-8'*, *xml_declaration: bool | None = True*, *pretty_print: bool | None = True*, *parser: XMLParsers | None = 'lxml'*, *stylesheet: FilePath | ReadBuffer[str] | ReadBuffer[bytes] | None = None*, *compression: CompressionOptions = 'infer'*, *storage_options: StorageOptions | None = None*) → str | None

Render a DataFrame to an XML document.

New in version 1.3.0.

Parameters

path_or_buffer
[str, path object, file-like object, or None, default None] String, path object (implementing `os.PathLike[str]`), or file-like object implementing a `write()` function. If None, the result is returned as a string.

index
[bool, default True] Whether to include index in XML document.

root_name
[str, default 'data'] The name of root element in XML document.

row_name
[str, default 'row'] The name of row element in XML document.

na_rep
[str, optional] Missing data representation.

attr_cols
[list-like, optional] List of columns to write as attributes in row element. Hierarchical columns will be flattened with underscore delimiting the different levels.

elem_cols
[list-like, optional] List of columns to write as children in row element. By default, all columns output as children of row element. Hierarchical columns will be flattened with underscore delimiting the different levels.

namespaces
[dict, optional] All namespaces to be defined in root element. Keys of dict should be prefix names and values of dict corresponding URIs. Default namespaces should be given empty string key. For example,

```
namespaces = {"": "https://example.com"}
```

prefix
[str, optional] Namespace prefix to be used for every element and/or attribute in document. This should be one of the keys in `namespaces` dict.

encoding
[str, default 'utf-8'] Encoding of the resulting document.

xml_declaration
[bool, default True] Whether to include the XML declaration at start of document.

pretty_print
[bool, default True] Whether output should be pretty printed with indentation and line breaks.

parser
[{'lxml','etree'}, default 'lxml'] Parser module to use for building of tree. Only 'lxml' and 'etree' are supported. With 'lxml', the ability to use XSLT stylesheet is supported.

stylesheet
[str, path object or file-like object, optional] A URL, file-like object, or a raw string containing an XSLT script used to transform the raw XML output. Script should use layout of elements and attributes from original output. This argument requires `lxml` to be installed. Only XSLT 1.0 scripts and not later versions is currently supported.

compression
[str or dict, default 'infer'] For on-the-fly compression of the output data. If 'infer' and 'path_or_buffer' is path-like, then detect compression from the following extensions: '.gz', '.bz2', '.zip', '.xz', '.zst', '.tar', '.tar.gz', '.tar.xz' or '.tar.bz2' (otherwise no compression). Set to `None` for no compression. Can also be a dict with key `'method'` set to one of {`'zip'`, `'gzip'`, `'bz2'`, `'zstd'`, `'xz'`, `'tar'`} and other key-value pairs are forwarded to `zipfile.ZipFile`, `gzip.GzipFile`, `bz2.BZ2File`, `zstandard.ZstdCompressor`, `lzma.LZMAFile` or `tarfile.TarFile`, respectively. As an example, the following could be passed for faster compression and to create a reproducible gzip archive: `compression={'method': 'gzip', 'compresslevel': 1, 'mtime': 1}`.

New in version 1.5.0: Added support for *.tar* files.

Changed in version 1.4.0: Zstandard support.

storage_options
[dict, optional] Extra options that make sense for a particular storage connection, e.g. host, port, username, password, etc. For HTTP(S) URLs the key-value pairs are forwarded to `urllib.request.Request` as header options. For other URLs (e.g. starting with "s3://", and "gcs://") the key-value pairs are forwarded to `fsspec.open`. Please see `fsspec` and `urllib` for more details, and for more examples on storage options refer here.

Returns

None or str
If `io` is None, returns the resulting XML format as a string. Otherwise returns None.

See also:

to_json
Convert the pandas object to a JSON string.

to_html
Convert DataFrame to a html.

Examples

```
>>> df = pd.DataFrame({'shape': ['square', 'circle', 'triangle'],
...                    'degrees': [360, 360, 180],
...                    'sides': [4, np.nan, 3]})
```

```
>>> df.to_xml()
<?xml version='1.0' encoding='utf-8'?>
<data>
  <row>
    <index>0</index>
    <shape>square</shape>
    <degrees>360</degrees>
    <sides>4.0</sides>
  </row>
  <row>
    <index>1</index>
    <shape>circle</shape>
    <degrees>360</degrees>
    <sides/>
  </row>
  <row>
    <index>2</index>
    <shape>triangle</shape>
    <degrees>180</degrees>
    <sides>3.0</sides>
  </row>
</data>
```

```
>>> df.to_xml(attr_cols=[
...           'index', 'shape', 'degrees', 'sides'
...           ])
<?xml version='1.0' encoding='utf-8'?>
<data>
  <row index="0" shape="square" degrees="360" sides="4.0"/>
  <row index="1" shape="circle" degrees="360"/>
  <row index="2" shape="triangle" degrees="180" sides="3.0"/>
</data>
```

```
>>> df.to_xml(namespaces={"doc": "https://example.com"},
...           prefix="doc")
<?xml version='1.0' encoding='utf-8'?>
<doc:data xmlns:doc="https://example.com">
  <doc:row>
    <doc:index>0</doc:index>
    <doc:shape>square</doc:shape>
```

(continues on next page)

(continued from previous page)

```
    <doc:degrees>360</doc:degrees>
    <doc:sides>4.0</doc:sides>
  </doc:row>
  <doc:row>
    <doc:index>1</doc:index>
    <doc:shape>circle</doc:shape>
    <doc:degrees>360</doc:degrees>
    <doc:sides/>
  </doc:row>
  <doc:row>
    <doc:index>2</doc:index>
    <doc:shape>triangle</doc:shape>
    <doc:degrees>180</doc:degrees>
    <doc:sides>3.0</doc:sides>
  </doc:row>
</doc:data>
```

AlloViz.AlloViz.Elements.Element.transform

`Element.transform`(*func: AggFuncType*, *axis: Axis = 0*, **args*, ***kwargs*) → DataFrame

Call `func` on self producing a DataFrame with the same axis shape as self.

Parameters

func
[function, str, list-like or dict-like] Function to use for transforming the data. If a function, must either work when passed a DataFrame or when passed to DataFrame.apply. If func is both list-like and dict-like, dict-like behavior takes precedence.

Accepted combinations are:

- function
- string function name
- list-like of functions and/or function names, e.g. `[np.exp, 'sqrt']`
- dict-like of axis labels -> functions, function names or list-like of such.

axis
[{0 or 'index', 1 or 'columns'}, default 0] If 0 or 'index': apply function to each column. If 1 or 'columns': apply function to each row.

***args**
Positional arguments to pass to *func*.

****kwargs**
Keyword arguments to pass to *func*.

Returns

DataFrame
A DataFrame that must have the same length as self.

Raises

ValueError
[If the returned DataFrame has a different length than self.]

See also:

DataFrame.agg
Only perform aggregating type operations.

DataFrame.apply
Invoke function on a DataFrame.

Notes

Functions that mutate the passed object can produce unexpected behavior or errors and are not supported. See Mutating with User Defined Function (UDF) methods for more details.

Examples

```
>>> df = pd.DataFrame({'A': range(3), 'B': range(1, 4)})
>>> df
   A  B
0  0  1
1  1  2
2  2  3
>>> df.transform(lambda x: x + 1)
   A  B
0  1  2
1  2  3
2  3  4
```

Even though the resulting DataFrame must have the same length as the input DataFrame, it is possible to provide several input functions:

```
>>> s = pd.Series(range(3))
>>> s
0    0
1    1
2    2
dtype: int64
>>> s.transform([np.sqrt, np.exp])
       sqrt        exp
0  0.000000   1.000000
1  1.000000   2.718282
2  1.414214   7.389056
```

You can call transform on a GroupBy object:

```
>>> df = pd.DataFrame({
...     "Date": [
...         "2015-05-08", "2015-05-07", "2015-05-06", "2015-05-05",
...         "2015-05-08", "2015-05-07", "2015-05-06", "2015-05-05"],
...     "Data": [5, 8, 6, 1, 50, 100, 60, 120],
... })
```

(continues on next page)

(continued from previous page)

```
>>> df
         Date  Data
0  2015-05-08     5
1  2015-05-07     8
2  2015-05-06     6
3  2015-05-05     1
4  2015-05-08    50
5  2015-05-07   100
6  2015-05-06    60
7  2015-05-05   120
>>> df.groupby('Date')['Data'].transform('sum')
0     55
1    108
2     66
3    121
4     55
5    108
6     66
7    121
Name: Data, dtype: int64
```

```
>>> df = pd.DataFrame({
...     "c": [1, 1, 1, 2, 2, 2, 2],
...     "type": ["m", "n", "o", "m", "m", "n", "n"]
... })
>>> df
   c type
0  1    m
1  1    n
2  1    o
3  2    m
4  2    m
5  2    n
6  2    n
>>> df['size'] = df.groupby('c')['type'].transform(len)
>>> df
   c type size
0  1    m    3
1  1    n    3
2  1    o    3
3  2    m    4
4  2    m    4
5  2    n    4
6  2    n    4
```

AlloViz.AlloViz.Elements.Element.transpose

`Element.transpose`(**args*, *copy: bool = False*) → DataFrame

Transpose index and columns.

Reflect the DataFrame over its main diagonal by writing rows as columns and vice-versa. The property `T` is an accessor to the method *transpose()*.

Parameters

***args**
[tuple, optional] Accepted for compatibility with NumPy.

copy
[bool, default False] Whether to copy the data after transposing, even for DataFrames with a single dtype.

Note that a copy is always required for mixed dtype DataFrames, or for DataFrames with any extension types.

Returns

DataFrame
The transposed DataFrame.

See also:

numpy.transpose
Permute the dimensions of a given array.

Notes

Transposing a DataFrame with mixed dtypes will result in a homogeneous DataFrame with the *object* dtype. In such a case, a copy of the data is always made.

Examples

Square DataFrame with homogeneous dtype

```
>>> d1 = {'col1': [1, 2], 'col2': [3, 4]}
>>> df1 = pd.DataFrame(data=d1)
>>> df1
   col1  col2
0     1     3
1     2     4
```

```
>>> df1_transposed = df1.T  # or df1.transpose()
>>> df1_transposed
      0  1
col1  1  2
col2  3  4
```

When the dtype is homogeneous in the original DataFrame, we get a transposed DataFrame with the same dtype:

```
>>> df1.dtypes
col1    int64
col2    int64
dtype: object
>>> df1_transposed.dtypes
0    int64
1    int64
dtype: object
```

Non-square DataFrame with mixed dtypes

```
>>> d2 = {'name': ['Alice', 'Bob'],
...       'score': [9.5, 8],
...       'employed': [False, True],
...       'kids': [0, 0]}
>>> df2 = pd.DataFrame(data=d2)
>>> df2
    name  score  employed  kids
0  Alice    9.5     False     0
1    Bob    8.0      True     0
```

```
>>> df2_transposed = df2.T  # or df2.transpose()
>>> df2_transposed
              0     1
name      Alice   Bob
score       9.5   8.0
employed  False  True
kids          0     0
```

When the DataFrame has mixed dtypes, we get a transposed DataFrame with the *object* dtype:

```
>>> df2.dtypes
name         object
score       float64
employed       bool
kids          int64
dtype: object
>>> df2_transposed.dtypes
0    object
1    object
dtype: object
```

AlloViz.AlloViz.Elements.Element.truediv

`Element.`**`truediv`**(*other*, *axis: Axis = 'columns'*, *level=None*, *fill_value=None*)

Get Floating division of dataframe and other, element-wise (binary operator *truediv*).

Equivalent to `dataframe / other`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *rtruediv*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *floordiv*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other
[scalar, sequence, Series, dict or DataFrame] Any single or multiple element data structure, or list-like object.

axis
[{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns. (1 or 'columns'). For Series input, axis to match Series index on.

level
[int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value
[float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame
Result of the arithmetic operation.

See also:

`DataFrame.add`
Add DataFrames.

`DataFrame.sub`
Subtract DataFrames.

`DataFrame.mul`
Multiply DataFrames.

`DataFrame.div`
Divide DataFrames (float division).

`DataFrame.truediv`
Divide DataFrames (float division).

`DataFrame.floordiv`
Divide DataFrames (integer division).

`DataFrame.mod`
Calculate modulo (remainder after division).

`DataFrame.pow`
Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...         axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a dictionary by axis.

```
>>> df.mul({'angles': 0, 'degrees': 2})
            angles  degrees
circle           0      720
triangle         0      360
rectangle        0      720
```

```
>>> df.mul({'circle': 0, 'triangle': 2, 'rectangle': 3}, axis='index')
            angles  degrees
circle           0        0
triangle         6      360
rectangle       12     1080
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

AlloViz.AlloViz.Elements.Element.truncate

Element.**truncate**(*before=None*, *after=None*, *axis: int | Literal['index', 'columns', 'rows'] | None = None*, *copy: bool | None = None*) → None

Truncate a Series or DataFrame before and after some index value.

This is a useful shorthand for boolean indexing based on index values above or below certain thresholds.

Parameters

before
[date, str, int] Truncate all rows before this index value.

after
[date, str, int] Truncate all rows after this index value.

axis
[{0 or 'index', 1 or 'columns'}, optional] Axis to truncate. Truncates the index (rows) by default. For *Series* this parameter is unused and defaults to 0.

copy
[bool, default is True,] Return a copy of the truncated section.

Returns

type of caller
The truncated Series or DataFrame.

See also:

DataFrame.loc
Select a subset of a DataFrame by label.

DataFrame.iloc
Select a subset of a DataFrame by position.

Notes

If the index being truncated contains only datetime values, *before* and *after* may be specified as strings instead of Timestamps.

Examples

```
>>> df = pd.DataFrame({'A': ['a', 'b', 'c', 'd', 'e'],
...                    'B': ['f', 'g', 'h', 'i', 'j'],
...                    'C': ['k', 'l', 'm', 'n', 'o']},
...                   index=[1, 2, 3, 4, 5])
>>> df
   A  B  C
1  a  f  k
2  b  g  l
3  c  h  m
4  d  i  n
5  e  j  o
```

```
>>> df.truncate(before=2, after=4)
   A  B  C
2  b  g  l
3  c  h  m
4  d  i  n
```

The columns of a DataFrame can be truncated.

```
>>> df.truncate(before="A", after="B", axis="columns")
   A  B
1  a  f
2  b  g
3  c  h
4  d  i
5  e  j
```

For Series, only rows can be truncated.

```
>>> df['A'].truncate(before=2, after=4)
2    b
3    c
4    d
Name: A, dtype: object
```

The index values in `truncate` can be datetimes or string dates.

```
>>> dates = pd.date_range('2016-01-01', '2016-02-01', freq='s')
>>> df = pd.DataFrame(index=dates, data={'A': 1})
>>> df.tail()
                     A
```

(continues on next page)

(continued from previous page)

```
2016-01-31 23:59:56  1
2016-01-31 23:59:57  1
2016-01-31 23:59:58  1
2016-01-31 23:59:59  1
2016-02-01 00:00:00  1
```

```
>>> df.truncate(before=pd.Timestamp('2016-01-05'),
...             after=pd.Timestamp('2016-01-10')).tail()
                     A
2016-01-09 23:59:56  1
2016-01-09 23:59:57  1
2016-01-09 23:59:58  1
2016-01-09 23:59:59  1
2016-01-10 00:00:00  1
```

Because the index is a DatetimeIndex containing only dates, we can specify *before* and *after* as strings. They will be coerced to Timestamps before truncation.

```
>>> df.truncate('2016-01-05', '2016-01-10').tail()
                     A
2016-01-09 23:59:56  1
2016-01-09 23:59:57  1
2016-01-09 23:59:58  1
2016-01-09 23:59:59  1
2016-01-10 00:00:00  1
```

Note that `truncate` assumes a 0 value for any unspecified time component (midnight). This differs from partial string slicing, which returns any partially matching dates.

```
>>> df.loc['2016-01-05':'2016-01-10', :].tail()
                     A
2016-01-10 23:59:55  1
2016-01-10 23:59:56  1
2016-01-10 23:59:57  1
2016-01-10 23:59:58  1
2016-01-10 23:59:59  1
```

AlloViz.AlloViz.Elements.Element.tz_convert

`Element.tz_convert`(*tz*, *axis: int | Literal['index', 'columns', 'rows'] = 0*, *level=None*, *copy: bool | None = None*) → None

Convert tz-aware axis to target time zone.

Parameters

tz
: [str or tzinfo object or None] Target time zone. Passing `None` will convert to UTC and remove the timezone information.

axis
: [{0 or 'index', 1 or 'columns'}, default 0] The axis to convert

level
[int, str, default None] If axis is a MultiIndex, convert a specific level. Otherwise must be None.

copy
[bool, default True] Also make a copy of the underlying data.

Returns

Series/DataFrame
Object with time zone converted axis.

Raises

TypeError
If the axis is tz-naive.

Examples

Change to another time zone:

```
>>> s = pd.Series(
...     [1],
...     index=pd.DatetimeIndex(['2018-09-15 01:30:00+02:00']),
... )
>>> s.tz_convert('Asia/Shanghai')
2018-09-15 07:30:00+08:00    1
dtype: int64
```

Pass None to convert to UTC and get a tz-naive index:

```
>>> s = pd.Series([1],
...     index=pd.DatetimeIndex(['2018-09-15 01:30:00+02:00']))
>>> s.tz_convert(None)
2018-09-14 23:30:00    1
dtype: int64
```

AlloViz.AlloViz.Elements.Element.tz_localize

`Element.tz_localize`(*tz*, *axis: Axis = 0*, *level=None*, *copy: bool_t | None = None*, *ambiguous: TimeAmbiguous = 'raise'*, *nonexistent: TimeNonexistent = 'raise'*) → Self

Localize tz-naive index of a Series or DataFrame to target time zone.

This operation localizes the Index. To localize the values in a timezone-naive Series, use `Series.dt.tz_localize()`.

Parameters

tz
[str or tzinfo or None] Time zone to localize. Passing `None` will remove the time zone information and preserve local time.

axis
[{0 or 'index', 1 or 'columns'}, default 0] The axis to localize

level
[int, str, default None] If axis ia a MultiIndex, localize a specific level. Otherwise must be None.

copy
[bool, default True] Also make a copy of the underlying data.

ambiguous
['infer', bool-ndarray, 'NaT', default 'raise'] When clocks moved backward due to DST, ambiguous times may arise. For example in Central European Time (UTC+01), when going from 03:00 DST to 02:00 non-DST, 02:30:00 local time occurs both at 00:30:00 UTC and at 01:30:00 UTC. In such a situation, the *ambiguous* parameter dictates how ambiguous times should be handled.

- 'infer' will attempt to infer fall dst-transition hours based on order
- bool-ndarray where True signifies a DST time, False designates a non-DST time (note that this flag is only applicable for ambiguous times)
- 'NaT' will return NaT where there are ambiguous times
- 'raise' will raise an AmbiguousTimeError if there are ambiguous times.

nonexistent
[str, default 'raise'] A nonexistent time does not exist in a particular timezone where clocks moved forward due to DST. Valid values are:

- 'shift_forward' will shift the nonexistent time forward to the closest existing time
- 'shift_backward' will shift the nonexistent time backward to the closest existing time
- 'NaT' will return NaT where there are nonexistent times
- timedelta objects will shift nonexistent times by the timedelta
- 'raise' will raise an NonExistentTimeError if there are nonexistent times.

Returns

Series/DataFrame
Same type as the input.

Raises

TypeError
If the TimeSeries is tz-aware and tz is not None.

Examples

Localize local times:

```
>>> s = pd.Series(
...     [1],
...     index=pd.DatetimeIndex(['2018-09-15 01:30:00']),
... )
>>> s.tz_localize('CET')
2018-09-15 01:30:00+02:00    1
dtype: int64
```

Pass None to convert to tz-naive index and preserve local time:

```
>>> s = pd.Series([1],
...     index=pd.DatetimeIndex(['2018-09-15 01:30:00+02:00']))
>>> s.tz_localize(None)
2018-09-15 01:30:00    1
dtype: int64
```

Be careful with DST changes. When there is sequential data, pandas can infer the DST time:

```
>>> s = pd.Series(range(7),
...               index=pd.DatetimeIndex(['2018-10-28 01:30:00',
...                                       '2018-10-28 02:00:00',
...                                       '2018-10-28 02:30:00',
...                                       '2018-10-28 02:00:00',
...                                       '2018-10-28 02:30:00',
...                                       '2018-10-28 03:00:00',
...                                       '2018-10-28 03:30:00']))
>>> s.tz_localize('CET', ambiguous='infer')
2018-10-28 01:30:00+02:00    0
2018-10-28 02:00:00+02:00    1
2018-10-28 02:30:00+02:00    2
2018-10-28 02:00:00+01:00    3
2018-10-28 02:30:00+01:00    4
2018-10-28 03:00:00+01:00    5
2018-10-28 03:30:00+01:00    6
dtype: int64
```

In some cases, inferring the DST is impossible. In such cases, you can pass an ndarray to the ambiguous parameter to set the DST explicitly

```
>>> s = pd.Series(range(3),
...               index=pd.DatetimeIndex(['2018-10-28 01:20:00',
...                                       '2018-10-28 02:36:00',
...                                       '2018-10-28 03:46:00']))
>>> s.tz_localize('CET', ambiguous=np.array([True, True, False]))
2018-10-28 01:20:00+02:00    0
2018-10-28 02:36:00+02:00    1
2018-10-28 03:46:00+01:00    2
dtype: int64
```

If the DST transition causes nonexistent times, you can shift these dates forward or backward with a timedelta object or *'shift_forward'* or *'shift_backward'*.

```
>>> s = pd.Series(range(2),
...               index=pd.DatetimeIndex(['2015-03-29 02:30:00',
...                                       '2015-03-29 03:30:00']))
>>> s.tz_localize('Europe/Warsaw', nonexistent='shift_forward')
2015-03-29 03:00:00+02:00    0
2015-03-29 03:30:00+02:00    1
dtype: int64
>>> s.tz_localize('Europe/Warsaw', nonexistent='shift_backward')
2015-03-29 01:59:59.999999999+01:00    0
2015-03-29 03:30:00+02:00              1
```

(continues on next page)

(continued from previous page)

```
dtype: int64
>>> s.tz_localize('Europe/Warsaw', nonexistent=pd.Timedelta('1H'))
2015-03-29 03:30:00+02:00    0
2015-03-29 03:30:00+02:00    1
dtype: int64
```

AlloViz.AlloViz.Elements.Element.unstack

Element.**unstack**(*level: IndexLabel = -1*, *fill_value=None*, *sort: bool = True*)

Pivot a level of the (necessarily hierarchical) index labels.

Returns a DataFrame having a new level of column labels whose inner-most level consists of the pivoted index labels.

If the index is not a MultiIndex, the output will be a Series (the analogue of stack when the columns are not a MultiIndex).

Parameters

level
[int, str, or list of these, default -1 (last level)] Level(s) of index to unstack, can pass level name.

fill_value
[int, str or dict] Replace NaN with this value if the unstack produces missing values.

sort
[bool, default True] Sort the level(s) in the resulting MultiIndex columns.

Returns

Series or DataFrame

See also:

`DataFrame.pivot`
Pivot a table based on column values.

`DataFrame.stack`
Pivot a level of the column labels (inverse operation from *unstack*).

Notes

Reference the user guide for more examples.

Examples

```
>>> index = pd.MultiIndex.from_tuples([('one', 'a'), ('one', 'b'),
...                                    ('two', 'a'), ('two', 'b')])
>>> s = pd.Series(np.arange(1.0, 5.0), index=index)
>>> s
one  a   1.0
     b   2.0
two  a   3.0
     b   4.0
dtype: float64
```

```
>>> s.unstack(level=-1)
     a   b
one  1.0  2.0
two  3.0  4.0
```

```
>>> s.unstack(level=0)
   one  two
a  1.0   3.0
b  2.0   4.0
```

```
>>> df = s.unstack(level=0)
>>> df.unstack()
one  a  1.0
     b  2.0
two  a  3.0
     b  4.0
dtype: float64
```

AlloViz.AlloViz.Elements.Element.update

Element.**update**(*other*, *join: UpdateJoin = 'left'*, *overwrite: bool = True*, *filter_func=None*, *errors: IgnoreRaise = 'ignore'*) → None

Modify in place using non-NA values from another DataFrame.

Aligns on indices. There is no return value.

Parameters

other
: [DataFrame, or object coercible into a DataFrame] Should have at least one matching index/column label with the original DataFrame. If a Series is passed, its name attribute must be set, and that will be used as the column name to align with the original DataFrame.

join
: [{'left'}, default 'left'] Only left join is implemented, keeping the index and columns of the original object.

overwrite
: [bool, default True] How to handle non-NA values for overlapping keys:

- True: overwrite original DataFrame's values with values from *other*.

- False: only update values that are NA in the original DataFrame.

filter_func
[callable(1d-array) -> bool 1d-array, optional] Can choose to replace values other than NA. Return True for values that should be updated.

errors
[{'raise', 'ignore'}, default 'ignore'] If 'raise', will raise a ValueError if the DataFrame and *other* both contain non-NA data in the same place.

Returns

None
This method directly changes calling object.

Raises

ValueError

- When *errors='raise'* and there's overlapping non-NA data.
- When *errors* is not either *'ignore'* or *'raise'*

NotImplementedError

- If *join != 'left'*

See also:

`dict.update`
Similar method for dictionaries.

`DataFrame.merge`
For column(s)-on-column(s) operations.

Examples

```
>>> df = pd.DataFrame({'A': [1, 2, 3],
...                    'B': [400, 500, 600]})
>>> new_df = pd.DataFrame({'B': [4, 5, 6],
...                        'C': [7, 8, 9]})
>>> df.update(new_df)
>>> df
   A  B
0  1  4
1  2  5
2  3  6
```

The DataFrame's length does not increase as a result of the update, only values at matching index/column labels are updated.

```
>>> df = pd.DataFrame({'A': ['a', 'b', 'c'],
...                    'B': ['x', 'y', 'z']})
>>> new_df = pd.DataFrame({'B': ['d', 'e', 'f', 'g', 'h', 'i']})
>>> df.update(new_df)
>>> df
   A  B
0  a  d
```

(continues on next page)

(continued from previous page)

```
1  b  e
2  c  f
```

For Series, its name attribute must be set.

```
>>> df = pd.DataFrame({'A': ['a', 'b', 'c'],
...                    'B': ['x', 'y', 'z']})
>>> new_column = pd.Series(['d', 'e'], name='B', index=[0, 2])
>>> df.update(new_column)
>>> df
   A  B
0  a  d
1  b  y
2  c  e
>>> df = pd.DataFrame({'A': ['a', 'b', 'c'],
...                    'B': ['x', 'y', 'z']})
>>> new_df = pd.DataFrame({'B': ['d', 'e']}, index=[1, 2])
>>> df.update(new_df)
>>> df
   A  B
0  a  x
1  b  d
2  c  e
```

If *other* contains NaNs the corresponding values are not updated in the original dataframe.

```
>>> df = pd.DataFrame({'A': [1, 2, 3],
...                    'B': [400, 500, 600]})
>>> new_df = pd.DataFrame({'B': [4, np.nan, 6]})
>>> df.update(new_df)
>>> df
   A    B
0  1    4
1  2  500
2  3    6
```

AlloViz.AlloViz.Elements.Element.value_counts

Element.**value_counts**(*subset: IndexLabel | None = None*, *normalize: bool = False*, *sort: bool = True*, *ascending: bool = False*, *dropna: bool = True*) → Series

Return a Series containing the frequency of each distinct row in the Dataframe.

Parameters

subset
[label or list of labels, optional] Columns to use when counting unique combinations.

normalize
[bool, default False] Return proportions rather than frequencies.

sort
[bool, default True] Sort by frequencies when True. Sort by DataFrame column

values when False.

ascending
[bool, default False] Sort in ascending order.

dropna
[bool, default True] Don't include counts of rows that contain NA values.

New in version 1.3.0.

Returns

Series

See also:

Series.value_counts
Equivalent method on Series.

Notes

The returned Series will have a MultiIndex with one level per input column but an Index (non-multi) for a single label. By default, rows that contain any NA values are omitted from the result. By default, the resulting Series will be in descending order so that the first element is the most frequently-occurring row.

Examples

```
>>> df = pd.DataFrame({'num_legs': [2, 4, 4, 6],
...                    'num_wings': [2, 0, 0, 0]},
...                   index=['falcon', 'dog', 'cat', 'ant'])
>>> df
        num_legs  num_wings
falcon         2          2
dog            4          0
cat            4          0
ant            6          0
```

```
>>> df.value_counts()
num_legs  num_wings
4         0            2
2         2            1
6         0            1
Name: count, dtype: int64
```

```
>>> df.value_counts(sort=False)
num_legs  num_wings
2         2            1
4         0            2
6         0            1
Name: count, dtype: int64
```

```
>>> df.value_counts(ascending=True)
num_legs  num_wings
2         2            1
```

(continues on next page)

(continued from previous page)

```
6         0            1
4         0            2
Name: count, dtype: int64
```

```
>>> df.value_counts(normalize=True)
num_legs  num_wings
4         0            0.50
2         2            0.25
6         0            0.25
Name: proportion, dtype: float64
```

With *dropna* set to *False* we can also count rows with NA values.

```
>>> df = pd.DataFrame({'first_name': ['John', 'Anne', 'John', 'Beth'],
...                    'middle_name': ['Smith', pd.NA, pd.NA, 'Louise']})
>>> df
  first_name middle_name
0       John       Smith
1       Anne        <NA>
2       John        <NA>
3       Beth      Louise
```

```
>>> df.value_counts()
first_name  middle_name
Beth        Louise         1
John        Smith          1
Name: count, dtype: int64
```

```
>>> df.value_counts(dropna=False)
first_name  middle_name
Anne        NaN            1
Beth        Louise         1
John        Smith          1
            NaN            1
Name: count, dtype: int64
```

```
>>> df.value_counts("first_name")
first_name
John    2
Anne    1
Beth    1
Name: count, dtype: int64
```

AlloViz.AlloViz.Elements.Element.var

Element.**var**(*axis: Axis | None = 0*, *skipna: bool = True*, *ddof: int = 1*, *numeric_only: bool = False*, ***kwargs*)

Return unbiased variance over requested axis.

Normalized by N-1 by default. This can be changed using the ddof argument.

Parameters

axis
[{index (0), columns (1)}] For *Series* this parameter is unused and defaults to 0.

skipna
[bool, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

ddof
[int, default 1] Delta Degrees of Freedom. The divisor used in calculations is N - ddof, where N represents the number of elements.

numeric_only
[bool, default False] Include only float, int, boolean columns. Not implemented for Series.

Returns

Series or DataFrame (if level specified)

Examples

```
>>> df = pd.DataFrame({'person_id': [0, 1, 2, 3],
...                    'age': [21, 25, 62, 43],
...                    'height': [1.61, 1.87, 1.49, 2.01]}
...                   ).set_index('person_id')
>>> df
           age  height
person_id
0           21    1.61
1           25    1.87
2           62    1.49
3           43    2.01
```

```
>>> df.var()
age       352.916667
height      0.056367
dtype: float64
```

Alternatively, `ddof=0` can be set to normalize by N instead of N-1:

```
>>> df.var(ddof=0)
age       264.687500
height      0.042275
dtype: float64
```

AlloViz.AlloViz.Elements.Element.view

Element.**view**(*metric*, *num=20*, *colors=['orange', 'turquoise']*, *nv=None*)

Represent the selected metric in the structure

Retrieves the analyzed data corresponding to the present Element (depending on the class) and from it the corresponding metric column from the DataFrame. It is used to obtain the elements' colors, sizes (inv. proportional to errors, if available) and names (resnames); and the parent instance attribute is used to retrieve the structure for representation using `nglview.NGLWidget`.

Data is sorted according to the selected metric in absolute value and descending order, a `LinearSegmentedColormap` is made with the passed colors. The colormap is represented in a colorbar through `_show_cbar()` and is used to establish the elements' colors through `_get_colors()`.

Errors, if available (i.e., if more than one trajectory has been used and averages were calculated), are used to establish the elements' sizes to be inversely proportional to them (interpolated between 1 and 0.1), thus directly proportional to the "confidence" in the calculated value in a way.

Elements are shown on a representation of the selected structure or added to the passed `NGLWidget` if applicable.

Parameters

metric
[str] Metric/Name of the column in the object's df attribute to represent.

num
[int, default: 20] Number of (each of the) network elements to show on the structure.

colors
[list, default: ["orange", "turquoise"]] List of two colors to assign to the minimum and maximum values of the network to be represented, respectively. Middle value is assigned "white" and it will be the mean of the network values or 0 if the network has both negative and positive values.

nv
[`nglview.NGLWidget`, optional] A structure representation into which the shapes representing the chosen network elements will be added.

See also:

AlloViz.Protein.view

AlloViz.AlloViz.Elements.Element.where

Element.**where**(*cond*, *other=nan*, ***, *inplace: bool_t = False*, *axis: Axis | None = None*, *level: Level | None = None*) → Self | None

Replace values where the condition is False.

Parameters

cond
[bool Series/DataFrame, array-like, or callable] Where *cond* is True, keep the original value. Where False, replace with corresponding value from *other*. If *cond* is callable, it is computed on the Series/DataFrame and should return boolean Series/DataFrame or array. The callable must not change input Series/DataFrame (though pandas doesn't check it).

other
[scalar, Series/DataFrame, or callable] Entries where *cond* is False are replaced with corresponding value from *other*. If other is callable, it is computed on the Series/DataFrame and should return scalar or Series/DataFrame. The callable must not change input Series/DataFrame (though pandas doesn't check it). If not specified, entries will be filled with the corresponding NULL value (`np.nan` for numpy dtypes, `pd.NA` for extension dtypes).

inplace
[bool, default False] Whether to perform the operation in place on the data.

axis
[int, default None] Alignment axis if needed. For *Series* this parameter is unused and defaults to 0.

level
[int, default None] Alignment level if needed.

Returns

Same type as caller or None if `inplace=True`.

See also:

`DataFrame.mask()`
Return an object of same shape as self.

Notes

The where method is an application of the if-then idiom. For each element in the calling DataFrame, if `cond` is `True` the element is used; otherwise the corresponding element from the DataFrame `other` is used. If the axis of `other` does not align with axis of `cond` Series/DataFrame, the misaligned index positions will be filled with False.

The signature for `DataFrame.where()` differs from `numpy.where()`. Roughly `df1.where(m, df2)` is equivalent to `np.where(m, df1, df2)`.

For further details and examples see the `where` documentation in indexing.

The dtype of the object takes precedence. The fill value is casted to the object's dtype, if this can be done losslessly.

Examples

```
>>> s = pd.Series(range(5))
>>> s.where(s > 0)
0    NaN
1    1.0
2    2.0
3    3.0
4    4.0
dtype: float64
>>> s.mask(s > 0)
0    0.0
1    NaN
2    NaN
```

(continues on next page)

(continued from previous page)

```
3    NaN
4    NaN
dtype: float64
```

```
>>> s = pd.Series(range(5))
>>> t = pd.Series([True, False])
>>> s.where(t, 99)
0     0
1    99
2    99
3    99
4    99
dtype: int64
>>> s.mask(t, 99)
0    99
1     1
2    99
3    99
4    99
dtype: int64
```

```
>>> s.where(s > 1, 10)
0    10
1    10
2    2
3    3
4    4
dtype: int64
>>> s.mask(s > 1, 10)
0     0
1     1
2    10
3    10
4    10
dtype: int64
```

```
>>> df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])
>>> df
   A  B
0  0  1
1  2  3
2  4  5
3  6  7
4  8  9
>>> m = df % 3 == 0
>>> df.where(m, -df)
   A  B
0  0 -1
1 -2  3
2 -4 -5
3  6 -7
```

(continues on next page)

(continued from previous page)

```
4 -8  9
>>> df.where(m, -df) == np.where(m, df, -df)
      A     B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True
>>> df.where(m, -df) == df.mask(~m, -df)
      A     B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True
```

AlloViz.AlloViz.Elements.Element.xs

Element.**xs**(*key: Hashable | Sequence[Hashable]*, *axis: int | Literal['index', 'columns', 'rows'] = 0*, *level: Hashable | Sequence[Hashable] | None = None*, *drop_level: bool = True*) → None

Return cross-section from the Series/DataFrame.

This method takes a *key* argument to select data at a particular level of a MultiIndex.

Parameters

key
: [label or tuple of label] Label contained in the index, or partially in a MultiIndex.

axis
: [{0 or 'index', 1 or 'columns'}, default 0] Axis to retrieve cross-section on.

level
: [object, defaults to first n levels (n=1 or len(key))] In case of a key partially contained in a MultiIndex, indicate which levels are used. Levels can be referred by label or position.

drop_level
: [bool, default True] If False, returns object with same levels as self.

Returns

Series or DataFrame
: Cross-section from the original Series or DataFrame corresponding to the selected index levels.

See also:

`DataFrame.loc`
: Access a group of rows and columns by label(s) or a boolean array.

`DataFrame.iloc`
: Purely integer-location based indexing for selection by position.

Notes

xs can not be used to set values.

MultiIndex Slicers is a generic way to get/set values on any level or levels. It is a superset of *xs* functionality, see MultiIndex Slicers.

Examples

```
>>> d = {'num_legs': [4, 4, 2, 2],
...      'num_wings': [0, 0, 2, 2],
...      'class': ['mammal', 'mammal', 'mammal', 'bird'],
...      'animal': ['cat', 'dog', 'bat', 'penguin'],
...      'locomotion': ['walks', 'walks', 'flies', 'walks']}
>>> df = pd.DataFrame(data=d)
>>> df = df.set_index(['class', 'animal', 'locomotion'])
>>> df
                           num_legs  num_wings
class  animal  locomotion
mammal cat     walks              4          0
       dog     walks              4          0
       bat     flies              2          2
bird   penguin walks              2          2
```

Get values at specified index

```
>>> df.xs('mammal')
                    num_legs  num_wings
animal locomotion
cat    walks              4          0
dog    walks              4          0
bat    flies              2          2
```

Get values at several indexes

```
>>> df.xs(('mammal', 'dog', 'walks'))
num_legs     4
num_wings    0
Name: (mammal, dog, walks), dtype: int64
```

Get values at specified index and level

```
>>> df.xs('cat', level=1)
                    num_legs  num_wings
class  locomotion
mammal walks              4          0
```

Get values at several indexes and levels

```
>>> df.xs(('bird', 'walks'),
...        level=[0, 'locomotion'])
         num_legs  num_wings
animal
penguin         2          2
```

Get values at specified column and axis

```
>>> df.xs('num_wings', axis=1)
class   animal   locomotion
mammal  cat      walks         0
        dog      walks         0
        bat      flies         2
bird    penguin  walks         2
Name: num_wings, dtype: int64
```

AlloViz.AlloViz.Elements.Nodes

class AlloViz.AlloViz.Elements.**Nodes**(*data*, *parent*, *index=None*, *columns=None*, *dtype=None*, *copy=True*)

Bases: *Element*

Class for storage and viz of Nodes

See this class' `_add_element()`

Attributes

T
The transpose of the DataFrame.

at
Access a single value for a row/column label pair.

attrs
Dictionary of global attributes of this dataset.

axes
Return a list representing the axes of the DataFrame.

columns
The column labels of the DataFrame.

```
>>> df = pd.DataFrame({'A': [1, 2], 'B': [3, 4]})
>>> df
     A  B
0    1  3
1    2  4
>>> df.columns
Index(['A', 'B'], dtype='object')
```

dtypes
Return the dtypes in the DataFrame.

empty
Indicator whether Series/DataFrame is empty.

flags
Get the properties associated with this pandas object.

iat
Access a single value for a row/column pair by integer position.

iloc
Purely integer-location based indexing for selection by position.

index
The index (row labels) of the DataFrame.

The index of a DataFrame is a series of labels that identify each row. The labels can be integers, strings, or any other hashable type. The index is used for label-based access and alignment, and can be accessed or modified using this attribute.

pandas.Index
The index labels of the DataFrame.

DataFrame.columns : The column labels of the DataFrame. DataFrame.to_numpy : Convert the DataFrame to a NumPy array.

```
>>> df = pd.DataFrame({'Name': ['Alice', 'Bob', 'Aritra'],
...                    'Age': [25, 30, 35],
...                    'Location': ['Seattle', 'New York', 'Kona
↪']},
...                   index=([10, 20, 30]))
>>> df.index
Index([10, 20, 30], dtype='int64')
```

In this example, we create a DataFrame with 3 rows and 3 columns, including Name, Age, and Location information. We set the index labels to be the integers 10, 20, and 30. We then access the *index* attribute of the DataFrame, which returns an *Index* object containing the index labels.

```
>>> df.index = [100, 200, 300]
>>> df
    Name  Age Location
100  Alice   25  Seattle
200    Bob   30 New York
300  Aritra  35    Kona
```

In this example, we modify the index labels of the DataFrame by assigning a new list of labels to the *index* attribute. The DataFrame is then updated with the new labels, and the output shows the modified DataFrame.

loc
Access a group of rows and columns by label(s) or a boolean array.

ndim
Return an int representing the number of axes / array dimensions.

shape
Return a tuple representing the dimensionality of the DataFrame.

size
Return an int representing the number of elements in this object.

style
Returns a Styler object.

values
Return a Numpy representation of the DataFrame.

Methods

abs()	Return a Series/DataFrame with absolute numeric value of each element.
add(other[, axis, level, fill_value])	Get Addition of dataframe and other, element-wise (binary operator *add*).
add_prefix(prefix[, axis])	Prefix labels with string *prefix*.
add_suffix(suffix[, axis])	Suffix labels with string *suffix*.
agg([func, axis])	Aggregate using one or more operations over the specified axis.
aggregate([func, axis])	Aggregate using one or more operations over the specified axis.
align(other[, join, axis, level, copy, ...])	Align two objects on their axes with the specified join method.
all([axis, bool_only, skipna])	Return whether all elements are True, potentially over an axis.
any(*[, axis, bool_only, skipna])	Return whether any element is True, potentially over an axis.
apply(func[, axis, raw, result_type, args, ...])	Apply a function along an axis of the DataFrame.
applymap(func[, na_action])	Apply a function to a Dataframe elementwise.
asfreq(freq[, method, how, normalize, ...])	Convert time series to specified frequency.
asof(where[, subset])	Return the last row(s) without any NaNs before *where*.
assign(**kwargs)	Assign new columns to a DataFrame.
astype(dtype[, copy, errors])	Cast a pandas object to a specified dtype `dtype`.
at_time(time[, asof, axis])	Select values at particular time of day (e.g., 9:30AM).
backfill(*[, axis, inplace, limit, downcast])	Fill NA/NaN values by using the next valid observation to fill the gap.
between_time(start_time, end_time[, ...])	Select values between particular times of the day (e.g., 9:00-9:30 AM).
bfill(*[, axis, inplace, limit, downcast])	Fill NA/NaN values by using the next valid observation to fill the gap.
bool()	Return the bool of a single element Series or DataFrame.
boxplot([column, by, ax, fontsize, rot, ...])	Make a box plot from DataFrame columns.
clip([lower, upper, axis, inplace])	Trim values at input threshold(s).
combine(other, func[, fill_value, overwrite])	Perform column-wise combine with another DataFrame.
combine_first(other)	Update null elements with value in the same location in *other*.
compare(other[, align_axis, keep_shape, ...])	Compare to another DataFrame and show the differences.
convert_dtypes([infer_objects, ...])	Convert columns to the best possible dtypes using dtypes supporting `pd.NA`.
copy([deep])	Make a copy of this object's indices and data.
corr([method, min_periods, numeric_only])	Compute pairwise correlation of columns, excluding NA/null values.
corrwith(other[, axis, drop, method, ...])	Compute pairwise correlation.
count([axis, numeric_only])	Count non-NA cells for each column or row.
cov([min_periods, ddof, numeric_only])	Compute pairwise covariance of columns, excluding NA/null values.

continues on next page

Table 3 – continued from previous page

cummax([axis, skipna])	Return cumulative maximum over a DataFrame or Series axis.
cummin([axis, skipna])	Return cumulative minimum over a DataFrame or Series axis.
cumprod([axis, skipna])	Return cumulative product over a DataFrame or Series axis.
cumsum([axis, skipna])	Return cumulative sum over a DataFrame or Series axis.
describe([percentiles, include, exclude])	Generate descriptive statistics.
diff([periods, axis])	First discrete difference of element.
div(other[, axis, level, fill_value])	Get Floating division of dataframe and other, element-wise (binary operator *truediv*).
divide(other[, axis, level, fill_value])	Get Floating division of dataframe and other, element-wise (binary operator *truediv*).
dot(other)	Compute the matrix multiplication between the DataFrame and other.
drop([labels, axis, index, columns, level, ...])	Drop specified labels from rows or columns.
drop_duplicates([subset, keep, inplace, ...])	Return DataFrame with duplicate rows removed.
droplevel(level[, axis])	Return Series/DataFrame with requested index / column level(s) removed.
dropna(*[, axis, how, thresh, subset, ...])	Remove missing values.
duplicated([subset, keep])	Return boolean Series denoting duplicate rows.
eq(other[, axis, level])	Get Equal to of dataframe and other, element-wise (binary operator *eq*).
equals(other)	Test whether two objects contain the same elements.
eval(expr, *[, inplace])	Evaluate a string describing operations on DataFrame columns.
ewm([com, span, halflife, alpha, ...])	Provide exponentially weighted (EW) calculations.
expanding([min_periods, axis, method])	Provide expanding window calculations.
explode(column[, ignore_index])	Transform each element of a list-like to a row, replicating index values.
ffill(*[, axis, inplace, limit, downcast])	Fill NA/NaN values by propagating the last valid observation to next valid.
fillna([value, method, axis, inplace, ...])	Fill NA/NaN values using the specified method.
filter([items, like, regex, axis])	Subset the dataframe rows or columns according to the specified index labels.
first(offset)	Select initial periods of time series data based on a date offset.
first_valid_index()	Return index for first non-NA value or None, if no non-NA value is found.
floordiv(other[, axis, level, fill_value])	Get Integer division of dataframe and other, element-wise (binary operator *floordiv*).
from_dict(data[, orient, dtype, columns])	Construct DataFrame from dict of array-like or dicts.
from_records(data[, index, exclude, ...])	Convert structured or record ndarray to DataFrame.
ge(other[, axis, level])	Get Greater than or equal to of dataframe and other, element-wise (binary operator *ge*).
get(key[, default])	Get item from object for given key (ex: DataFrame column).
groupby([by, axis, level, as_index, sort, ...])	Group DataFrame using a mapper or by a Series of columns.
gt(other[, axis, level])	Get Greater than of dataframe and other, element-wise (binary operator *gt*).

continues on next page

Table 3 – continued from previous page

head([n])	Return the first *n* rows.
hist([column, by, grid, xlabelsize, xrot, ...])	Make a histogram of the DataFrame's columns.
idxmax([axis, skipna, numeric_only])	Return index of first occurrence of maximum over requested axis.
idxmin([axis, skipna, numeric_only])	Return index of first occurrence of minimum over requested axis.
infer_objects([copy])	Attempt to infer better dtypes for object columns.
info([verbose, buf, max_cols, memory_usage, ...])	Print a concise summary of a DataFrame.
insert(loc, column, value[, allow_duplicates])	Insert column into DataFrame at specified location.
interpolate([method, axis, limit, inplace, ...])	Fill NaN values using an interpolation method.
isetitem(loc, value)	Set the given value in the column with position *loc*.
isin(values)	Whether each element in the DataFrame is contained in values.
isna()	Detect missing values.
isnull()	DataFrame.isnull is an alias for DataFrame.isna.
items()	Iterate over (column name, Series) pairs.
iterrows()	Iterate over DataFrame rows as (index, Series) pairs.
itertuples([index, name])	Iterate over DataFrame rows as namedtuples.
join(other[, on, how, lsuffix, rsuffix, ...])	Join columns of another DataFrame.
keys()	Get the 'info axis' (see Indexing for more).
kurt([axis, skipna, numeric_only])	Return unbiased kurtosis over requested axis.
kurtosis([axis, skipna, numeric_only])	Return unbiased kurtosis over requested axis.
last(offset)	Select final periods of time series data based on a date offset.
last_valid_index()	Return index for last non-NA value or None, if no non-NA value is found.
le(other[, axis, level])	Get Less than or equal to of dataframe and other, element-wise (binary operator *le*).
lt(other[, axis, level])	Get Less than of dataframe and other, element-wise (binary operator *lt*).
map(func[, na_action])	Apply a function to a Dataframe elementwise.
mask(cond[, other, inplace, axis, level])	Replace values where the condition is True.
max([axis, skipna, numeric_only])	Return the maximum of the values over the requested axis.
mean([axis, skipna, numeric_only])	Return the mean of the values over the requested axis.
median([axis, skipna, numeric_only])	Return the median of the values over the requested axis.
melt([id_vars, value_vars, var_name, ...])	Unpivot a DataFrame from wide to long format, optionally leaving identifiers set.
memory_usage([index, deep])	Return the memory usage of each column in bytes.
merge(right[, how, on, left_on, right_on, ...])	Merge DataFrame or named Series objects with a database-style join.
min([axis, skipna, numeric_only])	Return the minimum of the values over the requested axis.
mod(other[, axis, level, fill_value])	Get Modulo of dataframe and other, element-wise (binary operator *mod*).
mode([axis, numeric_only, dropna])	Get the mode(s) of each element along the selected axis.
mul(other[, axis, level, fill_value])	Get Multiplication of dataframe and other, element-wise (binary operator *mul*).
multiply(other[, axis, level, fill_value])	Get Multiplication of dataframe and other, element-wise (binary operator *mul*).

continues on next page

Table 3 – continued from previous page

ne(other[, axis, level])	Get Not equal to of dataframe and other, element-wise (binary operator *ne*).
nlargest(n, columns[, keep])	Return the first *n* rows ordered by *columns* in descending order.
notna()	Detect existing (non-missing) values.
notnull()	DataFrame.notnull is an alias for DataFrame.notna.
nsmallest(n, columns[, keep])	Return the first *n* rows ordered by *columns* in ascending order.
nunique([axis, dropna])	Count number of distinct elements in specified axis.
pad(*[, axis, inplace, limit, downcast])	Fill NA/NaN values by propagating the last valid observation to next valid.
pct_change([periods, fill_method, limit, freq])	Fractional change between the current and a prior element.
pipe(func, *args, **kwargs)	Apply chainable functions that expect Series or DataFrames.
pivot(*, columns[, index, values])	Return reshaped DataFrame organized by given index / column values.
pivot_table([values, index, columns, ...])	Create a spreadsheet-style pivot table as a DataFrame.
plot	alias of PlotAccessor
pop(item)	Return item and drop from frame.
pow(other[, axis, level, fill_value])	Get Exponential power of dataframe and other, element-wise (binary operator *pow*).
prod([axis, skipna, numeric_only, min_count])	Return the product of the values over the requested axis.
product([axis, skipna, numeric_only, min_count])	Return the product of the values over the requested axis.
quantile([q, axis, numeric_only, ...])	Return values at the given quantile over requested axis.
query(expr, *[, inplace])	Query the columns of a DataFrame with a boolean expression.
radd(other[, axis, level, fill_value])	Get Addition of dataframe and other, element-wise (binary operator *radd*).
rank([axis, method, numeric_only, ...])	Compute numerical data ranks (1 through n) along axis.
rdiv(other[, axis, level, fill_value])	Get Floating division of dataframe and other, element-wise (binary operator *rtruediv*).
reindex([labels, index, columns, axis, ...])	Conform DataFrame to new index with optional filling logic.
reindex_like(other[, method, copy, limit, ...])	Return an object with matching indices as other object.
rename([mapper, index, columns, axis, copy, ...])	Rename columns or index labels.
rename_axis([mapper, index, columns, axis, ...])	Set the name of the axis for the index or columns.
reorder_levels(order[, axis])	Rearrange index levels using input order.
replace([to_replace, value, inplace, limit, ...])	Replace values given in *to_replace* with *value*.
resample(rule[, axis, closed, label, ...])	Resample time-series data.
reset_index([level, drop, inplace, ...])	Reset the index, or a level of it.
rfloordiv(other[, axis, level, fill_value])	Get Integer division of dataframe and other, element-wise (binary operator *rfloordiv*).
rmod(other[, axis, level, fill_value])	Get Modulo of dataframe and other, element-wise (binary operator *rmod*).
rmul(other[, axis, level, fill_value])	Get Multiplication of dataframe and other, element-wise (binary operator *rmul*).

continues on next page

Table 3 – continued from previous page

rolling(window[, min_periods, center, ...])	Provide rolling window calculations.
round([decimals])	Round a DataFrame to a variable number of decimal places.
rpow(other[, axis, level, fill_value])	Get Exponential power of dataframe and other, element-wise (binary operator *rpow*).
rsub(other[, axis, level, fill_value])	Get Subtraction of dataframe and other, element-wise (binary operator *rsub*).
rtruediv(other[, axis, level, fill_value])	Get Floating division of dataframe and other, element-wise (binary operator *rtruediv*).
sample([n, frac, replace, weights, ...])	Return a random sample of items from an axis of object.
select_dtypes([include, exclude])	Return a subset of the DataFrame's columns based on the column dtypes.
sem([axis, skipna, ddof, numeric_only])	Return unbiased standard error of the mean over requested axis.
set_axis(labels, *[, axis, copy])	Assign desired index to given axis.
set_flags(*[, copy, allows_duplicate_labels])	Return a new object with updated flags.
set_index(keys, *[, drop, append, inplace, ...])	Set the DataFrame index using existing columns.
shift([periods, freq, axis, fill_value, suffix])	Shift index by desired number of periods with an optional time *freq*.
skew([axis, skipna, numeric_only])	Return unbiased skew over requested axis.
sort_index(*[, axis, level, ascending, ...])	Sort object by labels (along an axis).
sort_values(by, *[, axis, ascending, ...])	Sort by the values along either axis.
sparse	alias of `SparseFrameAccessor`
squeeze([axis])	Squeeze 1 dimensional axis objects into scalars.
stack([level, dropna, sort, future_stack])	Stack the prescribed level(s) from columns to index.
std([axis, skipna, ddof, numeric_only])	Return sample standard deviation over requested axis.
sub(other[, axis, level, fill_value])	Get Subtraction of dataframe and other, element-wise (binary operator *sub*).
subtract(other[, axis, level, fill_value])	Get Subtraction of dataframe and other, element-wise (binary operator *sub*).
sum([axis, skipna, numeric_only, min_count])	Return the sum of the values over the requested axis.
swapaxes(axis1, axis2[, copy])	Interchange axes and swap values axes appropriately.
swaplevel([i, j, axis])	Swap levels i and j in a `MultiIndex`.
tail([n])	Return the last *n* rows.
take(indices[, axis])	Return the elements in the given *positional* indices along an axis.
to_clipboard([excel, sep])	Copy object to the system clipboard.
to_csv([path_or_buf, sep, na_rep, ...])	Write object to a comma-separated values (csv) file.
to_dict([orient, into, index])	Convert the DataFrame to a dictionary.
to_excel(excel_writer[, sheet_name, na_rep, ...])	Write object to an Excel sheet.
to_feather(path, **kwargs)	Write a DataFrame to the binary Feather format.
to_gbq(destination_table[, project_id, ...])	Write a DataFrame to a Google BigQuery table.
to_hdf(path_or_buf, key[, mode, complevel, ...])	Write the contained data to an HDF5 file using HDFStore.
to_html([buf, columns, col_space, header, ...])	Render a DataFrame as an HTML table.
to_json([path_or_buf, orient, date_format, ...])	Convert the object to a JSON string.
to_latex([buf, columns, header, index, ...])	Render object to a LaTeX tabular, longtable, or nested table.
to_markdown([buf, mode, index, storage_options])	Print DataFrame in Markdown-friendly format.
to_numpy([dtype, copy, na_value])	Convert the DataFrame to a NumPy array.
to_orc([path, engine, index, engine_kwargs])	Write a DataFrame to the ORC format.

continues on next page

Table 3 – continued from previous page

to_parquet([path, engine, compression, ...])	Write a DataFrame to the binary parquet format.
to_period([freq, axis, copy])	Convert DataFrame from DatetimeIndex to PeriodIndex.
to_pickle(path[, compression, protocol, ...])	Pickle (serialize) object to file.
to_records([index, column_dtypes, index_dtypes])	Convert DataFrame to a NumPy record array.
to_sql(name, con, *[, schema, if_exists, ...])	Write records stored in a DataFrame to a SQL database.
to_stata(path, *[, convert_dates, ...])	Export DataFrame object to Stata dta format.
to_string([buf, columns, col_space, header, ...])	Render a DataFrame to a console-friendly tabular output.
to_timestamp([freq, how, axis, copy])	Cast to DatetimeIndex of timestamps, at *beginning* of period.
to_xarray()	Return an xarray object from the pandas object.
to_xml([path_or_buffer, index, root_name, ...])	Render a DataFrame to an XML document.
transform(func[, axis])	Call `func` on self producing a DataFrame with the same axis shape as self.
transpose(*args[, copy])	Transpose index and columns.
truediv(other[, axis, level, fill_value])	Get Floating division of dataframe and other, element-wise (binary operator *truediv*).
truncate([before, after, axis, copy])	Truncate a Series or DataFrame before and after some index value.
tz_convert(tz[, axis, level, copy])	Convert tz-aware axis to target time zone.
tz_localize(tz[, axis, level, copy, ...])	Localize tz-naive index of a Series or DataFrame to target time zone.
unstack([level, fill_value, sort])	Pivot a level of the (necessarily hierarchical) index labels.
update(other[, join, overwrite, ...])	Modify in place using non-NA values from another DataFrame.
value_counts([subset, normalize, sort, ...])	Return a Series containing the frequency of each distinct row in the Dataframe.
var([axis, skipna, ddof, numeric_only])	Return unbiased variance over requested axis.
view(metric[, num, colors, nv])	Represent the selected metric in the structure
where(cond[, other, inplace, axis, level])	Replace values where the condition is False.
xs(key[, axis, level, drop_level])	Return cross-section from the Series/DataFrame.

AlloViz.AlloViz.Elements.Nodes.abs

Nodes.**abs**() → None

Return a Series/DataFrame with absolute numeric value of each element.

This function only applies to elements that are all numeric.

Returns

abs
Series/DataFrame containing the absolute value of each element.

See also:

numpy.absolute
Calculate the absolute value element-wise.

Notes

For `complex` inputs, `1.2 + 1j`, the absolute value is $\sqrt{a^2 + b^2}$.

Examples

Absolute numeric values in a Series.

```
>>> s = pd.Series([-1.10, 2, -3.33, 4])
>>> s.abs()
0    1.10
1    2.00
2    3.33
3    4.00
dtype: float64
```

Absolute numeric values in a Series with complex numbers.

```
>>> s = pd.Series([1.2 + 1j])
>>> s.abs()
0    1.56205
dtype: float64
```

Absolute numeric values in a Series with a Timedelta element.

```
>>> s = pd.Series([pd.Timedelta('1 days')])
>>> s.abs()
0   1 days
dtype: timedelta64[ns]
```

Select rows with data closest to certain value using argsort (from StackOverflow).

```
>>> df = pd.DataFrame({
...     'a': [4, 5, 6, 7],
...     'b': [10, 20, 30, 40],
...     'c': [100, 50, -30, -50]
... })
>>> df
     a    b    c
0    4   10  100
1    5   20   50
2    6   30  -30
3    7   40  -50
>>> df.loc[(df.c - 43).abs().argsort()]
     a    b    c
1    5   20   50
0    4   10  100
2    6   30  -30
3    7   40  -50
```

AlloViz.AlloViz.Elements.Nodes.add

Nodes.**add**(*other*, *axis: Axis = 'columns'*, *level=None*, *fill_value=None*)

Get Addition of dataframe and other, element-wise (binary operator *add*).

Equivalent to `dataframe + other`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *radd*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *floordiv*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other
[scalar, sequence, Series, dict or DataFrame] Any single or multiple element data structure, or list-like object.

axis
[{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns. (1 or 'columns'). For Series input, axis to match Series index on.

level
[int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value
[float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame
Result of the arithmetic operation.

See also:

`DataFrame.add`
Add DataFrames.

`DataFrame.sub`
Subtract DataFrames.

`DataFrame.mul`
Multiply DataFrames.

`DataFrame.div`
Divide DataFrames (float division).

`DataFrame.truediv`
Divide DataFrames (float division).

`DataFrame.floordiv`
Divide DataFrames (integer division).

`DataFrame.mod`
Calculate modulo (remainder after division).

`DataFrame.pow`
Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...         axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a dictionary by axis.

```
>>> df.mul({'angles': 0, 'degrees': 2})
            angles  degrees
circle           0      720
triangle         0      360
rectangle        0      720
```

```
>>> df.mul({'circle': 0, 'triangle': 2, 'rectangle': 3}, axis='index')
            angles  degrees
circle           0        0
triangle         6      360
rectangle       12     1080
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

AlloViz.AlloViz.Elements.Nodes.add_prefix

Nodes.**add_prefix**(*prefix: str*, *axis: int | Literal['index', 'columns', 'rows'] | None = None*) → None

Prefix labels with string *prefix*.

For Series, the row labels are prefixed. For DataFrame, the column labels are prefixed.

Parameters

prefix
[str] The string to add before each label.

axis
[{0 or 'index', 1 or 'columns', None}, default None] Axis to add prefix on

New in version 2.0.0.

Returns

Series or DataFrame
New Series or DataFrame with updated labels.

See also:

Series.add_suffix
Suffix row labels with string *suffix*.

DataFrame.add_suffix
Suffix column labels with string *suffix*.

Examples

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s
0    1
1    2
2    3
3    4
dtype: int64
```

```
>>> s.add_prefix('item_')
item_0    1
item_1    2
item_2    3
item_3    4
dtype: int64
```

```
>>> df = pd.DataFrame({'A': [1, 2, 3, 4], 'B': [3, 4, 5, 6]})
>>> df
   A  B
0  1  3
1  2  4
2  3  5
3  4  6
```

```
>>> df.add_prefix('col_')
     col_A  col_B
0       1       3
1       2       4
2       3       5
3       4       6
```

AlloViz.AlloViz.Elements.Nodes.add_suffix

Nodes.**add_suffix**(*suffix: str*, *axis: int | Literal['index', 'columns', 'rows'] | None = None*) → None

Suffix labels with string *suffix*.

For Series, the row labels are suffixed. For DataFrame, the column labels are suffixed.

Parameters

suffix
[str] The string to add after each label.

axis
[{0 or 'index', 1 or 'columns', None}, default None] Axis to add suffix on

New in version 2.0.0.

Returns

Series or DataFrame
New Series or DataFrame with updated labels.

See also:

Series.add_prefix
Prefix row labels with string *prefix*.

DataFrame.add_prefix
Prefix column labels with string *prefix*.

Examples

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s
0    1
1    2
2    3
3    4
dtype: int64
```

```
>>> s.add_suffix('_item')
0_item    1
1_item    2
2_item    3
3_item    4
dtype: int64
```

```
>>> df = pd.DataFrame({'A': [1, 2, 3, 4], 'B': [3, 4, 5, 6]})
>>> df
   A  B
0  1  3
1  2  4
2  3  5
3  4  6
```

```
>>> df.add_suffix('_col')
     A_col  B_col
0       1       3
1       2       4
2       3       5
3       4       6
```

AlloViz.AlloViz.Elements.Nodes.agg

Nodes.**agg**(*func=None*, *axis: Axis = 0*, **args*, ***kwargs*)

Aggregate using one or more operations over the specified axis.

Parameters

func
[function, str, list or dict] Function to use for aggregating the data. If a function, must either work when passed a DataFrame or when passed to DataFrame.apply.

Accepted combinations are:

- function

- string function name
- list of functions and/or function names, e.g. `[np.sum, 'mean']`
- dict of axis labels -> functions, function names or list of such.

axis
[{0 or 'index', 1 or 'columns'}, default 0] If 0 or 'index': apply function to each column. If 1 or 'columns': apply function to each row.

***args**
Positional arguments to pass to *func*.

****kwargs**
Keyword arguments to pass to *func*.

Returns

scalar, Series or DataFrame
The return can be:

- scalar : when Series.agg is called with single function
- Series : when DataFrame.agg is called with a single function
- DataFrame : when DataFrame.agg is called with several functions

Return scalar, Series or DataFrame.

See also:

`DataFrame.apply`
Perform any type of operations.

`DataFrame.transform`
Perform transformation type operations.

`core.groupby.GroupBy`
Perform operations over groups.

`core.resample.Resampler`
Perform operations over resampled bins.

`core.window.Rolling`
Perform operations over rolling window.

`core.window.Expanding`
Perform operations over expanding window.

`core.window.ExponentialMovingWindow`
Perform operation over exponential weighted window.

Notes

The aggregation operations are always performed over an axis, either the index (default) or the column axis. This behavior is different from *numpy* aggregation functions (*mean*, *median*, *prod*, *sum*, *std*, *var*), where the default is to compute the aggregation of the flattened array, e.g., `numpy.mean(arr_2d)` as opposed to `numpy.mean(arr_2d, axis=0)`.

agg is an alias for *aggregate*. Use the alias.

Functions that mutate the passed object can produce unexpected behavior or errors and are not supported. See Mutating with User Defined Function (UDF) methods for more details.

A passed user-defined-function will be passed a Series for evaluation.

Examples

```
>>> df = pd.DataFrame([[1, 2, 3],
...                    [4, 5, 6],
...                    [7, 8, 9],
...                    [np.nan, np.nan, np.nan]],
...                   columns=['A', 'B', 'C'])
```

Aggregate these functions over the rows.

```
>>> df.agg(['sum', 'min'])
        A     B     C
sum  12.0  15.0  18.0
min   1.0   2.0   3.0
```

Different aggregations per column.

```
>>> df.agg({'A' : ['sum', 'min'], 'B' : ['min', 'max']})
        A    B
sum  12.0  NaN
min   1.0  2.0
max   NaN  8.0
```

Aggregate different functions over the columns and rename the index of the resulting DataFrame.

```
>>> df.agg(x=('A', 'max'), y=('B', 'min'), z=('C', 'mean'))
     A    B    C
x  7.0  NaN  NaN
y  NaN  2.0  NaN
z  NaN  NaN  6.0
```

Aggregate over the columns.

```
>>> df.agg("mean", axis="columns")
0    2.0
1    5.0
2    8.0
3    NaN
dtype: float64
```

AlloViz.AlloViz.Elements.Nodes.aggregate

`Nodes.`**`aggregate`**(*func=None*, *axis: Axis = 0*, **args*, ***kwargs*)

Aggregate using one or more operations over the specified axis.

Parameters

func

[function, str, list or dict] Function to use for aggregating the data. If a function, must either work when passed a DataFrame or when passed to DataFrame.apply.

Accepted combinations are:

- function
- string function name
- list of functions and/or function names, e.g. `[np.sum, 'mean']`
- dict of axis labels -> functions, function names or list of such.

axis
[{0 or 'index', 1 or 'columns'}, default 0] If 0 or 'index': apply function to each column. If 1 or 'columns': apply function to each row.

***args**
Positional arguments to pass to *func*.

****kwargs**
Keyword arguments to pass to *func*.

Returns

scalar, Series or DataFrame
The return can be:

- scalar : when Series.agg is called with single function
- Series : when DataFrame.agg is called with a single function
- DataFrame : when DataFrame.agg is called with several functions

Return scalar, Series or DataFrame.

See also:

`DataFrame.apply`
Perform any type of operations.

`DataFrame.transform`
Perform transformation type operations.

`core.groupby.GroupBy`
Perform operations over groups.

`core.resample.Resampler`
Perform operations over resampled bins.

`core.window.Rolling`
Perform operations over rolling window.

`core.window.Expanding`
Perform operations over expanding window.

`core.window.ExponentialMovingWindow`
Perform operation over exponential weighted window.

Notes

The aggregation operations are always performed over an axis, either the index (default) or the column axis. This behavior is different from *numpy* aggregation functions (*mean*, *median*, *prod*, *sum*, *std*, *var*), where the default is to compute the aggregation of the flattened array, e.g., `numpy.mean(arr_2d)` as opposed to `numpy.mean(arr_2d, axis=0)`.

agg is an alias for *aggregate*. Use the alias.

Functions that mutate the passed object can produce unexpected behavior or errors and are not supported. See Mutating with User Defined Function (UDF) methods for more details.

A passed user-defined-function will be passed a Series for evaluation.

Examples

```
>>> df = pd.DataFrame([[1, 2, 3],
...                    [4, 5, 6],
...                    [7, 8, 9],
...                    [np.nan, np.nan, np.nan]],
...                   columns=['A', 'B', 'C'])
```

Aggregate these functions over the rows.

```
>>> df.agg(['sum', 'min'])
        A     B     C
sum  12.0  15.0  18.0
min   1.0   2.0   3.0
```

Different aggregations per column.

```
>>> df.agg({'A' : ['sum', 'min'], 'B' : ['min', 'max']})
        A    B
sum  12.0  NaN
min   1.0  2.0
max   NaN  8.0
```

Aggregate different functions over the columns and rename the index of the resulting DataFrame.

```
>>> df.agg(x=('A', 'max'), y=('B', 'min'), z=('C', 'mean'))
     A    B    C
x  7.0  NaN  NaN
y  NaN  2.0  NaN
z  NaN  NaN  6.0
```

Aggregate over the columns.

```
>>> df.agg("mean", axis="columns")
0    2.0
1    5.0
2    8.0
3    NaN
dtype: float64
```

AlloViz.AlloViz.Elements.Nodes.align

Nodes.**align**(*other: NDFrameT*, *join: AlignJoin = 'outer'*, *axis: Axis | None = None*, *level: Level | None = None*, *copy: bool_t | None = None*, *fill_value: Hashable | None = None*, *method: FillnaOptions | None | lib.NoDefault = _NoDefault.no_default*, *limit: int | None | lib.NoDefault = _NoDefault.no_default*, *fill_axis: Axis | lib.NoDefault = _NoDefault.no_default*, *broadcast_axis: Axis | None | lib.NoDefault = _NoDefault.no_default*) → tuple[Self, NDFrameT]

Align two objects on their axes with the specified join method.

Join method is specified for each axis Index.

Parameters

other
[DataFrame or Series]

join
[{'outer', 'inner', 'left', 'right'}, default 'outer'] Type of alignment to be performed.

- left: use only keys from left frame, preserve key order.
- right: use only keys from right frame, preserve key order.
- outer: use union of keys from both frames, sort keys lexicographically.
- inner: use intersection of keys from both frames, preserve the order of the left keys.

axis
[allowed axis of the other object, default None] Align on index (0), columns (1), or both (None).

level
[int or level name, default None] Broadcast across a level, matching Index values on the passed MultiIndex level.

copy
[bool, default True] Always returns new objects. If copy=False and no reindexing is required then original objects are returned.

fill_value
[scalar, default np.nan] Value to use for missing values. Defaults to NaN, but can be any "compatible" value.

method
[{'backfill', 'bfill', 'pad', 'ffill', None}, default None] Method to use for filling holes in reindexed Series:

- pad / ffill: propagate last valid observation forward to next valid.
- backfill / bfill: use NEXT valid observation to fill gap.

Deprecated since version 2.1.

limit
[int, default None] If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled. Must be greater than 0 if not None.

Deprecated since version 2.1.

fill_axis
[{0 or 'index'} for Series, {0 or 'index', 1 or 'columns'} for DataFrame, default 0] Filling axis, method and limit.

Deprecated since version 2.1.

broadcast_axis
[{0 or 'index'} for Series, {0 or 'index', 1 or 'columns'} for DataFrame, default None] Broadcast values along this axis, if aligning two objects of different dimensions.

Deprecated since version 2.1.

Returns

tuple of (Series/DataFrame, type of other)
Aligned objects.

Examples

```
>>> df = pd.DataFrame(
...     [[1, 2, 3, 4], [6, 7, 8, 9]], columns=["D", "B", "E", "A"], index=[1,␣
↪2]
... )
>>> other = pd.DataFrame(
...     [[10, 20, 30, 40], [60, 70, 80, 90], [600, 700, 800, 900]],
...     columns=["A", "B", "C", "D"],
...     index=[2, 3, 4],
... )
>>> df
   D  B  E  A
1  1  2  3  4
2  6  7  8  9
>>> other
    A    B    C    D
2   10   20   30   40
3   60   70   80   90
4  600  700  800  900
```

Align on columns:

```
>>> left, right = df.align(other, join="outer", axis=1)
>>> left
   A  B   C  D  E
1  4  2 NaN  1  3
2  9  7 NaN  6  8
>>> right
    A    B    C    D   E
2   10   20   30   40 NaN
3   60   70   80   90 NaN
4  600  700  800  900 NaN
```

We can also align on the index:

```
>>> left, right = df.align(other, join="outer", axis=0)
>>> left
    D    B    E    A
1  1.0  2.0  3.0  4.0
2  6.0  7.0  8.0  9.0
3  NaN  NaN  NaN  NaN
4  NaN  NaN  NaN  NaN
>>> right
    A      B      C      D
1   NaN    NaN    NaN    NaN
2  10.0   20.0   30.0   40.0
3  60.0   70.0   80.0   90.0
4  600.0  700.0  800.0  900.0
```

Finally, the default *axis=None* will align on both index and columns:

```
>>> left, right = df.align(other, join="outer", axis=None)
>>> left
     A    B   C    D    E
1  4.0  2.0 NaN  1.0  3.0
2  9.0  7.0 NaN  6.0  8.0
3  NaN  NaN NaN  NaN  NaN
4  NaN  NaN NaN  NaN  NaN
>>> right
       A      B      C      D   E
1   NaN    NaN    NaN    NaN NaN
2  10.0   20.0   30.0   40.0 NaN
3  60.0   70.0   80.0   90.0 NaN
4  600.0  700.0  800.0  900.0 NaN
```

AlloViz.AlloViz.Elements.Nodes.all

`Nodes.all`(*axis: Axis = 0*, *bool_only: bool = False*, *skipna: bool = True*, ***kwargs*) → Series | bool

Return whether all elements are True, potentially over an axis.

Returns True unless there at least one element within a series or along a Dataframe axis that is False or equivalent (e.g. zero or empty).

Parameters

axis
[{0 or 'index', 1 or 'columns', None}, default 0] Indicate which axis or axes should be reduced. For *Series* this parameter is unused and defaults to 0.

- 0 / 'index' : reduce the index, return a Series whose index is the original column labels.
- 1 / 'columns' : reduce the columns, return a Series whose index is the original index.
- None : reduce all axes, return a scalar.

bool_only
[bool, default False] Include only boolean columns. Not implemented for Series.

skipna
[bool, default True] Exclude NA/null values. If the entire row/column is NA and skipna is True, then the result will be True, as for an empty row/column. If skipna is False, then NA are treated as True, because these are not equal to zero.

****kwargs**
[any, default None] Additional keywords have no effect but might be accepted for compatibility with NumPy.

Returns

Series or DataFrame
If level is specified, then, DataFrame is returned; otherwise, Series is returned.

See also:

`Series.all`
Return True if all elements are True.

`DataFrame.any`
Return True if one (or more) elements are True.

Examples

Series

```
>>> pd.Series([True, True]).all()
True
>>> pd.Series([True, False]).all()
False
>>> pd.Series([], dtype="float64").all()
True
>>> pd.Series([np.nan]).all()
True
>>> pd.Series([np.nan]).all(skipna=False)
True
```

DataFrames

Create a dataframe from a dictionary.

```
>>> df = pd.DataFrame({'col1': [True, True], 'col2': [True, False]})
>>> df
   col1   col2
0  True   True
1  True  False
```

Default behaviour checks if values in each column all return True.

```
>>> df.all()
col1     True
col2    False
dtype: bool
```

Specify `axis='columns'` to check if values in each row all return True.

```
>>> df.all(axis='columns')
0     True
1    False
dtype: bool
```

Or `axis=None` for whether every value is True.

```
>>> df.all(axis=None)
False
```

AlloViz.AlloViz.Elements.Nodes.any

Nodes.**any**(*, *axis: Axis = 0*, *bool_only: bool = False*, *skipna: bool = True*, ***kwargs*) → Series | bool

Return whether any element is True, potentially over an axis.

Returns False unless there is at least one element within a series or along a Dataframe axis that is True or equivalent (e.g. non-zero or non-empty).

Parameters

axis
[{0 or 'index', 1 or 'columns', None}, default 0] Indicate which axis or axes should be reduced. For *Series* this parameter is unused and defaults to 0.

- 0 / 'index' : reduce the index, return a Series whose index is the original column labels.
- 1 / 'columns' : reduce the columns, return a Series whose index is the original index.
- None : reduce all axes, return a scalar.

bool_only
[bool, default False] Include only boolean columns. Not implemented for Series.

skipna
[bool, default True] Exclude NA/null values. If the entire row/column is NA and skipna is True, then the result will be False, as for an empty row/column. If skipna is False, then NA are treated as True, because these are not equal to zero.

****kwargs**
[any, default None] Additional keywords have no effect but might be accepted for compatibility with NumPy.

Returns

Series or DataFrame
If level is specified, then, DataFrame is returned; otherwise, Series is returned.

See also:

`numpy.any`
Numpy version of this method.

`Series.any`
Return whether any element is True.

`Series.all`
Return whether all elements are True.

DataFrame.any
Return whether any element is True over requested axis.

DataFrame.all
Return whether all elements are True over requested axis.

Examples

Series

For Series input, the output is a scalar indicating whether any element is True.

```
>>> pd.Series([False, False]).any()
False
>>> pd.Series([True, False]).any()
True
>>> pd.Series([], dtype="float64").any()
False
>>> pd.Series([np.nan]).any()
False
>>> pd.Series([np.nan]).any(skipna=False)
True
```

DataFrame

Whether each column contains at least one True element (the default).

```
>>> df = pd.DataFrame({"A": [1, 2], "B": [0, 2], "C": [0, 0]})
>>> df
   A  B  C
0  1  0  0
1  2  2  0
```

```
>>> df.any()
A     True
B     True
C    False
dtype: bool
```

Aggregating over the columns.

```
>>> df = pd.DataFrame({"A": [True, False], "B": [1, 2]})
>>> df
       A  B
0   True  1
1  False  2
```

```
>>> df.any(axis='columns')
0    True
1    True
dtype: bool
```

```
>>> df = pd.DataFrame({"A": [True, False], "B": [1, 0]})
>>> df
```

(continues on next page)

(continued from previous page)

```
       A  B
0   True  1
1  False  0
```

```
>>> df.any(axis='columns')
0    True
1    False
dtype: bool
```

Aggregating over the entire DataFrame with `axis=None`.

```
>>> df.any(axis=None)
True
```

any for an empty DataFrame is an empty Series.

```
>>> pd.DataFrame([]).any()
Series([], dtype: bool)
```

AlloViz.AlloViz.Elements.Nodes.apply

`Nodes.apply`(*func: AggFuncType*, *axis: Axis = 0*, *raw: bool = False*, *result_type: Literal['expand', 'reduce', 'broadcast'] | None = None*, *args=()*, *by_row: Literal[False, 'compat'] = 'compat'*, ***kwargs*)

Apply a function along an axis of the DataFrame.

Objects passed to the function are Series objects whose index is either the DataFrame's index (`axis=0`) or the DataFrame's columns (`axis=1`). By default (`result_type=None`), the final return type is inferred from the return type of the applied function. Otherwise, it depends on the *result_type* argument.

Parameters

func
[function] Function to apply to each column or row.

axis
[{0 or 'index', 1 or 'columns'}, default 0] Axis along which the function is applied:

- 0 or 'index': apply function to each column.
- 1 or 'columns': apply function to each row.

raw
[bool, default False] Determines if row or column is passed as a Series or ndarray object:

- `False` : passes each row or column as a Series to the function.
- `True` : the passed function will receive ndarray objects instead. If you are just applying a NumPy reduction function this will achieve much better performance.

result_type
[{'expand', 'reduce', 'broadcast', None}, default None] These only act when `axis=1` (columns):

- 'expand' : list-like results will be turned into columns.

- 'reduce' : returns a Series if possible rather than expanding list-like results. This is the opposite of 'expand'.
- 'broadcast' : results will be broadcast to the original shape of the DataFrame, the original index and columns will be retained.

The default behaviour (None) depends on the return value of the applied function: list-like results will be returned as a Series of those. However if the apply function returns a Series these are expanded to columns.

args
[tuple] Positional arguments to pass to *func* in addition to the array/series.

by_row
[False or "compat", default "compat"] Only has an effect when `func` is a listlike or dictlike of funcs and the func isn't a string. If "compat", will if possible first translate the func into pandas methods (e.g. `Series().apply(np.sum)` will be translated to `Series().sum()`). If that doesn't work, will try call to apply again with `by_row=True` and if that fails, will call apply again with `by_row=False` (backward compatible). If False, the funcs will be passed the whole Series at once.

New in version 2.1.0.

****kwargs**
Additional keyword arguments to pass as keywords arguments to *func*.

Returns

Series or DataFrame
Result of applying `func` along the given axis of the DataFrame.

See also:

`DataFrame.map`
For elementwise operations.

`DataFrame.aggregate`
Only perform aggregating type operations.

`DataFrame.transform`
Only perform transforming type operations.

Notes

Functions that mutate the passed object can produce unexpected behavior or errors and are not supported. See Mutating with User Defined Function (UDF) methods for more details.

Examples

```
>>> df = pd.DataFrame([[4, 9]] * 3, columns=['A', 'B'])
>>> df
   A  B
0  4  9
1  4  9
2  4  9
```

Using a numpy universal function (in this case the same as `np.sqrt(df)`):

```
>>> df.apply(np.sqrt)
     A    B
0  2.0  3.0
1  2.0  3.0
2  2.0  3.0
```

Using a reducing function on either axis

```
>>> df.apply(np.sum, axis=0)
A    12
B    27
dtype: int64
```

```
>>> df.apply(np.sum, axis=1)
0    13
1    13
2    13
dtype: int64
```

Returning a list-like will result in a Series

```
>>> df.apply(lambda x: [1, 2], axis=1)
0    [1, 2]
1    [1, 2]
2    [1, 2]
dtype: object
```

Passing `result_type='expand'` will expand list-like results to columns of a Dataframe

```
>>> df.apply(lambda x: [1, 2], axis=1, result_type='expand')
   0  1
0  1  2
1  1  2
2  1  2
```

Returning a Series inside the function is similar to passing `result_type='expand'`. The resulting column names will be the Series index.

```
>>> df.apply(lambda x: pd.Series([1, 2], index=['foo', 'bar']), axis=1)
   foo  bar
0    1    2
1    1    2
2    1    2
```

Passing `result_type='broadcast'` will ensure the same shape result, whether list-like or scalar is returned by the function, and broadcast it along the axis. The resulting column names will be the originals.

```
>>> df.apply(lambda x: [1, 2], axis=1, result_type='broadcast')
   A  B
0  1  2
1  1  2
2  1  2
```

AlloViz.AlloViz.Elements.Nodes.applymap

Nodes.**applymap**(*func: PythonFuncType*, *na_action: NaAction | None = None*, ***kwargs*) → DataFrame

Apply a function to a Dataframe elementwise.

Deprecated since version 2.1.0: DataFrame.applymap has been deprecated. Use DataFrame.map instead.

This method applies a function that accepts and returns a scalar to every element of a DataFrame.

Parameters

func
[callable] Python function, returns a single value from a single value.

na_action
[{None, 'ignore'}, default None] If 'ignore', propagate NaN values, without passing them to func.

****kwargs**
Additional keyword arguments to pass as keywords arguments to *func*.

Returns

DataFrame
Transformed DataFrame.

See also:

DataFrame.apply
Apply a function along input axis of DataFrame.

DataFrame.map
Apply a function along input axis of DataFrame.

DataFrame.replace
Replace values given in *to_replace* with *value*.

Examples

```
>>> df = pd.DataFrame([[1, 2.12], [3.356, 4.567]])
>>> df
       0      1
0  1.000  2.120
1  3.356  4.567
```

```
>>> df.map(lambda x: len(str(x)))
   0  1
0  3  4
1  5  5
```

AlloViz.AlloViz.Elements.Nodes.asfreq

Nodes.**asfreq**(*freq: Frequency*, *method: FillnaOptions | None = None*, *how: Literal['start', 'end'] | None = None*, *normalize: bool_t = False*, *fill_value: Hashable | None = None*) → Self

Convert time series to specified frequency.

Returns the original data conformed to a new index with the specified frequency.

If the index of this Series/DataFrame is a `PeriodIndex`, the new index is the result of transforming the original index with `PeriodIndex.asfreq` (so the original index will map one-to-one to the new index).

Otherwise, the new index will be equivalent to `pd.date_range(start, end, freq=freq)` where `start` and `end` are, respectively, the first and last entries in the original index (see `pandas.date_range()`). The values corresponding to any timesteps in the new index which were not present in the original index will be null (`NaN`), unless a method for filling such unknowns is provided (see the `method` parameter below).

The *`resample()`* method is more appropriate if an operation on each group of timesteps (such as an aggregate) is necessary to represent the data at the new frequency.

Parameters

freq
[DateOffset or str] Frequency DateOffset or string.

method
[{'backfill'/'bfill', 'pad'/'ffill'}, default None] Method to use for filling holes in reindexed Series (note this does not fill NaNs that already were present):

- 'pad' / 'ffill': propagate last valid observation forward to next valid
- 'backfill' / 'bfill': use NEXT valid observation to fill.

how
[{'start', 'end'}, default end] For PeriodIndex only (see PeriodIndex.asfreq).

normalize
[bool, default False] Whether to reset output index to midnight.

fill_value
[scalar, optional] Value to use for missing values, applied during upsampling (note this does not fill NaNs that already were present).

Returns

Series/DataFrame
Series/DataFrame object reindexed to the specified frequency.

See also:

`reindex`
Conform DataFrame to new index with optional filling logic.

Notes

To learn more about the frequency strings, please see this link.

Examples

Start by creating a series with 4 one minute timestamps.

```
>>> index = pd.date_range('1/1/2000', periods=4, freq='T')
>>> series = pd.Series([0.0, None, 2.0, 3.0], index=index)
>>> df = pd.DataFrame({'s': series})
>>> df
                     s
2000-01-01 00:00:00    0.0
2000-01-01 00:01:00    NaN
2000-01-01 00:02:00    2.0
2000-01-01 00:03:00    3.0
```

Upsample the series into 30 second bins.

```
>>> df.asfreq(freq='30S')
                     s
2000-01-01 00:00:00    0.0
2000-01-01 00:00:30    NaN
2000-01-01 00:01:00    NaN
2000-01-01 00:01:30    NaN
2000-01-01 00:02:00    2.0
2000-01-01 00:02:30    NaN
2000-01-01 00:03:00    3.0
```

Upsample again, providing a `fill value`.

```
>>> df.asfreq(freq='30S', fill_value=9.0)
                     s
2000-01-01 00:00:00    0.0
2000-01-01 00:00:30    9.0
2000-01-01 00:01:00    NaN
2000-01-01 00:01:30    9.0
2000-01-01 00:02:00    2.0
2000-01-01 00:02:30    9.0
2000-01-01 00:03:00    3.0
```

Upsample again, providing a `method`.

```
>>> df.asfreq(freq='30S', method='bfill')
                     s
2000-01-01 00:00:00    0.0
2000-01-01 00:00:30    NaN
2000-01-01 00:01:00    NaN
2000-01-01 00:01:30    2.0
2000-01-01 00:02:00    2.0
2000-01-01 00:02:30    3.0
2000-01-01 00:03:00    3.0
```

AlloViz.AlloViz.Elements.Nodes.asof

`Nodes.asof`(*where*, *subset=None*)

Return the last row(s) without any NaNs before *where*.

The last row (for each element in *where*, if list) without any NaN is taken. In case of a `DataFrame`, the last row without NaN considering only the subset of columns (if not *None*)

If there is no good value, NaN is returned for a Series or a Series of NaN values for a DataFrame

Parameters

where
[date or array-like of dates] Date(s) before which the last row(s) are returned.

subset
[str or array-like of str, default *None*] For DataFrame, if not *None*, only use these columns to check for NaNs.

Returns

scalar, Series, or DataFrame
The return can be:

- scalar : when *self* is a Series and *where* is a scalar
- Series: when *self* is a Series and *where* is an array-like, or when *self* is a DataFrame and *where* is a scalar
- DataFrame : when *self* is a DataFrame and *where* is an array-like

Return scalar, Series, or DataFrame.

See also:

`merge_asof`
Perform an asof merge. Similar to left join.

Notes

Dates are assumed to be sorted. Raises if this is not the case.

Examples

A Series and a scalar *where*.

```
>>> s = pd.Series([1, 2, np.nan, 4], index=[10, 20, 30, 40])
>>> s
10    1.0
20    2.0
30    NaN
40    4.0
dtype: float64
```

```
>>> s.asof(20)
2.0
```

For a sequence *where*, a Series is returned. The first value is NaN, because the first element of *where* is before the first index value.

```
>>> s.asof([5, 20])
5     NaN
20    2.0
dtype: float64
```

Missing values are not considered. The following is `2.0`, not NaN, even though NaN is at the index location for `30`.

```
>>> s.asof(30)
2.0
```

Take all columns into consideration

```
>>> df = pd.DataFrame({'a': [10., 20., 30., 40., 50.],
...                    'b': [None, None, None, None, 500]},
...                   index=pd.DatetimeIndex(['2018-02-27 09:01:00',
...                                           '2018-02-27 09:02:00',
...                                           '2018-02-27 09:03:00',
...                                           '2018-02-27 09:04:00',
...                                           '2018-02-27 09:05:00']))
>>> df.asof(pd.DatetimeIndex(['2018-02-27 09:03:30',
...                           '2018-02-27 09:04:30']))
                     a   b
2018-02-27 09:03:30 NaN NaN
2018-02-27 09:04:30 NaN NaN
```

Take a single column into consideration

```
>>> df.asof(pd.DatetimeIndex(['2018-02-27 09:03:30',
...                           '2018-02-27 09:04:30']),
...         subset=['a'])
                       a   b
2018-02-27 09:03:30  30.0 NaN
2018-02-27 09:04:30  40.0 NaN
```

AlloViz.AlloViz.Elements.Nodes.assign

`Nodes.`**`assign`**(***kwargs*) → DataFrame

Assign new columns to a DataFrame.

Returns a new object with all original columns in addition to new ones. Existing columns that are re-assigned will be overwritten.

Parameters

****kwargs**

[dict of {str: callable or Series}] The column names are keywords. If the values are callable, they are computed on the DataFrame and assigned to the new columns. The callable must not change input DataFrame (though pandas doesn't check it). If the values are not callable, (e.g. a Series, scalar, or array), they are simply assigned.

Returns

DataFrame
A new DataFrame with the new columns in addition to all the existing columns.

Notes

Assigning multiple columns within the same `assign` is possible. Later items in '**kwargs' may refer to newly created or modified columns in 'df'; items are computed and assigned into 'df' in order.

Examples

```
>>> df = pd.DataFrame({'temp_c': [17.0, 25.0]},
...                   index=['Portland', 'Berkeley'])
>>> df
          temp_c
Portland    17.0
Berkeley    25.0
```

Where the value is a callable, evaluated on *df*:

```
>>> df.assign(temp_f=lambda x: x.temp_c * 9 / 5 + 32)
          temp_c  temp_f
Portland    17.0    62.6
Berkeley    25.0    77.0
```

Alternatively, the same behavior can be achieved by directly referencing an existing Series or sequence:

```
>>> df.assign(temp_f=df['temp_c'] * 9 / 5 + 32)
          temp_c  temp_f
Portland    17.0    62.6
Berkeley    25.0    77.0
```

You can create multiple columns within the same assign where one of the columns depends on another one defined within the same assign:

```
>>> df.assign(temp_f=lambda x: x['temp_c'] * 9 / 5 + 32,
...           temp_k=lambda x: (x['temp_f'] + 459.67) * 5 / 9)
          temp_c  temp_f  temp_k
Portland    17.0    62.6  290.15
Berkeley    25.0    77.0  298.15
```

AlloViz.AlloViz.Elements.Nodes.astype

`Nodes.`**`astype`**(*dtype*, *copy: bool | None = None*, *errors: Literal['ignore', 'raise'] = 'raise'*) → None

Cast a pandas object to a specified dtype `dtype`.

Parameters

dtype
[str, data type, Series or Mapping of column name -> data type] Use a str, numpy.dtype, pandas.ExtensionDtype or Python type to cast entire pandas object to the same type. Alternatively, use a mapping, e.g. {col: dtype, ...}, where col is

a column label and dtype is a numpy.dtype or Python type to cast one or more of the DataFrame's columns to column-specific types.

copy
[bool, default True] Return a copy when `copy=True` (be very careful setting `copy=False` as changes to values then may propagate to other pandas objects).

errors
[{'raise', 'ignore'}, default 'raise'] Control raising of exceptions on invalid data for provided dtype.

- `raise` : allow exceptions to be raised
- `ignore` : suppress exceptions. On error return original object.

Returns

same type as caller

See also:

`to_datetime`
Convert argument to datetime.

`to_timedelta`
Convert argument to timedelta.

`to_numeric`
Convert argument to a numeric type.

`numpy.ndarray.astype`
Cast a numpy array to a specified type.

Notes

Changed in version 2.0.0: Using `astype` to convert from timezone-naive dtype to timezone-aware dtype will raise an exception. Use `Series.dt.tz_localize()` instead.

Examples

Create a DataFrame:

```
>>> d = {'col1': [1, 2], 'col2': [3, 4]}
>>> df = pd.DataFrame(data=d)
>>> df.dtypes
col1    int64
col2    int64
dtype: object
```

Cast all columns to int32:

```
>>> df.astype('int32').dtypes
col1    int32
col2    int32
dtype: object
```

Cast col1 to int32 using a dictionary:

```
>>> df.astype({'col1': 'int32'}).dtypes
col1    int32
col2    int64
dtype: object
```

Create a series:

```
>>> ser = pd.Series([1, 2], dtype='int32')
>>> ser
0    1
1    2
dtype: int32
>>> ser.astype('int64')
0    1
1    2
dtype: int64
```

Convert to categorical type:

```
>>> ser.astype('category')
0    1
1    2
dtype: category
Categories (2, int32): [1, 2]
```

Convert to ordered categorical type with custom ordering:

```
>>> from pandas.api.types import CategoricalDtype
>>> cat_dtype = CategoricalDtype(
...     categories=[2, 1], ordered=True)
>>> ser.astype(cat_dtype)
0    1
1    2
dtype: category
Categories (2, int64): [2 < 1]
```

Create a series of dates:

```
>>> ser_date = pd.Series(pd.date_range('20200101', periods=3))
>>> ser_date
0   2020-01-01
1   2020-01-02
2   2020-01-03
dtype: datetime64[ns]
```

AlloViz.AlloViz.Elements.Nodes.at_time

Nodes.**at_time**(*time*, *asof: bool = False*, *axis: int | Literal['index', 'columns', 'rows'] | None = None*) → None

Select values at particular time of day (e.g., 9:30AM).

Parameters

time
[datetime.time or str] The values to select.

axis
[{0 or 'index', 1 or 'columns'}, default 0] For *Series* this parameter is unused and defaults to 0.

Returns

Series or DataFrame

Raises

TypeError
If the index is not a `DatetimeIndex`

See also:

between_time
Select values between particular times of the day.

first
Select initial periods of time series based on a date offset.

last
Select final periods of time series based on a date offset.

DatetimeIndex.indexer_at_time
Get just the index locations for values at particular time of the day.

Examples

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='12H')
>>> ts = pd.DataFrame({'A': [1, 2, 3, 4]}, index=i)
>>> ts
                     A
2018-04-09 00:00:00  1
2018-04-09 12:00:00  2
2018-04-10 00:00:00  3
2018-04-10 12:00:00  4
```

```
>>> ts.at_time('12:00')
                     A
2018-04-09 12:00:00  2
2018-04-10 12:00:00  4
```

AlloViz.AlloViz.Elements.Nodes.backfill

Nodes.**backfill**(*, *axis: None | Axis = None*, *inplace: bool_t = False*, *limit: None | int = None*, *downcast: dict | None | lib.NoDefault = _NoDefault.no_default*) → Self | None

Fill NA/NaN values by using the next valid observation to fill the gap.

Deprecated since version 2.0: Series/DataFrame.backfill is deprecated. Use Series/DataFrame.bfill instead.

Returns

Series/DataFrame or None
Object with missing values filled or None if `inplace=True`.

Examples

Please see examples for `DataFrame.bfill()` or `Series.bfill()`.

AlloViz.AlloViz.Elements.Nodes.between_time

Nodes.**between_time**(*start_time*, *end_time*, *inclusive: Literal['left', 'right'] | Literal['both', 'neither'] = 'both'*, *axis: int | Literal['index', 'columns', 'rows'] | None = None*) → None

Select values between particular times of the day (e.g., 9:00-9:30 AM).

By setting `start_time` to be later than `end_time`, you can get the times that are *not* between the two times.

Parameters

start_time
[datetime.time or str] Initial time as a time filter limit.

end_time
[datetime.time or str] End time as a time filter limit.

inclusive
[{"both", "neither", "left", "right"}, default "both"] Include boundaries; whether to set each bound as closed or open.

axis
[{0 or 'index', 1 or 'columns'}, default 0] Determine range time on index or columns value. For *Series* this parameter is unused and defaults to 0.

Returns

Series or DataFrame
Data from the original object filtered to the specified dates range.

Raises

TypeError
If the index is not a `DatetimeIndex`

See also:

at_time
Select values at a particular time of the day.

first
Select initial periods of time series based on a date offset.

last
Select final periods of time series based on a date offset.

DatetimeIndex.indexer_between_time
Get just the index locations for values between particular times of the day.

Examples

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='1D20min')
>>> ts = pd.DataFrame({'A': [1, 2, 3, 4]}, index=i)
>>> ts
                     A
2018-04-09 00:00:00  1
2018-04-10 00:20:00  2
2018-04-11 00:40:00  3
2018-04-12 01:00:00  4
```

```
>>> ts.between_time('0:15', '0:45')
                     A
2018-04-10 00:20:00  2
2018-04-11 00:40:00  3
```

You get the times that are *not* between two times by setting `start_time` later than `end_time`:

```
>>> ts.between_time('0:45', '0:15')
                     A
2018-04-09 00:00:00  1
2018-04-12 01:00:00  4
```

AlloViz.AlloViz.Elements.Nodes.bfill

Nodes.**bfill**(*, *axis: None | Axis = None*, *inplace: bool_t = False*, *limit: None | int = None*, *downcast: dict | None | lib.NoDefault = _NoDefault.no_default*) → Self | None

Fill NA/NaN values by using the next valid observation to fill the gap.

Parameters

axis
[{0 or 'index'} for Series, {0 or 'index', 1 or 'columns'} for DataFrame] Axis along which to fill missing values. For *Series* this parameter is unused and defaults to 0.

inplace
[bool, default False] If True, fill in-place. Note: this will modify any other views on this object (e.g., a no-copy slice for a column in a DataFrame).

limit
[int, default None] If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled. Must be greater than 0 if not None.

downcast
[dict, default is None] A dict of item->dtype of what to downcast if possible, or the string 'infer' which will try to downcast to an appropriate equal type (e.g. float64 to int64 if possible).

Returns

Series/DataFrame or None
Object with missing values filled or None if `inplace=True`.

Examples

For Series:

```
>>> s = pd.Series([1, None, None, 2])
>>> s.bfill()
0    1.0
1    2.0
2    2.0
3    2.0
dtype: float64
>>> s.bfill(limit=1)
0    1.0
1    NaN
2    2.0
3    2.0
dtype: float64
```

With DataFrame:

```
>>> df = pd.DataFrame({'A': [1, None, None, 4], 'B': [None, 5, None, 7]})
>>> df
     A     B
0   1.0   NaN
1   NaN   5.0
2   NaN   NaN
3   4.0   7.0
>>> df.bfill()
     A     B
0   1.0   5.0
1   4.0   5.0
2   4.0   7.0
3   4.0   7.0
>>> df.bfill(limit=1)
     A     B
0   1.0   5.0
1   NaN   5.0
2   4.0   7.0
3   4.0   7.0
```

AlloViz.AlloViz.Elements.Nodes.bool

Nodes.**bool**() → bool

Return the bool of a single element Series or DataFrame.

Deprecated since version 2.1.0: bool is deprecated and will be removed in future version of pandas

This must be a boolean scalar value, either True or False. It will raise a ValueError if the Series or DataFrame does not have exactly 1 element, or that element is not boolean (integer values 0 and 1 will also raise an exception).

Returns

bool
The value in the Series or DataFrame.

See also:

Series.astype
Change the data type of a Series, including to boolean.

DataFrame.astype
Change the data type of a DataFrame, including to boolean.

numpy.bool_
NumPy boolean data type, used by pandas for boolean values.

Examples

The method will only work for single element objects with a boolean value:

```
>>> pd.Series([True]).bool()
True
>>> pd.Series([False]).bool()
False
```

```
>>> pd.DataFrame({'col': [True]}).bool()
True
>>> pd.DataFrame({'col': [False]}).bool()
False
```

AlloViz.AlloViz.Elements.Nodes.boxplot

Nodes.**boxplot**(*column=None*, *by=None*, *ax=None*, *fontsize: None | int = None*, *rot: int = 0*, *grid: bool = True*, *figsize: tuple[float, float] | None = None*, *layout=None*, *return_type=None*, *backend=None*, ***kwargs*)

Make a box plot from DataFrame columns.

Make a box-and-whisker plot from DataFrame columns, optionally grouped by some other columns. A box plot is a method for graphically depicting groups of numerical data through their quartiles. The box extends from the Q1 to Q3 quartile values of the data, with a line at the median (Q2). The whiskers extend from the edges of box to show the range of the data. By default, they extend no more than *1.5 * IQR (IQR = Q3 - Q1)* from the edges of the box, ending at the farthest data point within that interval. Outliers are plotted as separate dots.

For further details see Wikipedia's entry for boxplot.

Parameters

column
[str or list of str, optional] Column name or list of names, or vector. Can be any valid input to `pandas.DataFrame.groupby()`.

by
[str or array-like, optional] Column in the DataFrame to `pandas.DataFrame.groupby()`. One box-plot will be done per value of columns in *by*.

ax
[object of class matplotlib.axes.Axes, optional] The matplotlib axes to be used by boxplot.

fontsize
[float or str] Tick label font size in points or as a string (e.g., *large*).

rot
[float, default 0] The rotation angle of labels (in degrees) with respect to the screen coordinate system.

grid
[bool, default True] Setting this to True will show the grid.

figsize
[A tuple (width, height) in inches] The size of the figure to create in matplotlib.

layout
[tuple (rows, columns), optional] For example, (3, 5) will display the subplots using 3 rows and 5 columns, starting from the top-left.

return_type
[{'axes', 'dict', 'both'} or None, default 'axes'] The kind of object to return. The default is `axes`.

- 'axes' returns the matplotlib axes the boxplot is drawn on.
- 'dict' returns a dictionary whose values are the matplotlib Lines of the boxplot.
- 'both' returns a namedtuple with the axes and dict.
- when grouping with `by`, a Series mapping columns to `return_type` is returned.

  If `return_type` is *None*, a NumPy array of axes with the same shape as `layout` is returned.

backend
[str, default None] Backend to use instead of the backend specified in the option `plotting.backend`. For instance, 'matplotlib'. Alternatively, to specify the `plotting.backend` for the whole session, set `pd.options.plotting.backend`.

****kwargs**
All other plotting keyword arguments to be passed to `matplotlib.pyplot.boxplot()`.

Returns

result
See Notes.

See also:

`pandas.Series.plot.hist`
: Make a histogram.

`matplotlib.pyplot.boxplot`
: Matplotlib equivalent plot.

Notes

The return type depends on the *return_type* parameter:

- 'axes' : object of class matplotlib.axes.Axes
- 'dict' : dict of matplotlib.lines.Line2D objects
- 'both' : a namedtuple with structure (ax, lines)

For data grouped with `by`, return a Series of the above or a numpy array:

- `Series`
- `array` (for `return_type = None`)

Use `return_type='dict'` when you want to tweak the appearance of the lines after plotting. In this case a dict containing the Lines making up the boxes, caps, fliers, medians, and whiskers is returned.

Examples

Boxplots can be created for every column in the dataframe by `df.boxplot()` or indicating the columns to be used:

```
>>> np.random.seed(1234)
>>> df = pd.DataFrame(np.random.randn(10, 4),
...                   columns=['Col1', 'Col2', 'Col3', 'Col4'])
>>> boxplot = df.boxplot(column=['Col1', 'Col2', 'Col3'])
```

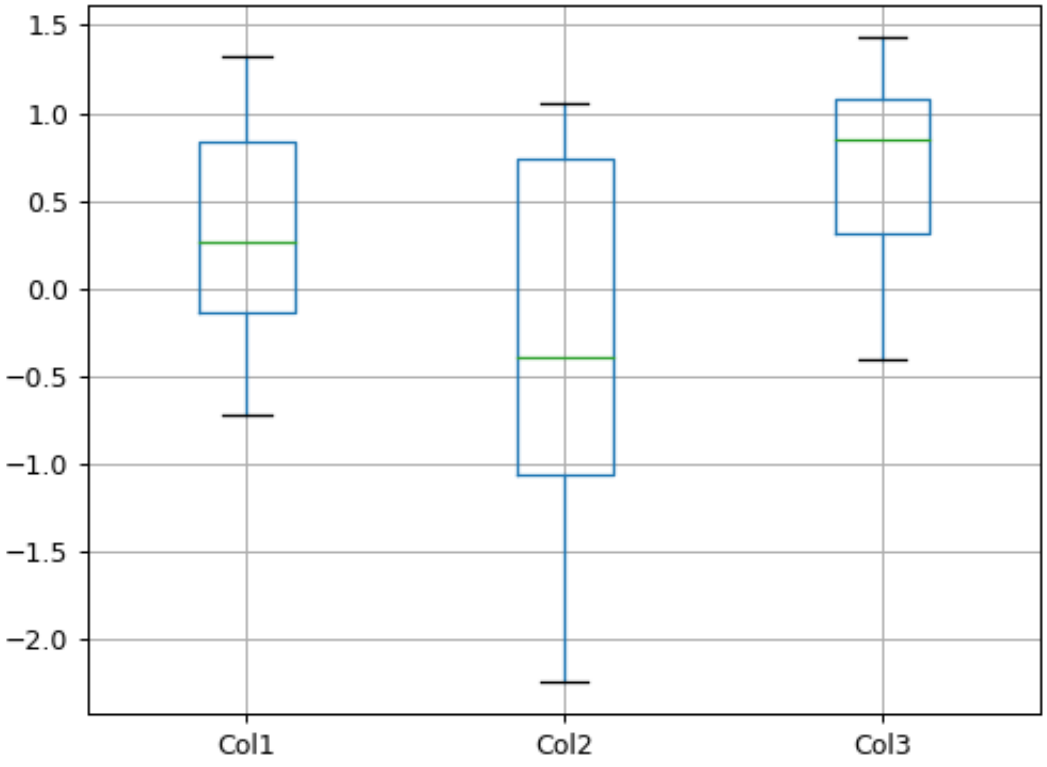
Boxplots of variables distributions grouped by the values of a third variable can be created using the option `by`. For instance:

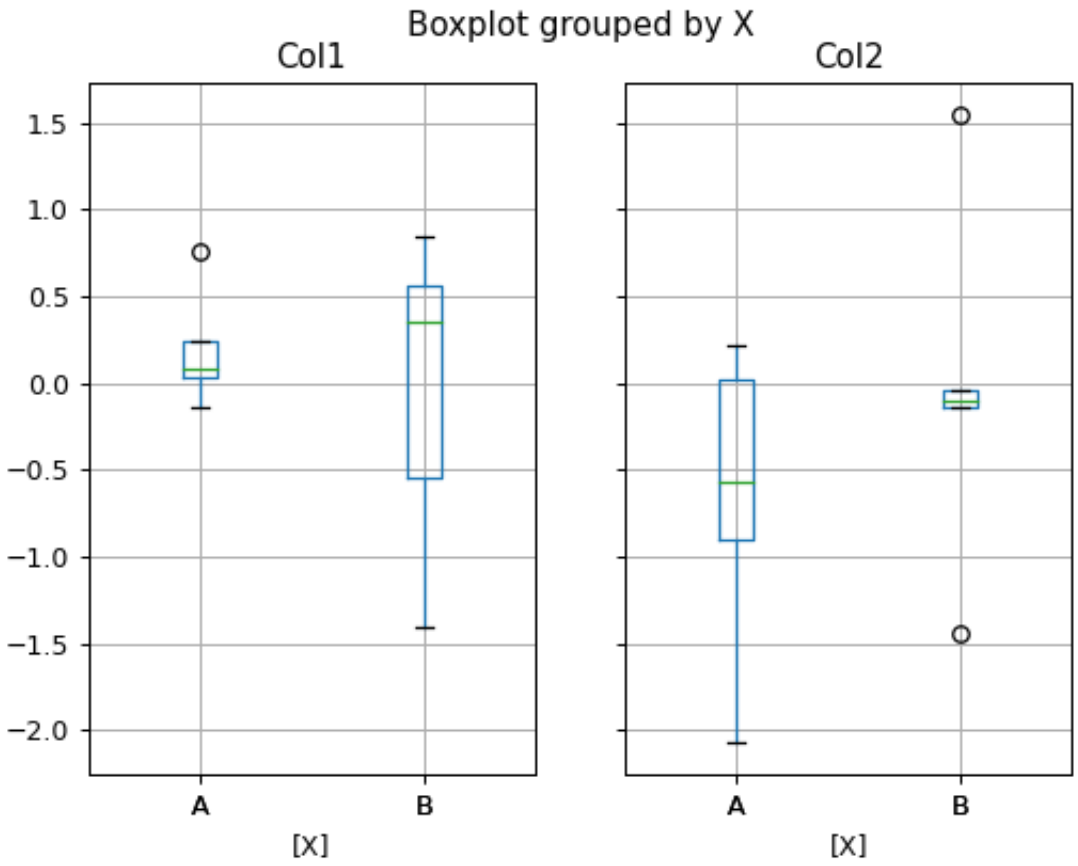
```
>>> df = pd.DataFrame(np.random.randn(10, 2),
...                   columns=['Col1', 'Col2'])
>>> df['X'] = pd.Series(['A', 'A', 'A', 'A', 'A',
...                      'B', 'B', 'B', 'B', 'B'])
>>> boxplot = df.boxplot(by='X')
```

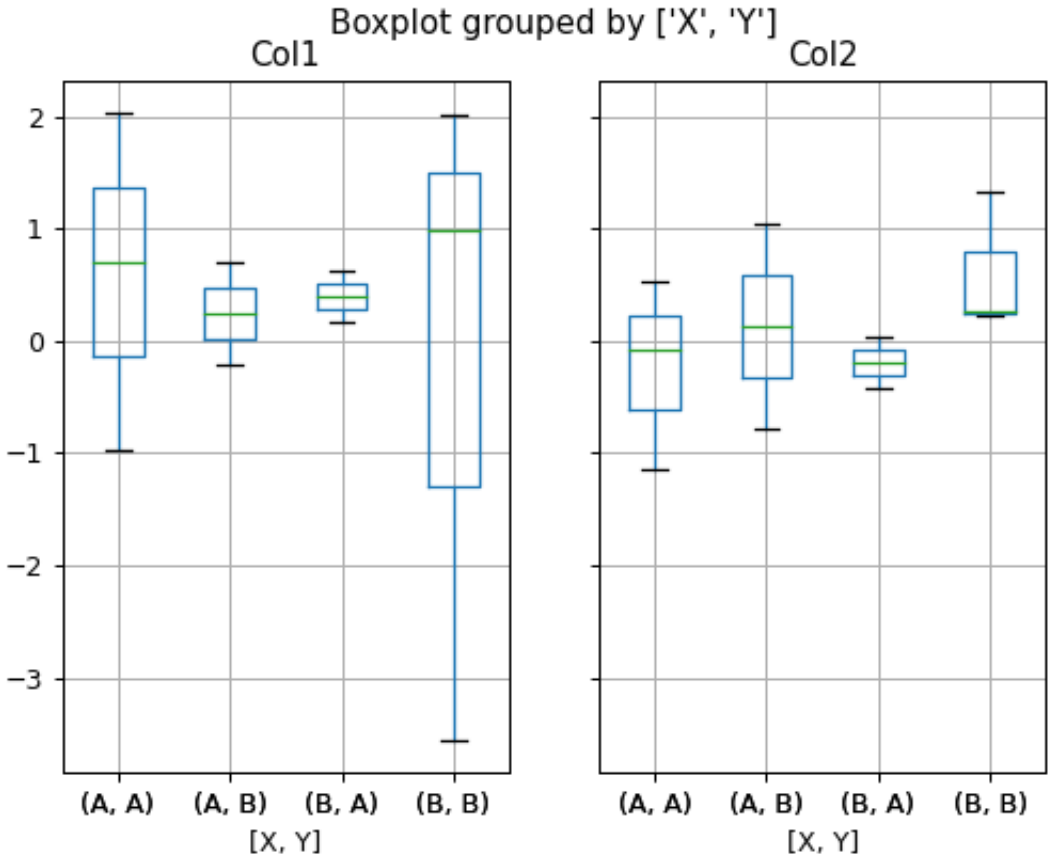
A list of strings (i.e. `['X', 'Y']`) can be passed to boxplot in order to group the data by combination of the variables in the x-axis:

```
>>> df = pd.DataFrame(np.random.randn(10, 3),
...                   columns=['Col1', 'Col2', 'Col3'])
>>> df['X'] = pd.Series(['A', 'A', 'A', 'A', 'A',
...                      'B', 'B', 'B', 'B', 'B'])
>>> df['Y'] = pd.Series(['A', 'B', 'A', 'B', 'A',
...                      'B', 'A', 'B', 'A', 'B'])
>>> boxplot = df.boxplot(column=['Col1', 'Col2'], by=['X', 'Y'])
```

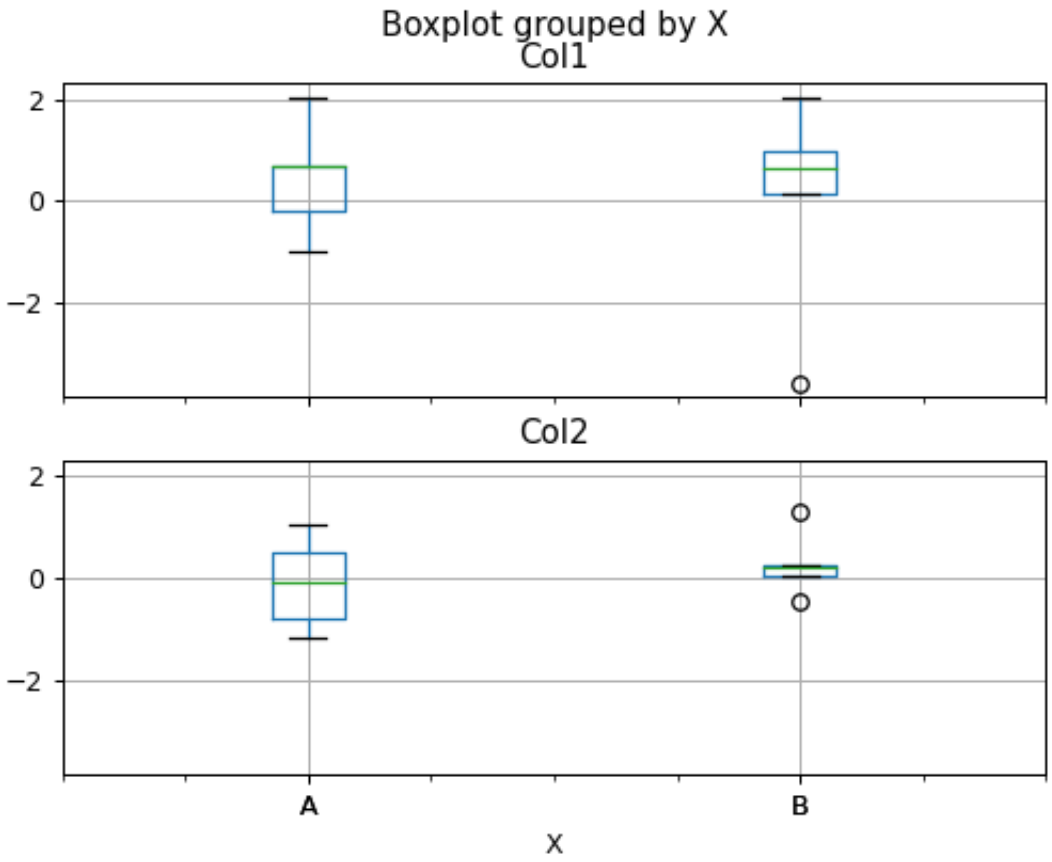
The layout of boxplot can be adjusted giving a tuple to `layout`:


1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
Col1
Col2
Col3


Boxplot grouped by X
Col1
Col2
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
A
B
[X]
A
B
[X]


Boxplot grouped by ['X', 'Y']
Col1
Col2
2
1
0
−1
−2
−3
(A, A)
(A, B)
(B, A)
(B, B)
[X, Y]
(A, A)
(A, B)
(B, A)
(B, B)
[X, Y]

```
>>> boxplot = df.boxplot(column=['Col1', 'Col2'], by='X',
...                      layout=(2, 1))
```



Additional formatting can be done to the boxplot, like suppressing the grid (`grid=False`), rotating the labels in the x-axis (i.e. `rot=45`) or changing the fontsize (i.e. `fontsize=15`):

```
>>> boxplot = df.boxplot(grid=False, rot=45, fontsize=15)
```

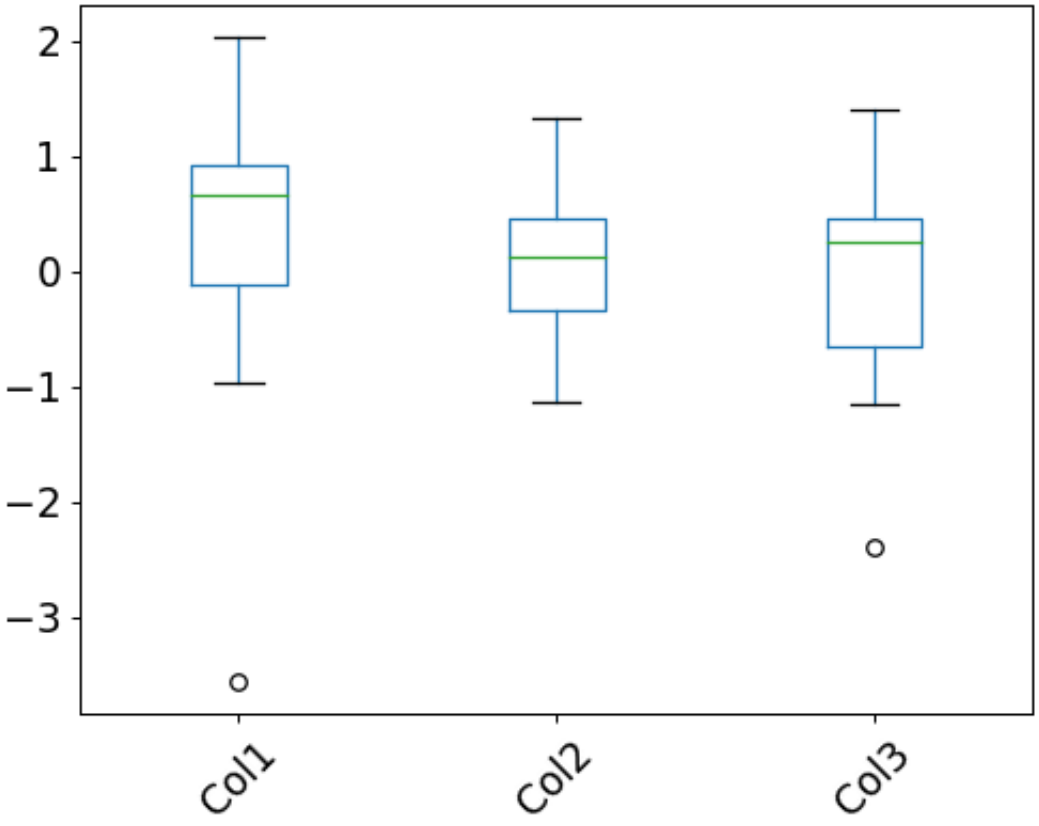
The parameter `return_type` can be used to select the type of element returned by *boxplot*. When `return_type='axes'` is selected, the matplotlib axes on which the boxplot is drawn are returned:

```
>>> boxplot = df.boxplot(column=['Col1', 'Col2'], return_type='axes')
>>> type(boxplot)
<class 'matplotlib.axes._axes.Axes'>
```

When grouping with `by`, a Series mapping columns to `return_type` is returned:

```
>>> boxplot = df.boxplot(column=['Col1', 'Col2'], by='X',
...                      return_type='axes')
>>> type(boxplot)
<class 'pandas.core.series.Series'>
```

If `return_type` is *None*, a NumPy array of axes with the same shape as `layout` is returned:


2
1
0
−1
−2
−3
Col1
Col2
Col3

```
>>> boxplot = df.boxplot(column=['Col1', 'Col2'], by='X',
...                      return_type=None)
>>> type(boxplot)
<class 'numpy.ndarray'>
```

AlloViz.AlloViz.Elements.Nodes.clip

Nodes.**clip**(*lower=None*, *upper=None*, ***, *axis: Axis | None = None*, *inplace: bool_t = False*, ***kwargs*) → Self | None

Trim values at input threshold(s).

Assigns values outside boundary to boundary values. Thresholds can be singular values or array like, and in the latter case the clipping is performed element-wise in the specified axis.

Parameters

lower
[float or array-like, default None] Minimum threshold value. All values below this threshold will be set to it. A missing threshold (e.g *NA*) will not clip the value.

upper
[float or array-like, default None] Maximum threshold value. All values above this threshold will be set to it. A missing threshold (e.g *NA*) will not clip the value.

axis
[{{0 or 'index', 1 or 'columns', None}}, default None] Align object with lower and upper along the given axis. For *Series* this parameter is unused and defaults to *None*.

inplace
[bool, default False] Whether to perform the operation in place on the data.

***args, **kwargs**
Additional keywords have no effect but might be accepted for compatibility with numpy.

Returns

Series or DataFrame or None
Same type as calling object with the values outside the clip boundaries replaced or None if `inplace=True`.

See also:

`Series.clip`
Trim values at input threshold in series.

`DataFrame.clip`
Trim values at input threshold in dataframe.

`numpy.clip`
Clip (limit) the values in an array.

Examples

```
>>> data = {'col_0': [9, -3, 0, -1, 5], 'col_1': [-2, -7, 6, 8, -5]}
>>> df = pd.DataFrame(data)
>>> df
   col_0  col_1
0      9     -2
1     -3     -7
2      0      6
3     -1      8
4      5     -5
```

Clips per column using lower and upper thresholds:

```
>>> df.clip(-4, 6)
   col_0  col_1
0      6     -2
1     -3     -4
2      0      6
3     -1      6
4      5     -4
```

Clips using specific lower and upper thresholds per column element:

```
>>> t = pd.Series([2, -4, -1, 6, 3])
>>> t
0    2
1   -4
2   -1
3    6
4    3
dtype: int64
```

```
>>> df.clip(t, t + 4, axis=0)
   col_0  col_1
0      6      2
1     -3     -4
2      0      3
3      6      8
4      5      3
```

Clips using specific lower threshold per column element, with missing values:

```
>>> t = pd.Series([2, -4, np.nan, 6, 3])
>>> t
0    2.0
1   -4.0
2    NaN
3    6.0
4    3.0
dtype: float64
```

```
>>> df.clip(t, axis=0)
col_0  col_1
0      9      2
1     -3     -4
2      0      6
3      6      8
4      5      3
```

AlloViz.AlloViz.Elements.Nodes.combine

Nodes.**combine**(*other: DataFrame*, *func: Callable[[Series, Series], Series | Hashable]*, *fill_value=None*, *overwrite: bool = True*) → DataFrame

Perform column-wise combine with another DataFrame.

Combines a DataFrame with *other* DataFrame using *func* to element-wise combine columns. The row and column indexes of the resulting DataFrame will be the union of the two.

Parameters

other
[DataFrame] The DataFrame to merge column-wise.

func
[function] Function that takes two series as inputs and return a Series or a scalar. Used to merge the two dataframes column by columns.

fill_value
[scalar value, default None] The value to fill NaNs with prior to passing any column to the merge func.

overwrite
[bool, default True] If True, columns in *self* that do not exist in *other* will be overwritten with NaNs.

Returns

DataFrame
Combination of the provided DataFrames.

See also:

DataFrame.combine_first
Combine two DataFrame objects and default to non-null values in frame calling the method.

Examples

Combine using a simple function that chooses the smaller column.

```
>>> df1 = pd.DataFrame({'A': [0, 0], 'B': [4, 4]})
>>> df2 = pd.DataFrame({'A': [1, 1], 'B': [3, 3]})
>>> take_smaller = lambda s1, s2: s1 if s1.sum() < s2.sum() else s2
>>> df1.combine(df2, take_smaller)
   A  B
0  0  3
1  0  3
```

Example using a true element-wise combine function.

```
>>> df1 = pd.DataFrame({'A': [5, 0], 'B': [2, 4]})
>>> df2 = pd.DataFrame({'A': [1, 1], 'B': [3, 3]})
>>> df1.combine(df2, np.minimum)
   A  B
0  1  2
1  0  3
```

Using *fill_value* fills Nones prior to passing the column to the merge function.

```
>>> df1 = pd.DataFrame({'A': [0, 0], 'B': [None, 4]})
>>> df2 = pd.DataFrame({'A': [1, 1], 'B': [3, 3]})
>>> df1.combine(df2, take_smaller, fill_value=-5)
   A    B
0  0 -5.0
1  0  4.0
```

However, if the same element in both dataframes is None, that None is preserved

```
>>> df1 = pd.DataFrame({'A': [0, 0], 'B': [None, 4]})
>>> df2 = pd.DataFrame({'A': [1, 1], 'B': [None, 3]})
>>> df1.combine(df2, take_smaller, fill_value=-5)
    A    B
0  0 -5.0
1  0  3.0
```

Example that demonstrates the use of *overwrite* and behavior when the axis differ between the dataframes.

```
>>> df1 = pd.DataFrame({'A': [0, 0], 'B': [4, 4]})
>>> df2 = pd.DataFrame({'B': [3, 3], 'C': [-10, 1], }, index=[1, 2])
>>> df1.combine(df2, take_smaller)
     A    B     C
0  NaN  NaN   NaN
1  NaN  3.0 -10.0
2  NaN  3.0   1.0
```

```
>>> df1.combine(df2, take_smaller, overwrite=False)
     A    B     C
0  0.0  NaN   NaN
1  0.0  3.0 -10.0
2  NaN  3.0   1.0
```

Demonstrating the preference of the passed in dataframe.

```
>>> df2 = pd.DataFrame({'B': [3, 3], 'C': [1, 1], }, index=[1, 2])
>>> df2.combine(df1, take_smaller)
   A    B   C
0  0.0  NaN NaN
1  0.0  3.0 NaN
2  NaN  3.0 NaN
```

```
>>> df2.combine(df1, take_smaller, overwrite=False)
     A    B   C
```

(continues on next page)

(continued from previous page)

```
0  0.0  NaN NaN
1  0.0  3.0 1.0
2  NaN  3.0 1.0
```

AlloViz.AlloViz.Elements.Nodes.combine_first

Nodes.**combine_first**(*other: DataFrame*) → DataFrame

Update null elements with value in the same location in *other*.

Combine two DataFrame objects by filling null values in one DataFrame with non-null values from other DataFrame. The row and column indexes of the resulting DataFrame will be the union of the two. The resulting dataframe contains the 'first' dataframe values and overrides the second one values where both first.loc[index, col] and second.loc[index, col] are not missing values, upon calling first.combine_first(second).

Parameters

other
[DataFrame] Provided DataFrame to use to fill null values.

Returns

DataFrame
The result of combining the provided DataFrame with the other object.

See also:

DataFrame.combine
Perform series-wise operation on two DataFrames using a given function.

Examples

```
>>> df1 = pd.DataFrame({'A': [None, 0], 'B': [None, 4]})
>>> df2 = pd.DataFrame({'A': [1, 1], 'B': [3, 3]})
>>> df1.combine_first(df2)
     A    B
0  1.0  3.0
1  0.0  4.0
```

Null values still persist if the location of that null value does not exist in *other*

```
>>> df1 = pd.DataFrame({'A': [None, 0], 'B': [4, None]})
>>> df2 = pd.DataFrame({'B': [3, 3], 'C': [1, 1]}, index=[1, 2])
>>> df1.combine_first(df2)
     A    B    C
0  NaN  4.0  NaN
1  0.0  3.0  1.0
2  NaN  3.0  1.0
```

AlloViz.AlloViz.Elements.Nodes.compare

Nodes.**compare**(*other: DataFrame*, *align_axis: Axis = 1*, *keep_shape: bool = False*, *keep_equal: bool = False*, *result_names: Suffixes = ('self', 'other')*) → DataFrame

Compare to another DataFrame and show the differences.

Parameters

other
[DataFrame] Object to compare with.

align_axis
[{0 or 'index', 1 or 'columns'}, default 1] Determine which axis to align the comparison on.

- **0, or 'index'**
  [Resulting differences are stacked vertically] with rows drawn alternately from self and other.
- **1, or 'columns'**
  [Resulting differences are aligned horizontally] with columns drawn alternately from self and other.

keep_shape
[bool, default False] If true, all rows and columns are kept. Otherwise, only the ones with different values are kept.

keep_equal
[bool, default False] If true, the result keeps values that are equal. Otherwise, equal values are shown as NaNs.

result_names
[tuple, default ('self', 'other')] Set the dataframes names in the comparison.

New in version 1.5.0.

Returns

DataFrame
DataFrame that shows the differences stacked side by side.

The resulting index will be a MultiIndex with 'self' and 'other' stacked alternately at the inner level.

Raises

ValueError
When the two DataFrames don't have identical labels or shape.

See also:

`Series.compare`
Compare with another Series and show differences.

`DataFrame.equals`
Test whether two objects contain the same elements.

Notes

Matching NaNs will not appear as a difference.

Can only compare identically-labeled (i.e. same shape, identical row and column labels) DataFrames

Examples

```
>>> df = pd.DataFrame(
...     {
...         "col1": ["a", "a", "b", "b", "a"],
...         "col2": [1.0, 2.0, 3.0, np.nan, 5.0],
...         "col3": [1.0, 2.0, 3.0, 4.0, 5.0]
...     },
...     columns=["col1", "col2", "col3"],
... )
>>> df
  col1  col2  col3
0    a   1.0   1.0
1    a   2.0   2.0
2    b   3.0   3.0
3    b   NaN   4.0
4    a   5.0   5.0
```

```
>>> df2 = df.copy()
>>> df2.loc[0, 'col1'] = 'c'
>>> df2.loc[2, 'col3'] = 4.0
>>> df2
  col1  col2  col3
0    c   1.0   1.0
1    a   2.0   2.0
2    b   3.0   4.0
3    b   NaN   4.0
4    a   5.0   5.0
```

Align the differences on columns

```
>>> df.compare(df2)
  col1       col3
  self other self other
0    a     c  NaN   NaN
2  NaN   NaN  3.0   4.0
```

Assign result_names

```
>>> df.compare(df2, result_names=("left", "right"))
  col1       col3
  left right left right
0    a     c  NaN   NaN
2  NaN   NaN  3.0   4.0
```

Stack the differences on rows

```
>>> df.compare(df2, align_axis=0)
        col1  col3
0 self     a   NaN
  other    c   NaN
2 self   NaN   3.0
  other  NaN   4.0
```

Keep the equal values

```
>>> df.compare(df2, keep_equal=True)
  col1       col3
  self other self other
0    a     c  1.0   1.0
2    b     b  3.0   4.0
```

Keep all original rows and columns

```
>>> df.compare(df2, keep_shape=True)
  col1       col2       col3
  self other self other self other
0    a     c  NaN   NaN  NaN   NaN
1  NaN   NaN  NaN   NaN  NaN   NaN
2  NaN   NaN  NaN   NaN  3.0   4.0
3  NaN   NaN  NaN   NaN  NaN   NaN
4  NaN   NaN  NaN   NaN  NaN   NaN
```

Keep all original rows and columns and also all original values

```
>>> df.compare(df2, keep_shape=True, keep_equal=True)
  col1       col2       col3
  self other self other self other
0    a     c  1.0   1.0  1.0   1.0
1    a     a  2.0   2.0  2.0   2.0
2    b     b  3.0   3.0  3.0   4.0
3    b     b  NaN   NaN  4.0   4.0
4    a     a  5.0   5.0  5.0   5.0
```

AlloViz.AlloViz.Elements.Nodes.convert_dtypes

Nodes.**convert_dtypes**(*infer_objects: bool = True*, *convert_string: bool = True*, *convert_integer: bool = True*, *convert_boolean: bool = True*, *convert_floating: bool = True*, *dtype_backend: Literal['pyarrow', 'numpy_nullable'] = 'numpy_nullable'*) → None

Convert columns to the best possible dtypes using dtypes supporting `pd.NA`.

Parameters

infer_objects
[bool, default True] Whether object dtypes should be converted to the best possible types.

convert_string
[bool, default True] Whether object dtypes should be converted to `StringDtype()`.

convert_integer
[bool, default True] Whether, if possible, conversion can be done to integer extension types.

convert_boolean
[bool, defaults True] Whether object dtypes should be converted to `BooleanDtypes()`.

convert_floating
[bool, defaults True] Whether, if possible, conversion can be done to floating extension types. If *convert_integer* is also True, preference will be give to integer dtypes if the floats can be faithfully casted to integers.

New in version 1.2.0.

dtype_backend
[{'numpy_nullable', 'pyarrow'}, default 'numpy_nullable'] Back-end data type applied to the resultant `DataFrame` (still experimental). Behaviour is as follows:

- `"numpy_nullable"`: returns nullable-dtype-backed `DataFrame` (default).
- `"pyarrow"`: returns pyarrow-backed nullable `ArrowDtype` DataFrame.

New in version 2.0.

Returns

Series or DataFrame
Copy of input object with new dtype.

See also:

`infer_objects`
Infer dtypes of objects.

`to_datetime`
Convert argument to datetime.

`to_timedelta`
Convert argument to timedelta.

`to_numeric`
Convert argument to a numeric type.

Notes

By default, `convert_dtypes` will attempt to convert a Series (or each Series in a DataFrame) to dtypes that support `pd.NA`. By using the options `convert_string`, `convert_integer`, `convert_boolean` and `convert_floating`, it is possible to turn off individual conversions to `StringDtype`, the integer extension types, `BooleanDtype` or floating extension types, respectively.

For object-dtyped columns, if `infer_objects` is `True`, use the inference rules as during normal Series/DataFrame construction. Then, if possible, convert to `StringDtype`, `BooleanDtype` or an appropriate integer or floating extension type, otherwise leave as `object`.

If the dtype is integer, convert to an appropriate integer extension type.

If the dtype is numeric, and consists of all integers, convert to an appropriate integer extension type. Otherwise, convert to an appropriate floating extension type.

Changed in version 1.2: Starting with pandas 1.2, this method also converts float columns to the nullable floating extension type.

In the future, as new dtypes are added that support `pd.NA`, the results of this method will change to support those new dtypes.

Examples

```
>>> df = pd.DataFrame(
...     {
...         "a": pd.Series([1, 2, 3], dtype=np.dtype("int32")),
...         "b": pd.Series(["x", "y", "z"], dtype=np.dtype("O")),
...         "c": pd.Series([True, False, np.nan], dtype=np.dtype("O")),
...         "d": pd.Series(["h", "i", np.nan], dtype=np.dtype("O")),
...         "e": pd.Series([10, np.nan, 20], dtype=np.dtype("float")),
...         "f": pd.Series([np.nan, 100.5, 200], dtype=np.dtype("float")),
...     }
... )
```

Start with a DataFrame with default dtypes.

```
>>> df
   a  b      c    d     e      f
0  1  x   True    h  10.0    NaN
1  2  y  False    i   NaN  100.5
2  3  z    NaN  NaN  20.0  200.0
```

```
>>> df.dtypes
a      int32
b     object
c     object
d     object
e    float64
f    float64
dtype: object
```

Convert the DataFrame to use best possible dtypes.

```
>>> dfn = df.convert_dtypes()
>>> dfn
   a  b      c     d     e      f
0  1  x   True     h    10   <NA>
1  2  y  False     i  <NA>  100.5
2  3  z   <NA>  <NA>    20  200.0
```

```
>>> dfn.dtypes
a             Int32
b    string[python]
c           boolean
d    string[python]
e             Int64
f           Float64
dtype: object
```

Start with a Series of strings and missing data represented by `np.nan`.

```
>>> s = pd.Series(["a", "b", np.nan])
>>> s
0      a
1      b
2    NaN
dtype: object
```

Obtain a Series with dtype `StringDtype`.

```
>>> s.convert_dtypes()
0       a
1       b
2    <NA>
dtype: string
```

AlloViz.AlloViz.Elements.Nodes.copy

`Nodes.copy`(*deep: bool | None = True*) → None

Make a copy of this object's indices and data.

When `deep=True` (default), a new object will be created with a copy of the calling object's data and indices. Modifications to the data or indices of the copy will not be reflected in the original object (see notes below).

When `deep=False`, a new object will be created without copying the calling object's data or index (only references to the data and index are copied). Any changes to the data of the original will be reflected in the shallow copy (and vice versa).

Parameters

deep
[bool, default True] Make a deep copy, including a copy of the data and the indices. With `deep=False` neither the indices nor the data are copied.

Returns

Series or DataFrame
Object type matches caller.

Notes

When `deep=True`, data is copied but actual Python objects will not be copied recursively, only the reference to the object. This is in contrast to *copy.deepcopy* in the Standard Library, which recursively copies object data (see examples below).

While `Index` objects are copied when `deep=True`, the underlying numpy array is not copied for performance reasons. Since `Index` is immutable, the underlying data can be safely shared and a copy is not needed.

Since pandas is not thread safe, see the gotchas when copying in a threading environment.

When `copy_on_write` in pandas config is set to `True`, the `copy_on_write` config takes effect even when `deep=False`. This means that any changes to the copied data would make a new copy of the data upon write (and vice versa). Changes made to either the original or copied variable would not be reflected in the counterpart. See Copy_on_Write for more information.

Examples

```
>>> s = pd.Series([1, 2], index=["a", "b"])
>>> s
a    1
b    2
dtype: int64
```

```
>>> s_copy = s.copy()
>>> s_copy
a    1
b    2
dtype: int64
```

Shallow copy versus default (deep) copy:

```
>>> s = pd.Series([1, 2], index=["a", "b"])
>>> deep = s.copy()
>>> shallow = s.copy(deep=False)
```

Shallow copy shares data and index with original.

```
>>> s is shallow
False
>>> s.values is shallow.values and s.index is shallow.index
True
```

Deep copy has own copy of data and index.

```
>>> s is deep
False
>>> s.values is deep.values or s.index is deep.index
False
```

Updates to the data shared by shallow copy and original is reflected in both; deep copy remains unchanged.

```
>>> s.iloc[0] = 3
>>> shallow.iloc[1] = 4
>>> s
a    3
b    4
dtype: int64
>>> shallow
a    3
b    4
dtype: int64
>>> deep
a    1
b    2
dtype: int64
```

Note that when copying an object containing Python objects, a deep copy will copy the data, but will not do so recursively. Updating a nested data object will be reflected in the deep copy.

```
>>> s = pd.Series([[1, 2], [3, 4]])
>>> deep = s.copy()
>>> s[0][0] = 10
>>> s
0    [10, 2]
1     [3, 4]
dtype: object
>>> deep
0    [10, 2]
1     [3, 4]
dtype: object
```

** Copy-on-Write is set to true: **

```
>>> with pd.option_context("mode.copy_on_write", True):
...     s = pd.Series([1, 2], index=["a", "b"])
...     copy = s.copy(deep=False)
...     s.iloc[0] = 100
...     s
a    100
b      2
dtype: int64
>>> copy
a    1
b    2
dtype: int64
```

AlloViz.AlloViz.Elements.Nodes.corr

Nodes.**corr**(*method: CorrelationMethod = 'pearson'*, *min_periods: int = 1*, *numeric_only: bool = False*) → DataFrame

Compute pairwise correlation of columns, excluding NA/null values.

Parameters

method
[{'pearson', 'kendall', 'spearman'} or callable] Method of correlation:

- pearson : standard correlation coefficient
- kendall : Kendall Tau correlation coefficient
- spearman : Spearman rank correlation
- **callable: callable with input two 1d ndarrays**
  and returning a float. Note that the returned matrix from corr will have 1 along the diagonals and will be symmetric regardless of the callable's behavior.

min_periods
[int, optional] Minimum number of observations required per pair of columns to have a valid result. Currently only available for Pearson and Spearman correlation.

numeric_only
[bool, default False] Include only *float*, *int* or *boolean* data.

New in version 1.5.0.

Changed in version 2.0.0: The default value of `numeric_only` is now `False`.

Returns

DataFrame
Correlation matrix.

See also:

`DataFrame.corrwith`
Compute pairwise correlation with another DataFrame or Series.

`Series.corr`
Compute the correlation between two Series.

Notes

Pearson, Kendall and Spearman correlation are currently computed using pairwise complete observations.

- Pearson correlation coefficient
- Kendall rank correlation coefficient
- Spearman's rank correlation coefficient

Examples

```
>>> def histogram_intersection(a, b):
...     v = np.minimum(a, b).sum().round(decimals=1)
...     return v
>>> df = pd.DataFrame([(.2, .3), (.0, .6), (.6, .0), (.2, .1)],
...                   columns=['dogs', 'cats'])
>>> df.corr(method=histogram_intersection)
      dogs  cats
dogs   1.0   0.3
cats   0.3   1.0
```

```
>>> df = pd.DataFrame([(1, 1), (2, np.nan), (np.nan, 3), (4, 4)],
...                   columns=['dogs', 'cats'])
>>> df.corr(min_periods=3)
      dogs  cats
dogs   1.0   NaN
cats   NaN   1.0
```

AlloViz.AlloViz.Elements.Nodes.corrwith

Nodes.**corrwith**(*other: DataFrame | Series*, *axis: Axis = 0*, *drop: bool = False*, *method: CorrelationMethod = 'pearson'*, *numeric_only: bool = False*) → Series

Compute pairwise correlation.

Pairwise correlation is computed between rows or columns of DataFrame with rows or columns of Series or DataFrame. DataFrames are first aligned along both axes before computing the correlations.

Parameters

other
[DataFrame, Series] Object with which to compute correlations.

axis
[{0 or 'index', 1 or 'columns'}, default 0] The axis to use. 0 or 'index' to compute row-wise, 1 or 'columns' for column-wise.

drop
[bool, default False] Drop missing indices from result.

method
[{'pearson', 'kendall', 'spearman'} or callable] Method of correlation:

- pearson : standard correlation coefficient
- kendall : Kendall Tau correlation coefficient
- spearman : Spearman rank correlation
- **callable: callable with input two 1d ndarrays**
  and returning a float.

numeric_only
[bool, default False] Include only *float*, *int* or *boolean* data.

New in version 1.5.0.

Changed in version 2.0.0: The default value of `numeric_only` is now `False`.

Returns

Series
Pairwise correlations.

See also:

DataFrame.corr
Compute pairwise correlation of columns.

Examples

```
>>> index = ["a", "b", "c", "d", "e"]
>>> columns = ["one", "two", "three", "four"]
>>> df1 = pd.DataFrame(np.arange(20).reshape(5, 4), index=index,␣
↪columns=columns)
>>> df2 = pd.DataFrame(np.arange(16).reshape(4, 4), index=index[:4],␣
↪columns=columns)
>>> df1.corrwith(df2)
```

(continues on next page)

(continued from previous page)

```
one      1.0
two      1.0
three    1.0
four     1.0
dtype: float64
```

```
>>> df2.corrwith(df1, axis=1)
a    1.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

AlloViz.AlloViz.Elements.Nodes.count

Nodes.**count**(*axis: Axis = 0*, *numeric_only: bool = False*)

Count non-NA cells for each column or row.

The values *None*, *NaN*, *NaT*, `pandas.NA` are considered NA.

Parameters

axis
[{0 or 'index', 1 or 'columns'}, default 0] If 0 or 'index' counts are generated for each column. If 1 or 'columns' counts are generated for each row.

numeric_only
[bool, default False] Include only *float*, *int* or *boolean* data.

Returns

Series
For each column/row the number of non-NA/null entries.

See also:

`Series.count`
Number of non-NA elements in a Series.

`DataFrame.value_counts`
Count unique combinations of columns.

`DataFrame.shape`
Number of DataFrame rows and columns (including NA elements).

`DataFrame.isna`
Boolean same-sized DataFrame showing places of NA elements.

Examples

Constructing DataFrame from a dictionary:

```
>>> df = pd.DataFrame({"Person":
...                    ["John", "Myla", "Lewis", "John", "Myla"],
...                    "Age": [24., np.nan, 21., 33, 26],
...                    "Single": [False, True, True, True, False]})
>>> df
   Person   Age  Single
0    John  24.0   False
1    Myla   NaN    True
2   Lewis  21.0    True
3    John  33.0    True
4    Myla  26.0   False
```

Notice the uncounted NA values:

```
>>> df.count()
Person    5
Age       4
Single    5
dtype: int64
```

Counts for each **row**:

```
>>> df.count(axis='columns')
0    3
1    2
2    3
3    3
4    3
dtype: int64
```

AlloViz.AlloViz.Elements.Nodes.cov

`Nodes.cov`(*min_periods: None | int = None*, *ddof: int | None = 1*, *numeric_only: bool = False*) → DataFrame

Compute pairwise covariance of columns, excluding NA/null values.

Compute the pairwise covariance among the series of a DataFrame. The returned data frame is the covariance matrix of the columns of the DataFrame.

Both NA and null values are automatically excluded from the calculation. (See the note below about bias from missing values.) A threshold can be set for the minimum number of observations for each value created. Comparisons with observations below this threshold will be returned as `NaN`.

This method is generally used for the analysis of time series data to understand the relationship between different measures across time.

Parameters

min_periods
[int, optional] Minimum number of observations required per pair of columns to have a valid result.

ddof
[int, default 1] Delta degrees of freedom. The divisor used in calculations is `N - ddof`, where `N` represents the number of elements. This argument is applicable only when no `nan` is in the dataframe.

numeric_only
[bool, default False] Include only *float*, *int* or *boolean* data.

New in version 1.5.0.

Changed in version 2.0.0: The default value of `numeric_only` is now `False`.

Returns

DataFrame
The covariance matrix of the series of the DataFrame.

See also:

`Series.cov`
Compute covariance with another Series.

`core.window.ewm.ExponentialMovingWindow.cov`
Exponential weighted sample covariance.

`core.window.expanding.Expanding.cov`
Expanding sample covariance.

`core.window.rolling.Rolling.cov`
Rolling sample covariance.

Notes

Returns the covariance matrix of the DataFrame's time series. The covariance is normalized by N-ddof.

For DataFrames that have Series that are missing data (assuming that data is missing at random) the returned covariance matrix will be an unbiased estimate of the variance and covariance between the member Series.

However, for many applications this estimate may not be acceptable because the estimate covariance matrix is not guaranteed to be positive semi-definite. This could lead to estimate correlations having absolute values which are greater than one, and/or a non-invertible covariance matrix. See Estimation of covariance matrices for more details.

Examples

```
>>> df = pd.DataFrame([(1, 2), (0, 3), (2, 0), (1, 1)],
...                   columns=['dogs', 'cats'])
>>> df.cov()
          dogs      cats
dogs  0.666667 -1.000000
cats -1.000000  1.666667
```

```
>>> np.random.seed(42)
>>> df = pd.DataFrame(np.random.randn(1000, 5),
...                   columns=['a', 'b', 'c', 'd', 'e'])
>>> df.cov()
```

(continues on next page)

(continued from previous page)

```
          a         b         c         d         e
a  0.998438 -0.020161  0.059277 -0.008943  0.014144
b -0.020161  1.059352 -0.008543 -0.024738  0.009826
c  0.059277 -0.008543  1.010670 -0.001486 -0.000271
d -0.008943 -0.024738 -0.001486  0.921297 -0.013692
e  0.014144  0.009826 -0.000271 -0.013692  0.977795
```

Minimum number of periods

This method also supports an optional `min_periods` keyword that specifies the required minimum number of non-NA observations for each column pair in order to have a valid result:

```
>>> np.random.seed(42)
>>> df = pd.DataFrame(np.random.randn(20, 3),
...                   columns=['a', 'b', 'c'])
>>> df.loc[df.index[:5], 'a'] = np.nan
>>> df.loc[df.index[5:10], 'b'] = np.nan
>>> df.cov(min_periods=12)
          a         b         c
a  0.316741       NaN -0.150812
b       NaN  1.248003  0.191417
c -0.150812  0.191417  0.895202
```

AlloViz.AlloViz.Elements.Nodes.cummax

`Nodes.`**`cummax`**(*axis: Axis | None = None*, *skipna: bool = True*, **args*, ***kwargs*)

Return cumulative maximum over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative maximum.

Parameters

axis
[{0 or 'index', 1 or 'columns'}, default 0] The index or the name of the axis. 0 is equivalent to None or 'index'. For *Series* this parameter is unused and defaults to 0.

skipna
[bool, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

***args, **kwargs**
Additional keywords have no effect but might be accepted for compatibility with NumPy.

Returns

Series or DataFrame
Return cumulative maximum of Series or DataFrame.

See also:

`core.window.expanding.Expanding.max`
Similar functionality but ignores `NaN` values.

DataFrame.max
Return the maximum over DataFrame axis.

DataFrame.cummax
Return cumulative maximum over DataFrame axis.

DataFrame.cummin
Return cumulative minimum over DataFrame axis.

DataFrame.cumsum
Return cumulative sum over DataFrame axis.

DataFrame.cumprod
Return cumulative product over DataFrame axis.

Examples

Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cummax()
0    2.0
1    NaN
2    5.0
3    5.0
4    5.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cummax(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                   columns=list('AB'))
>>> df
```

(continues on next page)

(continued from previous page)

```
     A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the maximum in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cummax()
     A    B
0  2.0  1.0
1  3.0  NaN
2  3.0  1.0
```

To iterate over columns and find the maximum in each row, use `axis=1`

```
>>> df.cummax(axis=1)
     A    B
0  2.0  2.0
1  3.0  NaN
2  1.0  1.0
```

AlloViz.AlloViz.Elements.Nodes.cummin

`Nodes.cummin`(*axis: Axis | None = None*, *skipna: bool = True*, **args*, ***kwargs*)

Return cumulative minimum over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative minimum.

Parameters

axis
: [{0 or 'index', 1 or 'columns'}, default 0] The index or the name of the axis. 0 is equivalent to None or 'index'. For *Series* this parameter is unused and defaults to 0.

skipna
: [bool, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

***args, **kwargs**
: Additional keywords have no effect but might be accepted for compatibility with NumPy.

Returns

Series or DataFrame
: Return cumulative minimum of Series or DataFrame.

See also:

`core.window.expanding.Expanding.min`
: Similar functionality but ignores `NaN` values.

`DataFrame.min`
: Return the minimum over DataFrame axis.

DataFrame.cummax
Return cumulative maximum over DataFrame axis.

DataFrame.cummin
Return cumulative minimum over DataFrame axis.

DataFrame.cumsum
Return cumulative sum over DataFrame axis.

DataFrame.cumprod
Return cumulative product over DataFrame axis.

Examples

Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cummin()
0    2.0
1    NaN
2    2.0
3   -1.0
4   -1.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cummin(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                   columns=list('AB'))
>>> df
     A    B
0  2.0  1.0
```

(continues on next page)

(continued from previous page)

```
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the minimum in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cummin()
     A    B
0  2.0  1.0
1  2.0  NaN
2  1.0  0.0
```

To iterate over columns and find the minimum in each row, use `axis=1`

```
>>> df.cummin(axis=1)
     A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

AlloViz.AlloViz.Elements.Nodes.cumprod

`Nodes.cumprod`(*axis: Axis | None = None*, *skipna: bool = True*, **args*, ***kwargs*)

Return cumulative product over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative product.

Parameters

axis
: [{0 or 'index', 1 or 'columns'}, default 0] The index or the name of the axis. 0 is equivalent to None or 'index'. For *Series* this parameter is unused and defaults to 0.

skipna
: [bool, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

***args, **kwargs**
: Additional keywords have no effect but might be accepted for compatibility with NumPy.

Returns

Series or DataFrame
: Return cumulative product of Series or DataFrame.

See also:

`core.window.expanding.Expanding.prod`
: Similar functionality but ignores `NaN` values.

`DataFrame.prod`
: Return the product over DataFrame axis.

DataFrame.cummax
Return cumulative maximum over DataFrame axis.

DataFrame.cummin
Return cumulative minimum over DataFrame axis.

DataFrame.cumsum
Return cumulative sum over DataFrame axis.

DataFrame.cumprod
Return cumulative product over DataFrame axis.

Examples

Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cumprod()
0     2.0
1     NaN
2    10.0
3   -10.0
4    -0.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cumprod(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                   columns=list('AB'))
>>> df
     A    B
0  2.0  1.0
```

(continues on next page)

(continued from previous page)

```
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the product in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cumprod()
     A    B
0  2.0  1.0
1  6.0  NaN
2  6.0  0.0
```

To iterate over columns and find the product in each row, use `axis=1`

```
>>> df.cumprod(axis=1)
     A    B
0  2.0  2.0
1  3.0  NaN
2  1.0  0.0
```

AlloViz.AlloViz.Elements.Nodes.cumsum

Nodes.**cumsum**(*axis: Axis | None = None*, *skipna: bool = True*, **args*, ***kwargs*)

Return cumulative sum over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative sum.

Parameters

axis
: [{0 or 'index', 1 or 'columns'}, default 0] The index or the name of the axis. 0 is equivalent to None or 'index'. For *Series* this parameter is unused and defaults to 0.

skipna
: [bool, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

***args, **kwargs**
: Additional keywords have no effect but might be accepted for compatibility with NumPy.

Returns

Series or DataFrame
: Return cumulative sum of Series or DataFrame.

See also:

core.window.expanding.Expanding.sum
: Similar functionality but ignores `NaN` values.

DataFrame.sum
: Return the sum over DataFrame axis.

DataFrame.cummax
Return cumulative maximum over DataFrame axis.

DataFrame.cummin
Return cumulative minimum over DataFrame axis.

DataFrame.cumsum
Return cumulative sum over DataFrame axis.

DataFrame.cumprod
Return cumulative product over DataFrame axis.

Examples

Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cumsum()
0    2.0
1    NaN
2    7.0
3    6.0
4    6.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cumsum(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                   columns=list('AB'))
>>> df
     A    B
0  2.0  1.0
```

(continues on next page)

(continued from previous page)

```
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the sum in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cumsum()
     A    B
0  2.0  1.0
1  5.0  NaN
2  6.0  1.0
```

To iterate over columns and find the sum in each row, use `axis=1`

```
>>> df.cumsum(axis=1)
     A    B
0  2.0  3.0
1  3.0  NaN
2  1.0  1.0
```

AlloViz.AlloViz.Elements.Nodes.describe

`Nodes.describe`(*percentiles=None*, *include=None*, *exclude=None*) → None

Generate descriptive statistics.

Descriptive statistics include those that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding `NaN` values.

Analyzes both numeric and object series, as well as `DataFrame` column sets of mixed data types. The output will vary depending on what is provided. Refer to the notes below for more detail.

Parameters

percentiles
[list-like of numbers, optional] The percentiles to include in the output. All should fall between 0 and 1. The default is `[.25, .5, .75]`, which returns the 25th, 50th, and 75th percentiles.

include
['all', list-like of dtypes or None (default), optional] A white list of data types to include in the result. Ignored for `Series`. Here are the options:

- 'all' : All columns of the input will be included in the output.
- A list-like of dtypes : Limits the results to the provided data types. To limit the result to numeric types submit `numpy.number`. To limit it instead to object columns submit the `numpy.object` data type. Strings can also be used in the style of `select_dtypes` (e.g. `df.describe(include=['O'])`). To select pandas categorical columns, use `'category'`
- None (default) : The result will include all numeric columns.

exclude
[list-like of dtypes or None (default), optional,] A black list of data types to omit from the result. Ignored for `Series`. Here are the options:

- A list-like of dtypes : Excludes the provided data types from the result. To exclude numeric types submit `numpy.number`. To exclude object columns submit the data type `numpy.object`. Strings can also be used in the style of `select_dtypes` (e.g. `df.describe(exclude=['O'])`). To exclude pandas categorical columns, use `'category'`
- None (default) : The result will exclude nothing.

Returns

Series or DataFrame
Summary statistics of the Series or Dataframe provided.

See also:

`DataFrame.count`
Count number of non-NA/null observations.

`DataFrame.max`
Maximum of the values in the object.

`DataFrame.min`
Minimum of the values in the object.

`DataFrame.mean`
Mean of the values.

`DataFrame.std`
Standard deviation of the observations.

`DataFrame.select_dtypes`
Subset of a DataFrame including/excluding columns based on their dtype.

Notes

For numeric data, the result's index will include `count`, `mean`, `std`, `min`, `max` as well as lower, `50` and upper percentiles. By default the lower percentile is `25` and the upper percentile is `75`. The `50` percentile is the same as the median.

For object data (e.g. strings or timestamps), the result's index will include `count`, `unique`, `top`, and `freq`. The `top` is the most common value. The `freq` is the most common value's frequency. Timestamps also include the `first` and `last` items.

If multiple object values have the highest count, then the `count` and `top` results will be arbitrarily chosen from among those with the highest count.

For mixed data types provided via a `DataFrame`, the default is to return only an analysis of numeric columns. If the dataframe consists only of object and categorical data without any numeric columns, the default is to return an analysis of both the object and categorical columns. If `include='all'` is provided as an option, the result will include a union of attributes of each type.

The *include* and *exclude* parameters can be used to limit which columns in a `DataFrame` are analyzed for the output. The parameters are ignored when analyzing a `Series`.

Examples

Describing a numeric `Series`.

```
>>> s = pd.Series([1, 2, 3])
>>> s.describe()
count    3.0
mean     2.0
std      1.0
min      1.0
25%      1.5
50%      2.0
75%      2.5
max      3.0
dtype: float64
```

Describing a categorical `Series`.

```
>>> s = pd.Series(['a', 'a', 'b', 'c'])
>>> s.describe()
count     4
unique    3
top       a
freq      2
dtype: object
```

Describing a timestamp `Series`.

```
>>> s = pd.Series([
...     np.datetime64("2000-01-01"),
...     np.datetime64("2010-01-01"),
...     np.datetime64("2010-01-01")
... ])
>>> s.describe()
count                      3
mean     2006-09-01 08:00:00
min      2000-01-01 00:00:00
25%      2004-12-31 12:00:00
50%      2010-01-01 00:00:00
75%      2010-01-01 00:00:00
max      2010-01-01 00:00:00
dtype: object
```

Describing a `DataFrame`. By default only numeric fields are returned.

```
>>> df = pd.DataFrame({'categorical': pd.Categorical(['d','e','f']),
...                    'numeric': [1, 2, 3],
...                    'object': ['a', 'b', 'c']
...                   })
>>> df.describe()
       numeric
count      3.0
mean       2.0
std        1.0
```

(continues on next page)

(continued from previous page)

```
min        1.0
25%        1.5
50%        2.0
75%        2.5
max        3.0
```

Describing all columns of a `DataFrame` regardless of data type.

```
>>> df.describe(include='all')
       categorical  numeric object
count            3      3.0      3
unique           3      NaN      3
top              f      NaN      a
freq             1      NaN      1
mean           NaN      2.0    NaN
std            NaN      1.0    NaN
min            NaN      1.0    NaN
25%            NaN      1.5    NaN
50%            NaN      2.0    NaN
75%            NaN      2.5    NaN
max            NaN      3.0    NaN
```

Describing a column from a `DataFrame` by accessing it as an attribute.

```
>>> df.numeric.describe()
count    3.0
mean     2.0
std      1.0
min      1.0
25%      1.5
50%      2.0
75%      2.5
max      3.0
Name: numeric, dtype: float64
```

Including only numeric columns in a `DataFrame` description.

```
>>> df.describe(include=[np.number])
       numeric
count      3.0
mean       2.0
std        1.0
min        1.0
25%        1.5
50%        2.0
75%        2.5
max        3.0
```

Including only string columns in a `DataFrame` description.

```
>>> df.describe(include=[object])
       object
count       3
```

(continues on next page)

(continued from previous page)

```
unique      3
top         a
freq        1
```

Including only categorical columns from a `DataFrame` description.

```
>>> df.describe(include=['category'])
       categorical
count            3
unique           3
top              d
freq             1
```

Excluding numeric columns from a `DataFrame` description.

```
>>> df.describe(exclude=[np.number])
       categorical object
count            3      3
unique           3      3
top              f      a
freq             1      1
```

Excluding object columns from a `DataFrame` description.

```
>>> df.describe(exclude=[object])
       categorical  numeric
count            3      3.0
unique           3      NaN
top              f      NaN
freq             1      NaN
mean           NaN      2.0
std            NaN      1.0
min            NaN      1.0
25%            NaN      1.5
50%            NaN      2.0
75%            NaN      2.5
max            NaN      3.0
```

AlloViz.AlloViz.Elements.Nodes.diff

`Nodes.`**`diff`**(*periods: int = 1*, *axis: Axis = 0*) → DataFrame

First discrete difference of element.

Calculates the difference of a DataFrame element compared with another element in the DataFrame (default is element in previous row).

Parameters

periods
[int, default 1] Periods to shift for calculating difference, accepts negative values.

axis
[{0 or 'index', 1 or 'columns'}, default 0] Take difference over rows (0) or columns (1).

Returns

DataFrame
First differences of the Series.

See also:

`DataFrame.pct_change`
Percent change over given number of periods.

`DataFrame.shift`
Shift index by desired number of periods with an optional time freq.

`Series.diff`
First discrete difference of object.

Notes

For boolean dtypes, this uses `operator.xor()` rather than `operator.sub()`. The result is calculated according to current dtype in DataFrame, however dtype of the result is always float64.

Examples

Difference with previous row

```
>>> df = pd.DataFrame({'a': [1, 2, 3, 4, 5, 6],
...                    'b': [1, 1, 2, 3, 5, 8],
...                    'c': [1, 4, 9, 16, 25, 36]})
>>> df
   a  b   c
0  1  1   1
1  2  1   4
2  3  2   9
3  4  3  16
4  5  5  25
5  6  8  36
```

```
>>> df.diff()
     a    b     c
0  NaN  NaN   NaN
1  1.0  0.0   3.0
2  1.0  1.0   5.0
3  1.0  1.0   7.0
4  1.0  2.0   9.0
5  1.0  3.0  11.0
```

Difference with previous column

```
>>> df.diff(axis=1)
    a  b   c
0 NaN  0   0
1 NaN -1   3
2 NaN -1   7
3 NaN -1  13
```

(continues on next page)

(continued from previous page)

```
4 NaN  0  20
5 NaN  2  28
```

Difference with 3rd previous row

```
>>> df.diff(periods=3)
     a    b     c
0  NaN  NaN   NaN
1  NaN  NaN   NaN
2  NaN  NaN   NaN
3  3.0  2.0  15.0
4  3.0  4.0  21.0
5  3.0  6.0  27.0
```

Difference with following row

```
>>> df.diff(periods=-1)
     a    b     c
0 -1.0  0.0  -3.0
1 -1.0 -1.0  -5.0
2 -1.0 -1.0  -7.0
3 -1.0 -2.0  -9.0
4 -1.0 -3.0 -11.0
5  NaN  NaN   NaN
```

Overflow in input dtype

```
>>> df = pd.DataFrame({'a': [1, 0]}, dtype=np.uint8)
>>> df.diff()
       a
0    NaN
1  255.0
```

AlloViz.AlloViz.Elements.Nodes.div

`Nodes.`**`div`**(*other*, *axis: Axis = 'columns'*, *level=None*, *fill_value=None*)

Get Floating division of dataframe and other, element-wise (binary operator *truediv*).

Equivalent to `dataframe / other`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *rtruediv*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *floordiv*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other
: [scalar, sequence, Series, dict or DataFrame] Any single or multiple element data structure, or list-like object.

axis
: [{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns. (1 or 'columns'). For Series input, axis to match Series index on.

level
[int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value
[float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame
Result of the arithmetic operation.

See also:

`DataFrame.add`
Add DataFrames.

`DataFrame.sub`
Subtract DataFrames.

`DataFrame.mul`
Multiply DataFrames.

`DataFrame.div`
Divide DataFrames (float division).

`DataFrame.truediv`
Divide DataFrames (float division).

`DataFrame.floordiv`
Divide DataFrames (integer division).

`DataFrame.mod`
Calculate modulo (remainder after division).

`DataFrame.pow`
Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a dictionary by axis.

```
>>> df.mul({'angles': 0, 'degrees': 2})
            angles  degrees
circle           0      720
```

(continues on next page)

(continued from previous page)

```
triangle         0      360
rectangle        0      720
```

```
>>> df.mul({'circle': 0, 'triangle': 2, 'rectangle': 3}, axis='index')
           angles  degrees
circle          0        0
triangle        6      360
rectangle      12     1080
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
```

(continues on next page)

(continued from previous page)

```
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

AlloViz.AlloViz.Elements.Nodes.divide

Nodes.**divide**(*other*, *axis: Axis = 'columns'*, *level=None*, *fill_value=None*)

Get Floating division of dataframe and other, element-wise (binary operator *truediv*).

Equivalent to `dataframe / other`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *rtruediv*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *floordiv*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other
: [scalar, sequence, Series, dict or DataFrame] Any single or multiple element data structure, or list-like object.

axis
: [{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns. (1 or 'columns'). For Series input, axis to match Series index on.

level
: [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value
: [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame
: Result of the arithmetic operation.

See also:

`DataFrame.add`
: Add DataFrames.

`DataFrame.sub`
: Subtract DataFrames.

`DataFrame.mul`
: Multiply DataFrames.

`DataFrame.div`
: Divide DataFrames (float division).

`DataFrame.truediv`
: Divide DataFrames (float division).

DataFrame.floordiv
Divide DataFrames (integer division).

DataFrame.mod
Calculate modulo (remainder after division).

DataFrame.pow
Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a dictionary by axis.

```
>>> df.mul({'angles': 0, 'degrees': 2})
            angles  degrees
circle           0      720
triangle         0      360
rectangle        0      720
```

```
>>> df.mul({'circle': 0, 'triangle': 2, 'rectangle': 3}, axis='index')
            angles  degrees
circle           0        0
triangle         6      360
rectangle       12     1080
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
```

(continues on next page)

(continued from previous page)

```
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

AlloViz.AlloViz.Elements.Nodes.dot

`Nodes.dot`(*other: AnyArrayLike | DataFrame*) → DataFrame | Series

Compute the matrix multiplication between the DataFrame and other.

This method computes the matrix product between the DataFrame and the values of an other Series, DataFrame or a numpy array.

It can also be called using `self @ other`.

Parameters

other
[Series, DataFrame or array-like] The other object to compute the matrix product with.

Returns

Series or DataFrame
If other is a Series, return the matrix product between self and other as a Series. If other is a DataFrame or a numpy.array, return the matrix product of self and other in a DataFrame of a np.array.

See also:

`Series.dot`
Similar method for Series.

Notes

The dimensions of DataFrame and other must be compatible in order to compute the matrix multiplication. In addition, the column names of DataFrame and the index of other must contain the same values, as they will be aligned prior to the multiplication.

The dot method for Series computes the inner product, instead of the matrix product here.

Examples

Here we multiply a DataFrame with a Series.

```
>>> df = pd.DataFrame([[0, 1, -2, -1], [1, 1, 1, 1]])
>>> s = pd.Series([1, 1, 2, 1])
>>> df.dot(s)
0    -4
1     5
dtype: int64
```

Here we multiply a DataFrame with another DataFrame.

```
>>> other = pd.DataFrame([[0, 1], [1, 2], [-1, -1], [2, 0]])
>>> df.dot(other)
    0   1
0   1   4
1   2   2
```

Note that the dot method give the same result as @

```
>>> df @ other
    0   1
0   1   4
1   2   2
```

The dot method works also if other is an np.array.

```
>>> arr = np.array([[0, 1], [1, 2], [-1, -1], [2, 0]])
>>> df.dot(arr)
    0   1
0   1   4
1   2   2
```

Note how shuffling of the objects does not change the result.

```
>>> s2 = s.reindex([1, 0, 2, 3])
>>> df.dot(s2)
0    -4
1     5
dtype: int64
```

AlloViz.AlloViz.Elements.Nodes.drop

Nodes.**drop**(*labels: IndexLabel | None = None*, ***, *axis: Axis = 0*, *index: IndexLabel | None = None*, *columns: IndexLabel | None = None*, *level: Level | None = None*, *inplace: bool = False*, *errors: IgnoreRaise = 'raise'*) → DataFrame | None

Drop specified labels from rows or columns.

Remove rows or columns by specifying label names and corresponding axis, or by directly specifying index or column names. When using a multi-index, labels on different levels can be removed by specifying the level. See the user guide for more information about the now unused levels.

Parameters

labels
[single label or list-like] Index or column labels to drop. A tuple will be used as a single label and not treated as a list-like.

axis
[{0 or 'index', 1 or 'columns'}, default 0] Whether to drop labels from the index (0 or 'index') or columns (1 or 'columns').

index
[single label or list-like] Alternative to specifying axis (`labels, axis=0` is equivalent to `index=labels`).

columns
[single label or list-like] Alternative to specifying axis (`labels, axis=1` is equivalent to `columns=labels`).

level
[int or level name, optional] For MultiIndex, level from which the labels will be removed.

inplace
[bool, default False] If False, return a copy. Otherwise, do operation in place and return None.

errors
[{'ignore', 'raise'}, default 'raise'] If 'ignore', suppress error and only existing labels are dropped.

Returns

DataFrame or None
Returns DataFrame or None DataFrame with the specified index or column labels removed or None if inplace=True.

Raises

KeyError
If any of the labels is not found in the selected axis.

See also:

`DataFrame.loc`
Label-location based indexer for selection by label.

`DataFrame.dropna`
Return DataFrame with labels on given axis omitted where (all or any) data are missing.

DataFrame.drop_duplicates
Return DataFrame with duplicate rows removed, optionally only considering certain columns.

Series.drop
Return Series with specified index labels removed.

Examples

```
>>> df = pd.DataFrame(np.arange(12).reshape(3, 4),
...                   columns=['A', 'B', 'C', 'D'])
>>> df
   A  B   C   D
0  0  1   2   3
1  4  5   6   7
2  8  9  10  11
```

Drop columns

```
>>> df.drop(['B', 'C'], axis=1)
   A   D
0  0   3
1  4   7
2  8  11
```

```
>>> df.drop(columns=['B', 'C'])
   A   D
0  0   3
1  4   7
2  8  11
```

Drop a row by index

```
>>> df.drop([0, 1])
   A  B   C   D
2  8  9  10  11
```

Drop columns and/or rows of MultiIndex DataFrame

```
>>> midx = pd.MultiIndex(levels=[['llama', 'cow', 'falcon'],
...                              ['speed', 'weight', 'length']],
...                      codes=[[0, 0, 0, 1, 1, 1, 2, 2, 2],
...                             [0, 1, 2, 0, 1, 2, 0, 1, 2]])
>>> df = pd.DataFrame(index=midx, columns=['big', 'small'],
...                   data=[[45, 30], [200, 100], [1.5, 1], [30, 20],
...                         [250, 150], [1.5, 0.8], [320, 250],
...                         [1, 0.8], [0.3, 0.2]])
>>> df
                big     small
llama   speed   45.0    30.0
        weight  200.0   100.0
        length  1.5     1.0
cow     speed   30.0    20.0
        weight  250.0   150.0
```

(continues on next page)

(continued from previous page)

```
        length  1.5     0.8
falcon  speed   320.0   250.0
        weight  1.0     0.8
        length  0.3     0.2
```

Drop a specific index combination from the MultiIndex DataFrame, i.e., drop the combination `'falcon'` and `'weight'`, which deletes only the corresponding row

```
>>> df.drop(index=('falcon', 'weight'))
                big     small
llama   speed   45.0    30.0
        weight  200.0   100.0
        length  1.5     1.0
cow     speed   30.0    20.0
        weight  250.0   150.0
        length  1.5     0.8
falcon  speed   320.0   250.0
        length  0.3     0.2
```

```
>>> df.drop(index='cow', columns='small')
                big
llama   speed   45.0
        weight  200.0
        length  1.5
falcon  speed   320.0
        weight  1.0
        length  0.3
```

```
>>> df.drop(index='length', level=1)
                big     small
llama   speed   45.0    30.0
        weight  200.0   100.0
cow     speed   30.0    20.0
        weight  250.0   150.0
falcon  speed   320.0   250.0
        weight  1.0     0.8
```

AlloViz.AlloViz.Elements.Nodes.drop_duplicates

Nodes.**drop_duplicates**(*subset: Hashable | Sequence[Hashable] | None = None*, ***, *keep: DropKeep = 'first'*, *inplace: bool = False*, *ignore_index: bool = False*) → DataFrame | None

Return DataFrame with duplicate rows removed.

Considering certain columns is optional. Indexes, including time indexes are ignored.

Parameters

subset
: [column label or sequence of labels, optional] Only consider certain columns for identifying duplicates, by default use all of the columns.

keep
: [{'first', 'last', `False`}, default 'first'] Determines which duplicates (if any) to keep.

- 'first' : Drop duplicates except for the first occurrence.
- 'last' : Drop duplicates except for the last occurrence.
- `False` : Drop all duplicates.

inplace
[bool, default `False`] Whether to modify the DataFrame rather than creating a new one.

ignore_index
[bool, default `False`] If `True`, the resulting axis will be labeled 0, 1, …, n - 1.

Returns

DataFrame or None
DataFrame with duplicates removed or None if `inplace=True`.

See also:

`DataFrame.value_counts`
Count unique combinations of columns.

Examples

Consider dataset containing ramen rating.

```
>>> df = pd.DataFrame({
...     'brand': ['Yum Yum', 'Yum Yum', 'Indomie', 'Indomie', 'Indomie'],
...     'style': ['cup', 'cup', 'cup', 'pack', 'pack'],
...     'rating': [4, 4, 3.5, 15, 5]
... })
>>> df
    brand style  rating
0  Yum Yum   cup     4.0
1  Yum Yum   cup     4.0
2  Indomie   cup     3.5
3  Indomie  pack    15.0
4  Indomie  pack     5.0
```

By default, it removes duplicate rows based on all columns.

```
>>> df.drop_duplicates()
    brand style  rating
0  Yum Yum   cup     4.0
2  Indomie   cup     3.5
3  Indomie  pack    15.0
4  Indomie  pack     5.0
```

To remove duplicates on specific column(s), use `subset`.

```
>>> df.drop_duplicates(subset=['brand'])
    brand style  rating
0  Yum Yum   cup     4.0
2  Indomie   cup     3.5
```

To remove duplicates and keep last occurrences, use `keep`.

```
>>> df.drop_duplicates(subset=['brand', 'style'], keep='last')
    brand style  rating
1  Yum Yum   cup     4.0
2  Indomie   cup     3.5
4  Indomie  pack     5.0
```

AlloViz.AlloViz.Elements.Nodes.droplevel

Nodes.**droplevel**(*level: Hashable | Sequence[Hashable]*, *axis: int | Literal['index', 'columns', 'rows'] = 0*) → None

Return Series/DataFrame with requested index / column level(s) removed.

Parameters

level
[int, str, or list-like] If a string is given, must be the name of a level If list-like, elements must be names or positional indexes of levels.

axis
[{0 or 'index', 1 or 'columns'}, default 0] Axis along which the level(s) is removed:

- 0 or 'index': remove level(s) in column.
- 1 or 'columns': remove level(s) in row.

For *Series* this parameter is unused and defaults to 0.

Returns

Series/DataFrame
Series/DataFrame with requested index / column level(s) removed.

Examples

```
>>> df = pd.DataFrame([
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12]
... ]).set_index([0, 1]).rename_axis(['a', 'b'])
```

```
>>> df.columns = pd.MultiIndex.from_tuples([
...     ('c', 'e'), ('d', 'f')
... ], names=['level_1', 'level_2'])
```

```
>>> df
level_1   c   d
level_2   e   f
a b
1 2       3   4
5 6       7   8
9 10     11  12
```

```
>>> df.droplevel('a')
level_1   c   d
level_2   e   f
b
2         3   4
6         7   8
10       11  12
```

```
>>> df.droplevel('level_2', axis=1)
level_1   c   d
a b
1 2       3   4
5 6       7   8
9 10     11  12
```

AlloViz.AlloViz.Elements.Nodes.dropna

Nodes.**dropna**(*, *axis: Axis = 0*, *how: AnyAll | lib.NoDefault = _NoDefault.no_default*, *thresh: int | lib.NoDefault = _NoDefault.no_default*, *subset: IndexLabel | None = None*, *inplace: bool = False*, *ignore_index: bool = False*) → DataFrame | None

Remove missing values.

See the User Guide for more on which values are considered missing, and how to work with missing data.

Parameters

axis
[{0 or 'index', 1 or 'columns'}, default 0] Determine if rows or columns which contain missing values are removed.

- 0, or 'index' : Drop rows which contain missing values.
- 1, or 'columns' : Drop columns which contain missing value.

Only a single axis is allowed.

how
[{'any', 'all'}, default 'any'] Determine if row or column is removed from DataFrame, when we have at least one NA or all NA.

- 'any' : If any NA values are present, drop that row or column.
- 'all' : If all values are NA, drop that row or column.

thresh
[int, optional] Require that many non-NA values. Cannot be combined with how.

subset
[column label or sequence of labels, optional] Labels along other axis to consider, e.g. if you are dropping rows these would be a list of columns to include.

inplace
[bool, default False] Whether to modify the DataFrame rather than creating a new one.

ignore_index
[bool, default `False`] If `True`, the resulting axis will be labeled 0, 1, …, n - 1.

New in version 2.0.0.

Returns

DataFrame or None
DataFrame with NA entries dropped from it or None if `inplace=True`.

See also:

`DataFrame.isna`
Indicate missing values.

`DataFrame.notna`
Indicate existing (non-missing) values.

`DataFrame.fillna`
Replace missing values.

`Series.dropna`
Drop missing values.

`Index.dropna`
Drop missing indices.

Examples

```
>>> df = pd.DataFrame({"name": ['Alfred', 'Batman', 'Catwoman'],
...                    "toy": [np.nan, 'Batmobile', 'Bullwhip'],
...                    "born": [pd.NaT, pd.Timestamp("1940-04-25"),
...                             pd.NaT]})
>>> df
       name        toy       born
0    Alfred        NaN        NaT
1    Batman  Batmobile 1940-04-25
2  Catwoman   Bullwhip        NaT
```

Drop the rows where at least one element is missing.

```
>>> df.dropna()
     name        toy       born
1  Batman  Batmobile 1940-04-25
```

Drop the columns where at least one element is missing.

```
>>> df.dropna(axis='columns')
       name
0    Alfred
1    Batman
2  Catwoman
```

Drop the rows where all elements are missing.

```
>>> df.dropna(how='all')
       name        toy       born
0    Alfred        NaN        NaT
1    Batman  Batmobile 1940-04-25
2  Catwoman   Bullwhip        NaT
```

Keep only the rows with at least 2 non-NA values.

```
>>> df.dropna(thresh=2)
       name        toy       born
1    Batman  Batmobile 1940-04-25
2  Catwoman   Bullwhip        NaT
```

Define in which columns to look for missing values.

```
>>> df.dropna(subset=['name', 'toy'])
       name        toy       born
1    Batman  Batmobile 1940-04-25
2  Catwoman   Bullwhip        NaT
```

AlloViz.AlloViz.Elements.Nodes.duplicated

Nodes.**duplicated**(*subset: Hashable | Sequence[Hashable] | None = None*, *keep: DropKeep = 'first'*) → Series

Return boolean Series denoting duplicate rows.

Considering certain columns is optional.

Parameters

subset
[column label or sequence of labels, optional] Only consider certain columns for identifying duplicates, by default use all of the columns.

keep
[{'first', 'last', False}, default 'first'] Determines which duplicates (if any) to mark.

- `first` : Mark duplicates as `True` except for the first occurrence.
- `last` : Mark duplicates as `True` except for the last occurrence.
- False : Mark all duplicates as `True`.

Returns

Series
Boolean series for each duplicated rows.

See also:

`Index.duplicated`
Equivalent method on index.

`Series.duplicated`
Equivalent method on Series.

`Series.drop_duplicates`
Remove duplicate values from Series.

`DataFrame.drop_duplicates`
Remove duplicate values from DataFrame.

Examples

Consider dataset containing ramen rating.

```
>>> df = pd.DataFrame({
...     'brand': ['Yum Yum', 'Yum Yum', 'Indomie', 'Indomie', 'Indomie'],
...     'style': ['cup', 'cup', 'cup', 'pack', 'pack'],
...     'rating': [4, 4, 3.5, 15, 5]
... })
>>> df
    brand style  rating
0  Yum Yum   cup     4.0
1  Yum Yum   cup     4.0
2  Indomie   cup     3.5
3  Indomie  pack    15.0
4  Indomie  pack     5.0
```

By default, for each set of duplicated values, the first occurrence is set on False and all others on True.

```
>>> df.duplicated()
0    False
1     True
2    False
3    False
4    False
dtype: bool
```

By using ‘last’, the last occurrence of each set of duplicated values is set on False and all others on True.

```
>>> df.duplicated(keep='last')
0     True
1    False
2    False
3    False
4    False
dtype: bool
```

By setting `keep` on False, all duplicates are True.

```
>>> df.duplicated(keep=False)
0     True
1     True
2    False
3    False
4    False
dtype: bool
```

To find duplicates on specific column(s), use `subset`.

```
>>> df.duplicated(subset=['brand'])
0    False
1     True
2    False
3     True
```

(continues on next page)

(continued from previous page)

```
4     True
dtype: bool
```

AlloViz.AlloViz.Elements.Nodes.eq

Nodes.**eq**(*other*, *axis: Axis = 'columns'*, *level=None*)

Get Equal to of dataframe and other, element-wise (binary operator *eq*).

Among flexible wrappers (*eq*, *ne*, *le*, *lt*, *ge*, *gt*) to comparison operators.

Equivalent to ==, *!*=, <=, <, >=, > with support to choose axis (rows or columns) and level for comparison.

Parameters

other
: [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.

axis
: [{0 or 'index', 1 or 'columns'}, default 'columns'] Whether to compare by the index (0 or 'index') or columns (1 or 'columns').

level
: [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

Returns

DataFrame of bool
: Result of the comparison.

See also:

DataFrame.eq
: Compare DataFrames for equality elementwise.

DataFrame.ne
: Compare DataFrames for inequality elementwise.

DataFrame.le
: Compare DataFrames for less than inequality or equality elementwise.

DataFrame.lt
: Compare DataFrames for strictly less than inequality elementwise.

DataFrame.ge
: Compare DataFrames for greater than inequality or equality elementwise.

DataFrame.gt
: Compare DataFrames for strictly greater than inequality elementwise.

Notes

Mismatched indices will be unioned together. *NaN* values are considered different (i.e. *NaN* != *NaN*).

Examples

```
>>> df = pd.DataFrame({'cost': [250, 150, 100],
...                    'revenue': [100, 250, 300]},
...                   index=['A', 'B', 'C'])
>>> df
   cost  revenue
A   250      100
B   150      250
C   100      300
```

Comparison with a scalar, using either the operator or method:

```
>>> df == 100
    cost  revenue
A  False     True
B  False    False
C   True    False
```

```
>>> df.eq(100)
    cost  revenue
A  False     True
B  False    False
C   True    False
```

When *other* is a `Series`, the columns of a DataFrame are aligned with the index of *other* and broadcast:

```
>>> df != pd.Series([100, 250], index=["cost", "revenue"])
    cost  revenue
A   True     True
B   True    False
C  False     True
```

Use the method to control the broadcast axis:

```
>>> df.ne(pd.Series([100, 300], index=["A", "D"]), axis='index')
   cost  revenue
A  True    False
B  True     True
C  True     True
D  True     True
```

When comparing to an arbitrary sequence, the number of columns must match the number elements in *other*:

```
>>> df == [250, 100]
    cost  revenue
A   True     True
```

(continues on next page)

(continued from previous page)

```
B  False    False
C  False    False
```

Use the method to control the axis:

```
>>> df.eq([250, 250, 100], axis='index')
    cost  revenue
A   True    False
B  False     True
C   True    False
```

Compare to a DataFrame of different shape.

```
>>> other = pd.DataFrame({'revenue': [300, 250, 100, 150]},
...                      index=['A', 'B', 'C', 'D'])
>>> other
   revenue
A      300
B      250
C      100
D      150
```

```
>>> df.gt(other)
    cost  revenue
A  False    False
B  False    False
C  False     True
D  False    False
```

Compare to a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'cost': [250, 150, 100, 150, 300, 220],
...                              'revenue': [100, 250, 300, 200, 175, 225]},
...                             index=[['Q1', 'Q1', 'Q1', 'Q2', 'Q2', 'Q2'],
...                                    ['A', 'B', 'C', 'A', 'B', 'C']])
>>> df_multindex
      cost  revenue
Q1 A   250      100
   B   150      250
   C   100      300
Q2 A   150      200
   B   300      175
   C   220      225
```

```
>>> df.le(df_multindex, level=1)
       cost  revenue
Q1 A   True     True
   B   True     True
   C   True     True
Q2 A  False     True
   B   True    False
   C   True    False
```

AlloViz.AlloViz.Elements.Nodes.equals

Nodes.**equals**(*other: object*) → bool

Test whether two objects contain the same elements.

This function allows two Series or DataFrames to be compared against each other to see if they have the same shape and elements. NaNs in the same location are considered equal.

The row/column index do not need to have the same type, as long as the values are considered equal. Corresponding columns must be of the same dtype.

Parameters

other
[Series or DataFrame] The other Series or DataFrame to be compared with the first.

Returns

bool
True if all elements are the same in both objects, False otherwise.

See also:

Series.eq
Compare two Series objects of the same length and return a Series where each element is True if the element in each Series is equal, False otherwise.

DataFrame.eq
Compare two DataFrame objects of the same shape and return a DataFrame where each element is True if the respective element in each DataFrame is equal, False otherwise.

testing.assert_series_equal
Raises an AssertionError if left and right are not equal. Provides an easy interface to ignore inequality in dtypes, indexes and precision among others.

testing.assert_frame_equal
Like assert_series_equal, but targets DataFrames.

numpy.array_equal
Return True if two arrays have the same shape and elements, False otherwise.

Examples

```
>>> df = pd.DataFrame({1: [10], 2: [20]})
>>> df
    1   2
0  10  20
```

DataFrames df and exactly_equal have the same types and values for their elements and column labels, which will return True.

```
>>> exactly_equal = pd.DataFrame({1: [10], 2: [20]})
>>> exactly_equal
    1   2
0  10  20
>>> df.equals(exactly_equal)
True
```

DataFrames df and different_column_type have the same element types and values, but have different types for the column labels, which will still return True.

```
>>> different_column_type = pd.DataFrame({1.0: [10], 2.0: [20]})
>>> different_column_type
   1.0  2.0
0   10   20
>>> df.equals(different_column_type)
True
```

DataFrames df and different_data_type have different types for the same values for their elements, and will return False even though their column labels are the same values and types.

```
>>> different_data_type = pd.DataFrame({1: [10.0], 2: [20.0]})
>>> different_data_type
      1     2
0  10.0  20.0
>>> df.equals(different_data_type)
False
```

AlloViz.AlloViz.Elements.Nodes.eval

`Nodes.`**`eval`**(*expr: str*, *, *inplace: bool = False*, ***kwargs*) → Any | None

Evaluate a string describing operations on DataFrame columns.

Operates on columns only, not specific rows or elements. This allows *eval* to run arbitrary code, which can make you vulnerable to code injection if you pass user input to this function.

Parameters

expr
: [str] The expression string to evaluate.

inplace
: [bool, default False] If the expression contains an assignment, whether to perform the operation inplace and mutate the existing DataFrame. Otherwise, a new DataFrame is returned.

****kwargs**
: See the documentation for *eval()* for complete details on the keyword arguments accepted by `query()`.

Returns

ndarray, scalar, pandas object, or None
: The result of the evaluation or None if `inplace=True`.

See also:

`DataFrame.query`
: Evaluates a boolean expression to query the columns of a frame.

`DataFrame.assign`
: Can evaluate an expression or function to create new values for a column.

eval
: Evaluate a Python expression as a string using various backends.

Notes

For more details see the API documentation for *eval()*. For detailed examples see enhancing performance with eval.

Examples

```
>>> df = pd.DataFrame({'A': range(1, 6), 'B': range(10, 0, -2)})
>>> df
   A   B
0  1  10
1  2   8
2  3   6
3  4   4
4  5   2
>>> df.eval('A + B')
0    11
1    10
2     9
3     8
4     7
dtype: int64
```

Assignment is allowed though by default the original DataFrame is not modified.

```
>>> df.eval('C = A + B')
   A   B   C
0  1  10  11
1  2   8  10
2  3   6   9
3  4   4   8
4  5   2   7
>>> df
   A   B
0  1  10
1  2   8
2  3   6
3  4   4
4  5   2
```

Multiple columns can be assigned to using multi-line expressions:

```
>>> df.eval(
...     '''
... C = A + B
... D = A - B
... '''
... )
   A   B   C  D
0  1  10  11 -9
1  2   8  10 -6
2  3   6   9 -3
```

(continues on next page)

(continued from previous page)

```
3  4   4   8  0
4  5   2   7  3
```

AlloViz.AlloViz.Elements.Nodes.ewm

Nodes.**ewm**(*com: float | None = None*, *span: float | None = None*, *halflife: float | TimedeltaConvertibleTypes | None = None*, *alpha: float | None = None*, *min_periods: int | None = 0*, *adjust: bool_t = True*, *ignore_na: bool_t = False*, *axis: Axis | lib.NoDefault = _NoDefault.no_default*, *times: np.ndarray | DataFrame | Series | None = None*, *method: Literal['single', 'table'] = 'single'*) → ExponentialMovingWindow

Provide exponentially weighted (EW) calculations.

Exactly one of `com`, `span`, `halflife`, or `alpha` must be provided if `times` is not provided. If `times` is provided, `halflife` and one of `com`, `span` or `alpha` may be provided.

Parameters

com
[float, optional] Specify decay in terms of center of mass

$\alpha = 1/(1 + com)$, for $com \geq 0$.

span
[float, optional] Specify decay in terms of span

$\alpha = 2/(span + 1)$, for $span \geq 1$.

halflife
[float, str, timedelta, optional] Specify decay in terms of half-life

$\alpha = 1 - \exp(-\ln(2)/halflife)$, for $halflife > 0$.

If `times` is specified, a timedelta convertible unit over which an observation decays to half its value. Only applicable to `mean()`, and halflife value will not apply to the other functions.

alpha
[float, optional] Specify smoothing factor α directly

$0 < \alpha \leq 1$.

min_periods
[int, default 0] Minimum number of observations in window required to have a value; otherwise, result is `np.nan`.

adjust
[bool, default True] Divide by decaying adjustment factor in beginning periods to account for imbalance in relative weightings (viewing EWMA as a moving average).

- When `adjust=True` (default), the EW function is calculated using weights $w_i = (1 - \alpha)^i$. For example, the EW moving average of the series $[x_0, x_1, ..., x_t]$ would be:

$$y_t = \frac{x_t + (1 - \alpha)x_{t-1} + (1 - \alpha)^2 x_{t-2} + ... + (1 - \alpha)^t x_0}{1 + (1 - \alpha) + (1 - \alpha)^2 + ... + (1 - \alpha)^t}$$

- When `adjust=False`, the exponentially weighted function is calculated recursively:

$$y_0 = x_0$$
$$y_t = (1 - \alpha) y_{t-1} + \alpha x_t,$$

ignore_na
[bool, default False] Ignore missing values when calculating weights.

- When `ignore_na=False` (default), weights are based on absolute positions. For example, the weights of x_0 and x_2 used in calculating the final weighted average of [x_0, None, x_2] are $(1-\alpha)^2$ and 1 if `adjust=True`, and $(1-\alpha)^2$ and α if `adjust=False`.
- When `ignore_na=True`, weights are based on relative positions. For example, the weights of x_0 and x_2 used in calculating the final weighted average of [x_0, None, x_2] are $1-\alpha$ and 1 if `adjust=True`, and $1-\alpha$ and α if `adjust=False`.

axis
[{0, 1}, default 0] If `0` or `'index'`, calculate across the rows.

If `1` or `'columns'`, calculate across the columns.

For *Series* this parameter is unused and defaults to 0.

times
[np.ndarray, Series, default None] Only applicable to `mean()`.

Times corresponding to the observations. Must be monotonically increasing and `datetime64[ns]` dtype.

If 1-D array like, a sequence with the same shape as the observations.

method
[str {'single', 'table'}, default 'single'] New in version 1.4.0.

Execute the rolling operation per single column or row (`'single'`) or over the entire object (`'table'`).

This argument is only implemented when specifying `engine='numba'` in the method call.

Only applicable to `mean()`

Returns

pandas.api.typing.ExponentialMovingWindow

See also:

rolling
Provides rolling window calculations.

expanding
Provides expanding transformations.

Notes

See Windowing Operations for further usage details and examples.

Examples

```
>>> df = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]})
>>> df
     B
0  0.0
1  1.0
2  2.0
3  NaN
4  4.0
```

```
>>> df.ewm(com=0.5).mean()
          B
0  0.000000
1  0.750000
2  1.615385
3  1.615385
4  3.670213
>>> df.ewm(alpha=2 / 3).mean()
          B
0  0.000000
1  0.750000
2  1.615385
3  1.615385
4  3.670213
```

adjust

```
>>> df.ewm(com=0.5, adjust=True).mean()
          B
0  0.000000
1  0.750000
2  1.615385
3  1.615385
4  3.670213
>>> df.ewm(com=0.5, adjust=False).mean()
          B
0  0.000000
1  0.666667
2  1.555556
3  1.555556
4  3.650794
```

ignore_na

```
>>> df.ewm(com=0.5, ignore_na=True).mean()
          B
0  0.000000
```

(continues on next page)

(continued from previous page)

```
1  0.750000
2  1.615385
3  1.615385
4  3.225000
>>> df.ewm(com=0.5, ignore_na=False).mean()
          B
0  0.000000
1  0.750000
2  1.615385
3  1.615385
4  3.670213
```

times

Exponentially weighted mean with weights calculated with a timedelta `halflife` relative to `times`.

```
>>> times = ['2020-01-01', '2020-01-03', '2020-01-10', '2020-01-15', '2020-01-
↪17']
>>> df.ewm(halflife='4 days', times=pd.DatetimeIndex(times)).mean()
          B
0  0.000000
1  0.585786
2  1.523889
3  1.523889
4  3.233686
```

AlloViz.AlloViz.Elements.Nodes.expanding

Nodes.**expanding**(*min_periods: int = 1*, *axis: int | Literal['index', 'columns', 'rows'] | Literal[_NoDefault.no_default] = _NoDefault.no_default*, *method: Literal['single', 'table'] = 'single'*) → Expanding

Provide expanding window calculations.

Parameters

min_periods
: [int, default 1] Minimum number of observations in window required to have a value; otherwise, result is `np.nan`.

axis
: [int or str, default 0] If `0` or `'index'`, roll across the rows.

  If `1` or `'columns'`, roll across the columns.

  For *Series* this parameter is unused and defaults to 0.

method
: [str {'single', 'table'}, default 'single'] Execute the rolling operation per single column or row (`'single'`) or over the entire object (`'table'`).

  This argument is only implemented when specifying `engine='numba'` in the method call.

  New in version 1.3.0.

Returns

pandas.api.typing.Expanding

See also:

rolling
Provides rolling window calculations.

ewm
Provides exponential weighted functions.

Notes

See Windowing Operations for further usage details and examples.

Examples

```
>>> df = pd.DataFrame({"B": [0, 1, 2, np.nan, 4]})
>>> df
     B
0  0.0
1  1.0
2  2.0
3  NaN
4  4.0
```

min_periods

Expanding sum with 1 vs 3 observations needed to calculate a value.

```
>>> df.expanding(1).sum()
     B
0  0.0
1  1.0
2  3.0
3  3.0
4  7.0
>>> df.expanding(3).sum()
     B
0  NaN
1  NaN
2  3.0
3  3.0
4  7.0
```

AlloViz.AlloViz.Elements.Nodes.explode

Nodes.**explode**(*column: IndexLabel*, *ignore_index: bool = False*) → DataFrame

Transform each element of a list-like to a row, replicating index values.

Parameters

column
[IndexLabel] Column(s) to explode. For multiple columns, specify a non-empty list with each element be str or tuple, and all specified columns their list-like data on same row of the frame must have matching length.

New in version 1.3.0: Multi-column explode

ignore_index
[bool, default False] If True, the resulting index will be labeled 0, 1, …, n - 1.

Returns

DataFrame
Exploded lists to rows of the subset columns; index will be duplicated for these rows.

Raises

ValueError

- If columns of the frame are not unique.
- If specified columns to explode is empty list.
- If specified columns to explode have not matching count of elements rowwise in the frame.

See also:

`DataFrame.unstack`
Pivot a level of the (necessarily hierarchical) index labels.

`DataFrame.melt`
Unpivot a DataFrame from wide format to long format.

`Series.explode`
Explode a DataFrame from list-like columns to long format.

Notes

This routine will explode list-likes including lists, tuples, sets, Series, and np.ndarray. The result dtype of the subset rows will be object. Scalars will be returned unchanged, and empty list-likes will result in a np.nan for that row. In addition, the ordering of rows in the output will be non-deterministic when exploding sets.

Reference the user guide for more examples.

Examples

```
>>> df = pd.DataFrame({'A': [[0, 1, 2], 'foo', [], [3, 4]],
...                    'B': 1,
...                    'C': [['a', 'b', 'c'], np.nan, [], ['d', 'e']]})
>>> df
           A  B          C
0  [0, 1, 2]  1  [a, b, c]
1        foo  1        NaN
2         []  1         []
3     [3, 4]  1     [d, e]
```

Single-column explode.

```
>>> df.explode('A')
     A  B          C
0    0  1  [a, b, c]
0    1  1  [a, b, c]
0    2  1  [a, b, c]
1  foo  1        NaN
2  NaN  1         []
3    3  1     [d, e]
3    4  1     [d, e]
```

Multi-column explode.

```
>>> df.explode(list('AC'))
     A  B    C
0    0  1    a
0    1  1    b
0    2  1    c
1  foo  1  NaN
2  NaN  1  NaN
3    3  1    d
3    4  1    e
```

AlloViz.AlloViz.Elements.Nodes.ffill

Nodes.**ffill**(*, *axis: None | Axis = None*, *inplace: bool_t = False*, *limit: None | int = None*, *downcast: dict | None | lib.NoDefault = _NoDefault.no_default*) → Self | None

Fill NA/NaN values by propagating the last valid observation to next valid.

Parameters

axis
[{0 or 'index'} for Series, {0 or 'index', 1 or 'columns'} for DataFrame] Axis along which to fill missing values. For *Series* this parameter is unused and defaults to 0.

inplace
[bool, default False] If True, fill in-place. Note: this will modify any other views on this object (e.g., a no-copy slice for a column in a DataFrame).

limit
[int, default None] If method is specified, this is the maximum number of consecu-

tive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled. Must be greater than 0 if not None.

downcast
[dict, default is None] A dict of item->dtype of what to downcast if possible, or the string 'infer' which will try to downcast to an appropriate equal type (e.g. float64 to int64 if possible).

Returns

Series/DataFrame or None
Object with missing values filled or None if `inplace=True`.

Examples

```
>>> df = pd.DataFrame([[np.nan, 2, np.nan, 0],
...                    [3, 4, np.nan, 1],
...                    [np.nan, np.nan, np.nan, np.nan],
...                    [np.nan, 3, np.nan, 4]],
...                   columns=list("ABCD"))
>>> df
     A    B   C    D
0  NaN  2.0 NaN  0.0
1  3.0  4.0 NaN  1.0
2  NaN  NaN NaN  NaN
3  NaN  3.0 NaN  4.0
```

```
>>> df.ffill()
     A    B   C    D
0  NaN  2.0 NaN  0.0
1  3.0  4.0 NaN  1.0
2  3.0  4.0 NaN  1.0
3  3.0  3.0 NaN  4.0
```

```
>>> ser = pd.Series([1, np.nan, 2, 3])
>>> ser.ffill()
0   1.0
1   1.0
2   2.0
3   3.0
dtype: float64
```

AlloViz.AlloViz.Elements.Nodes.fillna

Nodes.**fillna**(*value: Hashable | Mapping | Series | DataFrame | None = None*, ***, *method: FillnaOptions | None = None*, *axis: Axis | None = None*, *inplace: bool_t = False*, *limit: int | None = None*, *downcast: dict | None | lib.NoDefault = _NoDefault.no_default*) → Self | None

Fill NA/NaN values using the specified method.

Parameters

value
[scalar, dict, Series, or DataFrame] Value to use to fill holes (e.g. 0), alternately a dict/Series/DataFrame of values specifying which value to use for each index (for a Series) or column (for a DataFrame). Values not in the dict/Series/DataFrame will not be filled. This value cannot be a list.

method
[{'backfill', 'bfill', 'ffill', None}, default None] Method to use for filling holes in reindexed Series:

- ffill: propagate last valid observation forward to next valid.
- backfill / bfill: use next valid observation to fill gap.

Deprecated since version 2.1.0: Use ffill or bfill instead.

axis
[{0 or 'index'} for Series, {0 or 'index', 1 or 'columns'} for DataFrame] Axis along which to fill missing values. For *Series* this parameter is unused and defaults to 0.

inplace
[bool, default False] If True, fill in-place. Note: this will modify any other views on this object (e.g., a no-copy slice for a column in a DataFrame).

limit
[int, default None] If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled. Must be greater than 0 if not None.

downcast
[dict, default is None] A dict of item->dtype of what to downcast if possible, or the string 'infer' which will try to downcast to an appropriate equal type (e.g. float64 to int64 if possible).

Returns

Series/DataFrame or None
Object with missing values filled or None if `inplace=True`.

See also:

ffill
Fill values by propagating the last valid observation to next valid.

bfill
Fill values by using the next valid observation to fill the gap.

interpolate
Fill NaN values using interpolation.

reindex
Conform object to new index.

asfreq
Convert TimeSeries to specified frequency.

Examples

```
>>> df = pd.DataFrame([[np.nan, 2, np.nan, 0],
...                    [3, 4, np.nan, 1],
...                    [np.nan, np.nan, np.nan, np.nan],
...                    [np.nan, 3, np.nan, 4]],
...                   columns=list("ABCD"))
>>> df
     A    B   C    D
0  NaN  2.0 NaN  0.0
1  3.0  4.0 NaN  1.0
2  NaN  NaN NaN  NaN
3  NaN  3.0 NaN  4.0
```

Replace all NaN elements with 0s.

```
>>> df.fillna(0)
     A    B    C    D
0  0.0  2.0  0.0  0.0
1  3.0  4.0  0.0  1.0
2  0.0  0.0  0.0  0.0
3  0.0  3.0  0.0  4.0
```

Replace all NaN elements in column 'A', 'B', 'C', and 'D', with 0, 1, 2, and 3 respectively.

```
>>> values = {"A": 0, "B": 1, "C": 2, "D": 3}
>>> df.fillna(value=values)
     A    B    C    D
0  0.0  2.0  2.0  0.0
1  3.0  4.0  2.0  1.0
2  0.0  1.0  2.0  3.0
3  0.0  3.0  2.0  4.0
```

Only replace the first NaN element.

```
>>> df.fillna(value=values, limit=1)
     A    B    C    D
0  0.0  2.0  2.0  0.0
1  3.0  4.0  NaN  1.0
2  NaN  1.0  NaN  3.0
3  NaN  3.0  NaN  4.0
```

When filling using a DataFrame, replacement happens along the same column names and same indices

```
>>> df2 = pd.DataFrame(np.zeros((4, 4)), columns=list("ABCE"))
>>> df.fillna(df2)
     A    B    C    D
```

(continues on next page)

(continued from previous page)

```
0  0.0  2.0  0.0  0.0
1  3.0  4.0  0.0  1.0
2  0.0  0.0  0.0  NaN
3  0.0  3.0  0.0  4.0
```

Note that column D is not affected since it is not present in df2.

AlloViz.AlloViz.Elements.Nodes.filter

Nodes.**filter**(*items=None*, *like: str | None = None*, *regex: str | None = None*, *axis: int | Literal['index', 'columns', 'rows'] | None = None*) → None

Subset the dataframe rows or columns according to the specified index labels.

Note that this routine does not filter a dataframe on its contents. The filter is applied to the labels of the index.

Parameters

items
[list-like] Keep labels from axis which are in items.

like
[str] Keep labels from axis for which "like in label == True".

regex
[str (regular expression)] Keep labels from axis for which re.search(regex, label) == True.

axis
[{0 or 'index', 1 or 'columns', None}, default None] The axis to filter on, expressed either as an index (int) or axis name (str). By default this is the info axis, 'columns' for DataFrame. For *Series* this parameter is unused and defaults to *None*.

Returns

same type as input object

See also:

`DataFrame.loc`
Access a group of rows and columns by label(s) or a boolean array.

Notes

The `items`, `like`, and `regex` parameters are enforced to be mutually exclusive.

`axis` defaults to the info axis that is used when indexing with `[]`.

Examples

```
>>> df = pd.DataFrame(np.array(([1, 2, 3], [4, 5, 6])),
...                   index=['mouse', 'rabbit'],
...                   columns=['one', 'two', 'three'])
>>> df
        one  two  three
mouse     1    2      3
rabbit    4    5      6
```

```
>>> # select columns by name
>>> df.filter(items=['one', 'three'])
         one  three
mouse     1      3
rabbit    4      6
```

```
>>> # select columns by regular expression
>>> df.filter(regex='e$', axis=1)
         one  three
mouse     1      3
rabbit    4      6
```

```
>>> # select rows containing 'bbi'
>>> df.filter(like='bbi', axis=0)
         one  two  three
rabbit    4    5      6
```

AlloViz.AlloViz.Elements.Nodes.first

`Nodes.`**`first`**(*offset*) → None

Select initial periods of time series data based on a date offset.

For a DataFrame with a sorted DatetimeIndex, this function can select the first few rows based on a date offset.

Parameters

offset
[str, DateOffset or dateutil.relativedelta] The offset length of the data that will be selected. For instance, '1M' will display all the rows having their index within the first month.

Returns

Series or DataFrame
A subset of the caller.

Raises

TypeError
If the index is not a `DatetimeIndex`

See also:

last
Select final periods of time series based on a date offset.

at_time
Select values at a particular time of the day.

between_time
Select values between particular times of the day.

Examples

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='2D')
>>> ts = pd.DataFrame({'A': [1, 2, 3, 4]}, index=i)
>>> ts
            A
2018-04-09  1
2018-04-11  2
2018-04-13  3
2018-04-15  4
```

Get the rows for the first 3 days:

```
>>> ts.first('3D')
            A
2018-04-09  1
2018-04-11  2
```

Notice the data for 3 first calendar days were returned, not the first 3 days observed in the dataset, and therefore data for 2018-04-13 was not returned.

AlloViz.AlloViz.Elements.Nodes.first_valid_index

Nodes.**first_valid_index**() → Hashable | None

Return index for first non-NA value or None, if no non-NA value is found.

Returns

type of index

Notes

If all elements are non-NA/null, returns None. Also returns None for empty Series/DataFrame.

Examples

For Series:

```
>>> s = pd.Series([None, 3, 4])
>>> s.first_valid_index()
1
>>> s.last_valid_index()
2
```

For DataFrame:

```
>>> df = pd.DataFrame({'A': [None, None, 2], 'B': [None, 3, 4]})
>>> df
     A      B
0  NaN    NaN
1  NaN    3.0
2  2.0    4.0
>>> df.first_valid_index()
1
>>> df.last_valid_index()
2
```

AlloViz.AlloViz.Elements.Nodes.floordiv

`Nodes.floordiv`(*other*, *axis: Axis = 'columns'*, *level=None*, *fill_value=None*)

Get Integer division of dataframe and other, element-wise (binary operator *floordiv*).

Equivalent to `dataframe // other`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *rfloordiv*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *floordiv*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other
: [scalar, sequence, Series, dict or DataFrame] Any single or multiple element data structure, or list-like object.

axis
: [{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns. (1 or 'columns'). For Series input, axis to match Series index on.

level
: [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value
: [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame
Result of the arithmetic operation.

See also:

DataFrame.add
Add DataFrames.

DataFrame.sub
Subtract DataFrames.

DataFrame.mul
Multiply DataFrames.

DataFrame.div
Divide DataFrames (float division).

DataFrame.truediv
Divide DataFrames (float division).

DataFrame.floordiv
Divide DataFrames (integer division).

DataFrame.mod
Calculate modulo (remainder after division).

DataFrame.pow
Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
```

(continues on next page)

(continued from previous page)

```
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a dictionary by axis.

```
>>> df.mul({'angles': 0, 'degrees': 2})
           angles  degrees
circle          0      720
triangle        0      360
rectangle       0      720
```

```
>>> df.mul({'circle': 0, 'triangle': 2, 'rectangle': 3}, axis='index')
           angles  degrees
circle          0        0
triangle        6      360
rectangle      12     1080
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

AlloViz.AlloViz.Elements.Nodes.from_dict

classmethod Nodes.**from_dict**(*data: dict*, *orient: FromDictOrient = 'columns'*, *dtype: Dtype | None = None*, *columns: Axes | None = None*) → DataFrame

Construct DataFrame from dict of array-like or dicts.

Creates DataFrame object from dictionary by columns or by index allowing dtype specification.

Parameters

data
[dict] Of the form {field : array-like} or {field : dict}.

orient
[{'columns', 'index', 'tight'}, default 'columns'] The "orientation" of the data. If the keys of the passed dict should be the columns of the resulting DataFrame, pass 'columns' (default). Otherwise if the keys should be rows, pass 'index'. If 'tight', assume a dict with keys ['index', 'columns', 'data', 'index_names', 'column_names'].

New in version 1.4.0: 'tight' as an allowed value for the `orient` argument

dtype
[dtype, default None] Data type to force after DataFrame construction, otherwise infer.

columns
[list, default None] Column labels to use when `orient='index'`. Raises a ValueError if used with `orient='columns'` or `orient='tight'`.

Returns

DataFrame

See also:

`DataFrame.from_records`
DataFrame from structured ndarray, sequence of tuples or dicts, or DataFrame.

`DataFrame`
DataFrame object creation using constructor.

`DataFrame.to_dict`
Convert the DataFrame to a dictionary.

Examples

By default the keys of the dict become the DataFrame columns:

```
>>> data = {'col_1': [3, 2, 1, 0], 'col_2': ['a', 'b', 'c', 'd']}
>>> pd.DataFrame.from_dict(data)
   col_1 col_2
0      3     a
1      2     b
2      1     c
3      0     d
```

Specify `orient='index'` to create the DataFrame using dictionary keys as rows:

```
>>> data = {'row_1': [3, 2, 1, 0], 'row_2': ['a', 'b', 'c', 'd']}
>>> pd.DataFrame.from_dict(data, orient='index')
       0  1  2  3
row_1  3  2  1  0
row_2  a  b  c  d
```

When using the 'index' orientation, the column names can be specified manually:

```
>>> pd.DataFrame.from_dict(data, orient='index',
...                        columns=['A', 'B', 'C', 'D'])
       A  B  C  D
row_1  3  2  1  0
row_2  a  b  c  d
```

Specify `orient='tight'` to create the DataFrame using a 'tight' format:

```
>>> data = {'index': [('a', 'b'), ('a', 'c')],
...         'columns': [('x', 1), ('y', 2)],
...         'data': [[1, 3], [2, 4]],
...         'index_names': ['n1', 'n2'],
...         'column_names': ['z1', 'z2']}
>>> pd.DataFrame.from_dict(data, orient='tight')
z1     x  y
z2     1  2
n1 n2
a  b   1  3
   c   2  4
```

AlloViz.AlloViz.Elements.Nodes.from_records

classmethod Nodes.**from_records**(*data*, *index=None*, *exclude=None*, *columns=None*, *coerce_float: bool = False*, *nrows: None | int = None*) → DataFrame

Convert structured or record ndarray to DataFrame.

Creates a DataFrame object from a structured ndarray, sequence of tuples or dicts, or DataFrame.

Parameters

data
: [structured ndarray, sequence of tuples or dicts, or DataFrame] Structured input data.

  Deprecated since version 2.1.0: Passing a DataFrame is deprecated.

index
: [str, list of fields, array-like] Field of array to use as the index, alternately a specific set of input labels to use.

exclude
: [sequence, default None] Columns or fields to exclude.

columns
: [sequence, default None] Column names to use. If the passed data do not have names associated with them, this argument provides names for the columns. Otherwise this argument indicates the order of the columns in the result (any names not found in the data will become all-NA columns).

coerce_float
[bool, default False] Attempt to convert values of non-string, non-numeric objects (like decimal.Decimal) to floating point, useful for SQL result sets.

nrows
[int, default None] Number of rows to read if data is an iterator.

Returns

DataFrame

See also:

`DataFrame.from_dict`
DataFrame from dict of array-like or dicts.

`DataFrame`
DataFrame object creation using constructor.

Examples

Data can be provided as a structured ndarray:

```
>>> data = np.array([(3, 'a'), (2, 'b'), (1, 'c'), (0, 'd')],
...                 dtype=[('col_1', 'i4'), ('col_2', 'U1')])
>>> pd.DataFrame.from_records(data)
   col_1 col_2
0      3     a
1      2     b
2      1     c
3      0     d
```

Data can be provided as a list of dicts:

```
>>> data = [{'col_1': 3, 'col_2': 'a'},
...         {'col_1': 2, 'col_2': 'b'},
...         {'col_1': 1, 'col_2': 'c'},
...         {'col_1': 0, 'col_2': 'd'}]
>>> pd.DataFrame.from_records(data)
   col_1 col_2
0      3     a
1      2     b
2      1     c
3      0     d
```

Data can be provided as a list of tuples with corresponding columns:

```
>>> data = [(3, 'a'), (2, 'b'), (1, 'c'), (0, 'd')]
>>> pd.DataFrame.from_records(data, columns=['col_1', 'col_2'])
   col_1 col_2
0      3     a
1      2     b
2      1     c
3      0     d
```

AlloViz.AlloViz.Elements.Nodes.ge

Nodes.**ge**(*other*, *axis: Axis = 'columns'*, *level=None*)

Get Greater than or equal to of dataframe and other, element-wise (binary operator *ge*).

Among flexible wrappers (*eq*, *ne*, *le*, *lt*, *ge*, *gt*) to comparison operators.

Equivalent to ==, *!*=, <=, <, >=, > with support to choose axis (rows or columns) and level for comparison.

Parameters

other
[scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.

axis
[{0 or 'index', 1 or 'columns'}, default 'columns'] Whether to compare by the index (0 or 'index') or columns (1 or 'columns').

level
[int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

Returns

DataFrame of bool
Result of the comparison.

See also:

`DataFrame.eq`
Compare DataFrames for equality elementwise.

`DataFrame.ne`
Compare DataFrames for inequality elementwise.

`DataFrame.le`
Compare DataFrames for less than inequality or equality elementwise.

`DataFrame.lt`
Compare DataFrames for strictly less than inequality elementwise.

`DataFrame.ge`
Compare DataFrames for greater than inequality or equality elementwise.

`DataFrame.gt`
Compare DataFrames for strictly greater than inequality elementwise.

Notes

Mismatched indices will be unioned together. *NaN* values are considered different (i.e. *NaN* != *NaN*).

Examples

```
>>> df = pd.DataFrame({'cost': [250, 150, 100],
...                    'revenue': [100, 250, 300]},
...                   index=['A', 'B', 'C'])
>>> df
   cost  revenue
A   250      100
B   150      250
C   100      300
```

Comparison with a scalar, using either the operator or method:

```
>>> df == 100
    cost  revenue
A  False     True
B  False    False
C   True    False
```

```
>>> df.eq(100)
    cost  revenue
A  False     True
B  False    False
C   True    False
```

When *other* is a `Series`, the columns of a DataFrame are aligned with the index of *other* and broadcast:

```
>>> df != pd.Series([100, 250], index=["cost", "revenue"])
    cost  revenue
A   True     True
B   True    False
C  False     True
```

Use the method to control the broadcast axis:

```
>>> df.ne(pd.Series([100, 300], index=["A", "D"]), axis='index')
   cost  revenue
A  True    False
B  True     True
C  True     True
D  True     True
```

When comparing to an arbitrary sequence, the number of columns must match the number elements in *other*:

```
>>> df == [250, 100]
    cost  revenue
A   True     True
B  False    False
C  False    False
```

Use the method to control the axis:

```
>>> df.eq([250, 250, 100], axis='index')
    cost  revenue
A   True    False
B  False     True
C   True    False
```

Compare to a DataFrame of different shape.

```
>>> other = pd.DataFrame({'revenue': [300, 250, 100, 150]},
...                      index=['A', 'B', 'C', 'D'])
>>> other
   revenue
A      300
B      250
C      100
D      150
```

```
>>> df.gt(other)
    cost  revenue
A  False    False
B  False    False
C  False     True
D  False    False
```

Compare to a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'cost': [250, 150, 100, 150, 300, 220],
...                              'revenue': [100, 250, 300, 200, 175, 225]},
...                             index=[['Q1', 'Q1', 'Q1', 'Q2', 'Q2', 'Q2'],
...                                    ['A', 'B', 'C', 'A', 'B', 'C']])
>>> df_multindex
      cost  revenue
Q1 A   250      100
   B   150      250
   C   100      300
Q2 A   150      200
   B   300      175
   C   220      225
```

```
>>> df.le(df_multindex, level=1)
       cost  revenue
Q1 A   True     True
   B   True     True
   C   True     True
Q2 A  False     True
   B   True    False
   C   True    False
```

AlloViz.AlloViz.Elements.Nodes.get

Nodes.**get**(*key*, *default=None*)

Get item from object for given key (ex: DataFrame column).

Returns default value if not found.

Parameters

key
[object]

Returns

same type as items contained in object

Examples

```
>>> df = pd.DataFrame(
...     [
...         [24.3, 75.7, "high"],
...         [31, 87.8, "high"],
...         [22, 71.6, "medium"],
...         [35, 95, "medium"],
...     ],
...     columns=["temp_celsius", "temp_fahrenheit", "windspeed"],
...     index=pd.date_range(start="2014-02-12", end="2014-02-15", freq="D"),
... )
```

```
>>> df
            temp_celsius  temp_fahrenheit windspeed
2014-02-12          24.3             75.7      high
2014-02-13          31.0             87.8      high
2014-02-14          22.0             71.6    medium
2014-02-15          35.0             95.0    medium
```

```
>>> df.get(["temp_celsius", "windspeed"])
            temp_celsius windspeed
2014-02-12          24.3      high
2014-02-13          31.0      high
2014-02-14          22.0    medium
2014-02-15          35.0    medium
```

```
>>> ser = df['windspeed']
>>> ser.get('2014-02-13')
'high'
```

If the key isn't found, the default value will be used.

```
>>> df.get(["temp_celsius", "temp_kelvin"], default="default_value")
'default_value'
```

```
>>> ser.get('2014-02-10', '[unknown]')
'[unknown]'
```

AlloViz.AlloViz.Elements.Nodes.groupby

Nodes.**groupby**(*by=None*, *axis: Axis | lib.NoDefault = _NoDefault.no_default*, *level: IndexLabel | None = None*, *as_index: bool = True*, *sort: bool = True*, *group_keys: bool = True*, *observed: bool | lib.NoDefault = _NoDefault.no_default*, *dropna: bool = True*) → DataFrameGroupBy

Group DataFrame using a mapper or by a Series of columns.

A groupby operation involves some combination of splitting the object, applying a function, and combining the results. This can be used to group large amounts of data and compute operations on these groups.

Parameters

by
[mapping, function, label, pd.Grouper or list of such] Used to determine the groups for the groupby. If `by` is a function, it's called on each value of the object's index. If a dict or Series is passed, the Series or dict VALUES will be used to determine the groups (the Series' values are first aligned; see `.align()` method). If a list or ndarray of length equal to the selected axis is passed (see the groupby user guide), the values are used as-is to determine the groups. A label or list of labels may be passed to group by the columns in `self`. Notice that a tuple is interpreted as a (single) key.

axis
[{0 or 'index', 1 or 'columns'}, default 0] Split along rows (0) or columns (1). For *Series* this parameter is unused and defaults to 0.

level
[int, level name, or sequence of such, default None] If the axis is a MultiIndex (hierarchical), group by a particular level or levels. Do not specify both `by` and `level`.

as_index
[bool, default True] Return object with group labels as the index. Only relevant for DataFrame input. as_index=False is effectively "SQL-style" grouped output. This argument has no effect on filtrations (see the filtrations in the user guide), such as `head()`, `tail()`, `nth()` and in transformations (see the transformations in the user guide).

sort
[bool, default True] Sort group keys. Get better performance by turning this off. Note this does not influence the order of observations within each group. Groupby preserves the order of rows within each group. If False, the groups will appear in the same order as they did in the original DataFrame. This argument has no effect on filtrations (see the filtrations in the user guide), such as `head()`, `tail()`, `nth()` and in transformations (see the transformations in the user guide).

Changed in version 2.0.0: Specifying `sort=False` with an ordered categorical grouper will no longer sort the values.

group_keys
[bool, default True] When calling apply and the `by` argument produces a like-indexed (i.e. a transform) result, add group keys to index to identify pieces. By default group keys are not included when the result's index (and column) labels match the inputs, and are included otherwise.

Changed in version 1.5.0: Warns that `group_keys` will no longer be ignored when the result from `apply` is a like-indexed Series or DataFrame. Specify `group_keys` explicitly to include the group keys or not.

Changed in version 2.0.0: `group_keys` now defaults to `True`.

observed
[bool, default False] This only applies if any of the groupers are Categoricals. If True: only show observed values for categorical groupers. If False: show all values for categorical groupers.

Deprecated since version 2.1.0: The default value will change to True in a future version of pandas.

dropna
[bool, default True] If True, and if group keys contain NA values, NA values together with row/column will be dropped. If False, NA values will also be treated as the key in groups.

Returns

pandas.api.typing.DataFrameGroupBy
Returns a groupby object that contains information about the groups.

See also:

resample
Convenience method for frequency conversion and resampling of time series.

Notes

See the user guide for more detailed usage and examples, including splitting an object into groups, iterating through groups, selecting a group, aggregation, and more.

Examples

```
>>> df = pd.DataFrame({'Animal': ['Falcon', 'Falcon',
...                               'Parrot', 'Parrot'],
...                    'Max Speed': [380., 370., 24., 26.]})
>>> df
   Animal  Max Speed
0  Falcon      380.0
1  Falcon      370.0
2  Parrot       24.0
3  Parrot       26.0
>>> df.groupby(['Animal']).mean()
        Max Speed
Animal
Falcon      375.0
Parrot       25.0
```

Hierarchical Indexes

We can groupby different levels of a hierarchical index using the *level* parameter:

```
>>> arrays = [['Falcon', 'Falcon', 'Parrot', 'Parrot'],
...           ['Captive', 'Wild', 'Captive', 'Wild']]
>>> index = pd.MultiIndex.from_arrays(arrays, names=('Animal', 'Type'))
>>> df = pd.DataFrame({'Max Speed': [390., 350., 30., 20.]},
```

(continues on next page)

(continued from previous page)

```
...                     index=index)
>>> df
                Max Speed
Animal Type
Falcon Captive      390.0
       Wild         350.0
Parrot Captive       30.0
       Wild          20.0
>>> df.groupby(level=0).mean()
        Max Speed
Animal
Falcon      370.0
Parrot       25.0
>>> df.groupby(level="Type").mean()
         Max Speed
Type
Captive      210.0
Wild         185.0
```

We can also choose to include NA in group keys or not by setting *dropna* parameter, the default setting is *True*.

```
>>> l = [[1, 2, 3], [1, None, 4], [2, 1, 3], [1, 2, 2]]
>>> df = pd.DataFrame(l, columns=["a", "b", "c"])
```

```
>>> df.groupby(by=["b"]).sum()
    a   c
b
1.0 2   3
2.0 2   5
```

```
>>> df.groupby(by=["b"], dropna=False).sum()
    a   c
b
1.0 2   3
2.0 2   5
NaN 1   4
```

```
>>> l = [["a", 12, 12], [None, 12.3, 33.], ["b", 12.3, 123], ["a", 1, 1]]
>>> df = pd.DataFrame(l, columns=["a", "b", "c"])
```

```
>>> df.groupby(by="a").sum()
    b     c
a
a   13.0   13.0
b   12.3  123.0
```

```
>>> df.groupby(by="a", dropna=False).sum()
    b     c
a
a   13.0   13.0
```

(continues on next page)

(continued from previous page)

```
b   12.3  123.0
NaN 12.3   33.0
```

When using `.apply()`, use `group_keys` to include or exclude the group keys. The `group_keys` argument defaults to `True` (include).

```
>>> df = pd.DataFrame({'Animal': ['Falcon', 'Falcon',
...                               'Parrot', 'Parrot'],
...                    'Max Speed': [380., 370., 24., 26.]})
>>> df.groupby("Animal", group_keys=True).apply(lambda x: x)
          Animal  Max Speed
Animal
Falcon 0  Falcon      380.0
       1  Falcon      370.0
Parrot 2  Parrot       24.0
       3  Parrot       26.0
```

```
>>> df.groupby("Animal", group_keys=False).apply(lambda x: x)
   Animal  Max Speed
0  Falcon      380.0
1  Falcon      370.0
2  Parrot       24.0
3  Parrot       26.0
```

AlloViz.AlloViz.Elements.Nodes.gt

`Nodes.gt`(*other*, *axis: Axis = 'columns'*, *level=None*)

Get Greater than of dataframe and other, element-wise (binary operator *gt*).

Among flexible wrappers (*eq*, *ne*, *le*, *lt*, *ge*, *gt*) to comparison operators.

Equivalent to ==, *!*=, <=, <, >=, > with support to choose axis (rows or columns) and level for comparison.

Parameters

other
: [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.

axis
: [{0 or 'index', 1 or 'columns'}, default 'columns'] Whether to compare by the index (0 or 'index') or columns (1 or 'columns').

level
: [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

Returns

DataFrame of bool
: Result of the comparison.

See also:

DataFrame.eq
Compare DataFrames for equality elementwise.

DataFrame.ne
Compare DataFrames for inequality elementwise.

DataFrame.le
Compare DataFrames for less than inequality or equality elementwise.

DataFrame.lt
Compare DataFrames for strictly less than inequality elementwise.

DataFrame.ge
Compare DataFrames for greater than inequality or equality elementwise.

DataFrame.gt
Compare DataFrames for strictly greater than inequality elementwise.

Notes

Mismatched indices will be unioned together. *NaN* values are considered different (i.e. *NaN* != *NaN*).

Examples

```
>>> df = pd.DataFrame({'cost': [250, 150, 100],
...                    'revenue': [100, 250, 300]},
...                   index=['A', 'B', 'C'])
>>> df
   cost  revenue
A   250      100
B   150      250
C   100      300
```

Comparison with a scalar, using either the operator or method:

```
>>> df == 100
    cost  revenue
A  False     True
B  False    False
C   True    False
```

```
>>> df.eq(100)
    cost  revenue
A  False     True
B  False    False
C   True    False
```

When *other* is a `Series`, the columns of a DataFrame are aligned with the index of *other* and broadcast:

```
>>> df != pd.Series([100, 250], index=["cost", "revenue"])
    cost  revenue
A   True     True
B   True    False
C  False     True
```

Use the method to control the broadcast axis:

```
>>> df.ne(pd.Series([100, 300], index=["A", "D"]), axis='index')
   cost  revenue
A  True    False
B  True     True
C  True     True
D  True     True
```

When comparing to an arbitrary sequence, the number of columns must match the number elements in *other*:

```
>>> df == [250, 100]
    cost  revenue
A   True     True
B  False    False
C  False    False
```

Use the method to control the axis:

```
>>> df.eq([250, 250, 100], axis='index')
    cost  revenue
A   True    False
B  False     True
C   True    False
```

Compare to a DataFrame of different shape.

```
>>> other = pd.DataFrame({'revenue': [300, 250, 100, 150]},
...                      index=['A', 'B', 'C', 'D'])
>>> other
   revenue
A      300
B      250
C      100
D      150
```

```
>>> df.gt(other)
    cost  revenue
A  False    False
B  False    False
C  False     True
D  False    False
```

Compare to a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'cost': [250, 150, 100, 150, 300, 220],
...                              'revenue': [100, 250, 300, 200, 175, 225]},
...                             index=[['Q1', 'Q1', 'Q1', 'Q2', 'Q2', 'Q2'],
...                                    ['A', 'B', 'C', 'A', 'B', 'C']])
>>> df_multindex
      cost  revenue
Q1 A   250      100
   B   150      250
```

(continues on next page)

(continued from previous page)

```
   C   100      300
Q2 A   150      200
   B   300      175
   C   220      225
```

```
>>> df.le(df_multindex, level=1)
       cost  revenue
Q1 A   True     True
   B   True     True
   C   True     True
Q2 A  False     True
   B   True    False
   C   True    False
```

AlloViz.AlloViz.Elements.Nodes.head

Nodes.**head**(*n: int = 5*) → None

Return the first *n* rows.

This function returns the first *n* rows for the object based on position. It is useful for quickly testing if your object has the right type of data in it.

For negative values of *n*, this function returns all rows except the last |*n*| rows, equivalent to `df[:n]`.

If n is larger than the number of rows, this function returns all rows.

Parameters

n
[int, default 5] Number of rows to select.

Returns

same type as caller
The first *n* rows of the caller object.

See also:

DataFrame.tail
Returns the last *n* rows.

Examples

```
>>> df = pd.DataFrame({'animal': ['alligator', 'bee', 'falcon', 'lion',
...                    'monkey', 'parrot', 'shark', 'whale', 'zebra']})
>>> df
      animal
0  alligator
1        bee
2     falcon
3       lion
4     monkey
5     parrot
```

(continues on next page)

(continued from previous page)

```
6      shark
7      whale
8      zebra
```

Viewing the first 5 lines

```
>>> df.head()
      animal
0  alligator
1        bee
2     falcon
3       lion
4     monkey
```

Viewing the first *n* lines (three in this case)

```
>>> df.head(3)
      animal
0  alligator
1        bee
2     falcon
```

For negative values of *n*

```
>>> df.head(-3)
      animal
0  alligator
1        bee
2     falcon
3       lion
4     monkey
5     parrot
```

AlloViz.AlloViz.Elements.Nodes.hist

`Nodes.hist`(*column: IndexLabel | None = None*, *by=None*, *grid: bool = True*, *xlabelsize: int | None = None*, *xrot: float | None = None*, *ylabelsize: int | None = None*, *yrot: float | None = None*, *ax=None*, *sharex: bool = False*, *sharey: bool = False*, *figsize: tuple[int, int] | None = None*, *layout: tuple[int, int] | None = None*, *bins: int | Sequence[int] = 10*, *backend: str | None = None*, *legend: bool = False*, ***kwargs*)

Make a histogram of the DataFrame's columns.

A histogram is a representation of the distribution of data. This function calls `matplotlib.pyplot.hist()`, on each series in the DataFrame, resulting in one histogram per column.

Parameters

data
[DataFrame] The pandas object holding the data.

column
[str or sequence, optional] If passed, will be used to limit data to a subset of columns.

by
[object, optional] If passed, then used to form histograms for separate groups.

grid
[bool, default True] Whether to show axis grid lines.

xlabelsize
[int, default None] If specified changes the x-axis label size.

xrot
[float, default None] Rotation of x axis labels. For example, a value of 90 displays the x labels rotated 90 degrees clockwise.

ylabelsize
[int, default None] If specified changes the y-axis label size.

yrot
[float, default None] Rotation of y axis labels. For example, a value of 90 displays the y labels rotated 90 degrees clockwise.

ax
[Matplotlib axes object, default None] The axes to plot the histogram on.

sharex
[bool, default True if ax is None else False] In case subplots=True, share x axis and set some x axis labels to invisible; defaults to True if ax is None otherwise False if an ax is passed in. Note that passing in both an ax and sharex=True will alter all x axis labels for all subplots in a figure.

sharey
[bool, default False] In case subplots=True, share y axis and set some y axis labels to invisible.

figsize
[tuple, optional] The size in inches of the figure to create. Uses the value in *matplotlib.rcParams* by default.

layout
[tuple, optional] Tuple of (rows, columns) for the layout of the histograms.

bins
[int or sequence, default 10] Number of histogram bins to be used. If an integer is given, bins + 1 bin edges are calculated and returned. If bins is a sequence, gives bin edges, including left edge of first bin and right edge of last bin. In this case, bins is returned unmodified.

backend
[str, default None] Backend to use instead of the backend specified in the option `plotting.backend`. For instance, 'matplotlib'. Alternatively, to specify the `plotting.backend` for the whole session, set `pd.options.plotting.backend`.

legend
[bool, default False] Whether to show the legend.

****kwargs**
All other plotting keyword arguments to be passed to `matplotlib.pyplot.hist()`.

Returns

matplotlib.AxesSubplot or numpy.ndarray of them

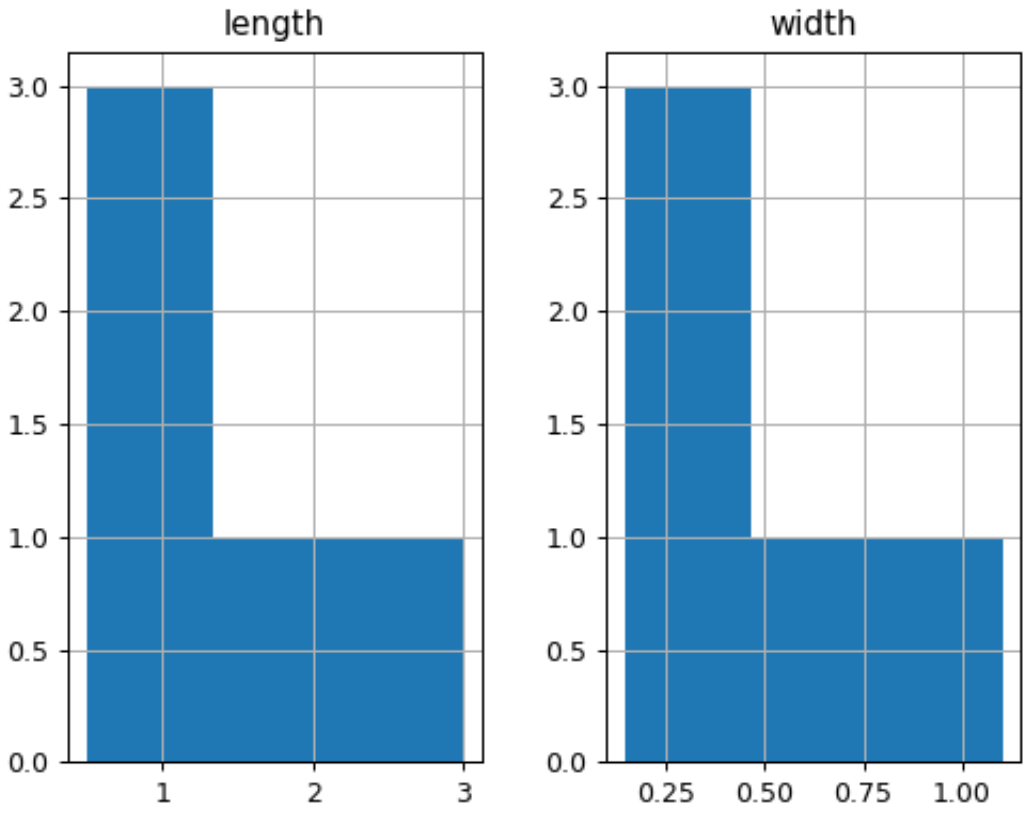
See also:

`matplotlib.pyplot.hist`
Plot a histogram using matplotlib.

Examples

This example draws a histogram based on the length and width of some animals, displayed in three bins

```
>>> df = pd.DataFrame({
...     'length': [1.5, 0.5, 1.2, 0.9, 3],
...     'width': [0.7, 0.2, 0.15, 0.2, 1.1]
...     }, index=['pig', 'rabbit', 'duck', 'chicken', 'horse'])
>>> hist = df.hist(bins=3)
```



AlloViz.AlloViz.Elements.Nodes.idxmax

Nodes.**idxmax**(*axis: Axis = 0*, *skipna: bool = True*, *numeric_only: bool = False*) → Series

Return index of first occurrence of maximum over requested axis.

NA/null values are excluded.

Parameters

axis
[{0 or 'index', 1 or 'columns'}, default 0] The axis to use. 0 or 'index' for row-wise, 1 or 'columns' for column-wise.

skipna
[bool, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

numeric_only
[bool, default False] Include only *float*, *int* or *boolean* data.

New in version 1.5.0.

Returns

Series
Indexes of maxima along the specified axis.

Raises

ValueError

- If the row/column is empty

See also:

Series.idxmax
Return index of the maximum element.

Notes

This method is the DataFrame version of `ndarray.argmax`.

Examples

Consider a dataset containing food consumption in Argentina.

```
>>> df = pd.DataFrame({'consumption': [10.51, 103.11, 55.48],
...                    'co2_emissions': [37.2, 19.66, 1712]},
...                   index=['Pork', 'Wheat Products', 'Beef'])
```

```
>>> df
                consumption  co2_emissions
Pork                  10.51         37.20
Wheat Products       103.11         19.66
Beef                  55.48       1712.00
```

By default, it returns the index for the maximum value in each column.

```
>>> df.idxmax()
consumption     Wheat Products
co2_emissions             Beef
dtype: object
```

To return the index for the maximum value in each row, use `axis="columns"`.

```
>>> df.idxmax(axis="columns")
Pork              co2_emissions
Wheat Products     consumption
Beef              co2_emissions
dtype: object
```

AlloViz.AlloViz.Elements.Nodes.idxmin

Nodes.**idxmin**(*axis: Axis = 0*, *skipna: bool = True*, *numeric_only: bool = False*) → Series

Return index of first occurrence of minimum over requested axis.

NA/null values are excluded.

Parameters

axis
[{0 or 'index', 1 or 'columns'}, default 0] The axis to use. 0 or 'index' for row-wise, 1 or 'columns' for column-wise.

skipna
[bool, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

numeric_only
[bool, default False] Include only *float*, *int* or *boolean* data.

New in version 1.5.0.

Returns

Series
Indexes of minima along the specified axis.

Raises

ValueError

- If the row/column is empty

See also:

Series.idxmin
Return index of the minimum element.

Notes

This method is the DataFrame version of `ndarray.argmin`.

Examples

Consider a dataset containing food consumption in Argentina.

```
>>> df = pd.DataFrame({'consumption': [10.51, 103.11, 55.48],
...                    'co2_emissions': [37.2, 19.66, 1712]},
...                   index=['Pork', 'Wheat Products', 'Beef'])
```

```
>>> df
                consumption  co2_emissions
Pork                  10.51         37.20
Wheat Products       103.11         19.66
Beef                  55.48       1712.00
```

By default, it returns the index for the minimum value in each column.

```
>>> df.idxmin()
consumption                Pork
co2_emissions    Wheat Products
dtype: object
```

To return the index for the minimum value in each row, use `axis="columns"`.

```
>>> df.idxmin(axis="columns")
Pork                consumption
Wheat Products    co2_emissions
Beef                consumption
dtype: object
```

AlloViz.AlloViz.Elements.Nodes.infer_objects

`Nodes.`**`infer_objects`**(*copy: bool | None = None*) → None

Attempt to infer better dtypes for object columns.

Attempts soft conversion of object-dtyped columns, leaving non-object and unconvertible columns unchanged. The inference rules are the same as during normal Series/DataFrame construction.

Parameters

copy
[bool, default True] Whether to make a copy for non-object or non-inferable columns or Series.

Returns

same type as input object

See also:

`to_datetime`
Convert argument to datetime.

to_timedelta
Convert argument to timedelta.

to_numeric
Convert argument to numeric type.

convert_dtypes
Convert argument to best possible dtype.

Examples

```
>>> df = pd.DataFrame({"A": ["a", 1, 2, 3]})
>>> df = df.iloc[1:]
>>> df
   A
1  1
2  2
3  3
```

```
>>> df.dtypes
A    object
dtype: object
```

```
>>> df.infer_objects().dtypes
A    int64
dtype: object
```

AlloViz.AlloViz.Elements.Nodes.info

Nodes.**info**(*verbose: bool | None = None*, *buf: WriteBuffer[str] | None = None*, *max_cols: int | None = None*, *memory_usage: bool | str | None = None*, *show_counts: bool | None = None*) → None

Print a concise summary of a DataFrame.

This method prints information about a DataFrame including the index dtype and columns, non-null values and memory usage.

Parameters

verbose
[bool, optional] Whether to print the full summary. By default, the setting in `pandas.options.display.max_info_columns` is followed.

buf
[writable buffer, defaults to sys.stdout] Where to send the output. By default, the output is printed to sys.stdout. Pass a writable buffer if you need to further process the output.

max_cols
[int, optional] When to switch from the verbose to the truncated output. If the DataFrame has more than *max_cols* columns, the truncated output is used. By default, the setting in `pandas.options.display.max_info_columns` is used.

memory_usage
[bool, str, optional] Specifies whether total memory usage of the DataFrame el-

ements (including the index) should be displayed. By default, this follows the `pandas.options.display.memory_usage` setting.

True always show memory usage. False never shows memory usage. A value of 'deep' is equivalent to "True with deep introspection". Memory usage is shown in human-readable units (base-2 representation). Without deep introspection a memory estimation is made based in column dtype and number of rows assuming values consume the same memory amount for corresponding dtypes. With deep memory introspection, a real memory usage calculation is performed at the cost of computational resources. See the Frequently Asked Questions for more details.

show_counts
[bool, optional] Whether to show the non-null counts. By default, this is shown only if the DataFrame is smaller than `pandas.options.display.max_info_rows` and `pandas.options.display.max_info_columns`. A value of True always shows the counts, and False never shows the counts.

Returns

None
This method prints a summary of a DataFrame and returns None.

See also:

`DataFrame.describe`
Generate descriptive statistics of DataFrame columns.

`DataFrame.memory_usage`
Memory usage of DataFrame columns.

Examples

```
>>> int_values = [1, 2, 3, 4, 5]
>>> text_values = ['alpha', 'beta', 'gamma', 'delta', 'epsilon']
>>> float_values = [0.0, 0.25, 0.5, 0.75, 1.0]
>>> df = pd.DataFrame({"int_col": int_values, "text_col": text_values,
...                    "float_col": float_values})
>>> df
    int_col text_col  float_col
0        1    alpha       0.00
1        2     beta       0.25
2        3    gamma       0.50
3        4    delta       0.75
4        5  epsilon       1.00
```

Prints information of all columns:

```
>>> df.info(verbose=True)
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 3 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   int_col    5 non-null      int64
 1   text_col   5 non-null      object
```

(continues on next page)

(continued from previous page)

```
 2   float_col  5 non-null      float64
dtypes: float64(1), int64(1), object(1)
memory usage: 248.0+ bytes
```

Prints a summary of columns count and its dtypes but not per column information:

```
>>> df.info(verbose=False)
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Columns: 3 entries, int_col to float_col
dtypes: float64(1), int64(1), object(1)
memory usage: 248.0+ bytes
```

Pipe output of DataFrame.info to buffer instead of sys.stdout, get buffer content and writes to a text file:

```
>>> import io
>>> buffer = io.StringIO()
>>> df.info(buf=buffer)
>>> s = buffer.getvalue()
>>> with open("df_info.txt", "w",
...           encoding="utf-8") as f:
...     f.write(s)
260
```

The *memory_usage* parameter allows deep introspection mode, specially useful for big DataFrames and fine-tune memory optimization:

```
>>> random_strings_array = np.random.choice(['a', 'b', 'c'], 10 ** 6)
>>> df = pd.DataFrame({
...     'column_1': np.random.choice(['a', 'b', 'c'], 10 ** 6),
...     'column_2': np.random.choice(['a', 'b', 'c'], 10 ** 6),
...     'column_3': np.random.choice(['a', 'b', 'c'], 10 ** 6)
... })
>>> df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000000 entries, 0 to 999999
Data columns (total 3 columns):
 #   Column    Non-Null Count    Dtype
---  ------    --------------    -----
 0   column_1  1000000 non-null  object
 1   column_2  1000000 non-null  object
 2   column_3  1000000 non-null  object
dtypes: object(3)
memory usage: 22.9+ MB
```

```
>>> df.info(memory_usage='deep')
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000000 entries, 0 to 999999
Data columns (total 3 columns):
 #   Column    Non-Null Count    Dtype
---  ------    --------------    -----
 0   column_1  1000000 non-null  object
```

(continues on next page)

(continued from previous page)

```
 1   column_2  1000000 non-null  object
 2   column_3  1000000 non-null  object
dtypes: object(3)
memory usage: 165.9 MB
```

AlloViz.AlloViz.Elements.Nodes.insert

Nodes.**insert**(*loc: int*, *column: Hashable*, *value: Scalar | AnyArrayLike*, *allow_duplicates: bool | lib.NoDefault = _NoDefault.no_default*) → None

Insert column into DataFrame at specified location.

Raises a ValueError if *column* is already contained in the DataFrame, unless *allow_duplicates* is set to True.

Parameters

loc
[int] Insertion index. Must verify 0 <= loc <= len(columns).

column
[str, number, or hashable object] Label of the inserted column.

value
[Scalar, Series, or array-like]

allow_duplicates
[bool, optional, default lib.no_default]

See also:

Index.insert
Insert new item by index.

Examples

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df
   col1  col2
0     1     3
1     2     4
>>> df.insert(1, "newcol", [99, 99])
>>> df
   col1  newcol  col2
0     1      99     3
1     2      99     4
>>> df.insert(0, "col1", [100, 100], allow_duplicates=True)
>>> df
   col1  col1  newcol  col2
0   100     1      99     3
1   100     2      99     4
```

Notice that pandas uses index alignment in case of *value* from type *Series*:

```
>>> df.insert(0, "col0", pd.Series([5, 6], index=[1, 2]))
>>> df
   col0  col1  col1  newcol  col2
0   NaN   100     1      99     3
1   5.0   100     2      99     4
```

AlloViz.AlloViz.Elements.Nodes.interpolate

Nodes.**interpolate**(*method: InterpolateOptions = 'linear'*, ***, *axis: Axis = 0*, *limit: int | None = None*, *inplace: bool_t = False*, *limit_direction: Literal['forward', 'backward', 'both'] | None = None*, *limit_area: Literal['inside', 'outside'] | None = None*, *downcast: Literal['infer'] | None | lib.NoDefault = _NoDefault.no_default*, ***kwargs*) → Self | None

Fill NaN values using an interpolation method.

Please note that only `method='linear'` is supported for DataFrame/Series with a MultiIndex.

Parameters

method
[str, default 'linear'] Interpolation technique to use. One of:

- 'linear': Ignore the index and treat the values as equally spaced. This is the only method supported on MultiIndexes.
- 'time': Works on daily and higher resolution data to interpolate given length of interval.
- 'index', 'values': use the actual numerical values of the index.
- 'pad': Fill in NaNs using existing values.
- 'nearest', 'zero', 'slinear', 'quadratic', 'cubic', 'barycentric', 'polynomial': Passed to *scipy.interpolate.interp1d*, whereas 'spline' is passed to *scipy.interpolate.UnivariateSpline*. These methods use the numerical values of the index. Both 'polynomial' and 'spline' require that you also specify an *order* (int), e.g. `df.interpolate(method='polynomial', order=5)`. Note that, *slinear* method in Pandas refers to the Scipy first order *spline* instead of Pandas first order *spline*.
- 'krogh', 'piecewise_polynomial', 'spline', 'pchip', 'akima', 'cubicspline': Wrappers around the SciPy interpolation methods of similar names. See *Notes*.
- 'from_derivatives': Refers to *scipy.interpolate.BPoly.from_derivatives*.

axis
[{{0 or 'index', 1 or 'columns', None}}, default None] Axis to interpolate along. For *Series* this parameter is unused and defaults to 0.

limit
[int, optional] Maximum number of consecutive NaNs to fill. Must be greater than 0.

inplace
[bool, default False] Update the data in place if possible.

limit_direction
[{{'forward', 'backward', 'both'}}, Optional] Consecutive NaNs will be filled in this direction.

If limit is specified:

- If 'method' is 'pad' or 'ffill', 'limit_direction' must be 'forward'.
- If 'method' is 'backfill' or 'bfill', 'limit_direction' must be 'backwards'.

If 'limit' is not specified:

- If 'method' is 'backfill' or 'bfill', the default is 'backward'
- else the default is 'forward'

raises ValueError if *limit_direction* is 'forward' or 'both' and
method is 'backfill' or 'bfill'.

raises ValueError if *limit_direction* is 'backward' or 'both' and
method is 'pad' or 'ffill'.

limit_area
[{{*None*, 'inside', 'outside'}}, default None] If limit is specified, consecutive NaNs will be filled with this restriction.

- `None`: No fill restriction.
- 'inside': Only fill NaNs surrounded by valid values (interpolate).
- 'outside': Only fill NaNs outside valid values (extrapolate).

downcast
[optional, 'infer' or None, defaults to None] Downcast dtypes if possible.

Deprecated since version 2.1.0.

``**kwargs``
[optional] Keyword arguments to pass on to the interpolating function.

Returns

Series or DataFrame or None
Returns the same object type as the caller, interpolated at some or all `NaN` values or None if `inplace=True`.

See also:

`fillna`
Fill missing values using different methods.

`scipy.interpolate.Akima1DInterpolator`
Piecewise cubic polynomials (Akima interpolator).

`scipy.interpolate.BPoly.from_derivatives`
Piecewise polynomial in the Bernstein basis.

`scipy.interpolate.interp1d`
Interpolate a 1-D function.

`scipy.interpolate.KroghInterpolator`
Interpolate polynomial (Krogh interpolator).

`scipy.interpolate.PchipInterpolator`
PCHIP 1-d monotonic cubic interpolation.

`scipy.interpolate.CubicSpline`
Cubic spline data interpolator.

Notes

The 'krogh', 'piecewise_polynomial', 'spline', 'pchip' and 'akima' methods are wrappers around the respective SciPy implementations of similar names. These use the actual numerical values of the index. For more information on their behavior, see the SciPy documentation.

Examples

Filling in `NaN` in a `Series` via linear interpolation.

```
>>> s = pd.Series([0, 1, np.nan, 3])
>>> s
0    0.0
1    1.0
2    NaN
3    3.0
dtype: float64
>>> s.interpolate()
0    0.0
1    1.0
2    2.0
3    3.0
dtype: float64
```

Filling in `NaN` in a Series via polynomial interpolation or splines: Both 'polynomial' and 'spline' methods require that you also specify an `order` (int).

```
>>> s = pd.Series([0, 2, np.nan, 8])
>>> s.interpolate(method='polynomial', order=2)
0    0.000000
1    2.000000
2    4.666667
3    8.000000
dtype: float64
```

Fill the DataFrame forward (that is, going down) along each column using linear interpolation.

Note how the last entry in column 'a' is interpolated differently, because there is no entry after it to use for interpolation. Note how the first entry in column 'b' remains `NaN`, because there is no entry before it to use for interpolation.

```
>>> df = pd.DataFrame([(0.0, np.nan, -1.0, 1.0),
...                    (np.nan, 2.0, np.nan, np.nan),
...                    (2.0, 3.0, np.nan, 9.0),
...                    (np.nan, 4.0, -4.0, 16.0)],
...                   columns=list('abcd'))
>>> df
     a    b    c     d
0  0.0  NaN -1.0   1.0
1  NaN  2.0  NaN   NaN
2  2.0  3.0  NaN   9.0
3  NaN  4.0 -4.0  16.0
>>> df.interpolate(method='linear', limit_direction='forward', axis=0)
     a    b    c     d
```

(continues on next page)

(continued from previous page)

```
0  0.0  NaN -1.0   1.0
1  1.0  2.0 -2.0   5.0
2  2.0  3.0 -3.0   9.0
3  2.0  4.0 -4.0  16.0
```

Using polynomial interpolation.

```
>>> df['d'].interpolate(method='polynomial', order=2)
0     1.0
1     4.0
2     9.0
3    16.0
Name: d, dtype: float64
```

AlloViz.AlloViz.Elements.Nodes.isetitem

Nodes.**isetitem**(*loc*, *value*) → None

Set the given value in the column with position *loc*.

This is a positional analogue to `__setitem__`.

Parameters

loc
[int or sequence of ints] Index position for the column.

value
[scalar or arraylike] Value(s) for the column.

Notes

`frame.isetitem(loc, value)` is an in-place method as it will modify the DataFrame in place (not returning a new object). In contrast to `frame.iloc[:, i] = value` which will try to update the existing values in place, `frame.isetitem(loc, value)` will not update the values of the column itself in place, it will instead insert a new array.

In cases where `frame.columns` is unique, this is equivalent to `frame[frame.columns[i]] = value`.

AlloViz.AlloViz.Elements.Nodes.isin

Nodes.**isin**(*values: Series | DataFrame | Sequence | Mapping*) → DataFrame

Whether each element in the DataFrame is contained in values.

Parameters

values
[iterable, Series, DataFrame or dict] The result will only be true at a location if all the labels match. If *values* is a Series, that's the index. If *values* is a dict, the keys must be the column names, which must match. If *values* is a DataFrame, then both the index and column labels must match.

Returns

DataFrame
DataFrame of booleans showing whether each element in the DataFrame is contained in values.

See also:

`DataFrame.eq`
Equality test for DataFrame.

`Series.isin`
Equivalent method on Series.

`Series.str.contains`
Test if pattern or regex is contained within a string of a Series or Index.

Examples

```
>>> df = pd.DataFrame({'num_legs': [2, 4], 'num_wings': [2, 0]},
...                   index=['falcon', 'dog'])
>>> df
        num_legs  num_wings
falcon         2          2
dog            4          0
```

When `values` is a list check whether every value in the DataFrame is present in the list (which animals have 0 or 2 legs or wings)

```
>>> df.isin([0, 2])
        num_legs  num_wings
falcon      True       True
dog        False       True
```

To check if `values` is *not* in the DataFrame, use the ~ operator:

```
>>> ~df.isin([0, 2])
        num_legs  num_wings
falcon     False      False
dog         True      False
```

When `values` is a dict, we can pass values to check for each column separately:

```
>>> df.isin({'num_wings': [0, 3]})
        num_legs  num_wings
falcon     False      False
dog        False       True
```

When `values` is a Series or DataFrame the index and column must match. Note that 'falcon' does not match based on the number of legs in other.

```
>>> other = pd.DataFrame({'num_legs': [8, 3], 'num_wings': [0, 2]},
...                      index=['spider', 'falcon'])
>>> df.isin(other)
        num_legs  num_wings
falcon     False       True
dog        False      False
```

AlloViz.AlloViz.Elements.Nodes.isna

`Nodes.isna()` → DataFrame

Detect missing values.

Return a boolean same-sized object indicating if the values are NA. NA values, such as None or `numpy.NaN`, gets mapped to True values. Everything else gets mapped to False values. Characters such as empty strings `''` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`).

Returns

DataFrame
Mask of bool values for each element in DataFrame that indicates whether an element is an NA value.

See also:

`DataFrame.isnull`
Alias of isna.

`DataFrame.notna`
Boolean inverse of isna.

`DataFrame.dropna`
Omit axes labels with missing values.

`isna`
Top-level isna.

Examples

Show which entries in a DataFrame are NA.

```
>>> df = pd.DataFrame(dict(age=[5, 6, np.nan],
...                        born=[pd.NaT, pd.Timestamp('1939-05-27'),
...                              pd.Timestamp('1940-04-25')],
...                        name=['Alfred', 'Batman', ''],
...                        toy=[None, 'Batmobile', 'Joker']))
>>> df
   age       born    name        toy
0  5.0        NaT  Alfred       None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25              Joker
```

```
>>> df.isna()
     age   born   name    toy
0  False   True  False   True
1  False  False  False  False
2   True  False  False  False
```

Show which entries in a Series are NA.

```
>>> ser = pd.Series([5, 6, np.nan])
>>> ser
```

(continues on next page)

(continued from previous page)

```
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.isna()
0    False
1    False
2     True
dtype: bool
```

AlloViz.AlloViz.Elements.Nodes.isnull

Nodes.**isnull**() → DataFrame

DataFrame.isnull is an alias for DataFrame.isna.

Detect missing values.

Return a boolean same-sized object indicating if the values are NA. NA values, such as None or `numpy.NaN`, gets mapped to True values. Everything else gets mapped to False values. Characters such as empty strings `''` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`).

Returns

DataFrame
Mask of bool values for each element in DataFrame that indicates whether an element is an NA value.

See also:

DataFrame.isnull
Alias of isna.

DataFrame.notna
Boolean inverse of isna.

DataFrame.dropna
Omit axes labels with missing values.

isna
Top-level isna.

Examples

Show which entries in a DataFrame are NA.

```
>>> df = pd.DataFrame(dict(age=[5, 6, np.nan],
...                        born=[pd.NaT, pd.Timestamp('1939-05-27'),
...                              pd.Timestamp('1940-04-25')],
...                        name=['Alfred', 'Batman', ''],
...                        toy=[None, 'Batmobile', 'Joker']))
>>> df
```

(continues on next page)

(continued from previous page)

```
   age       born    name        toy
0  5.0        NaT  Alfred       None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25              Joker
```

```
>>> df.isna()
     age   born   name    toy
0  False   True  False   True
1  False  False  False  False
2   True  False  False  False
```

Show which entries in a Series are NA.

```
>>> ser = pd.Series([5, 6, np.nan])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.isna()
0    False
1    False
2     True
dtype: bool
```

AlloViz.AlloViz.Elements.Nodes.items

`Nodes.items()` → Iterable[tuple[Hashable, Series]]

Iterate over (column name, Series) pairs.

Iterates over the DataFrame columns, returning a tuple with the column name and the content as a Series.

Yields

label
[object] The column names for the DataFrame being iterated over.

content
[Series] The column entries belonging to each label, as a Series.

See also:

`DataFrame.iterrows`
Iterate over DataFrame rows as (index, Series) pairs.

`DataFrame.itertuples`
Iterate over DataFrame rows as namedtuples of the values.

Examples

```
>>> df = pd.DataFrame({'species': ['bear', 'bear', 'marsupial'],
...                    'population': [1864, 22000, 80000]},
...                    index=['panda', 'polar', 'koala'])
>>> df
        species   population
panda   bear      1864
polar   bear      22000
koala   marsupial 80000
>>> for label, content in df.items():
...     print(f'label: {label}')
...     print(f'content: {content}', sep='\n')
...
label: species
content:
panda         bear
polar         bear
koala    marsupial
Name: species, dtype: object
label: population
content:
panda     1864
polar    22000
koala    80000
Name: population, dtype: int64
```

AlloViz.AlloViz.Elements.Nodes.iterrows

Nodes.**iterrows**() → Iterable[tuple[Hashable, Series]]

Iterate over DataFrame rows as (index, Series) pairs.

Yields

index
[label or tuple of label] The index of the row. A tuple for a *MultiIndex*.

data
[Series] The data of the row as a Series.

See also:

DataFrame.itertuples
Iterate over DataFrame rows as namedtuples of the values.

DataFrame.items
Iterate over (column name, Series) pairs.

Notes

1. Because `iterrows` returns a Series for each row, it does **not** preserve dtypes across the rows (dtypes are preserved across columns for DataFrames).

   To preserve dtypes while iterating over the rows, it is better to use *`itertuples()`* which returns namedtuples of the values and which is generally faster than `iterrows`.

2. You should **never modify** something you are iterating over. This is not guaranteed to work in all cases. Depending on the data types, the iterator returns a copy and not a view, and writing to it will have no effect.

Examples

```
>>> df = pd.DataFrame([[1, 1.5]], columns=['int', 'float'])
>>> row = next(df.iterrows())[1]
>>> row
int      1.0
float    1.5
Name: 0, dtype: float64
>>> print(row['int'].dtype)
float64
>>> print(df['int'].dtype)
int64
```

AlloViz.AlloViz.Elements.Nodes.itertuples

`Nodes.itertuples`(*index: bool = True*, *name: str | None = 'Pandas'*) → Iterable[tuple[Any, ...]]

Iterate over DataFrame rows as namedtuples.

Parameters

index
: [bool, default True] If True, return the index as the first element of the tuple.

name
: [str or None, default "Pandas"] The name of the returned namedtuples or None to return regular tuples.

Returns

iterator
: An object to iterate over namedtuples for each row in the DataFrame with the first field possibly being the index and following fields being the column values.

See also:

`DataFrame.iterrows`
: Iterate over DataFrame rows as (index, Series) pairs.

`DataFrame.items`
: Iterate over (column name, Series) pairs.

Notes

The column names will be renamed to positional names if they are invalid Python identifiers, repeated, or start with an underscore.

Examples

```
>>> df = pd.DataFrame({'num_legs': [4, 2], 'num_wings': [0, 2]},
...                   index=['dog', 'hawk'])
>>> df
      num_legs  num_wings
dog          4          0
hawk         2          2
>>> for row in df.itertuples():
...     print(row)
...
Pandas(Index='dog', num_legs=4, num_wings=0)
Pandas(Index='hawk', num_legs=2, num_wings=2)
```

By setting the *index* parameter to False we can remove the index as the first element of the tuple:

```
>>> for row in df.itertuples(index=False):
...     print(row)
...
Pandas(num_legs=4, num_wings=0)
Pandas(num_legs=2, num_wings=2)
```

With the *name* parameter set we set a custom name for the yielded namedtuples:

```
>>> for row in df.itertuples(name='Animal'):
...     print(row)
...
Animal(Index='dog', num_legs=4, num_wings=0)
Animal(Index='hawk', num_legs=2, num_wings=2)
```

AlloViz.AlloViz.Elements.Nodes.join

Nodes.**join**(*other: DataFrame | Series | Iterable[DataFrame | Series]*, *on: IndexLabel | None = None*, *how: MergeHow = 'left'*, *lsuffix: str = ''*, *rsuffix: str = ''*, *sort: bool = False*, *validate: JoinValidate | None = None*) → DataFrame

Join columns of another DataFrame.

Join columns with *other* DataFrame either on index or on a key column. Efficiently join multiple DataFrame objects by index at once by passing a list.

Parameters

other
[DataFrame, Series, or a list containing any combination of them] Index should be similar to one of the columns in this one. If a Series is passed, its name attribute must be set, and that will be used as the column name in the resulting joined DataFrame.

on
[str, list of str, or array-like, optional] Column or index level name(s) in the caller to join on the index in *other*, otherwise joins index-on-index. If multiple values given, the *other* DataFrame must have a MultiIndex. Can pass an array as the join key if it is not already contained in the calling DataFrame. Like an Excel VLOOKUP operation.

how
[{'left', 'right', 'outer', 'inner', 'cross'}, default 'left'] How to handle the operation of the two objects.

- left: use calling frame's index (or column if on is specified)
- right: use *other*'s index.
- outer: form union of calling frame's index (or column if on is specified) with *other*'s index, and sort it lexicographically.
- inner: form intersection of calling frame's index (or column if on is specified) with *other*'s index, preserving the order of the calling's one.
- cross: creates the cartesian product from both frames, preserves the order of the left keys.

  New in version 1.2.0.

lsuffix
[str, default ''] Suffix to use from left frame's overlapping columns.

rsuffix
[str, default ''] Suffix to use from right frame's overlapping columns.

sort
[bool, default False] Order result DataFrame lexicographically by the join key. If False, the order of the join key depends on the join type (how keyword).

validate
[str, optional] If specified, checks if join is of specified type.

- "one_to_one" or "1:1": check if join keys are unique in both left and right datasets.
- "one_to_many" or "1:m": check if join keys are unique in left dataset.
- "many_to_one" or "m:1": check if join keys are unique in right dataset.
- "many_to_many" or "m:m": allowed, but does not result in checks.

New in version 1.5.0.

Returns

DataFrame
A dataframe containing columns from both the caller and *other*.

See also:

`DataFrame.merge`
For column(s)-on-column(s) operations.

Notes

Parameters *on*, *lsuffix*, and *rsuffix* are not supported when passing a list of *DataFrame* objects.

Examples

```
>>> df = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3', 'K4', 'K5'],
...                    'A': ['A0', 'A1', 'A2', 'A3', 'A4', 'A5']})
```

```
>>> df
  key   A
0  K0  A0
1  K1  A1
2  K2  A2
3  K3  A3
4  K4  A4
5  K5  A5
```

```
>>> other = pd.DataFrame({'key': ['K0', 'K1', 'K2'],
...                       'B': ['B0', 'B1', 'B2']})
```

```
>>> other
  key   B
0  K0  B0
1  K1  B1
2  K2  B2
```

Join DataFrames using their indexes.

```
>>> df.join(other, lsuffix='_caller', rsuffix='_other')
  key_caller   A key_other    B
0         K0  A0        K0   B0
1         K1  A1        K1   B1
2         K2  A2        K2   B2
3         K3  A3       NaN  NaN
4         K4  A4       NaN  NaN
5         K5  A5       NaN  NaN
```

If we want to join using the key columns, we need to set key to be the index in both *df* and *other*. The joined DataFrame will have key as its index.

```
>>> df.set_index('key').join(other.set_index('key'))
      A    B
key
K0   A0   B0
K1   A1   B1
K2   A2   B2
K3   A3  NaN
K4   A4  NaN
K5   A5  NaN
```

Another option to join using the key columns is to use the *on* parameter. DataFrame.join always uses

other's index but we can use any column in *df*. This method preserves the original DataFrame's index in the result.

```
>>> df.join(other.set_index('key'), on='key')
  key   A    B
0  K0  A0   B0
1  K1  A1   B1
2  K2  A2   B2
3  K3  A3  NaN
4  K4  A4  NaN
5  K5  A5  NaN
```

Using non-unique key values shows how they are matched.

```
>>> df = pd.DataFrame({'key': ['K0', 'K1', 'K1', 'K3', 'K0', 'K1'],
...                    'A': ['A0', 'A1', 'A2', 'A3', 'A4', 'A5']})
```

```
>>> df
  key   A
0  K0  A0
1  K1  A1
2  K1  A2
3  K3  A3
4  K0  A4
5  K1  A5
```

```
>>> df.join(other.set_index('key'), on='key', validate='m:1')
  key   A    B
0  K0  A0   B0
1  K1  A1   B1
2  K1  A2   B1
3  K3  A3  NaN
4  K0  A4   B0
5  K1  A5   B1
```

AlloViz.AlloViz.Elements.Nodes.keys

`Nodes.keys()` → Index

Get the 'info axis' (see Indexing for more).

This is index for Series, columns for DataFrame.

Returns

Index
Info axis.

Examples

```
>>> d = pd.DataFrame(data={'A': [1, 2, 3], 'B': [0, 4, 8]},
...                  index=['a', 'b', 'c'])
>>> d
   A  B
a  1  0
b  2  4
c  3  8
>>> d.keys()
Index(['A', 'B'], dtype='object')
```

AlloViz.AlloViz.Elements.Nodes.kurt

Nodes.**kurt**(*axis: Axis | None = 0*, *skipna: bool = True*, *numeric_only: bool = False*, ***kwargs*)

Return unbiased kurtosis over requested axis.

Kurtosis obtained using Fisher's definition of kurtosis (kurtosis of normal == 0.0). Normalized by N-1.

Parameters

axis
[{index (0), columns (1)}] Axis for the function to be applied on. For *Series* this parameter is unused and defaults to 0.

For DataFrames, specifying `axis=None` will apply the aggregation across both axes.

New in version 2.0.0.

skipna
[bool, default True] Exclude NA/null values when computing the result.

numeric_only
[bool, default False] Include only float, int, boolean columns. Not implemented for Series.

****kwargs**
Additional keyword arguments to be passed to the function.

Returns

Series or scalar

Examples

```
>>> s = pd.Series([1, 2, 2, 3], index=['cat', 'dog', 'dog', 'mouse'])
>>> s
cat    1
dog    2
dog    2
mouse  3
dtype: int64
>>> s.kurt()
1.5
```

With a DataFrame

```
>>> df = pd.DataFrame({'a': [1, 2, 2, 3], 'b': [3, 4, 4, 4]},
...                   index=['cat', 'dog', 'dog', 'mouse'])
>>> df
       a   b
  cat  1   3
  dog  2   4
  dog  2   4
mouse  3   4
>>> df.kurt()
a   1.5
b   4.0
dtype: float64
```

With axis=None

```
>>> df.kurt(axis=None).round(6)
-0.988693
```

Using axis=1

```
>>> df = pd.DataFrame({'a': [1, 2], 'b': [3, 4], 'c': [3, 4], 'd': [1, 2]},
...                   index=['cat', 'dog'])
>>> df.kurt(axis=1)
cat   -6.0
dog   -6.0
dtype: float64
```

AlloViz.AlloViz.Elements.Nodes.kurtosis

`Nodes.`**`kurtosis`**(*axis: Axis | None = 0*, *skipna: bool = True*, *numeric_only: bool = False*, ***kwargs*)

Return unbiased kurtosis over requested axis.

Kurtosis obtained using Fisher's definition of kurtosis (kurtosis of normal == 0.0). Normalized by N-1.

Parameters

axis
: [{index (0), columns (1)}] Axis for the function to be applied on. For *Series* this parameter is unused and defaults to 0.

  For DataFrames, specifying `axis=None` will apply the aggregation across both axes.

  New in version 2.0.0.

skipna
: [bool, default True] Exclude NA/null values when computing the result.

numeric_only
: [bool, default False] Include only float, int, boolean columns. Not implemented for Series.

****kwargs**
: Additional keyword arguments to be passed to the function.

Returns

Series or scalar

Examples

```
>>> s = pd.Series([1, 2, 2, 3], index=['cat', 'dog', 'dog', 'mouse'])
>>> s
cat    1
dog    2
dog    2
mouse  3
dtype: int64
>>> s.kurt()
1.5
```

With a DataFrame

```
>>> df = pd.DataFrame({'a': [1, 2, 2, 3], 'b': [3, 4, 4, 4]},
...                   index=['cat', 'dog', 'dog', 'mouse'])
>>> df
       a   b
  cat  1   3
  dog  2   4
  dog  2   4
mouse  3   4
>>> df.kurt()
a   1.5
b   4.0
dtype: float64
```

With axis=None

```
>>> df.kurt(axis=None).round(6)
-0.988693
```

Using axis=1

```
>>> df = pd.DataFrame({'a': [1, 2], 'b': [3, 4], 'c': [3, 4], 'd': [1, 2]},
...                   index=['cat', 'dog'])
>>> df.kurt(axis=1)
cat   -6.0
dog   -6.0
dtype: float64
```

AlloViz.AlloViz.Elements.Nodes.last

`Nodes.last`(*offset*) → None

Select final periods of time series data based on a date offset.

For a DataFrame with a sorted DatetimeIndex, this function selects the last few rows based on a date offset.

Parameters

offset
[str, DateOffset, dateutil.relativedelta] The offset length of the data that will be selected. For instance, '3D' will display all the rows having their index within the last 3 days.

Returns

Series or DataFrame
A subset of the caller.

Raises

TypeError
If the index is not a `DatetimeIndex`

See also:

first
Select initial periods of time series based on a date offset.

at_time
Select values at a particular time of the day.

between_time
Select values between particular times of the day.

Notes

Deprecated since version 2.1.0: Please create a mask and filter using *.loc* instead

Examples

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='2D')
>>> ts = pd.DataFrame({'A': [1, 2, 3, 4]}, index=i)
>>> ts
            A
2018-04-09  1
2018-04-11  2
2018-04-13  3
2018-04-15  4
```

Get the rows for the last 3 days:

```
>>> ts.last('3D')
            A
2018-04-13  3
2018-04-15  4
```

Notice the data for 3 last calendar days were returned, not the last 3 observed days in the dataset, and therefore data for 2018-04-11 was not returned.

AlloViz.AlloViz.Elements.Nodes.last_valid_index

Nodes.**last_valid_index**() → Hashable | None

Return index for last non-NA value or None, if no non-NA value is found.

Returns

type of index

Notes

If all elements are non-NA/null, returns None. Also returns None for empty Series/DataFrame.

Examples

For Series:

```
>>> s = pd.Series([None, 3, 4])
>>> s.first_valid_index()
1
>>> s.last_valid_index()
2
```

For DataFrame:

```
>>> df = pd.DataFrame({'A': [None, None, 2], 'B': [None, 3, 4]})
>>> df
     A      B
0  NaN    NaN
1  NaN    3.0
2  2.0    4.0
>>> df.first_valid_index()
1
>>> df.last_valid_index()
2
```

AlloViz.AlloViz.Elements.Nodes.le

Nodes.**le**(*other*, *axis: Axis = 'columns'*, *level=None*)

Get Less than or equal to of dataframe and other, element-wise (binary operator *le*).

Among flexible wrappers (*eq*, *ne*, *le*, *lt*, *ge*, *gt*) to comparison operators.

Equivalent to ==, *!*=, <=, <, >=, > with support to choose axis (rows or columns) and level for comparison.

Parameters

other
[scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.

axis
[{0 or 'index', 1 or 'columns'}, default 'columns'] Whether to compare by the index (0 or 'index') or columns (1 or 'columns').

level
[int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

Returns

DataFrame of bool
Result of the comparison.

See also:

`DataFrame.eq`
Compare DataFrames for equality elementwise.

`DataFrame.ne`
Compare DataFrames for inequality elementwise.

`DataFrame.le`
Compare DataFrames for less than inequality or equality elementwise.

`DataFrame.lt`
Compare DataFrames for strictly less than inequality elementwise.

`DataFrame.ge`
Compare DataFrames for greater than inequality or equality elementwise.

`DataFrame.gt`
Compare DataFrames for strictly greater than inequality elementwise.

Notes

Mismatched indices will be unioned together. *NaN* values are considered different (i.e. *NaN* != *NaN*).

Examples

```
>>> df = pd.DataFrame({'cost': [250, 150, 100],
...                    'revenue': [100, 250, 300]},
...                   index=['A', 'B', 'C'])
>>> df
   cost  revenue
A   250      100
B   150      250
C   100      300
```

Comparison with a scalar, using either the operator or method:

```
>>> df == 100
    cost  revenue
A  False     True
```

(continues on next page)

(continued from previous page)

```
B  False    False
C   True    False
```

```
>>> df.eq(100)
    cost  revenue
A  False     True
B  False    False
C   True    False
```

When *other* is a `Series`, the columns of a DataFrame are aligned with the index of *other* and broadcast:

```
>>> df != pd.Series([100, 250], index=["cost", "revenue"])
    cost  revenue
A   True     True
B   True    False
C  False     True
```

Use the method to control the broadcast axis:

```
>>> df.ne(pd.Series([100, 300], index=["A", "D"]), axis='index')
   cost  revenue
A  True    False
B  True     True
C  True     True
D  True     True
```

When comparing to an arbitrary sequence, the number of columns must match the number elements in *other*:

```
>>> df == [250, 100]
    cost  revenue
A   True     True
B  False    False
C  False    False
```

Use the method to control the axis:

```
>>> df.eq([250, 250, 100], axis='index')
    cost  revenue
A   True    False
B  False     True
C   True    False
```

Compare to a DataFrame of different shape.

```
>>> other = pd.DataFrame({'revenue': [300, 250, 100, 150]},
...                      index=['A', 'B', 'C', 'D'])
>>> other
   revenue
A      300
B      250
C      100
D      150
```

```
>>> df.gt(other)
    cost  revenue
A  False    False
B  False    False
C  False     True
D  False    False
```

Compare to a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'cost': [250, 150, 100, 150, 300, 220],
...                              'revenue': [100, 250, 300, 200, 175, 225]},
...                             index=[['Q1', 'Q1', 'Q1', 'Q2', 'Q2', 'Q2'],
...                                    ['A', 'B', 'C', 'A', 'B', 'C']])
>>> df_multindex
      cost  revenue
Q1 A   250      100
   B   150      250
   C   100      300
Q2 A   150      200
   B   300      175
   C   220      225
```

```
>>> df.le(df_multindex, level=1)
       cost  revenue
Q1 A   True     True
   B   True     True
   C   True     True
Q2 A  False     True
   B   True    False
   C   True    False
```

AlloViz.AlloViz.Elements.Nodes.lt

`Nodes.lt`(*other*, *axis: Axis = 'columns'*, *level=None*)

Get Less than of dataframe and other, element-wise (binary operator *lt*).

Among flexible wrappers (*eq*, *ne*, *le*, *lt*, *ge*, *gt*) to comparison operators.

Equivalent to ==, *!*=, <=, <, >=, > with support to choose axis (rows or columns) and level for comparison.

Parameters

other
: [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.

axis
: [{0 or 'index', 1 or 'columns'}, default 'columns'] Whether to compare by the index (0 or 'index') or columns (1 or 'columns').

level
: [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

Returns

DataFrame of bool
Result of the comparison.

See also:

`DataFrame.eq`
Compare DataFrames for equality elementwise.

`DataFrame.ne`
Compare DataFrames for inequality elementwise.

`DataFrame.le`
Compare DataFrames for less than inequality or equality elementwise.

`DataFrame.lt`
Compare DataFrames for strictly less than inequality elementwise.

`DataFrame.ge`
Compare DataFrames for greater than inequality or equality elementwise.

`DataFrame.gt`
Compare DataFrames for strictly greater than inequality elementwise.

Notes

Mismatched indices will be unioned together. *NaN* values are considered different (i.e. *NaN* != *NaN*).

Examples

```
>>> df = pd.DataFrame({'cost': [250, 150, 100],
...                    'revenue': [100, 250, 300]},
...                   index=['A', 'B', 'C'])
>>> df
   cost  revenue
A   250      100
B   150      250
C   100      300
```

Comparison with a scalar, using either the operator or method:

```
>>> df == 100
    cost  revenue
A  False     True
B  False    False
C   True    False
```

```
>>> df.eq(100)
    cost  revenue
A  False     True
B  False    False
C   True    False
```

When *other* is a `Series`, the columns of a DataFrame are aligned with the index of *other* and broadcast:

```
>>> df != pd.Series([100, 250], index=["cost", "revenue"])
    cost  revenue
A   True     True
B   True    False
C  False     True
```

Use the method to control the broadcast axis:

```
>>> df.ne(pd.Series([100, 300], index=["A", "D"]), axis='index')
   cost  revenue
A  True    False
B  True     True
C  True     True
D  True     True
```

When comparing to an arbitrary sequence, the number of columns must match the number elements in *other*:

```
>>> df == [250, 100]
    cost  revenue
A   True     True
B  False    False
C  False    False
```

Use the method to control the axis:

```
>>> df.eq([250, 250, 100], axis='index')
    cost  revenue
A   True    False
B  False     True
C   True    False
```

Compare to a DataFrame of different shape.

```
>>> other = pd.DataFrame({'revenue': [300, 250, 100, 150]},
...                      index=['A', 'B', 'C', 'D'])
>>> other
   revenue
A      300
B      250
C      100
D      150
```

```
>>> df.gt(other)
    cost  revenue
A  False    False
B  False    False
C  False     True
D  False    False
```

Compare to a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'cost': [250, 150, 100, 150, 300, 220],
...                              'revenue': [100, 250, 300, 200, 175, 225]},
...                             index=[['Q1', 'Q1', 'Q1', 'Q2', 'Q2', 'Q2'],
...                                    ['A', 'B', 'C', 'A', 'B', 'C']])
>>> df_multindex
      cost  revenue
Q1 A   250      100
   B   150      250
   C   100      300
Q2 A   150      200
   B   300      175
   C   220      225
```

```
>>> df.le(df_multindex, level=1)
       cost  revenue
Q1 A   True     True
   B   True     True
   C   True     True
Q2 A  False     True
   B   True    False
   C   True    False
```

AlloViz.AlloViz.Elements.Nodes.map

Nodes.**map**(*func: PythonFuncType*, *na_action: str | None = None*, ***kwargs*) → DataFrame

Apply a function to a Dataframe elementwise.

New in version 2.1.0: DataFrame.applymap was deprecated and renamed to DataFrame.map.

This method applies a function that accepts and returns a scalar to every element of a DataFrame.

Parameters

func
[callable] Python function, returns a single value from a single value.

na_action
[{None, 'ignore'}, default None] If 'ignore', propagate NaN values, without passing them to func.

****kwargs**
Additional keyword arguments to pass as keywords arguments to *func*.

Returns

DataFrame
Transformed DataFrame.

See also:

DataFrame.apply
Apply a function along input axis of DataFrame.

DataFrame.replace
Replace values given in *to_replace* with *value*.

Series.map
Apply a function elementwise on a Series.

Examples

```
>>> df = pd.DataFrame([[1, 2.12], [3.356, 4.567]])
>>> df
       0      1
0  1.000  2.120
1  3.356  4.567
```

```
>>> df.map(lambda x: len(str(x)))
   0  1
0  3  4
1  5  5
```

Like Series.map, NA values can be ignored:

```
>>> df_copy = df.copy()
>>> df_copy.iloc[0, 0] = pd.NA
>>> df_copy.map(lambda x: len(str(x)), na_action='ignore')
     0  1
0  NaN  4
1  5.0  5
```

Note that a vectorized version of *func* often exists, which will be much faster. You could square each number elementwise.

```
>>> df.map(lambda x: x**2)
           0          1
0   1.000000   4.494400
1  11.262736  20.857489
```

But it's better to avoid map in that case.

```
>>> df ** 2
           0          1
0   1.000000   4.494400
1  11.262736  20.857489
```

AlloViz.AlloViz.Elements.Nodes.mask

Nodes.**mask**(*cond*, *other=_NoDefault.no_default*, ***, *inplace: bool_t = False*, *axis: Axis | None = None*, *level: Level | None = None*) → Self | None

Replace values where the condition is True.

Parameters

cond
[bool Series/DataFrame, array-like, or callable] Where *cond* is False, keep the original value. Where True, replace with corresponding value from *other*. If *cond* is

callable, it is computed on the Series/DataFrame and should return boolean Series/DataFrame or array. The callable must not change input Series/DataFrame (though pandas doesn't check it).

other
[scalar, Series/DataFrame, or callable] Entries where *cond* is True are replaced with corresponding value from *other*. If other is callable, it is computed on the Series/DataFrame and should return scalar or Series/DataFrame. The callable must not change input Series/DataFrame (though pandas doesn't check it). If not specified, entries will be filled with the corresponding NULL value (`np.nan` for numpy dtypes, `pd.NA` for extension dtypes).

inplace
[bool, default False] Whether to perform the operation in place on the data.

axis
[int, default None] Alignment axis if needed. For *Series* this parameter is unused and defaults to 0.

level
[int, default None] Alignment level if needed.

Returns

Same type as caller or None if `inplace=True`.

See also:

`DataFrame.where()`
Return an object of same shape as self.

Notes

The mask method is an application of the if-then idiom. For each element in the calling DataFrame, if `cond` is `False` the element is used; otherwise the corresponding element from the DataFrame `other` is used. If the axis of `other` does not align with axis of `cond` Series/DataFrame, the misaligned index positions will be filled with True.

The signature for `DataFrame.where()` differs from `numpy.where()`. Roughly `df1.where(m, df2)` is equivalent to `np.where(m, df1, df2)`.

For further details and examples see the `mask` documentation in indexing.

The dtype of the object takes precedence. The fill value is casted to the object's dtype, if this can be done losslessly.

Examples

```
>>> s = pd.Series(range(5))
>>> s.where(s > 0)
0    NaN
1    1.0
2    2.0
3    3.0
4    4.0
dtype: float64
```

(continues on next page)

(continued from previous page)

```
>>> s.mask(s > 0)
0    0.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

```
>>> s = pd.Series(range(5))
>>> t = pd.Series([True, False])
>>> s.where(t, 99)
0     0
1    99
2    99
3    99
4    99
dtype: int64
>>> s.mask(t, 99)
0    99
1     1
2    99
3    99
4    99
dtype: int64
```

```
>>> s.where(s > 1, 10)
0    10
1    10
2    2
3    3
4    4
dtype: int64
>>> s.mask(s > 1, 10)
0     0
1     1
2    10
3    10
4    10
dtype: int64
```

```
>>> df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])
>>> df
   A  B
0  0  1
1  2  3
2  4  5
3  6  7
4  8  9
>>> m = df % 3 == 0
>>> df.where(m, -df)
   A  B
```

(continues on next page)

(continued from previous page)

```
0  0 -1
1 -2  3
2 -4 -5
3  6 -7
4 -8  9
>>> df.where(m, -df) == np.where(m, df, -df)
      A     B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True
>>> df.where(m, -df) == df.mask(~m, -df)
      A     B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True
```

AlloViz.AlloViz.Elements.Nodes.max

Nodes.**max**(*axis: Axis | None = 0*, *skipna: bool = True*, *numeric_only: bool = False*, ***kwargs*)

Return the maximum of the values over the requested axis.

If you want the *index* of the maximum, use `idxmax`. This is the equivalent of the `numpy.ndarray` method `argmax`.

Parameters

axis
[{index (0), columns (1)}] Axis for the function to be applied on. For *Series* this parameter is unused and defaults to 0.

For DataFrames, specifying `axis=None` will apply the aggregation across both axes.

New in version 2.0.0.

skipna
[bool, default True] Exclude NA/null values when computing the result.

numeric_only
[bool, default False] Include only float, int, boolean columns. Not implemented for Series.

****kwargs**
Additional keyword arguments to be passed to the function.

Returns

Series or scalar

See also:

`Series.sum`
Return the sum.

Series.min
Return the minimum.

Series.max
Return the maximum.

Series.idxmin
Return the index of the minimum.

Series.idxmax
Return the index of the maximum.

DataFrame.sum
Return the sum over the requested axis.

DataFrame.min
Return the minimum over the requested axis.

DataFrame.max
Return the maximum over the requested axis.

DataFrame.idxmin
Return the index of the minimum over the requested axis.

DataFrame.idxmax
Return the index of the maximum over the requested axis.

Examples

```
>>> idx = pd.MultiIndex.from_arrays([
...     ['warm', 'warm', 'cold', 'cold'],
...     ['dog', 'falcon', 'fish', 'spider']],
...     names=['blooded', 'animal'])
>>> s = pd.Series([4, 2, 0, 8], name='legs', index=idx)
>>> s
blooded  animal
warm     dog       4
         falcon    2
cold     fish      0
         spider    8
Name: legs, dtype: int64
```

```
>>> s.max()
8
```

AlloViz.AlloViz.Elements.Nodes.mean

Nodes.**mean**(*axis: Axis | None = 0*, *skipna: bool = True*, *numeric_only: bool = False*, ***kwargs*)

Return the mean of the values over the requested axis.

Parameters

axis
[{index (0), columns (1)}] Axis for the function to be applied on. For *Series* this parameter is unused and defaults to 0.

For DataFrames, specifying `axis=None` will apply the aggregation across both axes.

New in version 2.0.0.

skipna
[bool, default True] Exclude NA/null values when computing the result.

numeric_only
[bool, default False] Include only float, int, boolean columns. Not implemented for Series.

****kwargs**
Additional keyword arguments to be passed to the function.

Returns

Series or scalar

Examples

```
>>> s = pd.Series([1, 2, 3])
>>> s.mean()
2.0
```

With a DataFrame

```
>>> df = pd.DataFrame({'a': [1, 2], 'b': [2, 3]}, index=['tiger', 'zebra'])
>>> df
       a   b
tiger  1   2
zebra  2   3
>>> df.mean()
a   1.5
b   2.5
dtype: float64
```

Using axis=1

```
>>> df.mean(axis=1)
tiger   1.5
zebra   2.5
dtype: float64
```

In this case, *numeric_only* should be set to *True* to avoid getting an error.

```
>>> df = pd.DataFrame({'a': [1, 2], 'b': ['T', 'Z']},
...                   index=['tiger', 'zebra'])
>>> df.mean(numeric_only=True)
a   1.5
dtype: float64
```

AlloViz.AlloViz.Elements.Nodes.median

Nodes.**median**(*axis: Axis | None = 0*, *skipna: bool = True*, *numeric_only: bool = False*, ***kwargs*)

Return the median of the values over the requested axis.

Parameters

axis
[{index (0), columns (1)}] Axis for the function to be applied on. For *Series* this parameter is unused and defaults to 0.

For DataFrames, specifying `axis=None` will apply the aggregation across both axes.

New in version 2.0.0.

skipna
[bool, default True] Exclude NA/null values when computing the result.

numeric_only
[bool, default False] Include only float, int, boolean columns. Not implemented for Series.

****kwargs**
Additional keyword arguments to be passed to the function.

Returns

Series or scalar

Examples

```
>>> s = pd.Series([1, 2, 3])
>>> s.median()
2.0
```

With a DataFrame

```
>>> df = pd.DataFrame({'a': [1, 2], 'b': [2, 3]}, index=['tiger', 'zebra'])
>>> df
       a   b
tiger  1   2
zebra  2   3
>>> df.median()
a   1.5
b   2.5
dtype: float64
```

Using axis=1

```
>>> df.median(axis=1)
tiger   1.5
zebra   2.5
dtype: float64
```

In this case, *numeric_only* should be set to *True* to avoid getting an error.

```
>>> df = pd.DataFrame({'a': [1, 2], 'b': ['T', 'Z']},
...                   index=['tiger', 'zebra'])
>>> df.median(numeric_only=True)
a   1.5
dtype: float64
```

AlloViz.AlloViz.Elements.Nodes.melt

Nodes.**melt**(*id_vars=None*, *value_vars=None*, *var_name=None*, *value_name: Hashable = 'value'*, *col_level: Level | None = None*, *ignore_index: bool = True*) → DataFrame

Unpivot a DataFrame from wide to long format, optionally leaving identifiers set.

This function is useful to massage a DataFrame into a format where one or more columns are identifier variables (*id_vars*), while all other columns, considered measured variables (*value_vars*), are "unpivoted" to the row axis, leaving just two non-identifier columns, 'variable' and 'value'.

Parameters

id_vars
[tuple, list, or ndarray, optional] Column(s) to use as identifier variables.

value_vars
[tuple, list, or ndarray, optional] Column(s) to unpivot. If not specified, uses all columns that are not set as *id_vars*.

var_name
[scalar] Name to use for the 'variable' column. If None it uses `frame.columns.name` or 'variable'.

value_name
[scalar, default 'value'] Name to use for the 'value' column.

col_level
[int or str, optional] If columns are a MultiIndex then use this level to melt.

ignore_index
[bool, default True] If True, original index is ignored. If False, the original index is retained. Index labels will be repeated as necessary.

Returns

DataFrame
Unpivoted DataFrame.

See also:

melt
Identical method.

pivot_table
Create a spreadsheet-style pivot table as a DataFrame.

`DataFrame.pivot`
Return reshaped DataFrame organized by given index / column values.

`DataFrame.explode`
Explode a DataFrame from list-like columns to long format.

Notes

Reference the user guide for more examples.

Examples

```
>>> df = pd.DataFrame({'A': {0: 'a', 1: 'b', 2: 'c'},
...                    'B': {0: 1, 1: 3, 2: 5},
...                    'C': {0: 2, 1: 4, 2: 6}})
>>> df
   A  B  C
0  a  1  2
1  b  3  4
2  c  5  6
```

```
>>> df.melt(id_vars=['A'], value_vars=['B'])
   A variable  value
0  a        B      1
1  b        B      3
2  c        B      5
```

```
>>> df.melt(id_vars=['A'], value_vars=['B', 'C'])
   A variable  value
0  a        B      1
1  b        B      3
2  c        B      5
3  a        C      2
4  b        C      4
5  c        C      6
```

The names of 'variable' and 'value' columns can be customized:

```
>>> df.melt(id_vars=['A'], value_vars=['B'],
...         var_name='myVarname', value_name='myValname')
   A myVarname  myValname
0  a         B          1
1  b         B          3
2  c         B          5
```

Original index values can be kept around:

```
>>> df.melt(id_vars=['A'], value_vars=['B', 'C'], ignore_index=False)
   A variable  value
0  a        B      1
1  b        B      3
2  c        B      5
0  a        C      2
1  b        C      4
2  c        C      6
```

If you have multi-index columns:

```
>>> df.columns = [list('ABC'), list('DEF')]
>>> df
   A  B  C
   D  E  F
0  a  1  2
1  b  3  4
2  c  5  6
```

```
>>> df.melt(col_level=0, id_vars=['A'], value_vars=['B'])
   A variable  value
0  a        B      1
1  b        B      3
2  c        B      5
```

```
>>> df.melt(id_vars=[('A', 'D')], value_vars=[('B', 'E')])
  (A, D) variable_0 variable_1  value
0      a          B          E      1
1      b          B          E      3
2      c          B          E      5
```

AlloViz.AlloViz.Elements.Nodes.memory_usage

Nodes.**memory_usage**(*index: bool = True*, *deep: bool = False*) → Series

Return the memory usage of each column in bytes.

The memory usage can optionally include the contribution of the index and elements of *object* dtype.

This value is displayed in *DataFrame.info* by default. This can be suppressed by setting `pandas.options.display.memory_usage` to False.

Parameters

index
: [bool, default True] Specifies whether to include the memory usage of the DataFrame's index in returned Series. If `index=True`, the memory usage of the index is the first item in the output.

deep
: [bool, default False] If True, introspect the data deeply by interrogating *object* dtypes for system-level memory consumption, and include it in the returned values.

Returns

Series
: A Series whose index is the original column names and whose values is the memory usage of each column in bytes.

See also:

`numpy.ndarray.nbytes`
: Total bytes consumed by the elements of an ndarray.

`Series.memory_usage`
: Bytes consumed by a Series.

Categorical
: Memory-efficient array for string values with many repeated values.

DataFrame.info
: Concise summary of a DataFrame.

Notes

See the Frequently Asked Questions for more details.

Examples

```
>>> dtypes = ['int64', 'float64', 'complex128', 'object', 'bool']
>>> data = dict([(t, np.ones(shape=5000, dtype=int).astype(t))
...              for t in dtypes])
>>> df = pd.DataFrame(data)
>>> df.head()
   int64  float64            complex128  object  bool
0      1      1.0              1.0+0.0j       1  True
1      1      1.0              1.0+0.0j       1  True
2      1      1.0              1.0+0.0j       1  True
3      1      1.0              1.0+0.0j       1  True
4      1      1.0              1.0+0.0j       1  True
```

```
>>> df.memory_usage()
Index           128
int64         40000
float64       40000
complex128    80000
object        40000
bool           5000
dtype: int64
```

```
>>> df.memory_usage(index=False)
int64         40000
float64       40000
complex128    80000
object        40000
bool           5000
dtype: int64
```

The memory footprint of *object* dtype columns is ignored by default:

```
>>> df.memory_usage(deep=True)
Index           128
int64         40000
float64       40000
complex128    80000
object       180000
bool           5000
dtype: int64
```

Use a Categorical for efficient storage of an object-dtype column with many repeated values.

```
>>> df['object'].astype('category').memory_usage(deep=True)
5244
```

AlloViz.AlloViz.Elements.Nodes.merge

Nodes.**merge**(*right: DataFrame | Series*, *how: MergeHow = 'inner'*, *on: IndexLabel | None = None*, *left_on: IndexLabel | None = None*, *right_on: IndexLabel | None = None*, *left_index: bool = False*, *right_index: bool = False*, *sort: bool = False*, *suffixes: Suffixes = ('_x', '_y')*, *copy: bool | None = None*, *indicator: str | bool = False*, *validate: MergeValidate | None = None*) → DataFrame

Merge DataFrame or named Series objects with a database-style join.

A named Series object is treated as a DataFrame with a single named column.

The join is done on columns or indexes. If joining columns on columns, the DataFrame indexes *will be ignored*. Otherwise if joining indexes on indexes or indexes on a column or columns, the index will be passed on. When performing a cross merge, no column specifications to merge on are allowed.

Warning: If both key columns contain rows where the key is a null value, those rows will be matched against each other. This is different from usual SQL join behaviour and can lead to unexpected results.

Parameters

right
[DataFrame or named Series] Object to merge with.

how
[{'left', 'right', 'outer', 'inner', 'cross'}, default 'inner'] Type of merge to be performed.

- left: use only keys from left frame, similar to a SQL left outer join; preserve key order.
- right: use only keys from right frame, similar to a SQL right outer join; preserve key order.
- outer: use union of keys from both frames, similar to a SQL full outer join; sort keys lexicographically.
- inner: use intersection of keys from both frames, similar to a SQL inner join; preserve the order of the left keys.
- cross: creates the cartesian product from both frames, preserves the order of the left keys.

  New in version 1.2.0.

on
[label or list] Column or index level names to join on. These must be found in both DataFrames. If *on* is None and not merging on indexes then this defaults to the intersection of the columns in both DataFrames.

left_on
[label or list, or array-like] Column or index level names to join on in the left DataFrame. Can also be an array or list of arrays of the length of the left DataFrame. These arrays are treated as if they are columns.

right_on
[label or list, or array-like] Column or index level names to join on in the right DataFrame. Can also be an array or list of arrays of the length of the right DataFrame. These arrays are treated as if they are columns.

left_index
[bool, default False] Use the index from the left DataFrame as the join key(s). If it is a MultiIndex, the number of keys in the other DataFrame (either the index or a number of columns) must match the number of levels.

right_index
[bool, default False] Use the index from the right DataFrame as the join key. Same caveats as left_index.

sort
[bool, default False] Sort the join keys lexicographically in the result DataFrame. If False, the order of the join keys depends on the join type (how keyword).

suffixes
[list-like, default is ("_x", "_y")] A length-2 sequence where each element is optionally a string indicating the suffix to add to overlapping column names in *left* and *right* respectively. Pass a value of *None* instead of a string to indicate that the column name from *left* or *right* should be left as-is, with no suffix. At least one of the values must not be None.

copy
[bool, default True] If False, avoid copy if possible.

indicator
[bool or str, default False] If True, adds a column to the output DataFrame called "_merge" with information on the source of each row. The column can be given a different name by providing a string argument. The column will have a Categorical type with the value of "left_only" for observations whose merge key only appears in the left DataFrame, "right_only" for observations whose merge key only appears in the right DataFrame, and "both" if the observation's merge key is found in both DataFrames.

validate
[str, optional] If specified, checks if merge is of specified type.

- "one_to_one" or "1:1": check if merge keys are unique in both left and right datasets.
- "one_to_many" or "1:m": check if merge keys are unique in left dataset.
- "many_to_one" or "m:1": check if merge keys are unique in right dataset.
- "many_to_many" or "m:m": allowed, but does not result in checks.

Returns

DataFrame
A DataFrame of the two merged objects.

See also:

`merge_ordered`
Merge with optional filling/interpolation.

`merge_asof`
Merge on nearest keys.

DataFrame.join
Similar method using indices.

Examples

```
>>> df1 = pd.DataFrame({'lkey': ['foo', 'bar', 'baz', 'foo'],
...                     'value': [1, 2, 3, 5]})
>>> df2 = pd.DataFrame({'rkey': ['foo', 'bar', 'baz', 'foo'],
...                     'value': [5, 6, 7, 8]})
>>> df1
    lkey value
0   foo      1
1   bar      2
2   baz      3
3   foo      5
>>> df2
    rkey value
0   foo      5
1   bar      6
2   baz      7
3   foo      8
```

Merge df1 and df2 on the lkey and rkey columns. The value columns have the default suffixes, _x and _y, appended.

```
>>> df1.merge(df2, left_on='lkey', right_on='rkey')
  lkey  value_x rkey  value_y
0  foo        1  foo        5
1  foo        1  foo        8
2  foo        5  foo        5
3  foo        5  foo        8
4  bar        2  bar        6
5  baz        3  baz        7
```

Merge DataFrames df1 and df2 with specified left and right suffixes appended to any overlapping columns.

```
>>> df1.merge(df2, left_on='lkey', right_on='rkey',
...           suffixes=('_left', '_right'))
  lkey  value_left rkey  value_right
0  foo           1  foo           5
1  foo           1  foo           8
2  foo           5  foo           5
3  foo           5  foo           8
4  bar           2  bar           6
5  baz           3  baz           7
```

Merge DataFrames df1 and df2, but raise an exception if the DataFrames have any overlapping columns.

```
>>> df1.merge(df2, left_on='lkey', right_on='rkey', suffixes=(False, False))
Traceback (most recent call last):
...
ValueError: columns overlap but no suffix specified:
    Index(['value'], dtype='object')
```

```
>>> df1 = pd.DataFrame({'a': ['foo', 'bar'], 'b': [1, 2]})
>>> df2 = pd.DataFrame({'a': ['foo', 'baz'], 'c': [3, 4]})
>>> df1
      a  b
0   foo  1
1   bar  2
>>> df2
      a  c
0   foo  3
1   baz  4
```

```
>>> df1.merge(df2, how='inner', on='a')
      a  b  c
0   foo  1  3
```

```
>>> df1.merge(df2, how='left', on='a')
      a  b  c
0   foo  1  3.0
1   bar  2  NaN
```

```
>>> df1 = pd.DataFrame({'left': ['foo', 'bar']})
>>> df2 = pd.DataFrame({'right': [7, 8]})
>>> df1
    left
0   foo
1   bar
>>> df2
    right
0   7
1   8
```

```
>>> df1.merge(df2, how='cross')
   left  right
0   foo      7
1   foo      8
2   bar      7
3   bar      8
```

AlloViz.AlloViz.Elements.Nodes.min

Nodes.**min**(*axis: Axis | None = 0*, *skipna: bool = True*, *numeric_only: bool = False*, ***kwargs*)

Return the minimum of the values over the requested axis.

If you want the *index* of the minimum, use `idxmin`. This is the equivalent of the `numpy.ndarray` method `argmin`.

Parameters

axis

[{index (0), columns (1)}] Axis for the function to be applied on. For *Series* this parameter is unused and defaults to 0.

For DataFrames, specifying `axis=None` will apply the aggregation across both axes.

New in version 2.0.0.

skipna
[bool, default True] Exclude NA/null values when computing the result.

numeric_only
[bool, default False] Include only float, int, boolean columns. Not implemented for Series.

****kwargs**
Additional keyword arguments to be passed to the function.

Returns

Series or scalar

See also:

`Series.sum`
Return the sum.

`Series.min`
Return the minimum.

`Series.max`
Return the maximum.

`Series.idxmin`
Return the index of the minimum.

`Series.idxmax`
Return the index of the maximum.

`DataFrame.sum`
Return the sum over the requested axis.

`DataFrame.min`
Return the minimum over the requested axis.

`DataFrame.max`
Return the maximum over the requested axis.

`DataFrame.idxmin`
Return the index of the minimum over the requested axis.

`DataFrame.idxmax`
Return the index of the maximum over the requested axis.

Examples

```
>>> idx = pd.MultiIndex.from_arrays([
...     ['warm', 'warm', 'cold', 'cold'],
...     ['dog', 'falcon', 'fish', 'spider']],
...     names=['blooded', 'animal'])
>>> s = pd.Series([4, 2, 0, 8], name='legs', index=idx)
>>> s
blooded  animal
```

(continues on next page)

(continued from previous page)

```
warm     dog       4
         falcon    2
cold     fish      0
         spider    8
Name: legs, dtype: int64
```

```
>>> s.min()
0
```

AlloViz.AlloViz.Elements.Nodes.mod

Nodes.**mod**(*other*, *axis: Axis = 'columns'*, *level=None*, *fill_value=None*)

Get Modulo of dataframe and other, element-wise (binary operator *mod*).

Equivalent to `dataframe % other`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *rmod*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *floordiv*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other
: [scalar, sequence, Series, dict or DataFrame] Any single or multiple element data structure, or list-like object.

axis
: [{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns. (1 or 'columns'). For Series input, axis to match Series index on.

level
: [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value
: [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame
: Result of the arithmetic operation.

See also:

`DataFrame.add`
: Add DataFrames.

`DataFrame.sub`
: Subtract DataFrames.

`DataFrame.mul`
: Multiply DataFrames.

DataFrame.div
Divide DataFrames (float division).

DataFrame.truediv
Divide DataFrames (float division).

DataFrame.floordiv
Divide DataFrames (integer division).

DataFrame.mod
Calculate modulo (remainder after division).

DataFrame.pow
Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a dictionary by axis.

```
>>> df.mul({'angles': 0, 'degrees': 2})
            angles  degrees
circle           0      720
triangle         0      360
rectangle        0      720
```

```
>>> df.mul({'circle': 0, 'triangle': 2, 'rectangle': 3}, axis='index')
            angles  degrees
circle           0        0
triangle         6      360
rectangle       12     1080
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

AlloViz.AlloViz.Elements.Nodes.mode

Nodes.**mode**(*axis: Axis = 0*, *numeric_only: bool = False*, *dropna: bool = True*) → DataFrame

Get the mode(s) of each element along the selected axis.

The mode of a set of values is the value that appears most often. It can be multiple values.

Parameters

axis

[{0 or 'index', 1 or 'columns'}, default 0] The axis to iterate over while searching for the mode:

- 0 or 'index' : get mode of each column
- 1 or 'columns' : get mode of each row.

numeric_only
[bool, default False] If True, only apply to numeric columns.

dropna
[bool, default True] Don't consider counts of NaN/NaT.

Returns

DataFrame
The modes of each column or row.

See also:

`Series.mode`
Return the highest frequency value in a Series.

`Series.value_counts`
Return the counts of values in a Series.

Examples

```
>>> df = pd.DataFrame([('bird', 2, 2),
...                    ('mammal', 4, np.nan),
...                    ('arthropod', 8, 0),
...                    ('bird', 2, np.nan)],
...                   index=('falcon', 'horse', 'spider', 'ostrich'),
...                   columns=('species', 'legs', 'wings'))
>>> df
           species  legs  wings
falcon        bird     2    2.0
horse       mammal     4    NaN
spider   arthropod     8    0.0
ostrich       bird     2    NaN
```

By default, missing values are not considered, and the mode of wings are both 0 and 2. Because the resulting DataFrame has two rows, the second row of `species` and `legs` contains `NaN`.

```
>>> df.mode()
  species  legs  wings
0    bird   2.0    0.0
1     NaN   NaN    2.0
```

Setting `dropna=False` `NaN` values are considered and they can be the mode (like for wings).

```
>>> df.mode(dropna=False)
  species  legs  wings
0    bird     2    NaN
```

Setting `numeric_only=True`, only the mode of numeric columns is computed, and columns of other types are ignored.

```
>>> df.mode(numeric_only=True)
   legs  wings
0   2.0    0.0
1   NaN    2.0
```

To compute the mode over columns and not rows, use the axis parameter:

```
>>> df.mode(axis='columns', numeric_only=True)
           0    1
falcon   2.0  NaN
horse    4.0  NaN
spider   0.0  8.0
ostrich  2.0  NaN
```

AlloViz.AlloViz.Elements.Nodes.mul

Nodes.**mul**(*other*, *axis: Axis = 'columns'*, *level=None*, *fill_value=None*)

Get Multiplication of dataframe and other, element-wise (binary operator *mul*).

Equivalent to `dataframe * other`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *rmul*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *floordiv*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other
: [scalar, sequence, Series, dict or DataFrame] Any single or multiple element data structure, or list-like object.

axis
: [{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns. (1 or 'columns'). For Series input, axis to match Series index on.

level
: [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value
: [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame
: Result of the arithmetic operation.

See also:

DataFrame.add
: Add DataFrames.

DataFrame.sub
: Subtract DataFrames.

DataFrame.mul
: Multiply DataFrames.

DataFrame.div
: Divide DataFrames (float division).

DataFrame.truediv
Divide DataFrames (float division).

DataFrame.floordiv
Divide DataFrames (integer division).

DataFrame.mod
Calculate modulo (remainder after division).

DataFrame.pow
Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
```

(continues on next page)

(continued from previous page)

```
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a dictionary by axis.

```
>>> df.mul({'angles': 0, 'degrees': 2})
            angles  degrees
circle           0      720
triangle         0      360
rectangle        0      720
```

```
>>> df.mul({'circle': 0, 'triangle': 2, 'rectangle': 3}, axis='index')
            angles  degrees
circle           0        0
triangle         6      360
rectangle       12     1080
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
```

(continues on next page)

(continued from previous page)

```
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

AlloViz.AlloViz.Elements.Nodes.multiply

Nodes.**multiply**(*other*, *axis: Axis = 'columns'*, *level=None*, *fill_value=None*)

Get Multiplication of dataframe and other, element-wise (binary operator *mul*).

Equivalent to `dataframe * other`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *rmul*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *floordiv*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other
[scalar, sequence, Series, dict or DataFrame] Any single or multiple element data structure, or list-like object.

axis
[{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or

columns. (1 or 'columns'). For Series input, axis to match Series index on.

level
[int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value
[float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame
Result of the arithmetic operation.

See also:

`DataFrame.add`
Add DataFrames.

`DataFrame.sub`
Subtract DataFrames.

`DataFrame.mul`
Multiply DataFrames.

`DataFrame.div`
Divide DataFrames (float division).

`DataFrame.truediv`
Divide DataFrames (float division).

`DataFrame.floordiv`
Divide DataFrames (integer division).

`DataFrame.mod`
Calculate modulo (remainder after division).

`DataFrame.pow`
Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a dictionary by axis.

```
>>> df.mul({'angles': 0, 'degrees': 2})
            angles  degrees
```

(continues on next page)

(continued from previous page)

```
circle           0      720
triangle         0      360
rectangle        0      720
```

```
>>> df.mul({'circle': 0, 'triangle': 2, 'rectangle': 3}, axis='index')
           angles  degrees
circle          0        0
triangle        6      360
rectangle      12     1080
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
```

(continues on next page)

(continued from previous page)

```
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

AlloViz.AlloViz.Elements.Nodes.ne

Nodes.**ne**(*other*, *axis: Axis = 'columns'*, *level=None*)

Get Not equal to of dataframe and other, element-wise (binary operator *ne*).

Among flexible wrappers (*eq*, *ne*, *le*, *lt*, *ge*, *gt*) to comparison operators.

Equivalent to ==, *!*=, <=, <, >=, > with support to choose axis (rows or columns) and level for comparison.

Parameters

other
[scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.

axis
[{0 or 'index', 1 or 'columns'}, default 'columns'] Whether to compare by the index (0 or 'index') or columns (1 or 'columns').

level
[int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

Returns

DataFrame of bool
Result of the comparison.

See also:

DataFrame.eq
Compare DataFrames for equality elementwise.

DataFrame.ne
Compare DataFrames for inequality elementwise.

DataFrame.le
Compare DataFrames for less than inequality or equality elementwise.

DataFrame.lt
Compare DataFrames for strictly less than inequality elementwise.

DataFrame.ge
Compare DataFrames for greater than inequality or equality elementwise.

DataFrame.gt
Compare DataFrames for strictly greater than inequality elementwise.

Notes

Mismatched indices will be unioned together. *NaN* values are considered different (i.e. *NaN* != *NaN*).

Examples

```
>>> df = pd.DataFrame({'cost': [250, 150, 100],
...                    'revenue': [100, 250, 300]},
...                   index=['A', 'B', 'C'])
>>> df
   cost  revenue
A   250      100
B   150      250
C   100      300
```

Comparison with a scalar, using either the operator or method:

```
>>> df == 100
    cost  revenue
A  False     True
B  False    False
C   True    False
```

```
>>> df.eq(100)
    cost  revenue
A  False     True
B  False    False
C   True    False
```

When *other* is a `Series`, the columns of a DataFrame are aligned with the index of *other* and broadcast:

```
>>> df != pd.Series([100, 250], index=["cost", "revenue"])
    cost  revenue
A   True     True
B   True    False
C  False     True
```

Use the method to control the broadcast axis:

```
>>> df.ne(pd.Series([100, 300], index=["A", "D"]), axis='index')
   cost  revenue
A  True    False
B  True     True
C  True     True
D  True     True
```

When comparing to an arbitrary sequence, the number of columns must match the number elements in *other*:

```
>>> df == [250, 100]
    cost  revenue
A   True     True
```

(continues on next page)

(continued from previous page)

```
B  False    False
C  False    False
```

Use the method to control the axis:

```
>>> df.eq([250, 250, 100], axis='index')
    cost  revenue
A   True    False
B  False     True
C   True    False
```

Compare to a DataFrame of different shape.

```
>>> other = pd.DataFrame({'revenue': [300, 250, 100, 150]},
...                      index=['A', 'B', 'C', 'D'])
>>> other
   revenue
A      300
B      250
C      100
D      150
```

```
>>> df.gt(other)
    cost  revenue
A  False    False
B  False    False
C  False     True
D  False    False
```

Compare to a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'cost': [250, 150, 100, 150, 300, 220],
...                              'revenue': [100, 250, 300, 200, 175, 225]},
...                             index=[['Q1', 'Q1', 'Q1', 'Q2', 'Q2', 'Q2'],
...                                    ['A', 'B', 'C', 'A', 'B', 'C']])
>>> df_multindex
      cost  revenue
Q1 A   250      100
   B   150      250
   C   100      300
Q2 A   150      200
   B   300      175
   C   220      225
```

```
>>> df.le(df_multindex, level=1)
       cost  revenue
Q1 A   True     True
   B   True     True
   C   True     True
Q2 A  False     True
   B   True    False
   C   True    False
```

AlloViz.AlloViz.Elements.Nodes.nlargest

`Nodes.nlargest`(*n: int*, *columns: IndexLabel*, *keep: NsmallestNlargestKeep = 'first'*) → DataFrame

Return the first *n* rows ordered by *columns* in descending order.

Return the first *n* rows with the largest values in *columns*, in descending order. The columns that are not specified are returned as well, but not used for ordering.

This method is equivalent to `df.sort_values(columns, ascending=False).head(n)`, but more performant.

Parameters

n
[int] Number of rows to return.

columns
[label or list of labels] Column label(s) to order by.

keep
[{'first', 'last', 'all'}, default 'first'] Where there are duplicate values:

- `first` : prioritize the first occurrence(s)
- `last` : prioritize the last occurrence(s)
- `all` : do not drop any duplicates, even it means selecting more than *n* items.

Returns

DataFrame
The first *n* rows ordered by the given columns in descending order.

See also:

`DataFrame.nsmallest`
Return the first *n* rows ordered by *columns* in ascending order.

`DataFrame.sort_values`
Sort DataFrame by the values.

`DataFrame.head`
Return the first *n* rows without re-ordering.

Notes

This function cannot be used with all column types. For example, when specifying columns with *object* or *category* dtypes, `TypeError` is raised.

Examples

```
>>> df = pd.DataFrame({'population': [59000000, 65000000, 434000,
...                                   434000, 434000, 337000, 11300,
...                                   11300, 11300],
...                    'GDP': [1937894, 2583560 , 12011, 4520, 12128,
...                            17036, 182, 38, 311],
...                    'alpha-2': ["IT", "FR", "MT", "MV", "BN",
...                                "IS", "NR", "TV", "AI"]},
```

(continues on next page)

(continued from previous page)

```
...                   index=["Italy", "France", "Malta",
...                          "Maldives", "Brunei", "Iceland",
...                          "Nauru", "Tuvalu", "Anguilla"])
>>> df
          population      GDP alpha-2
Italy       59000000  1937894      IT
France      65000000  2583560      FR
Malta         434000    12011      MT
Maldives      434000     4520      MV
Brunei        434000    12128      BN
Iceland       337000    17036      IS
Nauru          11300      182      NR
Tuvalu         11300       38      TV
Anguilla       11300      311      AI
```

In the following example, we will use `nlargest` to select the three rows having the largest values in column "population".

```
>>> df.nlargest(3, 'population')
        population      GDP alpha-2
France    65000000  2583560      FR
Italy     59000000  1937894      IT
Malta       434000    12011      MT
```

When using `keep='last'`, ties are resolved in reverse order:

```
>>> df.nlargest(3, 'population', keep='last')
        population      GDP alpha-2
France    65000000  2583560      FR
Italy     59000000  1937894      IT
Brunei      434000    12128      BN
```

When using `keep='all'`, all duplicate items are maintained:

```
>>> df.nlargest(3, 'population', keep='all')
          population      GDP alpha-2
France      65000000  2583560      FR
Italy       59000000  1937894      IT
Malta         434000    12011      MT
Maldives      434000     4520      MV
Brunei        434000    12128      BN
```

To order by the largest values in column "population" and then "GDP", we can specify multiple columns like in the next example.

```
>>> df.nlargest(3, ['population', 'GDP'])
        population      GDP alpha-2
France    65000000  2583560      FR
Italy     59000000  1937894      IT
Brunei      434000    12128      BN
```

AlloViz.AlloViz.Elements.Nodes.notna

Nodes.**notna**() → DataFrame

Detect existing (non-missing) values.

Return a boolean same-sized object indicating if the values are not NA. Non-missing values get mapped to True. Characters such as empty strings `''` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`). NA values, such as None or `numpy.NaN`, get mapped to False values.

Returns

DataFrame
Mask of bool values for each element in DataFrame that indicates whether an element is not an NA value.

See also:

DataFrame.notnull
Alias of notna.

DataFrame.isna
Boolean inverse of notna.

DataFrame.dropna
Omit axes labels with missing values.

notna
Top-level notna.

Examples

Show which entries in a DataFrame are not NA.

```
>>> df = pd.DataFrame(dict(age=[5, 6, np.nan],
...                        born=[pd.NaT, pd.Timestamp('1939-05-27'),
...                              pd.Timestamp('1940-04-25')],
...                        name=['Alfred', 'Batman', ''],
...                        toy=[None, 'Batmobile', 'Joker']))
>>> df
   age       born    name        toy
0  5.0        NaT  Alfred       None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25              Joker
```

```
>>> df.notna()
     age   born  name    toy
0   True  False  True  False
1   True   True  True   True
2  False   True  True   True
```

Show which entries in a Series are not NA.

```
>>> ser = pd.Series([5, 6, np.nan])
>>> ser
```

(continues on next page)

(continued from previous page)

```
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.notna()
0     True
1     True
2    False
dtype: bool
```

AlloViz.AlloViz.Elements.Nodes.notnull

Nodes.**notnull**() → DataFrame

DataFrame.notnull is an alias for DataFrame.notna.

Detect existing (non-missing) values.

Return a boolean same-sized object indicating if the values are not NA. Non-missing values get mapped to True. Characters such as empty strings `''` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`). NA values, such as None or `numpy.NaN`, get mapped to False values.

Returns

DataFrame
Mask of bool values for each element in DataFrame that indicates whether an element is not an NA value.

See also:

DataFrame.notnull
Alias of notna.

DataFrame.isna
Boolean inverse of notna.

DataFrame.dropna
Omit axes labels with missing values.

notna
Top-level notna.

Examples

Show which entries in a DataFrame are not NA.

```
>>> df = pd.DataFrame(dict(age=[5, 6, np.nan],
...                        born=[pd.NaT, pd.Timestamp('1939-05-27'),
...                              pd.Timestamp('1940-04-25')],
...                        name=['Alfred', 'Batman', ''],
...                        toy=[None, 'Batmobile', 'Joker']))
>>> df
```

(continues on next page)

(continued from previous page)

```
   age       born    name        toy
0  5.0        NaT  Alfred       None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25              Joker
```

```
>>> df.notna()
     age   born  name    toy
0   True  False  True  False
1   True   True  True   True
2  False   True  True   True
```

Show which entries in a Series are not NA.

```
>>> ser = pd.Series([5, 6, np.nan])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.notna()
0     True
1     True
2    False
dtype: bool
```

AlloViz.AlloViz.Elements.Nodes.nsmallest

`Nodes.`**`nsmallest`**(*n: int*, *columns: IndexLabel*, *keep: NsmallestNlargestKeep = 'first'*) → DataFrame

Return the first *n* rows ordered by *columns* in ascending order.

Return the first *n* rows with the smallest values in *columns*, in ascending order. The columns that are not specified are returned as well, but not used for ordering.

This method is equivalent to `df.sort_values(columns, ascending=True).head(n)`, but more performant.

Parameters

n
: [int] Number of items to retrieve.

columns
: [list or str] Column name or names to order by.

keep
: [{'first', 'last', 'all'}, default 'first'] Where there are duplicate values:

- `first` : take the first occurrence.
- `last` : take the last occurrence.
- `all` : do not drop any duplicates, even it means selecting more than *n* items.

Returns

DataFrame

See also:

`DataFrame.nlargest`
Return the first *n* rows ordered by *columns* in descending order.

`DataFrame.sort_values`
Sort DataFrame by the values.

`DataFrame.head`
Return the first *n* rows without re-ordering.

Examples

```
>>> df = pd.DataFrame({'population': [59000000, 65000000, 434000,
...                                   434000, 434000, 337000, 337000,
...                                   11300, 11300],
...                    'GDP': [1937894, 2583560 , 12011, 4520, 12128,
...                            17036, 182, 38, 311],
...                    'alpha-2': ["IT", "FR", "MT", "MV", "BN",
...                                "IS", "NR", "TV", "AI"]},
...                   index=["Italy", "France", "Malta",
...                          "Maldives", "Brunei", "Iceland",
...                          "Nauru", "Tuvalu", "Anguilla"])
>>> df
          population      GDP alpha-2
Italy       59000000  1937894      IT
France      65000000  2583560      FR
Malta         434000    12011      MT
Maldives      434000     4520      MV
Brunei        434000    12128      BN
Iceland       337000    17036      IS
Nauru         337000      182      NR
Tuvalu         11300       38      TV
Anguilla       11300      311      AI
```

In the following example, we will use `nsmallest` to select the three rows having the smallest values in column "population".

```
>>> df.nsmallest(3, 'population')
          population    GDP alpha-2
Tuvalu         11300     38      TV
Anguilla       11300    311      AI
Iceland       337000  17036      IS
```

When using `keep='last'`, ties are resolved in reverse order:

```
>>> df.nsmallest(3, 'population', keep='last')
          population  GDP alpha-2
Anguilla       11300  311      AI
Tuvalu         11300   38      TV
Nauru         337000  182      NR
```

When using `keep='all'`, all duplicate items are maintained:

```
>>> df.nsmallest(3, 'population', keep='all')
          population    GDP alpha-2
Tuvalu         11300     38      TV
Anguilla       11300    311      AI
Iceland       337000  17036      IS
Nauru         337000    182      NR
```

To order by the smallest values in column "population" and then "GDP", we can specify multiple columns like in the next example.

```
>>> df.nsmallest(3, ['population', 'GDP'])
          population  GDP alpha-2
Tuvalu         11300   38      TV
Anguilla       11300  311      AI
Nauru         337000  182      NR
```

AlloViz.AlloViz.Elements.Nodes.nunique

Nodes.**nunique**(*axis: Axis = 0*, *dropna: bool = True*) → Series

Count number of distinct elements in specified axis.

Return Series with number of distinct elements. Can ignore NaN values.

Parameters

axis
[{0 or 'index', 1 or 'columns'}, default 0] The axis to use. 0 or 'index' for row-wise, 1 or 'columns' for column-wise.

dropna
[bool, default True] Don't include NaN in the counts.

Returns

Series

See also:

Series.nunique
Method nunique for Series.

DataFrame.count
Count non-NA cells for each column or row.

Examples

```
>>> df = pd.DataFrame({'A': [4, 5, 6], 'B': [4, 1, 1]})
>>> df.nunique()
A    3
B    2
dtype: int64
```

```
>>> df.nunique(axis=1)
0    1
1    2
2    2
dtype: int64
```

AlloViz.AlloViz.Elements.Nodes.pad

Nodes.**pad**(*, *axis: None | Axis = None*, *inplace: bool_t = False*, *limit: None | int = None*, *downcast: dict | None | lib.NoDefault = _NoDefault.no_default*) → Self | None

Fill NA/NaN values by propagating the last valid observation to next valid.

Deprecated since version 2.0: Series/DataFrame.pad is deprecated. Use Series/DataFrame.ffill instead.

Returns

Series/DataFrame or None
Object with missing values filled or None if `inplace=True`.

Examples

Please see examples for `DataFrame.ffill()` or `Series.ffill()`.

AlloViz.AlloViz.Elements.Nodes.pct_change

Nodes.**pct_change**(*periods: int = 1*, *fill_method: Literal['backfill', 'bfill', 'ffill', 'pad'] | None | Literal[_NoDefault.no_default] = _NoDefault.no_default*, *limit: int | None | Literal[_NoDefault.no_default] = _NoDefault.no_default*, *freq=None*, ***kwargs*) → None

Fractional change between the current and a prior element.

Computes the fractional change from the immediately previous row by default. This is useful in comparing the fraction of change in a time series of elements.

Note: Despite the name of this method, it calculates fractional change (also known as per unit change or relative change) and not percentage change. If you need the percentage change, multiply these values by 100.

Parameters

periods
[int, default 1] Periods to shift for forming percent change.

fill_method
[{'backfill', 'bfill', 'pad', 'ffill', None}, default 'pad'] How to handle NAs **before** computing percent changes.

Deprecated since version 2.1.

limit
[int, default None] The number of consecutive NAs to fill before stopping.

Deprecated since version 2.1.

freq
[DateOffset, timedelta, or str, optional] Increment to use from time series API (e.g. 'M' or BDay()).

****kwargs**
Additional keyword arguments are passed into *DataFrame.shift* or *Series.shift*.

Returns

Series or DataFrame
The same type as the calling object.

See also:

`Series.diff`
Compute the difference of two elements in a Series.

`DataFrame.diff`
Compute the difference of two elements in a DataFrame.

`Series.shift`
Shift the index by some number of periods.

`DataFrame.shift`
Shift the index by some number of periods.

Examples

Series

```
>>> s = pd.Series([90, 91, 85])
>>> s
0    90
1    91
2    85
dtype: int64
```

```
>>> s.pct_change()
0         NaN
1    0.011111
2   -0.065934
dtype: float64
```

```
>>> s.pct_change(periods=2)
0         NaN
1         NaN
2   -0.055556
dtype: float64
```

See the percentage change in a Series where filling NAs with last valid observation forward to next valid.

```
>>> s = pd.Series([90, 91, None, 85])
>>> s
0    90.0
1    91.0
```

(continues on next page)

(continued from previous page)

```
2     NaN
3    85.0
dtype: float64
```

```
>>> s.ffill().pct_change()
0         NaN
1    0.011111
2    0.000000
3   -0.065934
dtype: float64
```

DataFrame

Percentage change in French franc, Deutsche Mark, and Italian lira from 1980-01-01 to 1980-03-01.

```
>>> df = pd.DataFrame({
...     'FR': [4.0405, 4.0963, 4.3149],
...     'GR': [1.7246, 1.7482, 1.8519],
...     'IT': [804.74, 810.01, 860.13]},
...     index=['1980-01-01', '1980-02-01', '1980-03-01'])
>>> df
                FR      GR      IT
1980-01-01  4.0405  1.7246  804.74
1980-02-01  4.0963  1.7482  810.01
1980-03-01  4.3149  1.8519  860.13
```

```
>>> df.pct_change()
                  FR        GR        IT
1980-01-01       NaN       NaN       NaN
1980-02-01  0.013810  0.013684  0.006549
1980-03-01  0.053365  0.059318  0.061876
```

Percentage of change in GOOG and APPL stock volume. Shows computing the percentage change between columns.

```
>>> df = pd.DataFrame({
...     '2016': [1769950, 30586265],
...     '2015': [1500923, 40912316],
...     '2014': [1371819, 41403351]},
...     index=['GOOG', 'APPL'])
>>> df
          2016      2015      2014
GOOG   1769950   1500923   1371819
APPL  30586265  40912316  41403351
```

```
>>> df.pct_change(axis='columns', periods=-1)
          2016      2015  2014
GOOG  0.179241  0.094112   NaN
APPL -0.252395 -0.011860   NaN
```

AlloViz.AlloViz.Elements.Nodes.pipe

`Nodes.pipe`(*func: Callable[[...], T] | tuple[Callable[[...], T], str]*, **args*, ***kwargs*) → T

Apply chainable functions that expect Series or DataFrames.

Parameters

func
[function] Function to apply to the Series/DataFrame. `args`, and `kwargs` are passed into `func`. Alternatively a `(callable, data_keyword)` tuple where `data_keyword` is a string indicating the keyword of `callable` that expects the Series/DataFrame.

***args**
[iterable, optional] Positional arguments passed into `func`.

****kwargs**
[mapping, optional] A dictionary of keyword arguments passed into `func`.

Returns

the return type of `func`.

See also:

`DataFrame.apply`
Apply a function along input axis of DataFrame.

`DataFrame.map`
Apply a function elementwise on a whole DataFrame.

`Series.map`
Apply a mapping correspondence on a `Series`.

Notes

Use `.pipe` when chaining together functions that expect Series, DataFrames or GroupBy objects.

Examples

Constructing a income DataFrame from a dictionary.

```
>>> data = [[8000, 1000], [9500, np.nan], [5000, 2000]]
>>> df = pd.DataFrame(data, columns=['Salary', 'Others'])
>>> df
   Salary  Others
0    8000  1000.0
1    9500     NaN
2    5000  2000.0
```

Functions that perform tax reductions on an income DataFrame.

```
>>> def subtract_federal_tax(df):
...     return df * 0.9
>>> def subtract_state_tax(df, rate):
...     return df * (1 - rate)
```

(continues on next page)

(continued from previous page)

```
>>> def subtract_national_insurance(df, rate, rate_increase):
...     new_rate = rate + rate_increase
...     return df * (1 - new_rate)
```

Instead of writing

```
>>> subtract_national_insurance(
...     subtract_state_tax(subtract_federal_tax(df), rate=0.12),
...     rate=0.05,
...     rate_increase=0.02)
```

You can write

```
>>> (
...     df.pipe(subtract_federal_tax)
...     .pipe(subtract_state_tax, rate=0.12)
...     .pipe(subtract_national_insurance, rate=0.05, rate_increase=0.02)
... )
    Salary   Others
0  5892.48   736.56
1  6997.32      NaN
2  3682.80  1473.12
```

If you have a function that takes the data as (say) the second argument, pass a tuple indicating which keyword expects the data. For example, suppose `national_insurance` takes its data as `df` in the second argument:

```
>>> def subtract_national_insurance(rate, df, rate_increase):
...     new_rate = rate + rate_increase
...     return df * (1 - new_rate)
>>> (
...     df.pipe(subtract_federal_tax)
...     .pipe(subtract_state_tax, rate=0.12)
...     .pipe(
...         (subtract_national_insurance, 'df'),
...         rate=0.05,
...         rate_increase=0.02
...     )
... )
    Salary   Others
0  5892.48   736.56
1  6997.32      NaN
2  3682.80  1473.12
```

AlloViz.AlloViz.Elements.Nodes.pivot

Nodes.**pivot**(*, *columns*, *index=_NoDefault.no_default*, *values=_NoDefault.no_default*) → DataFrame

Return reshaped DataFrame organized by given index / column values.

Reshape data (produce a "pivot" table) based on column values. Uses unique values from specified *index* / *columns* to form axes of the resulting DataFrame. This function does not support data aggregation, multiple values will result in a MultiIndex in the columns. See the User Guide for more on reshaping.

Parameters

columns
[str or object or a list of str] Column to use to make new frame's columns.

index
[str or object or a list of str, optional] Column to use to make new frame's index. If not given, uses existing index.

values
[str, object or a list of the previous, optional] Column(s) to use for populating new frame's values. If not specified, all remaining columns will be used and the result will have hierarchically indexed columns.

Returns

DataFrame
Returns reshaped DataFrame.

Raises

ValueError:
When there are any *index*, *columns* combinations with multiple values. *DataFrame.pivot_table* when you need to aggregate.

See also:

`DataFrame.pivot_table`
Generalization of pivot that can handle duplicate values for one index/column pair.

`DataFrame.unstack`
Pivot based on the index values instead of a column.

`wide_to_long`
Wide panel to long format. Less flexible but more user-friendly than melt.

Notes

For finer-tuned control, see hierarchical indexing documentation along with the related stack/unstack methods.

Reference the user guide for more examples.

Examples

```
>>> df = pd.DataFrame({'foo': ['one', 'one', 'one', 'two', 'two',
...                            'two'],
...                    'bar': ['A', 'B', 'C', 'A', 'B', 'C'],
...                    'baz': [1, 2, 3, 4, 5, 6],
...                    'zoo': ['x', 'y', 'z', 'q', 'w', 't']})
>>> df
    foo   bar  baz  zoo
0   one   A    1    x
1   one   B    2    y
2   one   C    3    z
3   two   A    4    q
4   two   B    5    w
5   two   C    6    t
```

```
>>> df.pivot(index='foo', columns='bar', values='baz')
bar  A   B   C
foo
one  1   2   3
two  4   5   6
```

```
>>> df.pivot(index='foo', columns='bar')['baz']
bar  A   B   C
foo
one  1   2   3
two  4   5   6
```

```
>>> df.pivot(index='foo', columns='bar', values=['baz', 'zoo'])
      baz       zoo
bar   A  B  C   A  B  C
foo
one   1  2  3   x  y  z
two   4  5  6   q  w  t
```

You could also assign a list of column names or a list of index names.

```
>>> df = pd.DataFrame({
...        "lev1": [1, 1, 1, 2, 2, 2],
...        "lev2": [1, 1, 2, 1, 1, 2],
...        "lev3": [1, 2, 1, 2, 1, 2],
...        "lev4": [1, 2, 3, 4, 5, 6],
...        "values": [0, 1, 2, 3, 4, 5]})
>>> df
    lev1 lev2 lev3 lev4 values
0   1    1    1    1    0
1   1    1    2    2    1
2   1    2    1    3    2
3   2    1    2    4    3
4   2    1    1    5    4
5   2    2    2    6    5
```

```
>>> df.pivot(index="lev1", columns=["lev2", "lev3"], values="values")
lev2    1         2
lev3    1    2    1    2
lev1
1     0.0  1.0  2.0  NaN
2     4.0  3.0  NaN  5.0
```

```
>>> df.pivot(index=["lev1", "lev2"], columns=["lev3"], values="values")
      lev3    1    2
lev1  lev2
   1     1  0.0  1.0
         2  2.0  NaN
   2     1  4.0  3.0
         2  NaN  5.0
```

A ValueError is raised if there are any duplicates.

```
>>> df = pd.DataFrame({"foo": ['one', 'one', 'two', 'two'],
...                    "bar": ['A', 'A', 'B', 'C'],
...                    "baz": [1, 2, 3, 4]})
>>> df
   foo bar  baz
0  one   A    1
1  one   A    2
2  two   B    3
3  two   C    4
```

Notice that the first two rows are the same for our *index* and *columns* arguments.

```
>>> df.pivot(index='foo', columns='bar', values='baz')
Traceback (most recent call last):
   ...
ValueError: Index contains duplicate entries, cannot reshape
```

AlloViz.AlloViz.Elements.Nodes.pivot_table

Nodes.**pivot_table**(*values=None*, *index=None*, *columns=None*, *aggfunc: AggFuncType = 'mean'*, *fill_value=None*, *margins: bool = False*, *dropna: bool = True*, *margins_name: Level = 'All'*, *observed: bool = False*, *sort: bool = True*) → DataFrame

Create a spreadsheet-style pivot table as a DataFrame.

The levels in the pivot table will be stored in MultiIndex objects (hierarchical indexes) on the index and columns of the result DataFrame.

Parameters

values
[list-like or scalar, optional] Column or columns to aggregate.

index
[column, Grouper, array, or list of the previous] Keys to group by on the pivot table index. If a list is passed, it can contain any of the other types (except list). If an array is passed, it must be the same length as the data and will be used in the same manner as column values.

columns
[column, Grouper, array, or list of the previous] Keys to group by on the pivot table column. If a list is passed, it can contain any of the other types (except list). If an array is passed, it must be the same length as the data and will be used in the same manner as column values.

aggfunc
[function, list of functions, dict, default "mean"] If a list of functions is passed, the resulting pivot table will have hierarchical columns whose top level are the function names (inferred from the function objects themselves). If a dict is passed, the key is column to aggregate and the value is function or list of functions. If `margin=True`, aggfunc will be used to calculate the partial aggregates.

fill_value
[scalar, default None] Value to replace missing values with (in the resulting pivot table, after aggregation).

margins
[bool, default False] If `margins=True`, special `All` columns and rows will be added with partial group aggregates across the categories on the rows and columns.

dropna
[bool, default True] Do not include columns whose entries are all NaN. If True, rows with a NaN value in any column will be omitted before computing margins.

margins_name
[str, default 'All'] Name of the row / column that will contain the totals when margins is True.

observed
[bool, default False] This only applies if any of the groupers are Categoricals. If True: only show observed values for categorical groupers. If False: show all values for categorical groupers.

sort
[bool, default True] Specifies if the result should be sorted.

New in version 1.3.0.

Returns

DataFrame
An Excel style pivot table.

See also:

`DataFrame.pivot`
Pivot without aggregation that can handle non-numeric data.

`DataFrame.melt`
Unpivot a DataFrame from wide to long format, optionally leaving identifiers set.

`wide_to_long`
Wide panel to long format. Less flexible but more user-friendly than melt.

Notes

Reference the user guide for more examples.

Examples

```
>>> df = pd.DataFrame({"A": ["foo", "foo", "foo", "foo", "foo",
...                          "bar", "bar", "bar", "bar"],
...                    "B": ["one", "one", "one", "two", "two",
...                          "one", "one", "two", "two"],
...                    "C": ["small", "large", "large", "small",
...                          "small", "large", "small", "small",
...                          "large"],
...                    "D": [1, 2, 2, 3, 3, 4, 5, 6, 7],
...                    "E": [2, 4, 5, 5, 6, 6, 8, 9, 9]})
>>> df
     A    B      C  D  E
0  foo  one  small  1  2
1  foo  one  large  2  4
2  foo  one  large  2  5
3  foo  two  small  3  5
4  foo  two  small  3  6
5  bar  one  large  4  6
6  bar  one  small  5  8
7  bar  two  small  6  9
8  bar  two  large  7  9
```

This first example aggregates values by taking the sum.

```
>>> table = pd.pivot_table(df, values='D', index=['A', 'B'],
...                        columns=['C'], aggfunc="sum")
>>> table
C        large  small
A   B
bar one    4.0    5.0
    two    7.0    6.0
foo one    4.0    1.0
    two    NaN    6.0
```

We can also fill missing values using the *fill_value* parameter.

```
>>> table = pd.pivot_table(df, values='D', index=['A', 'B'],
...                        columns=['C'], aggfunc="sum", fill_value=0)
>>> table
C        large  small
A   B
bar one      4      5
    two      7      6
foo one      4      1
    two      0      6
```

The next example aggregates by taking the mean across multiple columns.

```
>>> table = pd.pivot_table(df, values=['D', 'E'], index=['A', 'C'],
...                        aggfunc={'D': "mean", 'E': "mean"})
>>> table
                D         E
A   C
bar large  5.500000  7.500000
    small  5.500000  8.500000
foo large  2.000000  4.500000
    small  2.333333  4.333333
```

We can also calculate multiple types of aggregations for any given value column.

```
>>> table = pd.pivot_table(df, values=['D', 'E'], index=['A', 'C'],
...                        aggfunc={'D': "mean",
...                                 'E': ["min", "max", "mean"]})
>>> table
                  D   E
               mean max      mean  min
A   C
bar large  5.500000   9  7.500000    6
    small  5.500000   9  8.500000    8
foo large  2.000000   5  4.500000    4
    small  2.333333   6  4.333333    2
```

AlloViz.AlloViz.Elements.Nodes.pop

`Nodes.pop`(*item: Hashable*) → Series

Return item and drop from frame. Raise KeyError if not found.

Parameters

item
[label] Label of column to be popped.

Returns

Series

Examples

```
>>> df = pd.DataFrame([('falcon', 'bird', 389.0),
...                    ('parrot', 'bird', 24.0),
...                    ('lion', 'mammal', 80.5),
...                    ('monkey', 'mammal', np.nan)],
...                   columns=('name', 'class', 'max_speed'))
>>> df
     name   class  max_speed
0  falcon    bird      389.0
1  parrot    bird       24.0
2    lion  mammal       80.5
3  monkey  mammal        NaN
```

```
>>> df.pop('class')
0      bird
1      bird
2    mammal
3    mammal
Name: class, dtype: object
```

```
>>> df
     name  max_speed
0  falcon      389.0
1  parrot       24.0
2    lion       80.5
3  monkey        NaN
```

AlloViz.AlloViz.Elements.Nodes.pow

Nodes.**pow**(*other*, *axis: Axis = 'columns'*, *level=None*, *fill_value=None*)

Get Exponential power of dataframe and other, element-wise (binary operator *pow*).

Equivalent to `dataframe ** other`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *rpow*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *floordiv*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other
: [scalar, sequence, Series, dict or DataFrame] Any single or multiple element data structure, or list-like object.

axis
: [{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns. (1 or 'columns'). For Series input, axis to match Series index on.

level
: [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value
: [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame
: Result of the arithmetic operation.

See also:

`DataFrame.add`
: Add DataFrames.

`DataFrame.sub`
: Subtract DataFrames.

DataFrame.mul
Multiply DataFrames.

DataFrame.div
Divide DataFrames (float division).

DataFrame.truediv
Divide DataFrames (float division).

DataFrame.floordiv
Divide DataFrames (integer division).

DataFrame.mod
Calculate modulo (remainder after division).

DataFrame.pow
Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a dictionary by axis.

```
>>> df.mul({'angles': 0, 'degrees': 2})
            angles  degrees
circle           0      720
triangle         0      360
rectangle        0      720
```

```
>>> df.mul({'circle': 0, 'triangle': 2, 'rectangle': 3}, axis='index')
            angles  degrees
circle           0        0
triangle         6      360
rectangle       12     1080
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

AlloViz.AlloViz.Elements.Nodes.prod

`Nodes.`**`prod`**(*axis: Axis | None = 0*, *skipna: bool = True*, *numeric_only: bool = False*, *min_count: int = 0*, ***kwargs*)

Return the product of the values over the requested axis.

Parameters

axis

[{index (0), columns (1)}] Axis for the function to be applied on. For *Series* this parameter is unused and defaults to 0.

For DataFrames, specifying `axis=None` will apply the aggregation across both axes.

New in version 2.0.0.

skipna
[bool, default True] Exclude NA/null values when computing the result.

numeric_only
[bool, default False] Include only float, int, boolean columns. Not implemented for Series.

min_count
[int, default 0] The required number of valid values to perform the operation. If fewer than `min_count` non-NA values are present the result will be NA.

****kwargs**
Additional keyword arguments to be passed to the function.

Returns

Series or scalar

See also:

`Series.sum`
Return the sum.

`Series.min`
Return the minimum.

`Series.max`
Return the maximum.

`Series.idxmin`
Return the index of the minimum.

`Series.idxmax`
Return the index of the maximum.

`DataFrame.sum`
Return the sum over the requested axis.

`DataFrame.min`
Return the minimum over the requested axis.

`DataFrame.max`
Return the maximum over the requested axis.

`DataFrame.idxmin`
Return the index of the minimum over the requested axis.

`DataFrame.idxmax`
Return the index of the maximum over the requested axis.

Examples

By default, the product of an empty or all-NA Series is 1

```
>>> pd.Series([], dtype="float64").prod()
1.0
```

This can be controlled with the `min_count` parameter

```
>>> pd.Series([], dtype="float64").prod(min_count=1)
nan
```

Thanks to the `skipna` parameter, `min_count` handles all-NA and empty series identically.

```
>>> pd.Series([np.nan]).prod()
1.0
```

```
>>> pd.Series([np.nan]).prod(min_count=1)
nan
```

AlloViz.AlloViz.Elements.Nodes.product

`Nodes.`**`product`**(*axis: Axis | None = 0*, *skipna: bool = True*, *numeric_only: bool = False*, *min_count: int = 0*, ***kwargs*)

Return the product of the values over the requested axis.

Parameters

axis
[{index (0), columns (1)}] Axis for the function to be applied on. For *Series* this parameter is unused and defaults to 0.

For DataFrames, specifying `axis=None` will apply the aggregation across both axes.

New in version 2.0.0.

skipna
[bool, default True] Exclude NA/null values when computing the result.

numeric_only
[bool, default False] Include only float, int, boolean columns. Not implemented for Series.

min_count
[int, default 0] The required number of valid values to perform the operation. If fewer than `min_count` non-NA values are present the result will be NA.

****kwargs**
Additional keyword arguments to be passed to the function.

Returns

Series or scalar

See also:

`Series.sum`
Return the sum.

`Series.min`
Return the minimum.

`Series.max`
Return the maximum.

`Series.idxmin`
Return the index of the minimum.

`Series.idxmax`
Return the index of the maximum.

DataFrame.sum
Return the sum over the requested axis.

DataFrame.min
Return the minimum over the requested axis.

DataFrame.max
Return the maximum over the requested axis.

DataFrame.idxmin
Return the index of the minimum over the requested axis.

DataFrame.idxmax
Return the index of the maximum over the requested axis.

Examples

By default, the product of an empty or all-NA Series is 1

```
>>> pd.Series([], dtype="float64").prod()
1.0
```

This can be controlled with the `min_count` parameter

```
>>> pd.Series([], dtype="float64").prod(min_count=1)
nan
```

Thanks to the `skipna` parameter, `min_count` handles all-NA and empty series identically.

```
>>> pd.Series([np.nan]).prod()
1.0
```

```
>>> pd.Series([np.nan]).prod(min_count=1)
nan
```

AlloViz.AlloViz.Elements.Nodes.quantile

`Nodes.quantile`(*q: float | AnyArrayLike | Sequence[float] = 0.5*, *axis: Axis = 0*, *numeric_only: bool = False*, *interpolation: QuantileInterpolation = 'linear'*, *method: Literal['single', 'table'] = 'single'*) → Series | DataFrame

Return values at the given quantile over requested axis.

Parameters

q
[float or array-like, default 0.5 (50% quantile)] Value between 0 <= q <= 1, the quantile(s) to compute.

axis
[{0 or 'index', 1 or 'columns'}, default 0] Equals 0 or 'index' for row-wise, 1 or 'columns' for column-wise.

numeric_only
[bool, default False] Include only *float*, *int* or *boolean* data.

Changed in version 2.0.0: The default value of `numeric_only` is now `False`.

interpolation
[{'linear', 'lower', 'higher', 'midpoint', 'nearest'}] This optional parameter specifies the interpolation method to use, when the desired quantile lies between two data points *i* and *j*:

- linear: *i* + *(j - i) * fraction*, where *fraction* is the fractional part of the index surrounded by *i* and *j*.
- lower: *i*.
- higher: *j*.
- nearest: *i* or *j* whichever is nearest.
- midpoint: (*i* + *j*) / 2.

method
[{'single', 'table'}, default 'single'] Whether to compute quantiles per-column ('single') or over all columns ('table'). When 'table', the only allowed interpolation methods are 'nearest', 'lower', and 'higher'.

Returns

Series or DataFrame

If q is an array, a DataFrame will be returned where the
index is q, the columns are the columns of self, and the values are the quantiles.

If q is a float, a Series will be returned where the
index is the columns of self and the values are the quantiles.

See also:

`core.window.rolling.Rolling.quantile`
Rolling quantile.

`numpy.percentile`
Numpy function to compute the percentile.

Examples

```
>>> df = pd.DataFrame(np.array([[1, 1], [2, 10], [3, 100], [4, 100]]),
...                   columns=['a', 'b'])
>>> df.quantile(.1)
a    1.3
b    3.7
Name: 0.1, dtype: float64
>>> df.quantile([.1, .5])
       a     b
0.1  1.3   3.7
0.5  2.5  55.0
```

Specifying *method='table'* will compute the quantile over all columns.

```
>>> df.quantile(.1, method="table", interpolation="nearest")
a    1
b    1
Name: 0.1, dtype: int64
```

(continues on next page)

(continued from previous page)

```
>>> df.quantile([.1, .5], method="table", interpolation="nearest")
     a    b
0.1  1    1
0.5  3  100
```

Specifying *numeric_only=False* will also compute the quantile of datetime and timedelta data.

```
>>> df = pd.DataFrame({'A': [1, 2],
...                    'B': [pd.Timestamp('2010'),
...                          pd.Timestamp('2011')],
...                    'C': [pd.Timedelta('1 days'),
...                          pd.Timedelta('2 days')]})
>>> df.quantile(0.5, numeric_only=False)
A                    1.5
B    2010-07-02 12:00:00
C        1 days 12:00:00
Name: 0.5, dtype: object
```

AlloViz.AlloViz.Elements.Nodes.query

`Nodes.query`(*expr: str*, *, *inplace: bool = False*, ***kwargs*) → DataFrame | None

Query the columns of a DataFrame with a boolean expression.

Parameters

expr
[str] The query string to evaluate.

You can refer to variables in the environment by prefixing them with an '@' character like `@a + b`.

You can refer to column names that are not valid Python variable names by surrounding them in backticks. Thus, column names containing spaces or punctuations (besides underscores) or starting with digits must be surrounded by backticks. (For example, a column named "Area (cm^2)" would be referenced as `` `Area (cm^2)` ``). Column names which are Python keywords (like "list", "for", "import", etc) cannot be used.

For example, if one of your columns is called `a a` and you want to sum it with `b`, your query should be `` `a a` + b ``.

inplace
[bool] Whether to modify the DataFrame rather than creating a new one.

****kwargs**
See the documentation for *eval()* for complete details on the keyword arguments accepted by `DataFrame.query()`.

Returns

DataFrame or None
DataFrame resulting from the provided query expression or None if `inplace=True`.

See also:

eval
: Evaluate a string describing operations on DataFrame columns.

DataFrame.eval
: Evaluate a string describing operations on DataFrame columns.

Notes

The result of the evaluation of this expression is first passed to `DataFrame.loc` and if that fails because of a multidimensional key (e.g., a DataFrame) then the result will be passed to `DataFrame.__getitem__()`.

This method uses the top-level *eval()* function to evaluate the passed query.

The `query()` method uses a slightly modified Python syntax by default. For example, the `&` and `|` (bitwise) operators have the precedence of their boolean cousins, `and` and `or`. This *is* syntactically valid Python, however the semantics are different.

You can change the semantics of the expression by passing the keyword argument `parser='python'`. This enforces the same semantics as evaluation in Python space. Likewise, you can pass `engine='python'` to evaluate an expression using Python itself as a backend. This is not recommended as it is inefficient compared to using `numexpr` as the engine.

The `DataFrame.index` and `DataFrame.columns` attributes of the `DataFrame` instance are placed in the query namespace by default, which allows you to treat both the index and columns of the frame as a column in the frame. The identifier `index` is used for the frame index; you can also use the name of the index to identify it in a query. Please note that Python keywords may not be used as identifiers.

For further details and examples see the `query` documentation in indexing.

Backtick quoted variables

Backtick quoted variables are parsed as literal Python code and are converted internally to a Python valid identifier. This can lead to the following problems.

During parsing a number of disallowed characters inside the backtick quoted string are replaced by strings that are allowed as a Python identifier. These characters include all operators in Python, the space character, the question mark, the exclamation mark, the dollar sign, and the euro sign. For other characters that fall outside the ASCII range (U+0001..U+007F) and those that are not further specified in PEP 3131, the query parser will raise an error. This excludes whitespace different than the space character, but also the hashtag (as it is used for comments) and the backtick itself (backtick can also not be escaped).

In a special case, quotes that make a pair around a backtick can confuse the parser. For example, `` `it's` > `that's` `` will raise an error, as it forms a quoted string (`'s > `that'`) with a backtick inside.

See also the Python documentation about lexical analysis (https://docs.python.org/3/reference/lexical_analysis.html) in combination with the source code in `pandas.core.computation.parsing`.

Examples

```
>>> df = pd.DataFrame({'A': range(1, 6),
...                    'B': range(10, 0, -2),
...                    'C C': range(10, 5, -1)})
>>> df
   A   B  C C
0  1  10   10
1  2   8    9
2  3   6    8
```

(continues on next page)

(continued from previous page)

```
3  4   4    7
4  5   2    6
>>> df.query('A > B')
   A  B  C C
4  5  2    6
```

The previous expression is equivalent to

```
>>> df[df.A > df.B]
   A  B  C C
4  5  2    6
```

For columns with spaces in their name, you can use backtick quoting.

```
>>> df.query('B == `C C`')
   A   B  C C
0  1  10   10
```

The previous expression is equivalent to

```
>>> df[df.B == df['C C']]
   A   B  C C
0  1  10   10
```

AlloViz.AlloViz.Elements.Nodes.radd

`Nodes.`**`radd`**(*other*, *axis: Axis = 'columns'*, *level=None*, *fill_value=None*)

Get Addition of dataframe and other, element-wise (binary operator *radd*).

Equivalent to `other + dataframe`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *add*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *floordiv*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other
: [scalar, sequence, Series, dict or DataFrame] Any single or multiple element data structure, or list-like object.

axis
: [{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns. (1 or 'columns'). For Series input, axis to match Series index on.

level
: [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value
: [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame
Result of the arithmetic operation.

See also:

DataFrame.add
Add DataFrames.

DataFrame.sub
Subtract DataFrames.

DataFrame.mul
Multiply DataFrames.

DataFrame.div
Divide DataFrames (float division).

DataFrame.truediv
Divide DataFrames (float division).

DataFrame.floordiv
Divide DataFrames (integer division).

DataFrame.mod
Calculate modulo (remainder after division).

DataFrame.pow
Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
```

(continues on next page)

(continued from previous page)

```
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a dictionary by axis.

```
>>> df.mul({'angles': 0, 'degrees': 2})
           angles  degrees
circle          0      720
triangle        0      360
rectangle       0      720
```

```
>>> df.mul({'circle': 0, 'triangle': 2, 'rectangle': 3}, axis='index')
           angles  degrees
circle          0        0
triangle        6      360
rectangle      12     1080
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

AlloViz.AlloViz.Elements.Nodes.rank

Nodes.**rank**(*axis: int | Literal['index', 'columns', 'rows'] = 0*, *method: Literal['average', 'min', 'max', 'first', 'dense'] = 'average'*, *numeric_only: bool = False*, *na_option: Literal['keep', 'top', 'bottom'] = 'keep'*, *ascending: bool = True*, *pct: bool = False*) → None

Compute numerical data ranks (1 through n) along axis.

By default, equal values are assigned a rank that is the average of the ranks of those values.

Parameters

axis
[{0 or 'index', 1 or 'columns'}, default 0] Index to direct ranking. For *Series* this parameter is unused and defaults to 0.

method
[{'average', 'min', 'max', 'first', 'dense'}, default 'average'] How to rank the group of records that have the same value (i.e. ties):

- average: average rank of the group
- min: lowest rank in the group
- max: highest rank in the group
- first: ranks assigned in order they appear in the array
- dense: like 'min', but rank always increases by 1 between groups.

numeric_only
[bool, default False] For DataFrame objects, rank only numeric columns if set to True.

Changed in version 2.0.0: The default value of `numeric_only` is now `False`.

na_option
[{'keep', 'top', 'bottom'}, default 'keep'] How to rank NaN values:

- keep: assign NaN rank to NaN values
- top: assign lowest rank to NaN values
- bottom: assign highest rank to NaN values

ascending
[bool, default True] Whether or not the elements should be ranked in ascending order.

pct
[bool, default False] Whether or not to display the returned rankings in percentile form.

Returns

same type as caller
Return a Series or DataFrame with data ranks as values.

See also:

`core.groupby.DataFrameGroupBy.rank`
Rank of values within each group.

`core.groupby.SeriesGroupBy.rank`
Rank of values within each group.

Examples

```
>>> df = pd.DataFrame(data={'Animal': ['cat', 'penguin', 'dog',
...                                    'spider', 'snake'],
...                         'Number_legs': [4, 2, 4, 8, np.nan]})
>>> df
    Animal  Number_legs
0      cat          4.0
1  penguin          2.0
2      dog          4.0
3   spider          8.0
4    snake          NaN
```

Ties are assigned the mean of the ranks (by default) for the group.

```
>>> s = pd.Series(range(5), index=list("abcde"))
>>> s["d"] = s["b"]
>>> s.rank()
a    1.0
b    2.5
c    4.0
d    2.5
e    5.0
dtype: float64
```

The following example shows how the method behaves with the above parameters:

- default_rank: this is the default behaviour obtained without using any parameter.
- max_rank: setting `method = 'max'` the records that have the same values are ranked using the highest rank (e.g.: since 'cat' and 'dog' are both in the 2nd and 3rd position, rank 3 is assigned.)
- NA_bottom: choosing `na_option = 'bottom'`, if there are records with NaN values they are placed at the bottom of the ranking.
- pct_rank: when setting `pct = True`, the ranking is expressed as percentile rank.

```
>>> df['default_rank'] = df['Number_legs'].rank()
>>> df['max_rank'] = df['Number_legs'].rank(method='max')
>>> df['NA_bottom'] = df['Number_legs'].rank(na_option='bottom')
>>> df['pct_rank'] = df['Number_legs'].rank(pct=True)
>>> df
    Animal  Number_legs  default_rank  max_rank  NA_bottom  pct_rank
0      cat          4.0           2.5       3.0        2.5     0.625
1  penguin          2.0           1.0       1.0        1.0     0.250
2      dog          4.0           2.5       3.0        2.5     0.625
3   spider          8.0           4.0       4.0        4.0     1.000
4    snake          NaN           NaN       NaN        5.0       NaN
```

AlloViz.AlloViz.Elements.Nodes.rdiv

Nodes.**rdiv**(*other*, *axis: Axis = 'columns'*, *level=None*, *fill_value=None*)

Get Floating division of dataframe and other, element-wise (binary operator *rtruediv*).

Equivalent to `other / dataframe`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *truediv*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *floordiv*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other
[scalar, sequence, Series, dict or DataFrame] Any single or multiple element data structure, or list-like object.

axis
[{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns. (1 or 'columns'). For Series input, axis to match Series index on.

level
[int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value
[float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame
Result of the arithmetic operation.

See also:

`DataFrame.add`
Add DataFrames.

`DataFrame.sub`
Subtract DataFrames.

`DataFrame.mul`
Multiply DataFrames.

`DataFrame.div`
Divide DataFrames (float division).

`DataFrame.truediv`
Divide DataFrames (float division).

`DataFrame.floordiv`
Divide DataFrames (integer division).

`DataFrame.mod`
Calculate modulo (remainder after division).

`DataFrame.pow`
Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...         axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a dictionary by axis.

```
>>> df.mul({'angles': 0, 'degrees': 2})
            angles  degrees
circle           0      720
triangle         0      360
rectangle        0      720
```

```
>>> df.mul({'circle': 0, 'triangle': 2, 'rectangle': 3}, axis='index')
            angles  degrees
circle           0        0
triangle         6      360
rectangle       12     1080
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

AlloViz.AlloViz.Elements.Nodes.reindex

Nodes.**reindex**(*labels=None*, ***, *index=None*, *columns=None*, *axis: Axis | None = None*, *method: ReindexMethod | None = None*, *copy: bool | None = None*, *level: Level | None = None*, *fill_value: Scalar | None = nan*, *limit: int | None = None*, *tolerance=None*) → DataFrame

Conform DataFrame to new index with optional filling logic.

Places NA/NaN in locations having no value in the previous index. A new object is produced unless the new index is equivalent to the current one and `copy=False`.

Parameters

labels
[array-like, optional] New labels / index to conform the axis specified by 'axis' to.

index
[array-like, optional] New labels for the index. Preferably an Index object to avoid duplicating data.

columns
[array-like, optional] New labels for the columns. Preferably an Index object to avoid duplicating data.

axis
[int or str, optional] Axis to target. Can be either the axis name ('index', 'columns') or number (0, 1).

method
[{None, 'backfill'/'bfill', 'pad'/'ffill', 'nearest'}] Method to use for filling holes in reindexed DataFrame. Please note: this is only applicable to DataFrames/Series with a monotonically increasing/decreasing index.

- None (default): don't fill gaps
- pad / ffill: Propagate last valid observation forward to next valid.
- backfill / bfill: Use next valid observation to fill gap.
- nearest: Use nearest valid observations to fill gap.

copy
[bool, default True] Return a new object, even if the passed indexes are the same.

level
[int or name] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value
[scalar, default np.nan] Value to use for missing values. Defaults to NaN, but can be any "compatible" value.

limit
[int, default None] Maximum number of consecutive elements to forward or backward fill.

tolerance
[optional] Maximum distance between original and new labels for inexact matches. The values of the index at the matching locations most satisfy the equation `abs(index[indexer] - target) <= tolerance`.

Tolerance may be a scalar value, which applies the same tolerance to all values, or list-like, which applies variable tolerance per element. List-like includes list, tuple, array, Series, and must be the same size as the index and its dtype must exactly match the index's type.

Returns

DataFrame with changed index.

See also:

`DataFrame.set_index`
Set row labels.

`DataFrame.reset_index`
Remove row labels or move them to new columns.

`DataFrame.reindex_like`
Change to same indices as other DataFrame.

Examples

`DataFrame.reindex` supports two calling conventions

- `(index=index_labels, columns=column_labels, ...)`
- `(labels, axis={'index', 'columns'}, ...)`

We *highly* recommend using keyword arguments to clarify your intent.

Create a dataframe with some fictional data.

```
>>> index = ['Firefox', 'Chrome', 'Safari', 'IE10', 'Konqueror']
>>> df = pd.DataFrame({'http_status': [200, 200, 404, 404, 301],
...                    'response_time': [0.04, 0.02, 0.07, 0.08, 1.0]},
...                    index=index)
>>> df
           http_status  response_time
Firefox            200           0.04
Chrome             200           0.02
Safari             404           0.07
IE10               404           0.08
Konqueror          301           1.00
```

Create a new index and reindex the dataframe. By default values in the new index that do not have corresponding records in the dataframe are assigned `NaN`.

```
>>> new_index = ['Safari', 'Iceweasel', 'Comodo Dragon', 'IE10',
...              'Chrome']
>>> df.reindex(new_index)
               http_status  response_time
Safari               404.0           0.07
Iceweasel              NaN            NaN
Comodo Dragon          NaN            NaN
IE10                 404.0           0.08
Chrome               200.0           0.02
```

We can fill in the missing values by passing a value to the keyword `fill_value`. Because the index is not monotonically increasing or decreasing, we cannot use arguments to the keyword `method` to fill the `NaN` values.

```
>>> df.reindex(new_index, fill_value=0)
               http_status  response_time
Safari                 404           0.07
Iceweasel                0           0.00
Comodo Dragon            0           0.00
IE10                   404           0.08
Chrome                 200           0.02
```

```
>>> df.reindex(new_index, fill_value='missing')
              http_status response_time
Safari                404          0.07
Iceweasel         missing       missing
Comodo Dragon     missing       missing
IE10                  404          0.08
Chrome                200          0.02
```

We can also reindex the columns.

```
>>> df.reindex(columns=['http_status', 'user_agent'])
           http_status  user_agent
Firefox            200         NaN
Chrome             200         NaN
Safari             404         NaN
IE10               404         NaN
```

(continues on next page)

(continued from previous page)

```
Konqueror          301         NaN
```

Or we can use "axis-style" keyword arguments

```
>>> df.reindex(['http_status', 'user_agent'], axis="columns")
           http_status  user_agent
Firefox            200         NaN
Chrome             200         NaN
Safari             404         NaN
IE10               404         NaN
Konqueror          301         NaN
```

To further illustrate the filling functionality in `reindex`, we will create a dataframe with a monotonically increasing index (for example, a sequence of dates).

```
>>> date_index = pd.date_range('1/1/2010', periods=6, freq='D')
>>> df2 = pd.DataFrame({"prices": [100, 101, np.nan, 100, 89, 88]},
...                    index=date_index)
>>> df2
            prices
2010-01-01   100.0
2010-01-02   101.0
2010-01-03     NaN
2010-01-04   100.0
2010-01-05    89.0
2010-01-06    88.0
```

Suppose we decide to expand the dataframe to cover a wider date range.

```
>>> date_index2 = pd.date_range('12/29/2009', periods=10, freq='D')
>>> df2.reindex(date_index2)
            prices
2009-12-29     NaN
2009-12-30     NaN
2009-12-31     NaN
2010-01-01   100.0
2010-01-02   101.0
2010-01-03     NaN
2010-01-04   100.0
2010-01-05    89.0
2010-01-06    88.0
2010-01-07     NaN
```

The index entries that did not have a value in the original data frame (for example, '2009-12-29') are by default filled with `NaN`. If desired, we can fill in the missing values using one of several options.

For example, to back-propagate the last valid value to fill the `NaN` values, pass `bfill` as an argument to the `method` keyword.

```
>>> df2.reindex(date_index2, method='bfill')
            prices
2009-12-29   100.0
2009-12-30   100.0
```

(continues on next page)

(continued from previous page)

```
2009-12-31   100.0
2010-01-01   100.0
2010-01-02   101.0
2010-01-03     NaN
2010-01-04   100.0
2010-01-05    89.0
2010-01-06    88.0
2010-01-07     NaN
```

Please note that the `NaN` value present in the original dataframe (at index value 2010-01-03) will not be filled by any of the value propagation schemes. This is because filling while reindexing does not look at dataframe values, but only compares the original and desired indexes. If you do want to fill in the `NaN` values present in the original dataframe, use the `fillna()` method.

See the user guide for more.

AlloViz.AlloViz.Elements.Nodes.reindex_like

`Nodes.`**`reindex_like`**(*other*, *method: Literal['backfill', 'bfill', 'pad', 'ffill', 'nearest'] | None = None*, *copy: bool | None = None*, *limit: None | int = None*, *tolerance=None*) → None

Return an object with matching indices as other object.

Conform the object to the same index on all axes. Optional filling logic, placing NaN in locations having no value in the previous index. A new object is produced unless the new index is equivalent to the current one and copy=False.

Parameters

other
: [Object of the same data type] Its row and column indices are used to define the new indices of this object.

method
: [{None, 'backfill'/'bfill', 'pad'/'ffill', 'nearest'}] Method to use for filling holes in reindexed DataFrame. Please note: this is only applicable to DataFrames/Series with a monotonically increasing/decreasing index.

- None (default): don't fill gaps
- pad / ffill: propagate last valid observation forward to next valid
- backfill / bfill: use next valid observation to fill gap
- nearest: use nearest valid observations to fill gap.

copy
: [bool, default True] Return a new object, even if the passed indexes are the same.

limit
: [int, default None] Maximum number of consecutive labels to fill for inexact matches.

tolerance
: [optional] Maximum distance between original and new labels for inexact matches. The values of the index at the matching locations must satisfy the equation `abs(index[indexer] - target) <= tolerance`.

Tolerance may be a scalar value, which applies the same tolerance to all values, or list-like, which applies variable tolerance per element. List-like includes list, tuple, array, Series, and must be the same size as the index and its dtype must exactly match the index's type.

Returns

Series or DataFrame
Same type as caller, but with changed indices on each axis.

See also:

DataFrame.set_index
Set row labels.

DataFrame.reset_index
Remove row labels or move them to new columns.

DataFrame.reindex
Change to new indices or expand indices.

Notes

Same as calling `.reindex(index=other.index, columns=other.columns,...)`.

Examples

```
>>> df1 = pd.DataFrame([[24.3, 75.7, 'high'],
...                     [31, 87.8, 'high'],
...                     [22, 71.6, 'medium'],
...                     [35, 95, 'medium']],
...                    columns=['temp_celsius', 'temp_fahrenheit',
...                             'windspeed'],
...                    index=pd.date_range(start='2014-02-12',
...                                        end='2014-02-15', freq='D'))
```

```
>>> df1
            temp_celsius  temp_fahrenheit windspeed
2014-02-12          24.3             75.7      high
2014-02-13          31.0             87.8      high
2014-02-14          22.0             71.6    medium
2014-02-15          35.0             95.0    medium
```

```
>>> df2 = pd.DataFrame([[28, 'low'],
...                     [30, 'low'],
...                     [35.1, 'medium']],
...                    columns=['temp_celsius', 'windspeed'],
...                    index=pd.DatetimeIndex(['2014-02-12', '2014-02-13',
...                                            '2014-02-15']))
```

```
>>> df2
            temp_celsius windspeed
2014-02-12          28.0       low
```

(continues on next page)

(continued from previous page)

```
2014-02-13          30.0       low
2014-02-15          35.1    medium
```

```
>>> df2.reindex_like(df1)
            temp_celsius  temp_fahrenheit windspeed
2014-02-12          28.0              NaN       low
2014-02-13          30.0              NaN       low
2014-02-14           NaN              NaN       NaN
2014-02-15          35.1              NaN    medium
```

AlloViz.AlloViz.Elements.Nodes.rename

Nodes.**rename**(*mapper: Renamer | None = None*, ***, *index: Renamer | None = None*, *columns: Renamer | None = None*, *axis: Axis | None = None*, *copy: bool | None = None*, *inplace: bool = False*, *level: Level | None = None*, *errors: IgnoreRaise = 'ignore'*) → DataFrame | None

Rename columns or index labels.

Function / dict values must be unique (1-to-1). Labels not contained in a dict / Series will be left as-is. Extra labels listed don't throw an error.

See the user guide for more.

Parameters

mapper
[dict-like or function] Dict-like or function transformations to apply to that axis' values. Use either `mapper` and `axis` to specify the axis to target with `mapper`, or `index` and `columns`.

index
[dict-like or function] Alternative to specifying axis (`mapper, axis=0` is equivalent to `index=mapper`).

columns
[dict-like or function] Alternative to specifying axis (`mapper, axis=1` is equivalent to `columns=mapper`).

axis
[{0 or 'index', 1 or 'columns'}, default 0] Axis to target with `mapper`. Can be either the axis name ('index', 'columns') or number (0, 1). The default is 'index'.

copy
[bool, default True] Also copy underlying data.

inplace
[bool, default False] Whether to modify the DataFrame rather than creating a new one. If True then value of copy is ignored.

level
[int or level name, default None] In case of a MultiIndex, only rename labels in the specified level.

errors
[{'ignore', 'raise'}, default 'ignore'] If 'raise', raise a *KeyError* when a dict-like *mapper*, *index*, or *columns* contains labels that are not present in the Index being

transformed. If 'ignore', existing keys will be renamed and extra keys will be ignored.

Returns

DataFrame or None
DataFrame with the renamed axis labels or None if `inplace=True`.

Raises

KeyError
If any of the labels is not found in the selected axis and "errors='raise'".

See also:

`DataFrame.rename_axis`
Set the name of the axis.

Examples

`DataFrame.rename` supports two calling conventions

- `(index=index_mapper, columns=columns_mapper, ...)`
- `(mapper, axis={'index', 'columns'}, ...)`

We *highly* recommend using keyword arguments to clarify your intent.

Rename columns using a mapping:

```
>>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
>>> df.rename(columns={"A": "a", "B": "c"})
   a  c
0  1  4
1  2  5
2  3  6
```

Rename index using a mapping:

```
>>> df.rename(index={0: "x", 1: "y", 2: "z"})
   A  B
x  1  4
y  2  5
z  3  6
```

Cast index labels to a different type:

```
>>> df.index
RangeIndex(start=0, stop=3, step=1)
>>> df.rename(index=str).index
Index(['0', '1', '2'], dtype='object')
```

```
>>> df.rename(columns={"A": "a", "B": "b", "C": "c"}, errors="raise")
Traceback (most recent call last):
KeyError: ['C'] not found in axis
```

Using axis-style parameters:

```
>>> df.rename(str.lower, axis='columns')
   a  b
0  1  4
1  2  5
2  3  6
```

```
>>> df.rename({1: 2, 2: 4}, axis='index')
   A  B
0  1  4
2  2  5
4  3  6
```

AlloViz.AlloViz.Elements.Nodes.rename_axis

Nodes.**rename_axis**(*mapper: IndexLabel | lib.NoDefault = _NoDefault.no_default*, ***, *index=_NoDefault.no_default*, *columns=_NoDefault.no_default*, *axis: Axis = 0*, *copy: bool_t | None = None*, *inplace: bool_t = False*) → Self | None

Set the name of the axis for the index or columns.

Parameters

mapper
[scalar, list-like, optional] Value to set the axis name attribute.

index, columns
[scalar, list-like, dict-like or function, optional] A scalar, list-like, dict-like or functions transformations to apply to that axis' values. Note that the `columns` parameter is not allowed if the object is a Series. This parameter only apply for DataFrame type objects.

Use either `mapper` and `axis` to specify the axis to target with `mapper`, or `index` and/or `columns`.

axis
[{0 or 'index', 1 or 'columns'}, default 0] The axis to rename. For *Series* this parameter is unused and defaults to 0.

copy
[bool, default None] Also copy underlying data.

inplace
[bool, default False] Modifies the object directly, instead of creating a new Series or DataFrame.

Returns

Series, DataFrame, or None
The same type as the caller or None if `inplace=True`.

See also:

`Series.rename`
Alter Series index labels or name.

`DataFrame.rename`
Alter DataFrame index labels or name.

Index.rename
Set new names on index.

Notes

`DataFrame.rename_axis` supports two calling conventions

- `(index=index_mapper, columns=columns_mapper, ...)`
- `(mapper, axis={'index', 'columns'}, ...)`

The first calling convention will only modify the names of the index and/or the names of the Index object that is the columns. In this case, the parameter `copy` is ignored.

The second calling convention will modify the names of the corresponding index if mapper is a list or a scalar. However, if mapper is dict-like or a function, it will use the deprecated behavior of modifying the axis *labels*.

We *highly* recommend using keyword arguments to clarify your intent.

Examples

Series

```
>>> s = pd.Series(["dog", "cat", "monkey"])
>>> s
0       dog
1       cat
2    monkey
dtype: object
>>> s.rename_axis("animal")
animal
0    dog
1    cat
2    monkey
dtype: object
```

DataFrame

```
>>> df = pd.DataFrame({"num_legs": [4, 4, 2],
...                    "num_arms": [0, 0, 2]},
...                   ["dog", "cat", "monkey"])
>>> df
        num_legs  num_arms
dog            4         0
cat            4         0
monkey         2         2
>>> df = df.rename_axis("animal")
>>> df
        num_legs  num_arms
animal
dog            4         0
cat            4         0
monkey         2         2
>>> df = df.rename_axis("limbs", axis="columns")
```

(continues on next page)

(continued from previous page)

```
>>> df
limbs   num_legs  num_arms
animal
dog            4         0
cat            4         0
monkey         2         2
```

MultiIndex

```
>>> df.index = pd.MultiIndex.from_product([['mammal'],
...                                        ['dog', 'cat', 'monkey']],
...                                       names=['type', 'name'])
>>> df
limbs          num_legs  num_arms
type   name
mammal dog            4         0
       cat            4         0
       monkey         2         2
```

```
>>> df.rename_axis(index={'type': 'class'})
limbs          num_legs  num_arms
class  name
mammal dog            4         0
       cat            4         0
       monkey         2         2
```

```
>>> df.rename_axis(columns=str.upper)
LIMBS          num_legs  num_arms
type   name
mammal dog            4         0
       cat            4         0
       monkey         2         2
```

AlloViz.AlloViz.Elements.Nodes.reorder_levels

Nodes.**reorder_levels**(*order: Sequence[int | str]*, *axis: Axis = 0*) → DataFrame

Rearrange index levels using input order. May not drop or duplicate levels.

Parameters

order
[list of int or list of str] List representing new level order. Reference level by number (position) or by key (label).

axis
[{0 or 'index', 1 or 'columns'}, default 0] Where to reorder levels.

Returns

DataFrame

Examples

```
>>> data = {
...     "class": ["Mammals", "Mammals", "Reptiles"],
...     "diet": ["Omnivore", "Carnivore", "Carnivore"],
...     "species": ["Humans", "Dogs", "Snakes"],
... }
>>> df = pd.DataFrame(data, columns=["class", "diet", "species"])
>>> df = df.set_index(["class", "diet"])
>>> df
                                  species
class      diet
Mammals    Omnivore                Humans
           Carnivore                 Dogs
Reptiles   Carnivore               Snakes
```

Let's reorder the levels of the index:

```
>>> df.reorder_levels(["diet", "class"])
                                  species
diet      class
Omnivore  Mammals                 Humans
Carnivore Mammals                   Dogs
          Reptiles                Snakes
```

AlloViz.AlloViz.Elements.Nodes.replace

Nodes.**replace**(*to_replace=None*, *value=_NoDefault.no_default*, ***, *inplace: bool_t = False*, *limit: int | None = None*, *regex: bool_t = False*, *method: Literal['pad', 'ffill', 'bfill'] | lib.NoDefault = _NoDefault.no_default*) → Self | None

Replace values given in *to_replace* with *value*.

Values of the Series/DataFrame are replaced with other values dynamically. This differs from updating with `.loc` or `.iloc`, which require you to specify a location to update with some value.

Parameters

to_replace
[str, regex, list, dict, Series, int, float, or None] How to find the values that will be replaced.

- numeric, str or regex:
  - numeric: numeric values equal to *to_replace* will be replaced with *value*
  - str: string exactly matching *to_replace* will be replaced with *value*
  - regex: regexs matching *to_replace* will be replaced with *value*
- list of str, regex, or numeric:
  - First, if *to_replace* and *value* are both lists, they **must** be the same length.
  - Second, if `regex=True` then all of the strings in **both** lists will be interpreted as regexs otherwise they will match directly. This doesn't matter much for *value* since there are only a few possible substitution regexes you can use.
  - str, regex and numeric rules apply as above.

- dict:
  - Dicts can be used to specify different replacement values for different existing values. For example, `{'a': 'b', 'y': 'z'}` replaces the value 'a' with 'b' and 'y' with 'z'. To use a dict in this way, the optional *value* parameter should not be given.
  - For a DataFrame a dict can specify that different values should be replaced in different columns. For example, `{'a': 1, 'b': 'z'}` looks for the value 1 in column 'a' and the value 'z' in column 'b' and replaces these values with whatever is specified in *value*. The *value* parameter should not be `None` in this case. You can treat this as a special case of passing two lists except that you are specifying the column to search in.
  - For a DataFrame nested dictionaries, e.g., `{'a': {'b': np.nan}}`, are read as follows: look in column 'a' for the value 'b' and replace it with NaN. The optional *value* parameter should not be specified to use a nested dict in this way. You can nest regular expressions as well. Note that column names (the top-level dictionary keys in a nested dictionary) **cannot** be regular expressions.
- None:
  - This means that the *regex* argument must be a string, compiled regular expression, or list, dict, ndarray or Series of such elements. If *value* is also `None` then this **must** be a nested dictionary or Series.

See the examples section for examples of each of these.

value
: [scalar, dict, list, str, regex, default None] Value to replace any values matching *to_replace* with. For a DataFrame a dict of values can be used to specify which value to use for each column (columns not in the dict will not be filled). Regular expressions, strings and lists or dicts of such objects are also allowed.

inplace
: [bool, default False] If True, performs operation inplace and returns None.

limit
: [int, default None] Maximum size gap to forward or backward fill.

  Deprecated since version 2.1.0.

regex
: [bool or same types as *to_replace*, default False] Whether to interpret *to_replace* and/or *value* as regular expressions. If this is `True` then *to_replace must* be a string. Alternatively, this could be a regular expression or a list, dict, or array of regular expressions in which case *to_replace* must be `None`.

method
: [{'pad', 'ffill', 'bfill'}] The method to use when for replacement, when *to_replace* is a scalar, list or tuple and *value* is `None`.

  Deprecated since version 2.1.0.

Returns

Series/DataFrame
: Object after replacement.

Raises

AssertionError

- If *regex* is not a `bool` and *to_replace* is not `None`.

TypeError

- If *to_replace* is not a scalar, array-like, `dict`, or `None`
- If *to_replace* is a `dict` and *value* is not a `list`, `dict`, `ndarray`, or `Series`
- If *to_replace* is `None` and *regex* is not compilable into a regular expression or is a list, dict, ndarray, or Series.
- When replacing multiple `bool` or `datetime64` objects and the arguments to *to_replace* does not match the type of the value being replaced

ValueError

- If a `list` or an `ndarray` is passed to *to_replace* and *value* but they are not the same length.

See also:

`Series.fillna`
Fill NA values.

`DataFrame.fillna`
Fill NA values.

`Series.where`
Replace values based on boolean condition.

`DataFrame.where`
Replace values based on boolean condition.

`DataFrame.map`
Apply a function to a Dataframe elementwise.

`Series.map`
Map values of Series according to an input mapping or function.

`Series.str.replace`
Simple string replacement.

Notes

- Regex substitution is performed under the hood with `re.sub`. The rules for substitution for `re.sub` are the same.
- Regular expressions will only substitute on strings, meaning you cannot provide, for example, a regular expression matching floating point numbers and expect the columns in your frame that have a numeric dtype to be matched. However, if those floating point numbers *are* strings, then you can do this.
- This method has *a lot* of options. You are encouraged to experiment and play with this method to gain intuition about how it works.
- When dict is used as the *to_replace* value, it is like key(s) in the dict are the to_replace part and value(s) in the dict are the value parameter.

Examples

Scalar `to_replace` and `value`

```
>>> s = pd.Series([1, 2, 3, 4, 5])
>>> s.replace(1, 5)
0    5
1    2
2    3
3    4
4    5
dtype: int64
```

```
>>> df = pd.DataFrame({'A': [0, 1, 2, 3, 4],
...                    'B': [5, 6, 7, 8, 9],
...                    'C': ['a', 'b', 'c', 'd', 'e']})
>>> df.replace(0, 5)
    A  B  C
0  5  5  a
1  1  6  b
2  2  7  c
3  3  8  d
4  4  9  e
```

List-like `to_replace`

```
>>> df.replace([0, 1, 2, 3], 4)
    A  B  C
0  4  5  a
1  4  6  b
2  4  7  c
3  4  8  d
4  4  9  e
```

```
>>> df.replace([0, 1, 2, 3], [4, 3, 2, 1])
    A  B  C
0  4  5  a
1  3  6  b
2  2  7  c
3  1  8  d
4  4  9  e
```

```
>>> s.replace([1, 2], method='bfill')
0    3
1    3
2    3
3    4
4    5
dtype: int64
```

dict-like `to_replace`

```
>>> df.replace({0: 10, 1: 100})
        A  B  C
0   10  5  a
1  100  6  b
2    2  7  c
3    3  8  d
4    4  9  e
```

```
>>> df.replace({'A': 0, 'B': 5}, 100)
        A    B  C
0  100  100  a
1    1    6  b
2    2    7  c
3    3    8  d
4    4    9  e
```

```
>>> df.replace({'A': {0: 100, 4: 400}})
        A  B  C
0  100  5  a
1    1  6  b
2    2  7  c
3    3  8  d
4  400  9  e
```

Regular expression `to_replace`

```
>>> df = pd.DataFrame({'A': ['bat', 'foo', 'bait'],
...                    'B': ['abc', 'bar', 'xyz']})
>>> df.replace(to_replace=r'^ba.$', value='new', regex=True)
        A    B
0   new  abc
1   foo  new
2  bait  xyz
```

```
>>> df.replace({'A': r'^ba.$'}, {'A': 'new'}, regex=True)
        A    B
0   new  abc
1   foo  bar
2  bait  xyz
```

```
>>> df.replace(regex=r'^ba.$', value='new')
        A    B
0   new  abc
1   foo  new
2  bait  xyz
```

```
>>> df.replace(regex={r'^ba.$': 'new', 'foo': 'xyz'})
        A    B
0   new  abc
1   xyz  new
2  bait  xyz
```

```
>>> df.replace(regex=[r'^ba.$', 'foo'], value='new')
        A    B
0   new  abc
1   new  new
2  bait  xyz
```

Compare the behavior of `s.replace({'a': None})` and `s.replace('a', None)` to understand the peculiarities of the *to_replace* parameter:

```
>>> s = pd.Series([10, 'a', 'a', 'b', 'a'])
```

When one uses a dict as the *to_replace* value, it is like the value(s) in the dict are equal to the *value* parameter. `s.replace({'a': None})` is equivalent to `s.replace(to_replace={'a': None}, value=None, method=None)`:

```
>>> s.replace({'a': None})
0      10
1    None
2    None
3       b
4    None
dtype: object
```

When `value` is not explicitly passed and *to_replace* is a scalar, list or tuple, *replace* uses the method parameter (default 'pad') to do the replacement. So this is why the 'a' values are being replaced by 10 in rows 1 and 2 and 'b' in row 4 in this case.

```
>>> s.replace('a')
0    10
1    10
2    10
3     b
4     b
dtype: object
```

Deprecated since version 2.1.0: The 'method' parameter and padding behavior are deprecated.

On the other hand, if `None` is explicitly passed for `value`, it will be respected:

```
>>> s.replace('a', None)
0      10
1    None
2    None
3       b
4    None
dtype: object
```

Changed in version 1.4.0: Previously the explicit `None` was silently ignored.

AlloViz.AlloViz.Elements.Nodes.resample

Nodes.**resample**(*rule*, *axis: Axis | lib.NoDefault = _NoDefault.no_default*, *closed: Literal['right', 'left'] | None = None*, *label: Literal['right', 'left'] | None = None*, *convention: Literal['start', 'end', 's', 'e'] = 'start'*, *kind: Literal['timestamp', 'period'] | None = None*, *on: Level | None = None*, *level: Level | None = None*, *origin: str | TimestampConvertibleTypes = 'start_day'*, *offset: TimedeltaConvertibleTypes | None = None*, *group_keys: bool_t = False*) → Resampler

Resample time-series data.

Convenience method for frequency conversion and resampling of time series. The object must have a datetime-like index (*DatetimeIndex*, *PeriodIndex*, or *TimedeltaIndex*), or the caller must pass the label of a datetime-like series/index to the `on`/`level` keyword parameter.

Parameters

rule
[DateOffset, Timedelta or str] The offset string or object representing target conversion.

axis
[{0 or 'index', 1 or 'columns'}, default 0] Which axis to use for up- or down-sampling. For *Series* this parameter is unused and defaults to 0. Must be *DatetimeIndex*, *TimedeltaIndex* or *PeriodIndex*.

Deprecated since version 2.0.0: Use frame.T.resample(...) instead.

closed
[{'right', 'left'}, default None] Which side of bin interval is closed. The default is 'left' for all frequency offsets except for 'M', 'A', 'Q', 'BM', 'BA', 'BQ', and 'W' which all have a default of 'right'.

label
[{'right', 'left'}, default None] Which bin edge label to label bucket with. The default is 'left' for all frequency offsets except for 'M', 'A', 'Q', 'BM', 'BA', 'BQ', and 'W' which all have a default of 'right'.

convention
[{'start', 'end', 's', 'e'}, default 'start'] For *PeriodIndex* only, controls whether to use the start or end of *rule*.

kind
[{'timestamp', 'period'}, optional, default None] Pass 'timestamp' to convert the resulting index to a *DateTimeIndex* or 'period' to convert it to a *PeriodIndex*. By default the input representation is retained.

on
[str, optional] For a DataFrame, column to use instead of index for resampling. Column must be datetime-like.

level
[str or int, optional] For a MultiIndex, level (name or number) to use for resampling. *level* must be datetime-like.

origin
[Timestamp or str, default 'start_day'] The timestamp on which to adjust the grouping. The timezone of origin must match the timezone of the index. If string, must be one of the following:

- 'epoch': *origin* is 1970-01-01

- 'start': *origin* is the first value of the timeseries
- 'start_day': *origin* is the first day at midnight of the timeseries
- 'end': *origin* is the last value of the timeseries
- 'end_day': *origin* is the ceiling midnight of the last day

New in version 1.3.0.

offset
[Timedelta or str, default is None] An offset timedelta added to the origin.

group_keys
[bool, default False] Whether to include the group keys in the result index when using `.apply()` on the resampled object.

New in version 1.5.0: Not specifying `group_keys` will retain values-dependent behavior from pandas 1.4 and earlier (see pandas 1.5.0 Release notes for examples).

Changed in version 2.0.0: `group_keys` now defaults to `False`.

Returns

pandas.api.typing.Resampler
`Resampler` object.

See also:

`Series.resample`
Resample a Series.

`DataFrame.resample`
Resample a DataFrame.

groupby
Group Series/DataFrame by mapping, function, label, or list of labels.

asfreq
Reindex a Series/DataFrame with the given frequency without grouping.

Notes

See the user guide for more.

To learn more about the offset strings, please see this link.

Examples

Start by creating a series with 9 one minute timestamps.

```
>>> index = pd.date_range('1/1/2000', periods=9, freq='T')
>>> series = pd.Series(range(9), index=index)
>>> series
2000-01-01 00:00:00    0
2000-01-01 00:01:00    1
2000-01-01 00:02:00    2
2000-01-01 00:03:00    3
2000-01-01 00:04:00    4
```

(continues on next page)

(continued from previous page)

```
2000-01-01 00:05:00    5
2000-01-01 00:06:00    6
2000-01-01 00:07:00    7
2000-01-01 00:08:00    8
Freq: T, dtype: int64
```

Downsample the series into 3 minute bins and sum the values of the timestamps falling into a bin.

```
>>> series.resample('3T').sum()
2000-01-01 00:00:00     3
2000-01-01 00:03:00    12
2000-01-01 00:06:00    21
Freq: 3T, dtype: int64
```

Downsample the series into 3 minute bins as above, but label each bin using the right edge instead of the left. Please note that the value in the bucket used as the label is not included in the bucket, which it labels. For example, in the original series the bucket `2000-01-01 00:03:00` contains the value 3, but the summed value in the resampled bucket with the label `2000-01-01 00:03:00` does not include 3 (if it did, the summed value would be 6, not 3). To include this value close the right side of the bin interval as illustrated in the example below this one.

```
>>> series.resample('3T', label='right').sum()
2000-01-01 00:03:00     3
2000-01-01 00:06:00    12
2000-01-01 00:09:00    21
Freq: 3T, dtype: int64
```

Downsample the series into 3 minute bins as above, but close the right side of the bin interval.

```
>>> series.resample('3T', label='right', closed='right').sum()
2000-01-01 00:00:00     0
2000-01-01 00:03:00     6
2000-01-01 00:06:00    15
2000-01-01 00:09:00    15
Freq: 3T, dtype: int64
```

Upsample the series into 30 second bins.

```
>>> series.resample('30S').asfreq()[0:5]   # Select first 5 rows
2000-01-01 00:00:00   0.0
2000-01-01 00:00:30   NaN
2000-01-01 00:01:00   1.0
2000-01-01 00:01:30   NaN
2000-01-01 00:02:00   2.0
Freq: 30S, dtype: float64
```

Upsample the series into 30 second bins and fill the `NaN` values using the `ffill` method.

```
>>> series.resample('30S').ffill()[0:5]
2000-01-01 00:00:00    0
2000-01-01 00:00:30    0
2000-01-01 00:01:00    1
2000-01-01 00:01:30    1
```

(continues on next page)

(continued from previous page)

```
2000-01-01 00:02:00    2
Freq: 30S, dtype: int64
```

Upsample the series into 30 second bins and fill the `NaN` values using the `bfill` method.

```
>>> series.resample('30S').bfill()[0:5]
2000-01-01 00:00:00    0
2000-01-01 00:00:30    1
2000-01-01 00:01:00    1
2000-01-01 00:01:30    2
2000-01-01 00:02:00    2
Freq: 30S, dtype: int64
```

Pass a custom function via `apply`

```
>>> def custom_resampler(arraylike):
...     return np.sum(arraylike) + 5
...
>>> series.resample('3T').apply(custom_resampler)
2000-01-01 00:00:00     8
2000-01-01 00:03:00    17
2000-01-01 00:06:00    26
Freq: 3T, dtype: int64
```

For a Series with a PeriodIndex, the keyword *convention* can be used to control whether to use the start or end of *rule*.

Resample a year by quarter using 'start' *convention*. Values are assigned to the first quarter of the period.

```
>>> s = pd.Series([1, 2], index=pd.period_range('2012-01-01',
...                                             freq='A',
...                                             periods=2))
>>> s
2012    1
2013    2
Freq: A-DEC, dtype: int64
>>> s.resample('Q', convention='start').asfreq()
2012Q1    1.0
2012Q2    NaN
2012Q3    NaN
2012Q4    NaN
2013Q1    2.0
2013Q2    NaN
2013Q3    NaN
2013Q4    NaN
Freq: Q-DEC, dtype: float64
```

Resample quarters by month using 'end' *convention*. Values are assigned to the last month of the period.

```
>>> q = pd.Series([1, 2, 3, 4], index=pd.period_range('2018-01-01',
...                                                   freq='Q',
...                                                   periods=4))
>>> q
```

(continues on next page)

(continued from previous page)

```
2018Q1    1
2018Q2    2
2018Q3    3
2018Q4    4
Freq: Q-DEC, dtype: int64
>>> q.resample('M', convention='end').asfreq()
2018-03    1.0
2018-04    NaN
2018-05    NaN
2018-06    2.0
2018-07    NaN
2018-08    NaN
2018-09    3.0
2018-10    NaN
2018-11    NaN
2018-12    4.0
Freq: M, dtype: float64
```

For DataFrame objects, the keyword *on* can be used to specify the column instead of the index for resampling.

```
>>> d = {'price': [10, 11, 9, 13, 14, 18, 17, 19],
...      'volume': [50, 60, 40, 100, 50, 100, 40, 50]}
>>> df = pd.DataFrame(d)
>>> df['week_starting'] = pd.date_range('01/01/2018',
...                                     periods=8,
...                                     freq='W')
>>> df
   price  volume week_starting
0     10      50    2018-01-07
1     11      60    2018-01-14
2      9      40    2018-01-21
3     13     100    2018-01-28
4     14      50    2018-02-04
5     18     100    2018-02-11
6     17      40    2018-02-18
7     19      50    2018-02-25
>>> df.resample('M', on='week_starting').mean()
               price  volume
week_starting
2018-01-31     10.75    62.5
2018-02-28     17.00    60.0
```

For a DataFrame with MultiIndex, the keyword *level* can be used to specify on which level the resampling needs to take place.

```
>>> days = pd.date_range('1/1/2000', periods=4, freq='D')
>>> d2 = {'price': [10, 11, 9, 13, 14, 18, 17, 19],
...       'volume': [50, 60, 40, 100, 50, 100, 40, 50]}
>>> df2 = pd.DataFrame(
...     d2,
...     index=pd.MultiIndex.from_product(
```

(continues on next page)

(continued from previous page)

```
...         [days, ['morning', 'afternoon']]
...     )
... )
>>> df2
                     price  volume
2000-01-01 morning      10      50
           afternoon    11      60
2000-01-02 morning       9      40
           afternoon    13     100
2000-01-03 morning      14      50
           afternoon    18     100
2000-01-04 morning      17      40
           afternoon    19      50
>>> df2.resample('D', level=0).sum()
            price  volume
2000-01-01     21     110
2000-01-02     22     140
2000-01-03     32     150
2000-01-04     36      90
```

If you want to adjust the start of the bins based on a fixed timestamp:

```
>>> start, end = '2000-10-01 23:30:00', '2000-10-02 00:30:00'
>>> rng = pd.date_range(start, end, freq='7min')
>>> ts = pd.Series(np.arange(len(rng)) * 3, index=rng)
>>> ts
2000-10-01 23:30:00     0
2000-10-01 23:37:00     3
2000-10-01 23:44:00     6
2000-10-01 23:51:00     9
2000-10-01 23:58:00    12
2000-10-02 00:05:00    15
2000-10-02 00:12:00    18
2000-10-02 00:19:00    21
2000-10-02 00:26:00    24
Freq: 7T, dtype: int64
```

```
>>> ts.resample('17min').sum()
2000-10-01 23:14:00     0
2000-10-01 23:31:00     9
2000-10-01 23:48:00    21
2000-10-02 00:05:00    54
2000-10-02 00:22:00    24
Freq: 17T, dtype: int64
```

```
>>> ts.resample('17min', origin='epoch').sum()
2000-10-01 23:18:00     0
2000-10-01 23:35:00    18
2000-10-01 23:52:00    27
2000-10-02 00:09:00    39
2000-10-02 00:26:00    24
Freq: 17T, dtype: int64
```

```
>>> ts.resample('17W', origin='2000-01-01').sum()
2000-01-02      0
2000-04-30      0
2000-08-27      0
2000-12-24    108
Freq: 17W-SUN, dtype: int64
```

If you want to adjust the start of the bins with an *offset* Timedelta, the two following lines are equivalent:

```
>>> ts.resample('17min', origin='start').sum()
2000-10-01 23:30:00     9
2000-10-01 23:47:00    21
2000-10-02 00:04:00    54
2000-10-02 00:21:00    24
Freq: 17T, dtype: int64
```

```
>>> ts.resample('17min', offset='23h30min').sum()
2000-10-01 23:30:00     9
2000-10-01 23:47:00    21
2000-10-02 00:04:00    54
2000-10-02 00:21:00    24
Freq: 17T, dtype: int64
```

If you want to take the largest Timestamp as the end of the bins:

```
>>> ts.resample('17min', origin='end').sum()
2000-10-01 23:35:00     0
2000-10-01 23:52:00    18
2000-10-02 00:09:00    27
2000-10-02 00:26:00    63
Freq: 17T, dtype: int64
```

In contrast with the *start_day*, you can use *end_day* to take the ceiling midnight of the largest Timestamp as the end of the bins and drop the bins not containing data:

```
>>> ts.resample('17min', origin='end_day').sum()
2000-10-01 23:38:00     3
2000-10-01 23:55:00    15
2000-10-02 00:12:00    45
2000-10-02 00:29:00    45
Freq: 17T, dtype: int64
```

AlloViz.AlloViz.Elements.Nodes.reset_index

Nodes.**reset_index**(*level: IndexLabel | None = None*, ***, *drop: bool = False*, *inplace: bool = False*, *col_level: Hashable = 0*, *col_fill: Hashable = ''*, *allow_duplicates: bool | lib.NoDefault = _NoDefault.no_default*, *names: Hashable | Sequence[Hashable] | None = None*) → DataFrame | None

Reset the index, or a level of it.

Reset the index of the DataFrame, and use the default one instead. If the DataFrame has a MultiIndex, this method can remove one or more levels.

Parameters

level
[int, str, tuple, or list, default None] Only remove the given levels from the index. Removes all levels by default.

drop
[bool, default False] Do not try to insert index into dataframe columns. This resets the index to the default integer index.

inplace
[bool, default False] Whether to modify the DataFrame rather than creating a new one.

col_level
[int or str, default 0] If the columns have multiple levels, determines which level the labels are inserted into. By default it is inserted into the first level.

col_fill
[object, default ‘’] If the columns have multiple levels, determines how the other levels are named. If None then the index name is repeated.

allow_duplicates
[bool, optional, default lib.no_default] Allow duplicate column labels to be created.

New in version 1.5.0.

names
[int, str or 1-dimensional list, default None] Using the given string, rename the DataFrame column which contains the index data. If the DataFrame has a MultiIndex, this has to be a list or tuple with length equal to the number of levels.

New in version 1.5.0.

Returns

DataFrame or None
DataFrame with the new index or None if `inplace=True`.

See also:

`DataFrame.set_index`
Opposite of reset_index.

`DataFrame.reindex`
Change to new indices or expand indices.

`DataFrame.reindex_like`
Change to same indices as other DataFrame.

Examples

```
>>> df = pd.DataFrame([('bird', 389.0),
...                    ('bird', 24.0),
...                    ('mammal', 80.5),
...                    ('mammal', np.nan)],
...                   index=['falcon', 'parrot', 'lion', 'monkey'],
...                   columns=('class', 'max_speed'))
>>> df
```

(continues on next page)

(continued from previous page)

```
         class  max_speed
falcon    bird      389.0
parrot    bird       24.0
lion    mammal       80.5
monkey  mammal        NaN
```

When we reset the index, the old index is added as a column, and a new sequential index is used:

```
>>> df.reset_index()
    index   class  max_speed
0  falcon    bird      389.0
1  parrot    bird       24.0
2    lion  mammal       80.5
3  monkey  mammal        NaN
```

We can use the *drop* parameter to avoid the old index being added as a column:

```
>>> df.reset_index(drop=True)
    class  max_speed
0    bird      389.0
1    bird       24.0
2  mammal       80.5
3  mammal        NaN
```

You can also use *reset_index* with *MultiIndex*.

```
>>> index = pd.MultiIndex.from_tuples([('bird', 'falcon'),
...                                    ('bird', 'parrot'),
...                                    ('mammal', 'lion'),
...                                    ('mammal', 'monkey')],
...                                   names=['class', 'name'])
>>> columns = pd.MultiIndex.from_tuples([('speed', 'max'),
...                                      ('species', 'type')])
>>> df = pd.DataFrame([(389.0, 'fly'),
...                    (24.0, 'fly'),
...                    (80.5, 'run'),
...                    (np.nan, 'jump')],
...                   index=index,
...                   columns=columns)
>>> df
               speed species
                 max    type
class  name
bird   falcon  389.0     fly
       parrot   24.0     fly
mammal lion     80.5     run
       monkey    NaN    jump
```

Using the *names* parameter, choose a name for the index column:

```
>>> df.reset_index(names=['classes', 'names'])
  classes   names  speed species
                     max    type
```

(continues on next page)

(continued from previous page)

```
0    bird  falcon  389.0     fly
1    bird  parrot   24.0     fly
2  mammal    lion   80.5     run
3  mammal  monkey    NaN    jump
```

If the index has multiple levels, we can reset a subset of them:

```
>>> df.reset_index(level='class')
         class  speed species
                  max    type
name
falcon    bird  389.0     fly
parrot    bird   24.0     fly
lion    mammal   80.5     run
monkey  mammal    NaN    jump
```

If we are not dropping the index, by default, it is placed in the top level. We can place it in another level:

```
>>> df.reset_index(level='class', col_level=1)
                speed species
         class    max    type
name
falcon    bird  389.0     fly
parrot    bird   24.0     fly
lion    mammal   80.5     run
monkey  mammal    NaN    jump
```

When the index is inserted under another level, we can specify under which one with the parameter *col_fill*:

```
>>> df.reset_index(level='class', col_level=1, col_fill='species')
              species  speed species
                class    max    type
name
falcon           bird  389.0     fly
parrot           bird   24.0     fly
lion           mammal   80.5     run
monkey         mammal    NaN    jump
```

If we specify a nonexistent level for *col_fill*, it is created:

```
>>> df.reset_index(level='class', col_level=1, col_fill='genus')
               genus  speed species
               class    max    type
name
falcon           bird  389.0     fly
parrot           bird   24.0     fly
lion           mammal   80.5     run
monkey         mammal    NaN    jump
```

AlloViz.AlloViz.Elements.Nodes.rfloordiv

Nodes.**rfloordiv**(*other*, *axis: Axis = 'columns'*, *level=None*, *fill_value=None*)

Get Integer division of dataframe and other, element-wise (binary operator *rfloordiv*).

Equivalent to `other // dataframe`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *floordiv*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *floordiv*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other
[scalar, sequence, Series, dict or DataFrame] Any single or multiple element data structure, or list-like object.

axis
[{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns. (1 or 'columns'). For Series input, axis to match Series index on.

level
[int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value
[float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame
Result of the arithmetic operation.

See also:

DataFrame.add
Add DataFrames.

DataFrame.sub
Subtract DataFrames.

DataFrame.mul
Multiply DataFrames.

DataFrame.div
Divide DataFrames (float division).

DataFrame.truediv
Divide DataFrames (float division).

DataFrame.floordiv
Divide DataFrames (integer division).

DataFrame.mod
Calculate modulo (remainder after division).

DataFrame.pow
Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...         axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a dictionary by axis.

```
>>> df.mul({'angles': 0, 'degrees': 2})
            angles  degrees
circle           0      720
triangle         0      360
rectangle        0      720
```

```
>>> df.mul({'circle': 0, 'triangle': 2, 'rectangle': 3}, axis='index')
            angles  degrees
circle           0        0
triangle         6      360
rectangle       12     1080
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

AlloViz.AlloViz.Elements.Nodes.rmod

Nodes.**rmod**(*other*, *axis: Axis = 'columns'*, *level=None*, *fill_value=None*)

Get Modulo of dataframe and other, element-wise (binary operator *rmod*).

Equivalent to `other % dataframe`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *mod*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *floordiv*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other
: [scalar, sequence, Series, dict or DataFrame] Any single or multiple element data structure, or list-like object.

axis
: [{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns. (1 or 'columns'). For Series input, axis to match Series index on.

level
: [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value
: [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame
Result of the arithmetic operation.

See also:

DataFrame.add
Add DataFrames.

DataFrame.sub
Subtract DataFrames.

DataFrame.mul
Multiply DataFrames.

DataFrame.div
Divide DataFrames (float division).

DataFrame.truediv
Divide DataFrames (float division).

DataFrame.floordiv
Divide DataFrames (integer division).

DataFrame.mod
Calculate modulo (remainder after division).

DataFrame.pow
Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
```

(continues on next page)

(continued from previous page)

```
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a dictionary by axis.

```
>>> df.mul({'angles': 0, 'degrees': 2})
           angles  degrees
circle          0      720
triangle        0      360
rectangle       0      720
```

```
>>> df.mul({'circle': 0, 'triangle': 2, 'rectangle': 3}, axis='index')
           angles  degrees
circle          0        0
triangle        6      360
rectangle      12     1080
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

AlloViz.AlloViz.Elements.Nodes.rmul

Nodes.**rmul**(*other*, *axis: Axis = 'columns'*, *level=None*, *fill_value=None*)

Get Multiplication of dataframe and other, element-wise (binary operator *rmul*).

Equivalent to `other * dataframe`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *mul*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *floordiv*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other
[scalar, sequence, Series, dict or DataFrame] Any single or multiple element data structure, or list-like object.

axis
[{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns. (1 or 'columns'). For Series input, axis to match Series index on.

level
[int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value
[float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame
Result of the arithmetic operation.

See also:

DataFrame.add
Add DataFrames.

DataFrame.sub
Subtract DataFrames.

DataFrame.mul
Multiply DataFrames.

DataFrame.div
Divide DataFrames (float division).

DataFrame.truediv
Divide DataFrames (float division).

DataFrame.floordiv
Divide DataFrames (integer division).

DataFrame.mod
Calculate modulo (remainder after division).

DataFrame.pow
Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...         axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a dictionary by axis.

```
>>> df.mul({'angles': 0, 'degrees': 2})
            angles  degrees
circle           0      720
triangle         0      360
rectangle        0      720
```

```
>>> df.mul({'circle': 0, 'triangle': 2, 'rectangle': 3}, axis='index')
            angles  degrees
circle           0        0
triangle         6      360
rectangle       12     1080
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

AlloViz.AlloViz.Elements.Nodes.rolling

Nodes.**rolling**(*window: int | dt.timedelta | str | BaseOffset | BaseIndexer*, *min_periods: int | None = None*, *center: bool_t = False*, *win_type: str | None = None*, *on: str | None = None*, *axis: Axis | lib.NoDefault = _NoDefault.no_default*, *closed: IntervalClosedType | None = None*, *step: int | None = None*, *method: str = 'single'*) → Window | Rolling

Provide rolling window calculations.

Parameters

window

[int, timedelta, str, offset, or BaseIndexer subclass] Size of the moving window.

If an integer, the fixed number of observations used for each window.

If a timedelta, str, or offset, the time period of each window. Each window will be a variable sized based on the observations included in the time-period. This is only valid for datetimelike indexes. To learn more about the offsets & frequency strings, please see this link.

If a BaseIndexer subclass, the window boundaries based on the defined `get_window_bounds` method. Additional rolling keyword arguments, namely `min_periods`, `center`, `closed` and `step` will be passed to `get_window_bounds`.

min_periods

[int, default None] Minimum number of observations in window required to have a value; otherwise, result is `np.nan`.

For a window that is specified by an offset, `min_periods` will default to 1.

For a window that is specified by an integer, `min_periods` will default to the size of the window.

center
[bool, default False] If False, set the window labels as the right edge of the window index.

If True, set the window labels as the center of the window index.

win_type
[str, default None] If `None`, all points are evenly weighted.

If a string, it must be a valid scipy.signal window function.

Certain Scipy window types require additional parameters to be passed in the aggregation function. The additional parameters must match the keywords specified in the Scipy window type method signature.

on
[str, optional] For a DataFrame, a column label or Index level on which to calculate the rolling window, rather than the DataFrame's index.

Provided integer column is ignored and excluded from result since an integer index is not used to calculate the rolling window.

axis
[int or str, default 0] If `0` or `'index'`, roll across the rows.

If `1` or `'columns'`, roll across the columns.

For *Series* this parameter is unused and defaults to 0.

closed
[str, default None] If `'right'`, the first point in the window is excluded from calculations.

If `'left'`, the last point in the window is excluded from calculations.

If `'both'`, the no points in the window are excluded from calculations.

If `'neither'`, the first and last points in the window are excluded from calculations.

Default `None` (`'right'`).

Changed in version 1.2.0: The closed parameter with fixed windows is now supported.

step
[int, default None] New in version 1.5.0.

Evaluate the window at every `step` result, equivalent to slicing as `[::step]`. `window` must be an integer. Using a step argument other than None or 1 will produce a result with a different shape than the input.

method
[str {'single', 'table'}, default 'single'] New in version 1.3.0.

Execute the rolling operation per single column or row (`'single'`) or over the entire object (`'table'`).

This argument is only implemented when specifying `engine='numba'` in the method call.

Returns

pandas.api.typing.Window or pandas.api.typing.Rolling
An instance of Window is returned if `win_type` is passed. Otherwise, an instance of Rolling is returned.

See also:

expanding
: Provides expanding transformations.

ewm
: Provides exponential weighted functions.

Notes

See Windowing Operations for further usage details and examples.

Examples

```
>>> df = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]})
>>> df
     B
0  0.0
1  1.0
2  2.0
3  NaN
4  4.0
```

window

Rolling sum with a window length of 2 observations.

```
>>> df.rolling(2).sum()
     B
0  NaN
1  1.0
2  3.0
3  NaN
4  NaN
```

Rolling sum with a window span of 2 seconds.

```
>>> df_time = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]},
...                        index=[pd.Timestamp('20130101 09:00:00'),
...                               pd.Timestamp('20130101 09:00:02'),
...                               pd.Timestamp('20130101 09:00:03'),
...                               pd.Timestamp('20130101 09:00:05'),
...                               pd.Timestamp('20130101 09:00:06')])
```

```
>>> df_time
                       B
2013-01-01 09:00:00  0.0
2013-01-01 09:00:02  1.0
2013-01-01 09:00:03  2.0
2013-01-01 09:00:05  NaN
2013-01-01 09:00:06  4.0
```

```
>>> df_time.rolling('2s').sum()
                       B
2013-01-01 09:00:00  0.0
2013-01-01 09:00:02  1.0
2013-01-01 09:00:03  3.0
2013-01-01 09:00:05  NaN
2013-01-01 09:00:06  4.0
```

Rolling sum with forward looking windows with 2 observations.

```
>>> indexer = pd.api.indexers.FixedForwardWindowIndexer(window_size=2)
>>> df.rolling(window=indexer, min_periods=1).sum()
     B
0  1.0
1  3.0
2  2.0
3  4.0
4  4.0
```

min_periods

Rolling sum with a window length of 2 observations, but only needs a minimum of 1 observation to calculate a value.

```
>>> df.rolling(2, min_periods=1).sum()
     B
0  0.0
1  1.0
2  3.0
3  2.0
4  4.0
```

center

Rolling sum with the result assigned to the center of the window index.

```
>>> df.rolling(3, min_periods=1, center=True).sum()
     B
0  1.0
1  3.0
2  3.0
3  6.0
4  4.0
```

```
>>> df.rolling(3, min_periods=1, center=False).sum()
     B
0  0.0
1  1.0
2  3.0
3  3.0
4  6.0
```

step

Rolling sum with a window length of 2 observations, minimum of 1 observation to calculate a value, and

a step of 2.

```
>>> df.rolling(2, min_periods=1, step=2).sum()
     B
0  0.0
2  3.0
4  4.0
```

win_type

Rolling sum with a window length of 2, using the Scipy `'gaussian'` window type. `std` is required in the aggregation function.

```
>>> df.rolling(2, win_type='gaussian').sum(std=3)
          B
0       NaN
1  0.986207
2  2.958621
3       NaN
4       NaN
```

on

Rolling sum with a window length of 2 days.

```
>>> df = pd.DataFrame({
...     'A': [pd.to_datetime('2020-01-01'),
...           pd.to_datetime('2020-01-01'),
...           pd.to_datetime('2020-01-02'),],
...     'B': [1, 2, 3], },
...     index=pd.date_range('2020', periods=3))
```

```
>>> df
                   A  B
2020-01-01 2020-01-01  1
2020-01-02 2020-01-01  2
2020-01-03 2020-01-02  3
```

```
>>> df.rolling('2D', on='A').sum()
                   A    B
2020-01-01 2020-01-01  1.0
2020-01-02 2020-01-01  3.0
2020-01-03 2020-01-02  6.0
```

AlloViz.AlloViz.Elements.Nodes.round

`Nodes.`**round**(*decimals: int | dict[IndexLabel, int] | Series = 0*, **args*, ***kwargs*) → DataFrame

Round a DataFrame to a variable number of decimal places.

Parameters

decimals

[int, dict, Series] Number of decimal places to round each column to. If an int is given, round each column to the same number of places. Otherwise dict and

Series round to variable numbers of places. Column names should be in the keys if *decimals* is a dict-like, or in the index if *decimals* is a Series. Any columns not included in *decimals* will be left as is. Elements of *decimals* which are not columns of the input will be ignored.

***args**
Additional keywords have no effect but might be accepted for compatibility with numpy.

****kwargs**
Additional keywords have no effect but might be accepted for compatibility with numpy.

Returns

DataFrame
A DataFrame with the affected columns rounded to the specified number of decimal places.

See also:

`numpy.around`
Round a numpy array to the given number of decimals.

`Series.round`
Round a Series to the given number of decimals.

Examples

```
>>> df = pd.DataFrame([(.21, .32), (.01, .67), (.66, .03), (.21, .18)],
...                   columns=['dogs', 'cats'])
>>> df
    dogs  cats
0  0.21  0.32
1  0.01  0.67
2  0.66  0.03
3  0.21  0.18
```

By providing an integer each column is rounded to the same number of decimal places

```
>>> df.round(1)
    dogs  cats
0   0.2   0.3
1   0.0   0.7
2   0.7   0.0
3   0.2   0.2
```

With a dict, the number of places for specific columns can be specified with the column names as key and the number of decimal places as value

```
>>> df.round({'dogs': 1, 'cats': 0})
    dogs  cats
0   0.2   0.0
1   0.0   1.0
2   0.7   0.0
3   0.2   0.0
```

Using a Series, the number of places for specific columns can be specified with the column names as index and the number of decimal places as value

```
>>> decimals = pd.Series([0, 1], index=['cats', 'dogs'])
>>> df.round(decimals)
    dogs  cats
0   0.2   0.0
1   0.0   1.0
2   0.7   0.0
3   0.2   0.0
```

AlloViz.AlloViz.Elements.Nodes.rpow

Nodes.**rpow**(*other*, *axis: Axis = 'columns'*, *level=None*, *fill_value=None*)

Get Exponential power of dataframe and other, element-wise (binary operator *rpow*).

Equivalent to `other ** dataframe`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *pow*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *floordiv*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other
: [scalar, sequence, Series, dict or DataFrame] Any single or multiple element data structure, or list-like object.

axis
: [{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns. (1 or 'columns'). For Series input, axis to match Series index on.

level
: [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value
: [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame
: Result of the arithmetic operation.

See also:

`DataFrame.add`
: Add DataFrames.

`DataFrame.sub`
: Subtract DataFrames.

`DataFrame.mul`
: Multiply DataFrames.

DataFrame.div
Divide DataFrames (float division).

DataFrame.truediv
Divide DataFrames (float division).

DataFrame.floordiv
Divide DataFrames (integer division).

DataFrame.mod
Calculate modulo (remainder after division).

DataFrame.pow
Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a dictionary by axis.

```
>>> df.mul({'angles': 0, 'degrees': 2})
            angles  degrees
circle           0      720
triangle         0      360
rectangle        0      720
```

```
>>> df.mul({'circle': 0, 'triangle': 2, 'rectangle': 3}, axis='index')
            angles  degrees
circle           0        0
triangle         6      360
rectangle       12     1080
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

AlloViz.AlloViz.Elements.Nodes.rsub

`Nodes.`**`rsub`**(*other*, *axis: Axis = 'columns'*, *level=None*, *fill_value=None*)

Get Subtraction of dataframe and other, element-wise (binary operator *rsub*).

Equivalent to `other - dataframe`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *sub*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *floordiv*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other

[scalar, sequence, Series, dict or DataFrame] Any single or multiple element data structure, or list-like object.

axis
[{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns. (1 or 'columns'). For Series input, axis to match Series index on.

level
[int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value
[float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame
Result of the arithmetic operation.

See also:

`DataFrame.add`
Add DataFrames.

`DataFrame.sub`
Subtract DataFrames.

`DataFrame.mul`
Multiply DataFrames.

`DataFrame.div`
Divide DataFrames (float division).

`DataFrame.truediv`
Divide DataFrames (float division).

`DataFrame.floordiv`
Divide DataFrames (integer division).

`DataFrame.mod`
Calculate modulo (remainder after division).

`DataFrame.pow`
Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a dictionary by axis.

```
>>> df.mul({'angles': 0, 'degrees': 2})
           angles  degrees
circle          0      720
triangle        0      360
rectangle       0      720
```

```
>>> df.mul({'circle': 0, 'triangle': 2, 'rectangle': 3}, axis='index')
           angles  degrees
circle          0        0
triangle        6      360
rectangle      12     1080
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
```

(continues on next page)

(continued from previous page)

```
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

AlloViz.AlloViz.Elements.Nodes.rtruediv

Nodes.**rtruediv**(*other*, *axis: Axis = 'columns'*, *level=None*, *fill_value=None*)

Get Floating division of dataframe and other, element-wise (binary operator *rtruediv*).

Equivalent to `other / dataframe`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *truediv*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *floordiv*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other
: [scalar, sequence, Series, dict or DataFrame] Any single or multiple element data structure, or list-like object.

axis
: [{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns. (1 or 'columns'). For Series input, axis to match Series index on.

level
: [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value
: [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame
: Result of the arithmetic operation.

See also:

DataFrame.add
: Add DataFrames.

DataFrame.sub
Subtract DataFrames.

DataFrame.mul
Multiply DataFrames.

DataFrame.div
Divide DataFrames (float division).

DataFrame.truediv
Divide DataFrames (float division).

DataFrame.floordiv
Divide DataFrames (integer division).

DataFrame.mod
Calculate modulo (remainder after division).

DataFrame.pow
Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
```

(continues on next page)

(continued from previous page)

```
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a dictionary by axis.

```
>>> df.mul({'angles': 0, 'degrees': 2})
            angles  degrees
circle           0      720
triangle         0      360
rectangle        0      720
```

```
>>> df.mul({'circle': 0, 'triangle': 2, 'rectangle': 3}, axis='index')
            angles  degrees
circle           0        0
triangle         6      360
rectangle       12     1080
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
```

(continues on next page)

(continued from previous page)

```
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

AlloViz.AlloViz.Elements.Nodes.sample

Nodes.**sample**(*n: None | int = None*, *frac: float | None = None*, *replace: bool = False*, *weights=None*, *random_state: int | ndarray | Generator | BitGenerator | RandomState | None = None*, *axis: int | Literal['index', 'columns', 'rows'] | None = None*, *ignore_index: bool = False*) → None

Return a random sample of items from an axis of object.

You can use *random_state* for reproducibility.

Parameters

n
[int, optional] Number of items from axis to return. Cannot be used with *frac*. Default = 1 if *frac* = None.

frac
[float, optional] Fraction of axis items to return. Cannot be used with *n*.

replace
[bool, default False] Allow or disallow sampling of the same row more than once.

weights
[str or ndarray-like, optional] Default 'None' results in equal probability weighting. If passed a Series, will align with target object on index. Index values in weights not found in sampled object will be ignored and index values in sampled object not in weights will be assigned weights of zero. If called on a DataFrame, will accept the name of a column when axis = 0. Unless weights are a Series, weights must be same length as axis being sampled. If weights do not sum to 1, they will be normalized to sum to 1. Missing values in the weights column will be treated as zero. Infinite values not allowed.

random_state
[int, array-like, BitGenerator, np.random.RandomState, np.random.Generator, optional] If int, array-like, or BitGenerator, seed for random number generator. If np.random.RandomState or np.random.Generator, use as given.

Changed in version 1.4.0: np.random.Generator objects now accepted

axis
[{0 or 'index', 1 or 'columns', None}, default None] Axis to sample. Accepts axis number or name. Default is stat axis for given data type. For *Series* this parameter is unused and defaults to *None*.

ignore_index
[bool, default False] If True, the resulting index will be labeled 0, 1, …, n - 1.

New in version 1.3.0.

Returns

Series or DataFrame
A new object of same type as caller containing *n* items randomly sampled from the caller object.

See also:

`DataFrameGroupBy.sample`
Generates random samples from each group of a DataFrame object.

`SeriesGroupBy.sample`
Generates random samples from each group of a Series object.

`numpy.random.choice`
Generates a random sample from a given 1-D numpy array.

Notes

If *frac* > 1, *replacement* should be set to *True*.

Examples

```
>>> df = pd.DataFrame({'num_legs': [2, 4, 8, 0],
...                    'num_wings': [2, 0, 0, 0],
...                    'num_specimen_seen': [10, 2, 1, 8]},
...                   index=['falcon', 'dog', 'spider', 'fish'])
>>> df
        num_legs  num_wings  num_specimen_seen
falcon         2          2                 10
dog            4          0                  2
spider         8          0                  1
fish           0          0                  8
```

Extract 3 random elements from the `Series` `df['num_legs']`: Note that we use *random_state* to ensure the reproducibility of the examples.

```
>>> df['num_legs'].sample(n=3, random_state=1)
fish      0
spider    8
falcon    2
Name: num_legs, dtype: int64
```

A random 50% sample of the `DataFrame` with replacement:

```
>>> df.sample(frac=0.5, replace=True, random_state=1)
      num_legs  num_wings  num_specimen_seen
dog          4          0                  2
fish         0          0                  8
```

An upsample sample of the `DataFrame` with replacement: Note that *replace* parameter has to be *True* for *frac* parameter > 1.

```
>>> df.sample(frac=2, replace=True, random_state=1)
        num_legs  num_wings  num_specimen_seen
dog            4          0                  2
fish           0          0                  8
falcon         2          2                 10
falcon         2          2                 10
fish           0          0                  8
dog            4          0                  2
fish           0          0                  8
dog            4          0                  2
```

Using a DataFrame column as weights. Rows with larger value in the *num_specimen_seen* column are more likely to be sampled.

```
>>> df.sample(n=2, weights='num_specimen_seen', random_state=1)
        num_legs  num_wings  num_specimen_seen
```

(continues on next page)

(continued from previous page)

```
falcon         2          2                 10
fish           0          0                  8
```

AlloViz.AlloViz.Elements.Nodes.select_dtypes

`Nodes.select_dtypes`(*include=None*, *exclude=None*) → Self

Return a subset of the DataFrame's columns based on the column dtypes.

Parameters

include, exclude
[scalar or list-like] A selection of dtypes or strings to be included/excluded. At least one of these parameters must be supplied.

Returns

DataFrame
The subset of the frame including the dtypes in `include` and excluding the dtypes in `exclude`.

Raises

ValueError

- If both of `include` and `exclude` are empty
- If `include` and `exclude` have overlapping elements
- If any kind of string dtype is passed in.

See also:

`DataFrame.dtypes`
Return Series with the data type of each column.

Notes

- To select all *numeric* types, use `np.number` or `'number'`
- To select strings you must use the `object` dtype, but note that this will return *all* object dtype columns
- See the numpy dtype hierarchy
- To select datetimes, use `np.datetime64`, `'datetime'` or `'datetime64'`
- To select timedeltas, use `np.timedelta64`, `'timedelta'` or `'timedelta64'`
- To select Pandas categorical dtypes, use `'category'`
- To select Pandas datetimetz dtypes, use `'datetimetz'` or `'datetime64[ns, tz]'`

Examples

```
>>> df = pd.DataFrame({'a': [1, 2] * 3,
...                    'b': [True, False] * 3,
...                    'c': [1.0, 2.0] * 3})
>>> df
        a      b  c
0       1   True  1.0
1       2  False  2.0
2       1   True  1.0
3       2  False  2.0
4       1   True  1.0
5       2  False  2.0
```

```
>>> df.select_dtypes(include='bool')
   b
0  True
1  False
2  True
3  False
4  True
5  False
```

```
>>> df.select_dtypes(include=['float64'])
   c
0  1.0
1  2.0
2  1.0
3  2.0
4  1.0
5  2.0
```

```
>>> df.select_dtypes(exclude=['int64'])
       b    c
0   True  1.0
1  False  2.0
2   True  1.0
3  False  2.0
4   True  1.0
5  False  2.0
```

AlloViz.AlloViz.Elements.Nodes.sem

Nodes.**sem**(*axis: Axis | None = 0*, *skipna: bool = True*, *ddof: int = 1*, *numeric_only: bool = False*, ***kwargs*)

Return unbiased standard error of the mean over requested axis.

Normalized by N-1 by default. This can be changed using the ddof argument

Parameters

axis

[{index (0), columns (1)}] For *Series* this parameter is unused and defaults to 0.

skipna
: [bool, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

ddof
: [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is N - ddof, where N represents the number of elements.

numeric_only
: [bool, default False] Include only float, int, boolean columns. Not implemented for Series.

Returns

Series or DataFrame (if level specified)

Examples

```
>>> s = pd.Series([1, 2, 3])
>>> s.sem().round(6)
0.57735
```

With a DataFrame

```
>>> df = pd.DataFrame({'a': [1, 2], 'b': [2, 3]}, index=['tiger', 'zebra'])
>>> df
       a   b
tiger  1   2
zebra  2   3
>>> df.sem()
a   0.5
b   0.5
dtype: float64
```

Using axis=1

```
>>> df.sem(axis=1)
tiger   0.5
zebra   0.5
dtype: float64
```

In this case, *numeric_only* should be set to *True* to avoid getting an error.

```
>>> df = pd.DataFrame({'a': [1, 2], 'b': ['T', 'Z']},
...                   index=['tiger', 'zebra'])
>>> df.sem(numeric_only=True)
a   0.5
dtype: float64
```

AlloViz.AlloViz.Elements.Nodes.set_axis

Nodes.**set_axis**(*labels*, *, *axis: Axis = 0*, *copy: bool | None = None*) → DataFrame

Assign desired index to given axis.

Indexes for column or row labels can be changed by assigning a list-like or Index.

Parameters

labels
[list-like, Index] The values for the new index.

axis
[{0 or 'index', 1 or 'columns'}, default 0] The axis to update. The value 0 identifies the rows. For *Series* this parameter is unused and defaults to 0.

copy
[bool, default True] Whether to make a copy of the underlying data.

New in version 1.5.0.

Returns

DataFrame
An object of type DataFrame.

See also:

DataFrame.rename_axis
Alter the name of the index or columns.

Examples

```
>>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
```

Change the row labels.

```
>>> df.set_axis(['a', 'b', 'c'], axis='index')
   A  B
a  1  4
b  2  5
c  3  6
```

Change the column labels.

```
>>> df.set_axis(['I', 'II'], axis='columns')
   I  II
0  1   4
1  2   5
2  3   6
```

AlloViz.AlloViz.Elements.Nodes.set_flags

Nodes.**set_flags**(*, *copy: bool = False*, *allows_duplicate_labels: bool | None = None*) → None

Return a new object with updated flags.

Parameters

copy
[bool, default False] Specify if a copy of the object should be made.

allows_duplicate_labels
[bool, optional] Whether the returned object allows duplicate labels.

Returns

Series or DataFrame
The same type as the caller.

See also:

DataFrame.attrs
Global metadata applying to this dataset.

DataFrame.flags
Global flags applying to this object.

Notes

This method returns a new object that's a view on the same data as the input. Mutating the input or the output values will be reflected in the other.

This method is intended to be used in method chains.

"Flags" differ from "metadata". Flags reflect properties of the pandas object (the Series or DataFrame). Metadata refer to properties of the dataset, and should be stored in `DataFrame.attrs`.

Examples

```
>>> df = pd.DataFrame({"A": [1, 2]})
>>> df.flags.allows_duplicate_labels
True
>>> df2 = df.set_flags(allows_duplicate_labels=False)
>>> df2.flags.allows_duplicate_labels
False
```

AlloViz.AlloViz.Elements.Nodes.set_index

Nodes.**set_index**(*keys*, *, *drop: bool = True*, *append: bool = False*, *inplace: bool = False*, *verify_integrity: bool = False*) → DataFrame | None

Set the DataFrame index using existing columns.

Set the DataFrame index (row labels) using one or more existing columns or arrays (of the correct length). The index can replace the existing index or expand on it.

Parameters

keys
[label or array-like or list of labels/arrays] This parameter can be either a single column key, a single array of the same length as the calling DataFrame, or a list containing an arbitrary combination of column keys and arrays. Here, "array" encompasses `Series`, `Index`, `np.ndarray`, and instances of `Iterator`.

drop
[bool, default True] Delete columns to be used as the new index.

append
[bool, default False] Whether to append columns to existing index.

inplace
[bool, default False] Whether to modify the DataFrame rather than creating a new one.

verify_integrity
[bool, default False] Check the new index for duplicates. Otherwise defer the check until necessary. Setting to False will improve the performance of this method.

Returns

DataFrame or None
Changed row labels or None if `inplace=True`.

See also:

`DataFrame.reset_index`
Opposite of set_index.

`DataFrame.reindex`
Change to new indices or expand indices.

`DataFrame.reindex_like`
Change to same indices as other DataFrame.

Examples

```
>>> df = pd.DataFrame({'month': [1, 4, 7, 10],
...                    'year': [2012, 2014, 2013, 2014],
...                    'sale': [55, 40, 84, 31]})
>>> df
   month  year  sale
0      1  2012    55
1      4  2014    40
2      7  2013    84
3     10  2014    31
```

Set the index to become the 'month' column:

```
>>> df.set_index('month')
       year  sale
month
1      2012    55
4      2014    40
7      2013    84
10     2014    31
```

Create a MultiIndex using columns 'year' and 'month':

```
>>> df.set_index(['year', 'month'])
            sale
year  month
2012  1     55
2014  4     40
2013  7     84
2014  10    31
```

Create a MultiIndex using an Index and a column:

```
>>> df.set_index([pd.Index([1, 2, 3, 4]), 'year'])
         month  sale
   year
1  2012  1      55
2  2014  4      40
3  2013  7      84
4  2014  10     31
```

Create a MultiIndex using two Series:

```
>>> s = pd.Series([1, 2, 3, 4])
>>> df.set_index([s, s**2])
      month  year  sale
1 1       1  2012    55
2 4       4  2014    40
3 9       7  2013    84
4 16     10  2014    31
```

AlloViz.AlloViz.Elements.Nodes.shift

Nodes.**shift**(*periods: int | Sequence[int] = 1*, *freq: Frequency | None = None*, *axis: Axis = 0*, *fill_value: Hashable = _NoDefault.no_default*, *suffix: str | None = None*) → DataFrame

Shift index by desired number of periods with an optional time *freq*.

When *freq* is not passed, shift the index without realigning the data. If *freq* is passed (in this case, the index must be date or datetime, or it will raise a *NotImplementedError*), the index will be increased using the periods and the *freq*. *freq* can be inferred when specified as "infer" as long as either freq or inferred_freq attribute is set in the index.

Parameters

periods

[int or Sequence] Number of periods to shift. Can be positive or negative. If an iterable of ints, the data will be shifted once by each int. This is equivalent to shifting by one value at a time and concatenating all resulting frames. The resulting columns will have the shift suffixed to their column names. For multiple periods, axis must not be 1.

freq

[DateOffset, tseries.offsets, timedelta, or str, optional] Offset to use from the tseries module or time rule (e.g. 'EOM'). If *freq* is specified then the index values are shifted but the data is not realigned. That is, use *freq* if you would like to extend the index when shifting and preserve the original data. If *freq* is specified as "infer"

then it will be inferred from the freq or inferred_freq attributes of the index. If neither of those attributes exist, a ValueError is thrown.

axis
[{0 or 'index', 1 or 'columns', None}, default None] Shift direction. For *Series* this parameter is unused and defaults to 0.

fill_value
[object, optional] The scalar value to use for newly introduced missing values. the default depends on the dtype of *self*. For numeric data, `np.nan` is used. For datetime, timedelta, or period data, etc. `NaT` is used. For extension dtypes, `self.dtype.na_value` is used.

suffix
[str, optional] If str and periods is an iterable, this is added after the column name and before the shift value for each shifted column name.

Returns

DataFrame
Copy of input object, shifted.

See also:

`Index.shift`
Shift values of Index.

`DatetimeIndex.shift`
Shift values of DatetimeIndex.

`PeriodIndex.shift`
Shift values of PeriodIndex.

Examples

```
>>> df = pd.DataFrame({"Col1": [10, 20, 15, 30, 45],
...                    "Col2": [13, 23, 18, 33, 48],
...                    "Col3": [17, 27, 22, 37, 52]},
...                   index=pd.date_range("2020-01-01", "2020-01-05"))
>>> df
            Col1  Col2  Col3
2020-01-01    10    13    17
2020-01-02    20    23    27
2020-01-03    15    18    22
2020-01-04    30    33    37
2020-01-05    45    48    52
```

```
>>> df.shift(periods=3)
            Col1  Col2  Col3
2020-01-01   NaN   NaN   NaN
2020-01-02   NaN   NaN   NaN
2020-01-03   NaN   NaN   NaN
2020-01-04  10.0  13.0  17.0
2020-01-05  20.0  23.0  27.0
```

```
>>> df.shift(periods=1, axis="columns")
            Col1  Col2  Col3
2020-01-01   NaN    10    13
2020-01-02   NaN    20    23
2020-01-03   NaN    15    18
2020-01-04   NaN    30    33
2020-01-05   NaN    45    48
```

```
>>> df.shift(periods=3, fill_value=0)
            Col1  Col2  Col3
2020-01-01     0     0     0
2020-01-02     0     0     0
2020-01-03     0     0     0
2020-01-04    10    13    17
2020-01-05    20    23    27
```

```
>>> df.shift(periods=3, freq="D")
            Col1  Col2  Col3
2020-01-04    10    13    17
2020-01-05    20    23    27
2020-01-06    15    18    22
2020-01-07    30    33    37
2020-01-08    45    48    52
```

```
>>> df.shift(periods=3, freq="infer")
            Col1  Col2  Col3
2020-01-04    10    13    17
2020-01-05    20    23    27
2020-01-06    15    18    22
2020-01-07    30    33    37
2020-01-08    45    48    52
```

```
>>> df['Col1'].shift(periods=[0, 1, 2])
            Col1_0  Col1_1  Col1_2
2020-01-01      10     NaN     NaN
2020-01-02      20    10.0     NaN
2020-01-03      15    20.0    10.0
2020-01-04      30    15.0    20.0
2020-01-05      45    30.0    15.0
```

AlloViz.AlloViz.Elements.Nodes.skew

Nodes.**skew**(*axis: Axis | None = 0*, *skipna: bool = True*, *numeric_only: bool = False*, ***kwargs*)

Return unbiased skew over requested axis.

Normalized by N-1.

Parameters

axis

[{index (0), columns (1)}] Axis for the function to be applied on. For *Series* this parameter is unused and defaults to 0.

For DataFrames, specifying `axis=None` will apply the aggregation across both axes.

New in version 2.0.0.

skipna
[bool, default True] Exclude NA/null values when computing the result.

numeric_only
[bool, default False] Include only float, int, boolean columns. Not implemented for Series.

****kwargs**
Additional keyword arguments to be passed to the function.

Returns

Series or scalar

Examples

```
>>> s = pd.Series([1, 2, 3])
>>> s.skew()
0.0
```

With a DataFrame

```
>>> df = pd.DataFrame({'a': [1, 2, 3], 'b': [2, 3, 4], 'c': [1, 3, 5]},
...                   index=['tiger', 'zebra', 'cow'])
>>> df
        a   b   c
tiger   1   2   1
zebra   2   3   3
cow     3   4   5
>>> df.skew()
a   0.0
b   0.0
c   0.0
dtype: float64
```

Using axis=1

```
>>> df.skew(axis=1)
tiger   1.732051
zebra  -1.732051
cow     0.000000
dtype: float64
```

In this case, *numeric_only* should be set to *True* to avoid getting an error.

```
>>> df = pd.DataFrame({'a': [1, 2, 3], 'b': ['T', 'Z', 'X']},
...                   index=['tiger', 'zebra', 'cow'])
>>> df.skew(numeric_only=True)
a   0.0
dtype: float64
```

AlloViz.AlloViz.Elements.Nodes.sort_index

Nodes.**sort_index**(*, *axis: Axis = 0*, *level: IndexLabel | None = None*, *ascending: bool | Sequence[bool] = True*, *inplace: bool = False*, *kind: SortKind = 'quicksort'*, *na_position: NaPosition = 'last'*, *sort_remaining: bool = True*, *ignore_index: bool = False*, *key: IndexKeyFunc | None = None*) → DataFrame | None

Sort object by labels (along an axis).

Returns a new DataFrame sorted by label if *inplace* argument is `False`, otherwise updates the original DataFrame and returns None.

Parameters

axis
[{0 or 'index', 1 or 'columns'}, default 0] The axis along which to sort. The value 0 identifies the rows, and 1 identifies the columns.

level
[int or level name or list of ints or list of level names] If not None, sort on values in specified index level(s).

ascending
[bool or list-like of bools, default True] Sort ascending vs. descending. When the index is a MultiIndex the sort direction can be controlled for each level individually.

inplace
[bool, default False] Whether to modify the DataFrame rather than creating a new one.

kind
[{'quicksort', 'mergesort', 'heapsort', 'stable'}, default 'quicksort'] Choice of sorting algorithm. See also `numpy.sort()` for more information. *mergesort* and *stable* are the only stable algorithms. For DataFrames, this option is only applied when sorting on a single column or label.

na_position
[{'first', 'last'}, default 'last'] Puts NaNs at the beginning if *first*; *last* puts NaNs at the end. Not implemented for MultiIndex.

sort_remaining
[bool, default True] If True and sorting by level and index is multilevel, sort by other levels too (in order) after sorting by specified level.

ignore_index
[bool, default False] If True, the resulting axis will be labeled 0, 1, …, n - 1.

key
[callable, optional] If not None, apply the key function to the index values before sorting. This is similar to the *key* argument in the builtin `sorted()` function, with the notable difference that this *key* function should be *vectorized*. It should expect an `Index` and return an `Index` of the same shape. For MultiIndex inputs, the key is applied *per level*.

Returns

DataFrame or None
The original DataFrame sorted by the labels or None if `inplace=True`.

See also:

Series.sort_index
Sort Series by the index.

DataFrame.sort_values
Sort DataFrame by the value.

Series.sort_values
Sort Series by the value.

Examples

```
>>> df = pd.DataFrame([1, 2, 3, 4, 5], index=[100, 29, 234, 1, 150],
...                   columns=['A'])
>>> df.sort_index()
     A
1    4
29   2
100  1
150  5
234  3
```

By default, it sorts in ascending order, to sort in descending order, use `ascending=False`

```
>>> df.sort_index(ascending=False)
     A
234  3
150  5
100  1
29   2
1    4
```

A key function can be specified which is applied to the index before sorting. For a `MultiIndex` this is applied to each level separately.

```
>>> df = pd.DataFrame({"a": [1, 2, 3, 4]}, index=['A', 'b', 'C', 'd'])
>>> df.sort_index(key=lambda x: x.str.lower())
   a
A  1
b  2
C  3
d  4
```

AlloViz.AlloViz.Elements.Nodes.sort_values

Nodes.**sort_values**(*by: IndexLabel*, ***, *axis: Axis = 0*, *ascending: bool | list[bool] | tuple[bool, ...] = True*, *inplace: bool = False*, *kind: SortKind = 'quicksort'*, *na_position: str = 'last'*, *ignore_index: bool = False*, *key: ValueKeyFunc | None = None*) → DataFrame | None

Sort by the values along either axis.

Parameters

by
[str or list of str] Name or list of names to sort by.

- if *axis* is 0 or *'index'* then *by* may contain index levels and/or column labels.
- if *axis* is 1 or *'columns'* then *by* may contain column levels and/or index labels.

axis
["{0 or 'index', 1 or 'columns'}", default 0] Axis to be sorted.

ascending
[bool or list of bool, default True] Sort ascending vs. descending. Specify list for multiple sort orders. If this is a list of bools, must match the length of the by.

inplace
[bool, default False] If True, perform operation in-place.

kind
[{'quicksort', 'mergesort', 'heapsort', 'stable'}, default 'quicksort'] Choice of sorting algorithm. See also `numpy.sort()` for more information. *mergesort* and *stable* are the only stable algorithms. For DataFrames, this option is only applied when sorting on a single column or label.

na_position
[{'first', 'last'}, default 'last'] Puts NaNs at the beginning if *first*; *last* puts NaNs at the end.

ignore_index
[bool, default False] If True, the resulting axis will be labeled 0, 1, . . . , n - 1.

key
[callable, optional] Apply the key function to the values before sorting. This is similar to the *key* argument in the builtin `sorted()` function, with the notable difference that this *key* function should be *vectorized*. It should expect a `Series` and return a Series with the same shape as the input. It will be applied to each column in *by* independently.

Returns

DataFrame or None
DataFrame with sorted values or None if `inplace=True`.

See also:

`DataFrame.sort_index`
Sort a DataFrame by the index.

`Series.sort_values`
Similar method for a Series.

Examples

```
>>> df = pd.DataFrame({
...     'col1': ['A', 'A', 'B', np.nan, 'D', 'C'],
...     'col2': [2, 1, 9, 8, 7, 4],
...     'col3': [0, 1, 9, 4, 2, 3],
...     'col4': ['a', 'B', 'c', 'D', 'e', 'F']
... })
>>> df
  col1  col2  col3 col4
0    A     2     0    a
```

(continues on next page)

(continued from previous page)

```
1    A     1     1    B
2    B     9     9    c
3  NaN     8     4    D
4    D     7     2    e
5    C     4     3    F
```

Sort by col1

```
>>> df.sort_values(by=['col1'])
  col1  col2  col3 col4
0    A     2     0    a
1    A     1     1    B
2    B     9     9    c
5    C     4     3    F
4    D     7     2    e
3  NaN     8     4    D
```

Sort by multiple columns

```
>>> df.sort_values(by=['col1', 'col2'])
  col1  col2  col3 col4
1    A     1     1    B
0    A     2     0    a
2    B     9     9    c
5    C     4     3    F
4    D     7     2    e
3  NaN     8     4    D
```

Sort Descending

```
>>> df.sort_values(by='col1', ascending=False)
  col1  col2  col3 col4
4    D     7     2    e
5    C     4     3    F
2    B     9     9    c
0    A     2     0    a
1    A     1     1    B
3  NaN     8     4    D
```

Putting NAs first

```
>>> df.sort_values(by='col1', ascending=False, na_position='first')
  col1  col2  col3 col4
3  NaN     8     4    D
4    D     7     2    e
5    C     4     3    F
2    B     9     9    c
0    A     2     0    a
1    A     1     1    B
```

Sorting with a key function

```
>>> df.sort_values(by='col4', key=lambda col: col.str.lower())
   col1  col2  col3 col4
0    A     2     0    a
1    A     1     1    B
2    B     9     9    c
3  NaN     8     4    D
4    D     7     2    e
5    C     4     3    F
```

Natural sort with the key argument, using the *natsort <https://github.com/SethMMorton/natsort>* package.

```
>>> df = pd.DataFrame({
...    "time": ['0hr', '128hr', '72hr', '48hr', '96hr'],
...    "value": [10, 20, 30, 40, 50]
... })
>>> df
    time  value
0    0hr     10
1  128hr     20
2   72hr     30
3   48hr     40
4   96hr     50
>>> from natsort import index_natsorted
>>> df.sort_values(
...     by="time",
...     key=lambda x: np.argsort(index_natsorted(df["time"]))
... )
    time  value
0    0hr     10
3   48hr     40
2   72hr     30
4   96hr     50
1  128hr     20
```

AlloViz.AlloViz.Elements.Nodes.squeeze

`Nodes.`**`squeeze`**(*axis: int | Literal['index', 'columns', 'rows'] | None = None*)

Squeeze 1 dimensional axis objects into scalars.

Series or DataFrames with a single element are squeezed to a scalar. DataFrames with a single column or a single row are squeezed to a Series. Otherwise the object is unchanged.

This method is most useful when you don't know if your object is a Series or DataFrame, but you do know it has just a single column. In that case you can safely call *squeeze* to ensure you have a Series.

Parameters

axis
[{0 or 'index', 1 or 'columns', None}, default None] A specific axis to squeeze. By default, all length-1 axes are squeezed. For *Series* this parameter is unused and defaults to *None*.

Returns

DataFrame, Series, or scalar
The projection after squeezing *axis* or all the axes.

See also:

`Series.iloc`
Integer-location based indexing for selecting scalars.

`DataFrame.iloc`
Integer-location based indexing for selecting Series.

`Series.to_frame`
Inverse of DataFrame.squeeze for a single-column DataFrame.

Examples

```
>>> primes = pd.Series([2, 3, 5, 7])
```

Slicing might produce a Series with a single value:

```
>>> even_primes = primes[primes % 2 == 0]
>>> even_primes
0    2
dtype: int64
```

```
>>> even_primes.squeeze()
2
```

Squeezing objects with more than one value in every axis does nothing:

```
>>> odd_primes = primes[primes % 2 == 1]
>>> odd_primes
1    3
2    5
3    7
dtype: int64
```

```
>>> odd_primes.squeeze()
1    3
2    5
3    7
dtype: int64
```

Squeezing is even more effective when used with DataFrames.

```
>>> df = pd.DataFrame([[1, 2], [3, 4]], columns=['a', 'b'])
>>> df
   a  b
0  1  2
1  3  4
```

Slicing a single column will produce a DataFrame with the columns having only one value:

```
>>> df_a = df[['a']]
>>> df_a
   a
0  1
1  3
```

So the columns can be squeezed down, resulting in a Series:

```
>>> df_a.squeeze('columns')
0    1
1    3
Name: a, dtype: int64
```

Slicing a single row from a single column will produce a single scalar DataFrame:

```
>>> df_0a = df.loc[df.index < 1, ['a']]
>>> df_0a
   a
0  1
```

Squeezing the rows produces a single scalar Series:

```
>>> df_0a.squeeze('rows')
a    1
Name: 0, dtype: int64
```

Squeezing all axes will project directly into a scalar:

```
>>> df_0a.squeeze()
1
```

AlloViz.AlloViz.Elements.Nodes.stack

Nodes.**stack**(*level: IndexLabel = -1*, *dropna: bool | lib.NoDefault = _NoDefault.no_default*, *sort: bool | lib.NoDefault = _NoDefault.no_default*, *future_stack: bool = False*)

Stack the prescribed level(s) from columns to index.

Return a reshaped DataFrame or Series having a multi-level index with one or more new inner-most levels compared to the current DataFrame. The new inner-most levels are created by pivoting the columns of the current dataframe:

- if the columns have a single level, the output is a Series;
- if the columns have multiple levels, the new index level(s) is (are) taken from the prescribed level(s) and the output is a DataFrame.

Parameters

level
[int, str, list, default -1] Level(s) to stack from the column axis onto the index axis, defined as one index or label, or a list of indices or labels.

dropna
[bool, default True] Whether to drop rows in the resulting Frame/Series with missing values. Stacking a column level onto the index axis can create combinations of

index and column values that are missing from the original dataframe. See Examples section.

sort
[bool, default True] Whether to sort the levels of the resulting MultiIndex.

future_stack
[bool, default False] Whether to use the new implementation that will replace the current implementation in pandas 3.0. When True, dropna and sort have no impact on the result and must remain unspecified. See pandas 2.1.0 Release notes for more details.

Returns

DataFrame or Series
Stacked dataframe or series.

See also:

`DataFrame.unstack`
Unstack prescribed level(s) from index axis onto column axis.

`DataFrame.pivot`
Reshape dataframe from long format to wide format.

`DataFrame.pivot_table`
Create a spreadsheet-style pivot table as a DataFrame.

Notes

The function is named by analogy with a collection of books being reorganized from being side by side on a horizontal position (the columns of the dataframe) to being stacked vertically on top of each other (in the index of the dataframe).

Reference the user guide for more examples.

Examples

Single level columns

```
>>> df_single_level_cols = pd.DataFrame([[0, 1], [2, 3]],
...                                     index=['cat', 'dog'],
...                                     columns=['weight', 'height'])
```

Stacking a dataframe with a single level column axis returns a Series:

```
>>> df_single_level_cols
     weight height
cat       0      1
dog       2      3
>>> df_single_level_cols.stack(future_stack=True)
cat  weight    0
     height    1
dog  weight    2
     height    3
dtype: int64
```

Multi level columns: simple case

```
>>> multicol1 = pd.MultiIndex.from_tuples([('weight', 'kg'),
...                                        ('weight', 'pounds')])
>>> df_multi_level_cols1 = pd.DataFrame([[1, 2], [2, 4]],
...                                     index=['cat', 'dog'],
...                                     columns=multicol1)
```

Stacking a dataframe with a multi-level column axis:

```
>>> df_multi_level_cols1
     weight
         kg    pounds
cat       1         2
dog       2         4
>>> df_multi_level_cols1.stack(future_stack=True)
            weight
cat kg           1
    pounds       2
dog kg           2
    pounds       4
```

Missing values

```
>>> multicol2 = pd.MultiIndex.from_tuples([('weight', 'kg'),
...                                        ('height', 'm')])
>>> df_multi_level_cols2 = pd.DataFrame([[1.0, 2.0], [3.0, 4.0]],
...                                     index=['cat', 'dog'],
...                                     columns=multicol2)
```

It is common to have missing values when stacking a dataframe with multi-level columns, as the stacked dataframe typically has more values than the original dataframe. Missing values are filled with NaNs:

```
>>> df_multi_level_cols2
    weight height
        kg      m
cat    1.0    2.0
dog    3.0    4.0
>>> df_multi_level_cols2.stack(future_stack=True)
        weight  height
cat kg     1.0     NaN
    m      NaN     2.0
dog kg     3.0     NaN
    m      NaN     4.0
```

Prescribing the level(s) to be stacked

The first parameter controls which level or levels are stacked:

```
>>> df_multi_level_cols2.stack(0, future_stack=True)
             kg    m
cat weight  1.0  NaN
    height  NaN  2.0
dog weight  3.0  NaN
    height  NaN  4.0
```

(continues on next page)

(continued from previous page)

```
>>> df_multi_level_cols2.stack([0, 1], future_stack=True)
cat  weight  kg    1.0
     height  m     2.0
dog  weight  kg    3.0
     height  m     4.0
dtype: float64
```

Dropping missing values

```
>>> df_multi_level_cols3 = pd.DataFrame([[None, 1.0], [2.0, 3.0]],
...                                     index=['cat', 'dog'],
...                                     columns=multicol2)
```

Note that rows where all values are missing are dropped by default but this behaviour can be controlled via the dropna keyword parameter:

```
>>> df_multi_level_cols3
    weight height
        kg      m
cat    NaN    1.0
dog    2.0    3.0
>>> df_multi_level_cols3.stack(dropna=False)
        weight  height
cat kg     NaN     NaN
    m      NaN     1.0
dog kg     2.0     NaN
    m      NaN     3.0
>>> df_multi_level_cols3.stack(dropna=True)
        weight  height
cat m      NaN     1.0
dog kg     2.0     NaN
    m      NaN     3.0
```

AlloViz.AlloViz.Elements.Nodes.std

`Nodes.std`(*axis: Axis | None = 0*, *skipna: bool = True*, *ddof: int = 1*, *numeric_only: bool = False*, ***kwargs*)

Return sample standard deviation over requested axis.

Normalized by N-1 by default. This can be changed using the ddof argument.

Parameters

axis
: [{index (0), columns (1)}] For *Series* this parameter is unused and defaults to 0.

skipna
: [bool, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

ddof
: [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is N - ddof, where N represents the number of elements.

numeric_only
[bool, default False] Include only float, int, boolean columns. Not implemented for Series.

Returns

Series or DataFrame (if level specified)

Notes

To have the same behaviour as *numpy.std*, use *ddof=0* (instead of the default *ddof=1*)

Examples

```
>>> df = pd.DataFrame({'person_id': [0, 1, 2, 3],
...                    'age': [21, 25, 62, 43],
...                    'height': [1.61, 1.87, 1.49, 2.01]}
...                   ).set_index('person_id')
>>> df
           age  height
person_id
0           21    1.61
1           25    1.87
2           62    1.49
3           43    2.01
```

The standard deviation of the columns can be found as follows:

```
>>> df.std()
age       18.786076
height     0.237417
dtype: float64
```

Alternatively, *ddof=0* can be set to normalize by N instead of N-1:

```
>>> df.std(ddof=0)
age       16.269219
height     0.205609
dtype: float64
```

AlloViz.AlloViz.Elements.Nodes.sub

`Nodes.sub`(*other*, *axis: Axis = 'columns'*, *level=None*, *fill_value=None*)

Get Subtraction of dataframe and other, element-wise (binary operator *sub*).

Equivalent to `dataframe - other`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *rsub*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *floordiv*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other
[scalar, sequence, Series, dict or DataFrame] Any single or multiple element data structure, or list-like object.

axis
[{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns. (1 or 'columns'). For Series input, axis to match Series index on.

level
[int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value
[float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame
Result of the arithmetic operation.

See also:

`DataFrame.add`
Add DataFrames.

`DataFrame.sub`
Subtract DataFrames.

`DataFrame.mul`
Multiply DataFrames.

`DataFrame.div`
Divide DataFrames (float division).

`DataFrame.truediv`
Divide DataFrames (float division).

`DataFrame.floordiv`
Divide DataFrames (integer division).

`DataFrame.mod`
Calculate modulo (remainder after division).

`DataFrame.pow`
Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a dictionary by axis.

```
>>> df.mul({'angles': 0, 'degrees': 2})
           angles  degrees
circle          0      720
triangle        0      360
rectangle       0      720
```

```
>>> df.mul({'circle': 0, 'triangle': 2, 'rectangle': 3}, axis='index')
           angles  degrees
circle          0        0
triangle        6      360
rectangle      12     1080
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
```

(continues on next page)

(continued from previous page)

```
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

AlloViz.AlloViz.Elements.Nodes.subtract

Nodes.**subtract**(*other*, *axis: Axis = 'columns'*, *level=None*, *fill_value=None*)

Get Subtraction of dataframe and other, element-wise (binary operator *sub*).

Equivalent to `dataframe - other`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *rsub*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *floordiv*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other
: [scalar, sequence, Series, dict or DataFrame] Any single or multiple element data structure, or list-like object.

axis
: [{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns. (1 or 'columns'). For Series input, axis to match Series index on.

level
: [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value
: [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame
: Result of the arithmetic operation.

See also:

DataFrame.add
: Add DataFrames.

DataFrame.sub
Subtract DataFrames.

DataFrame.mul
Multiply DataFrames.

DataFrame.div
Divide DataFrames (float division).

DataFrame.truediv
Divide DataFrames (float division).

DataFrame.floordiv
Divide DataFrames (integer division).

DataFrame.mod
Calculate modulo (remainder after division).

DataFrame.pow
Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
```

(continues on next page)

(continued from previous page)

```
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a dictionary by axis.

```
>>> df.mul({'angles': 0, 'degrees': 2})
            angles  degrees
circle           0      720
triangle         0      360
rectangle        0      720
```

```
>>> df.mul({'circle': 0, 'triangle': 2, 'rectangle': 3}, axis='index')
            angles  degrees
circle           0        0
triangle         6      360
rectangle       12     1080
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
```

(continues on next page)

(continued from previous page)

```
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

AlloViz.AlloViz.Elements.Nodes.sum

Nodes.**sum**(*axis: Axis | None = 0*, *skipna: bool = True*, *numeric_only: bool = False*, *min_count: int = 0*, ***kwargs*)

Return the sum of the values over the requested axis.

This is equivalent to the method `numpy.sum`.

Parameters

axis

[{index (0), columns (1)}] Axis for the function to be applied on. For *Series* this

parameter is unused and defaults to 0.

For DataFrames, specifying `axis=None` will apply the aggregation across both axes.

New in version 2.0.0.

skipna
[bool, default True] Exclude NA/null values when computing the result.

numeric_only
[bool, default False] Include only float, int, boolean columns. Not implemented for Series.

min_count
[int, default 0] The required number of valid values to perform the operation. If fewer than `min_count` non-NA values are present the result will be NA.

****kwargs**
Additional keyword arguments to be passed to the function.

Returns

Series or scalar

See also:

`Series.sum`
Return the sum.

`Series.min`
Return the minimum.

`Series.max`
Return the maximum.

`Series.idxmin`
Return the index of the minimum.

`Series.idxmax`
Return the index of the maximum.

`DataFrame.sum`
Return the sum over the requested axis.

`DataFrame.min`
Return the minimum over the requested axis.

`DataFrame.max`
Return the maximum over the requested axis.

`DataFrame.idxmin`
Return the index of the minimum over the requested axis.

`DataFrame.idxmax`
Return the index of the maximum over the requested axis.

Examples

```
>>> idx = pd.MultiIndex.from_arrays([
...     ['warm', 'warm', 'cold', 'cold'],
...     ['dog', 'falcon', 'fish', 'spider']],
...     names=['blooded', 'animal'])
>>> s = pd.Series([4, 2, 0, 8], name='legs', index=idx)
>>> s
blooded  animal
warm     dog       4
         falcon    2
cold     fish      0
         spider    8
Name: legs, dtype: int64
```

```
>>> s.sum()
14
```

By default, the sum of an empty or all-NA Series is `0`.

```
>>> pd.Series([], dtype="float64").sum()  # min_count=0 is the default
0.0
```

This can be controlled with the `min_count` parameter. For example, if you'd like the sum of an empty series to be NaN, pass `min_count=1`.

```
>>> pd.Series([], dtype="float64").sum(min_count=1)
nan
```

Thanks to the `skipna` parameter, `min_count` handles all-NA and empty series identically.

```
>>> pd.Series([np.nan]).sum()
0.0
```

```
>>> pd.Series([np.nan]).sum(min_count=1)
nan
```

AlloViz.AlloViz.Elements.Nodes.swapaxes

`Nodes.swapaxes`(*axis1: int | Literal['index', 'columns', 'rows']*, *axis2: int | Literal['index', 'columns', 'rows']*, *copy: bool | None = None*) → None

Interchange axes and swap values axes appropriately.

Deprecated since version 2.1.0: `swapaxes` is deprecated and will be removed. Please use `transpose` instead.

Returns

same as input

Examples

Please see examples for `DataFrame.transpose()`.

AlloViz.AlloViz.Elements.Nodes.swaplevel

`Nodes.`**`swaplevel`**(*i: Axis = -2*, *j: Axis = -1*, *axis: Axis = 0*) → DataFrame

Swap levels i and j in a `MultiIndex`.

Default is to swap the two innermost levels of the index.

Parameters

i, j
[int or str] Levels of the indices to be swapped. Can pass level name as string.

axis
[{0 or 'index', 1 or 'columns'}, default 0] The axis to swap levels on. 0 or 'index' for row-wise, 1 or 'columns' for column-wise.

Returns

DataFrame
DataFrame with levels swapped in MultiIndex.

Examples

```
>>> df = pd.DataFrame(
...     {"Grade": ["A", "B", "A", "C"]},
...     index=[
...         ["Final exam", "Final exam", "Coursework", "Coursework"],
...         ["History", "Geography", "History", "Geography"],
...         ["January", "February", "March", "April"],
...     ],
... )
>>> df
                                Grade
Final exam  History     January      A
            Geography   February     B
Coursework  History     March        A
            Geography   April        C
```

In the following example, we will swap the levels of the indices. Here, we will swap the levels column-wise, but levels can be swapped row-wise in a similar manner. Note that column-wise is the default behaviour. By not supplying any arguments for i and j, we swap the last and second to last indices.

```
>>> df.swaplevel()
                                Grade
Final exam  January     History      A
            February    Geography    B
Coursework  March       History      A
            April       Geography    C
```

By supplying one argument, we can choose which index to swap the last index with. We can for example swap the first index with the last one as follows.

```
>>> df.swaplevel(0)
                                  Grade
January     History     Final exam      A
February    Geography   Final exam      B
March       History     Coursework      A
April       Geography   Coursework      C
```

We can also define explicitly which indices we want to swap by supplying values for both i and j. Here, we for example swap the first and second indices.

```
>>> df.swaplevel(0, 1)
                                  Grade
History     Final exam  January         A
Geography   Final exam  February        B
History     Coursework  March           A
Geography   Coursework  April           C
```

AlloViz.AlloViz.Elements.Nodes.tail

`Nodes.tail`(*n: int = 5*) → None

Return the last *n* rows.

This function returns last *n* rows from the object based on position. It is useful for quickly verifying data, for example, after sorting or appending rows.

For negative values of *n*, this function returns all rows except the first |*n*| rows, equivalent to `df[|n|:]`.

If n is larger than the number of rows, this function returns all rows.

Parameters

n
[int, default 5] Number of rows to select.

Returns

type of caller
The last *n* rows of the caller object.

See also:

`DataFrame.head`
The first *n* rows of the caller object.

Examples

```
>>> df = pd.DataFrame({'animal': ['alligator', 'bee', 'falcon', 'lion',
...                    'monkey', 'parrot', 'shark', 'whale', 'zebra']})
>>> df
      animal
0  alligator
1        bee
2     falcon
3       lion
```

(continues on next page)

(continued from previous page)

```
4     monkey
5     parrot
6      shark
7      whale
8      zebra
```

Viewing the last 5 lines

```
>>> df.tail()
   animal
4  monkey
5  parrot
6   shark
7   whale
8   zebra
```

Viewing the last *n* lines (three in this case)

```
>>> df.tail(3)
  animal
6  shark
7  whale
8  zebra
```

For negative values of *n*

```
>>> df.tail(-3)
   animal
3    lion
4  monkey
5  parrot
6   shark
7   whale
8   zebra
```

AlloViz.AlloViz.Elements.Nodes.take

`Nodes.take`(*indices*, *axis: int | Literal['index', 'columns', 'rows'] = 0*, ***kwargs*) → None

Return the elements in the given *positional* indices along an axis.

This means that we are not indexing according to actual values in the index attribute of the object. We are indexing according to the actual position of the element in the object.

Parameters

indices
[array-like] An array of ints indicating which positions to take.

axis
[{0 or 'index', 1 or 'columns', None}, default 0] The axis on which to select elements. `0` means that we are selecting rows, `1` means that we are selecting columns. For *Series* this parameter is unused and defaults to 0.

****kwargs**
For compatibility with `numpy.take()`. Has no effect on the output.

Returns

same type as caller
An array-like containing the elements taken from the object.

See also:

`DataFrame.loc`
Select a subset of a DataFrame by labels.

`DataFrame.iloc`
Select a subset of a DataFrame by positions.

`numpy.take`
Take elements from an array along an axis.

Examples

```
>>> df = pd.DataFrame([('falcon', 'bird', 389.0),
...                    ('parrot', 'bird', 24.0),
...                    ('lion', 'mammal', 80.5),
...                    ('monkey', 'mammal', np.nan)],
...                   columns=['name', 'class', 'max_speed'],
...                   index=[0, 2, 3, 1])
>>> df
     name   class  max_speed
0  falcon    bird      389.0
2  parrot    bird       24.0
3    lion  mammal       80.5
1  monkey  mammal        NaN
```

Take elements at positions 0 and 3 along the axis 0 (default).

Note how the actual indices selected (0 and 1) do not correspond to our selected indices 0 and 3. That's because we are selecting the 0th and 3rd rows, not rows whose indices equal 0 and 3.

```
>>> df.take([0, 3])
     name   class  max_speed
0  falcon    bird      389.0
1  monkey  mammal        NaN
```

Take elements at indices 1 and 2 along the axis 1 (column selection).

```
>>> df.take([1, 2], axis=1)
    class  max_speed
0    bird      389.0
2    bird       24.0
3  mammal       80.5
1  mammal        NaN
```

We may take elements using negative integers for positive indices, starting from the end of the object, just like with Python lists.

```
>>> df.take([-1, -2])
     name   class  max_speed
1  monkey  mammal        NaN
3    lion  mammal       80.5
```

AlloViz.AlloViz.Elements.Nodes.to_clipboard

`Nodes.to_clipboard`(*excel: bool = True*, *sep: str | None = None*, ***kwargs*) → None

Copy object to the system clipboard.

Write a text representation of object to the system clipboard. This can be pasted into Excel, for example.

Parameters

excel
[bool, default True] Produce output in a csv format for easy pasting into excel.

- True, use the provided separator for csv pasting.
- False, write a string representation of the object to the clipboard.

sep
[str, default `'\t'`] Field delimiter.

****kwargs**
These parameters will be passed to DataFrame.to_csv.

See also:

`DataFrame.to_csv`
Write a DataFrame to a comma-separated values (csv) file.

`read_clipboard`
Read text from clipboard and pass to read_csv.

Notes

Requirements for your platform.

- Linux : *xclip*, or *xsel* (with *PyQt4* modules)
- Windows : none
- macOS : none

This method uses the processes developed for the package *pyperclip*. A solution to render any output string format is given in the examples.

Examples

Copy the contents of a DataFrame to the clipboard.

```
>>> df = pd.DataFrame([[1, 2, 3], [4, 5, 6]], columns=['A', 'B', 'C'])
```

```
>>> df.to_clipboard(sep=',')
... # Wrote the following to the system clipboard:
... # ,A,B,C
... # 0,1,2,3
... # 1,4,5,6
```

We can omit the index by passing the keyword *index* and setting it to false.

```
>>> df.to_clipboard(sep=',', index=False)
... # Wrote the following to the system clipboard:
... # A,B,C
... # 1,2,3
... # 4,5,6
```

Using the original *pyperclip* package for any string output format.

```
import pyperclip
html = df.style.to_html()
pyperclip.copy(html)
```

AlloViz.AlloViz.Elements.Nodes.to_csv

Nodes.**to_csv**(*path_or_buf: FilePath | WriteBuffer[bytes] | WriteBuffer[str] | None = None, sep: str = ',', na_rep: str = '', float_format: str | Callable | None = None, columns: Sequence[Hashable] | None = None, header: bool_t | list[str] = True, index: bool_t = True, index_label: IndexLabel | None = None, mode: str = 'w', encoding: str | None = None, compression: CompressionOptions = 'infer', quoting: int | None = None, quotechar: str = '"', lineterminator: str | None = None, chunksize: int | None = None, date_format: str | None = None, doublequote: bool_t = True, escapechar: str | None = None, decimal: str = '.', errors: OpenFileErrors = 'strict', storage_options: StorageOptions | None = None*) → str | None

Write object to a comma-separated values (csv) file.

Parameters

path_or_buf
: [str, path object, file-like object, or None, default None] String, path object (implementing os.PathLike[str]), or file-like object implementing a write() function. If None, the result is returned as a string. If a non-binary file object is passed, it should be opened with *newline=''*, disabling universal newlines. If a binary file object is passed, *mode* might need to contain a *'b'*.

  Changed in version 1.2.0: Support for binary file objects was introduced.

sep
: [str, default ','] String of length 1. Field delimiter for the output file.

na_rep
: [str, default ''] Missing data representation.

float_format
[str, Callable, default None] Format string for floating point numbers. If a Callable is given, it takes precedence over other numeric formatting parameters, like decimal.

columns
[sequence, optional] Columns to write.

header
[bool or list of str, default True] Write out the column names. If a list of strings is given it is assumed to be aliases for the column names.

index
[bool, default True] Write row names (index).

index_label
[str or sequence, or False, default None] Column label for index column(s) if desired. If None is given, and *header* and *index* are True, then the index names are used. A sequence should be given if the object uses MultiIndex. If False do not print fields for index names. Use index_label=False for easier importing in R.

mode
[{'w', 'x', 'a'}, default 'w'] Forwarded to either *open(mode=)* or *fsspec.open(mode=)* to control the file opening. Typical values include:

- 'w', truncate the file first.
- 'x', exclusive creation, failing if the file already exists.
- 'a', append to the end of file if it exists.

encoding
[str, optional] A string representing the encoding to use in the output file, defaults to 'utf-8'. *encoding* is not supported if *path_or_buf* is a non-binary file object.

compression
[str or dict, default 'infer'] For on-the-fly compression of the output data. If 'infer' and 'path_or_buf' is path-like, then detect compression from the following extensions: '.gz', '.bz2', '.zip', '.xz', '.zst', '.tar', '.tar.gz', '.tar.xz' or '.tar.bz2' (otherwise no compression). Set to `None` for no compression. Can also be a dict with key `'method'` set to one of {`'zip'`, `'gzip'`, `'bz2'`, `'zstd'`, `'xz'`, `'tar'`} and other key-value pairs are forwarded to `zipfile.ZipFile`, `gzip.GzipFile`, `bz2.BZ2File`, `zstandard.ZstdCompressor`, `lzma.LZMAFile` or `tarfile.TarFile`, respectively. As an example, the following could be passed for faster compression and to create a reproducible gzip archive: `compression={'method': 'gzip', 'compresslevel': 1, 'mtime': 1}`.

New in version 1.5.0: Added support for *.tar* files.

May be a dict with key 'method' as compression mode and other entries as additional compression options if compression mode is 'zip'.

Passing compression options as keys in dict is supported for compression modes 'gzip', 'bz2', 'zstd', and 'zip'.

Changed in version 1.2.0: Compression is supported for binary file objects.

Changed in version 1.2.0: Previous versions forwarded dict entries for 'gzip' to *gzip.open* instead of *gzip.GzipFile* which prevented setting *mtime*.

quoting
[optional constant from csv module] Defaults to csv.QUOTE_MINIMAL. If

you have set a *float_format* then floats are converted to strings and thus csv.QUOTE_NONNUMERIC will treat them as non-numeric.

quotechar
[str, default '"'] String of length 1. Character used to quote fields.

lineterminator
[str, optional] The newline character or character sequence to use in the output file. Defaults to *os.linesep*, which depends on the OS in which this method is called ('\n' for linux, '\r\n' for Windows, i.e.).

Changed in version 1.5.0: Previously was line_terminator, changed for consistency with read_csv and the standard library 'csv' module.

chunksize
[int or None] Rows to write at a time.

date_format
[str, default None] Format string for datetime objects.

doublequote
[bool, default True] Control quoting of *quotechar* inside a field.

escapechar
[str, default None] String of length 1. Character used to escape *sep* and *quotechar* when appropriate.

decimal
[str, default '.'] Character recognized as decimal separator. E.g. use ',' for European data.

errors
[str, default 'strict'] Specifies how encoding and decoding errors are to be handled. See the errors argument for `open()` for a full list of options.

storage_options
[dict, optional] Extra options that make sense for a particular storage connection, e.g. host, port, username, password, etc. For HTTP(S) URLs the key-value pairs are forwarded to `urllib.request.Request` as header options. For other URLs (e.g. starting with "s3://", and "gcs://") the key-value pairs are forwarded to `fsspec.open`. Please see `fsspec` and `urllib` for more details, and for more examples on storage options refer here.

New in version 1.2.0.

Returns

None or str
If path_or_buf is None, returns the resulting csv format as a string. Otherwise returns None.

See also:

`read_csv`
Load a CSV file into a DataFrame.

`to_excel`
Write DataFrame to an Excel file.

Examples

```
>>> df = pd.DataFrame({'name': ['Raphael', 'Donatello'],
...                    'mask': ['red', 'purple'],
...                    'weapon': ['sai', 'bo staff']})
>>> df.to_csv(index=False)
'name,mask,weapon\nRaphael,red,sai\nDonatello,purple,bo staff\n'
```

Create 'out.zip' containing 'out.csv'

```
>>> compression_opts = dict(method='zip',
...                         archive_name='out.csv')
>>> df.to_csv('out.zip', index=False,
...           compression=compression_opts)
```

To write a csv file to a new folder or nested folder you will first need to create it using either Pathlib or os:

```
>>> from pathlib import Path
>>> filepath = Path('folder/subfolder/out.csv')
>>> filepath.parent.mkdir(parents=True, exist_ok=True)
>>> df.to_csv(filepath)
```

```
>>> import os
>>> os.makedirs('folder/subfolder', exist_ok=True)
>>> df.to_csv('folder/subfolder/out.csv')
```

AlloViz.AlloViz.Elements.Nodes.to_dict

Nodes.**to_dict**(*orient: ~typing.Literal['dict', 'list', 'series', 'split', 'tight', 'records', 'index'] = 'dict', into: type[dict] = <class 'dict'>, index: bool = True*) → dict | list[dict]

Convert the DataFrame to a dictionary.

The type of the key-value pairs can be customized with the parameters (see below).

Parameters

orient

[str {'dict', 'list', 'series', 'split', 'tight', 'records', 'index'}] Determines the type of the values of the dictionary.

- 'dict' (default) : dict like {column -> {index -> value}}
- 'list' : dict like {column -> [values]}
- 'series' : dict like {column -> Series(values)}
- 'split' : dict like {'index' -> [index], 'columns' -> [columns], 'data' -> [values]}
- 'tight' : dict like {'index' -> [index], 'columns' -> [columns], 'data' -> [values], 'index_names' -> [index.names], 'column_names' -> [column.names]}
- 'records' : list like [{column -> value}, … , {column -> value}]
- 'index' : dict like {index -> {column -> value}}

New in version 1.4.0: 'tight' as an allowed value for the `orient` argument

into
: [class, default dict] The collections.abc.Mapping subclass used for all Mappings in the return value. Can be the actual class or an empty instance of the mapping type you want. If you want a collections.defaultdict, you must pass it initialized.

index
: [bool, default True] Whether to include the index item (and index_names item if *orient* is 'tight') in the returned dictionary. Can only be `False` when *orient* is 'split' or 'tight'.

  New in version 2.0.0.

Returns

dict, list or collections.abc.Mapping
: Return a collections.abc.Mapping object representing the DataFrame. The resulting transformation depends on the *orient* parameter.

See also:

`DataFrame.from_dict`
: Create a DataFrame from a dictionary.

`DataFrame.to_json`
: Convert a DataFrame to JSON format.

Examples

```
>>> df = pd.DataFrame({'col1': [1, 2],
...                    'col2': [0.5, 0.75]},
...                   index=['row1', 'row2'])
>>> df
      col1  col2
row1     1  0.50
row2     2  0.75
>>> df.to_dict()
{'col1': {'row1': 1, 'row2': 2}, 'col2': {'row1': 0.5, 'row2': 0.75}}
```

You can specify the return orientation.

```
>>> df.to_dict('series')
{'col1': row1    1
         row2    2
Name: col1, dtype: int64,
'col2': row1    0.50
        row2    0.75
Name: col2, dtype: float64}
```

```
>>> df.to_dict('split')
{'index': ['row1', 'row2'], 'columns': ['col1', 'col2'],
 'data': [[1, 0.5], [2, 0.75]]}
```

```
>>> df.to_dict('records')
[{'col1': 1, 'col2': 0.5}, {'col1': 2, 'col2': 0.75}]
```

```
>>> df.to_dict('index')
{'row1': {'col1': 1, 'col2': 0.5}, 'row2': {'col1': 2, 'col2': 0.75}}
```

```
>>> df.to_dict('tight')
{'index': ['row1', 'row2'], 'columns': ['col1', 'col2'],
 'data': [[1, 0.5], [2, 0.75]], 'index_names': [None], 'column_names': [None]}
```

You can also specify the mapping type.

```
>>> from collections import OrderedDict, defaultdict
>>> df.to_dict(into=OrderedDict)
OrderedDict([('col1', OrderedDict([('row1', 1), ('row2', 2)])),
             ('col2', OrderedDict([('row1', 0.5), ('row2', 0.75)]))])
```

If you want a *defaultdict*, you need to initialize it:

```
>>> dd = defaultdict(list)
>>> df.to_dict('records', into=dd)
[defaultdict(<class 'list'>, {'col1': 1, 'col2': 0.5}),
 defaultdict(<class 'list'>, {'col1': 2, 'col2': 0.75})]
```

AlloViz.AlloViz.Elements.Nodes.to_excel

`Nodes.to_excel`(*excel_writer: FilePath | WriteExcelBuffer | ExcelWriter*, *sheet_name: str = 'Sheet1'*, *na_rep: str = ''*, *float_format: str | None = None*, *columns: Sequence[Hashable] | None = None*, *header: Sequence[Hashable] | bool_t = True*, *index: bool_t = True*, *index_label: IndexLabel | None = None*, *startrow: int = 0*, *startcol: int = 0*, *engine: Literal['openpyxl', 'xlsxwriter'] | None = None*, *merge_cells: bool_t = True*, *inf_rep: str = 'inf'*, *freeze_panes: tuple[int, int] | None = None*, *storage_options: StorageOptions | None = None*, *engine_kwargs: dict[str, Any] | None = None*) → None

Write object to an Excel sheet.

To write a single object to an Excel .xlsx file it is only necessary to specify a target file name. To write to multiple sheets it is necessary to create an *ExcelWriter* object with a target file name, and specify a sheet in the file to write to.

Multiple sheets may be written to by specifying unique *sheet_name*. With all data written to the file it is necessary to save the changes. Note that creating an *ExcelWriter* object with a file name that already exists will result in the contents of the existing file being erased.

Parameters

excel_writer
[path-like, file-like, or ExcelWriter object] File path or existing ExcelWriter.

sheet_name
[str, default 'Sheet1'] Name of sheet which will contain DataFrame.

na_rep
[str, default ''] Missing data representation.

float_format
[str, optional] Format string for floating point numbers. For example `float_format="%.2f"` will format 0.1234 to 0.12.

columns
[sequence or list of str, optional] Columns to write.

header
[bool or list of str, default True] Write out the column names. If a list of string is given it is assumed to be aliases for the column names.

index
[bool, default True] Write row names (index).

index_label
[str or sequence, optional] Column label for index column(s) if desired. If not specified, and *header* and *index* are True, then the index names are used. A sequence should be given if the DataFrame uses MultiIndex.

startrow
[int, default 0] Upper left cell row to dump data frame.

startcol
[int, default 0] Upper left cell column to dump data frame.

engine
[str, optional] Write engine to use, 'openpyxl' or 'xlsxwriter'. You can also set this via the options `io.excel.xlsx.writer` or `io.excel.xlsm.writer`.

merge_cells
[bool, default True] Write MultiIndex and Hierarchical Rows as merged cells.

inf_rep
[str, default 'inf'] Representation for infinity (there is no native representation for infinity in Excel).

freeze_panes
[tuple of int (length 2), optional] Specifies the one-based bottommost row and rightmost column that is to be frozen.

storage_options
[dict, optional] Extra options that make sense for a particular storage connection, e.g. host, port, username, password, etc. For HTTP(S) URLs the key-value pairs are forwarded to `urllib.request.Request` as header options. For other URLs (e.g. starting with "s3://", and "gcs://") the key-value pairs are forwarded to `fsspec.open`. Please see `fsspec` and `urllib` for more details, and for more examples on storage options refer here.

New in version 1.2.0.

engine_kwargs
[dict, optional] Arbitrary keyword arguments passed to excel engine.

See also:

`to_csv`
Write DataFrame to a comma-separated values (csv) file.

`ExcelWriter`
Class for writing DataFrame objects into excel sheets.

`read_excel`
Read an Excel file into a pandas DataFrame.

`read_csv`
Read a comma-separated values (csv) file into DataFrame.

io.formats.style.Styler.to_excel
Add styles to Excel sheet.

Notes

For compatibility with `to_csv()`, to_excel serializes lists and dicts to strings before writing.

Once a workbook has been saved it is not possible to write further data without rewriting the whole workbook.

Examples

Create, write to and save a workbook:

```
>>> df1 = pd.DataFrame([['a', 'b'], ['c', 'd']],
...                    index=['row 1', 'row 2'],
...                    columns=['col 1', 'col 2'])
>>> df1.to_excel("output.xlsx")
```

To specify the sheet name:

```
>>> df1.to_excel("output.xlsx",
...              sheet_name='Sheet_name_1')
```

If you wish to write to more than one sheet in the workbook, it is necessary to specify an ExcelWriter object:

```
>>> df2 = df1.copy()
>>> with pd.ExcelWriter('output.xlsx') as writer:
...     df1.to_excel(writer, sheet_name='Sheet_name_1')
...     df2.to_excel(writer, sheet_name='Sheet_name_2')
```

ExcelWriter can also be used to append to an existing Excel file:

```
>>> with pd.ExcelWriter('output.xlsx',
...                     mode='a') as writer:
...     df1.to_excel(writer, sheet_name='Sheet_name_3')
```

To set the library that is used to write the Excel file, you can pass the *engine* keyword (the default engine is automatically chosen depending on the file extension):

```
>>> df1.to_excel('output1.xlsx', engine='xlsxwriter')
```

AlloViz.AlloViz.Elements.Nodes.to_feather

`Nodes.`**`to_feather`**(*path: FilePath | WriteBuffer[bytes]*, ***kwargs*) → None

Write a DataFrame to the binary Feather format.

Parameters

path
[str, path object, file-like object] String, path object (implementing `os.PathLike[str]`), or file-like object implementing a binary `write()` function. If

a string or a path, it will be used as Root Directory path when writing a partitioned dataset.

****kwargs**
Additional keywords passed to `pyarrow.feather.write_feather()`. This includes the *compression*, *compression_level*, *chunksize* and *version* keywords.

Notes

This function writes the dataframe as a feather file. Requires a default index. For saving the DataFrame with your custom index use a method that supports custom indices e.g. *to_parquet*.

Examples

```
>>> df = pd.DataFrame([[1, 2, 3], [4, 5, 6]])
>>> df.to_feather("file.feather")
```

AlloViz.AlloViz.Elements.Nodes.to_gbq

`Nodes.to_gbq`(*destination_table: str*, *project_id: str | None = None*, *chunksize: int | None = None*, *reauth: bool = False*, *if_exists: ToGbqIfexist = 'fail'*, *auth_local_webserver: bool = True*, *table_schema: list[dict[str, str]] | None = None*, *location: str | None = None*, *progress_bar: bool = True*, *credentials=None*) → None

Write a DataFrame to a Google BigQuery table.

This function requires the pandas-gbq package.

See the How to authenticate with Google BigQuery guide for authentication instructions.

Parameters

destination_table
[str] Name of table to be written, in the form `dataset.tablename`.

project_id
[str, optional] Google BigQuery Account project ID. Optional when available from the environment.

chunksize
[int, optional] Number of rows to be inserted in each chunk from the dataframe. Set to `None` to load the whole dataframe at once.

reauth
[bool, default False] Force Google BigQuery to re-authenticate the user. This is useful if multiple accounts are used.

if_exists
[str, default 'fail'] Behavior when the destination table exists. Value can be one of:

`'fail'`
If table exists raise pandas_gbq.gbq.TableCreationError.

`'replace'`
If table exists, drop it, recreate it, and insert data.

`'append'`
If table exists, insert data. Create if does not exist.

auth_local_webserver
[bool, default True] Use the local webserver flow instead of the console flow when getting user credentials.

New in version 0.2.0 of pandas-gbq.

Changed in version 1.5.0: Default value is changed to `True`. Google has deprecated the `auth_local_webserver = False` "out of band" (copy-paste) flow.

table_schema
[list of dicts, optional] List of BigQuery table fields to which according DataFrame columns conform to, e.g. `[{'name': 'col1', 'type': 'STRING'},...]`. If schema is not provided, it will be generated according to dtypes of DataFrame columns. See BigQuery API documentation on available names of a field.

New in version 0.3.1 of pandas-gbq.

location
[str, optional] Location where the load job should run. See the BigQuery locations documentation for a list of available locations. The location must match that of the target dataset.

New in version 0.5.0 of pandas-gbq.

progress_bar
[bool, default True] Use the library *tqdm* to show the progress bar for the upload, chunk by chunk.

New in version 0.5.0 of pandas-gbq.

credentials
[google.auth.credentials.Credentials, optional] Credentials for accessing Google APIs. Use this parameter to override default credentials, such as to use Compute Engine `google.auth.compute_engine.Credentials` or Service Account `google.oauth2.service_account.Credentials` directly.

New in version 0.8.0 of pandas-gbq.

See also:

`pandas_gbq.to_gbq`
This function in the pandas-gbq library.

`read_gbq`
Read a DataFrame from Google BigQuery.

Examples

Example taken from Google BigQuery documentation

```
>>> project_id = "my-project"
>>> table_id = 'my_dataset.my_table'
>>> df = pd.DataFrame({
...                    "my_string": ["a", "b", "c"],
...                    "my_int64": [1, 2, 3],
...                    "my_float64": [4.0, 5.0, 6.0],
...                    "my_bool1": [True, False, True],
...                    "my_bool2": [False, True, False],
...                    "my_dates": pd.date_range("now", periods=3),
```

(continues on next page)

(continued from previous page)

```
...                     }
...                     )
```

```
>>> df.to_gbq(table_id, project_id=project_id)
```

AlloViz.AlloViz.Elements.Nodes.to_hdf

Nodes.**to_hdf**(*path_or_buf: FilePath | HDFStore*, *key: str*, *mode: Literal['a', 'w', 'r+'] = 'a'*, *complevel: int | None = None*, *complib: Literal['zlib', 'lzo', 'bzip2', 'blosc'] | None = None*, *append: bool_t = False*, *format: Literal['fixed', 'table'] | None = None*, *index: bool_t = True*, *min_itemsize: int | dict[str, int] | None = None*, *nan_rep=None*, *dropna: bool_t | None = None*, *data_columns: Literal[True] | list[str] | None = None*, *errors: OpenFileErrors = 'strict'*, *encoding: str = 'UTF-8'*) → None

Write the contained data to an HDF5 file using HDFStore.

Hierarchical Data Format (HDF) is self-describing, allowing an application to interpret the structure and contents of a file with no outside information. One HDF file can hold a mix of related objects which can be accessed as a group or as individual objects.

In order to add another DataFrame or Series to an existing HDF file please use append mode and a different a key.

Warning: One can store a subclass of `DataFrame` or `Series` to HDF5, but the type of the subclass is lost upon storing.

For more information see the user guide.

Parameters

path_or_buf
[str or pandas.HDFStore] File path or HDFStore object.

key
[str] Identifier for the group in the store.

mode
[{'a', 'w', 'r+'}, default 'a'] Mode to open file:

- 'w': write, a new file is created (an existing file with the same name would be deleted).
- 'a': append, an existing file is opened for reading and writing, and if the file does not exist it is created.
- 'r+': similar to 'a', but the file must already exist.

complevel
[{0-9}, default None] Specifies a compression level for data. A value of 0 or None disables compression.

complib
[{'zlib', 'lzo', 'bzip2', 'blosc'}, default 'zlib'] Specifies the compression library to be used. These additional compressors for Blosc are supported (default if no compressor specified: 'blosc:blosclz'): {'blosc:blosclz', 'blosc:lz4', 'blosc:lz4hc',

'blosc:snappy', 'blosc:zlib', 'blosc:zstd'}. Specifying a compression library which is not available issues a ValueError.

append
[bool, default False] For Table formats, append the input data to the existing.

format
[{'fixed', 'table', None}, default 'fixed'] Possible values:

- 'fixed': Fixed format. Fast writing/reading. Not-appendable, nor searchable.
- 'table': Table format. Write as a PyTables Table structure which may perform worse but allow more flexible operations like searching / selecting subsets of the data.
- If None, pd.get_option('io.hdf.default_format') is checked, followed by fallback to "fixed".

index
[bool, default True] Write DataFrame index as a column.

min_itemsize
[dict or int, optional] Map column names to minimum string sizes for columns.

nan_rep
[Any, optional] How to represent null values as str. Not allowed with append=True.

dropna
[bool, default False, optional] Remove missing values.

data_columns
[list of columns or True, optional] List of columns to create as indexed data columns for on-disk queries, or True to use all columns. By default only the axes of the object are indexed. See Query via data columns. for more information. Applicable only to format='table'.

errors
[str, default 'strict'] Specifies how encoding and decoding errors are to be handled. See the errors argument for `open()` for a full list of options.

encoding
[str, default "UTF-8"]

See also:

`read_hdf`
Read from HDF file.

`DataFrame.to_orc`
Write a DataFrame to the binary orc format.

`DataFrame.to_parquet`
Write a DataFrame to the binary parquet format.

`DataFrame.to_sql`
Write to a SQL table.

`DataFrame.to_feather`
Write out feather-format for DataFrames.

`DataFrame.to_csv`
Write out to a csv file.

Examples

```
>>> df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]},
...                   index=['a', 'b', 'c'])
>>> df.to_hdf('data.h5', key='df', mode='w')
```

We can add another object to the same file:

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s.to_hdf('data.h5', key='s')
```

Reading from HDF file:

```
>>> pd.read_hdf('data.h5', 'df')
A  B
a  1  4
b  2  5
c  3  6
>>> pd.read_hdf('data.h5', 's')
0    1
1    2
2    3
3    4
dtype: int64
```

AlloViz.AlloViz.Elements.Nodes.to_html

Nodes.**to_html**(*buf: FilePath | WriteBuffer[str] | None = None*, *columns: Axes | None = None*, *col_space: ColspaceArgType | None = None*, *header: bool = True*, *index: bool = True*, *na_rep: str = 'NaN'*, *formatters: FormattersType | None = None*, *float_format: FloatFormatType | None = None*, *sparsify: bool | None = None*, *index_names: bool = True*, *justify: str | None = None*, *max_rows: int | None = None*, *max_cols: int | None = None*, *show_dimensions: bool | str = False*, *decimal: str = '.'*, *bold_rows: bool = True*, *classes: str | list | tuple | None = None*, *escape: bool = True*, *notebook: bool = False*, *border: int | bool | None = None*, *table_id: str | None = None*, *render_links: bool = False*, *encoding: str | None = None*) → str | None

Render a DataFrame as an HTML table.

Parameters

buf
: [str, Path or StringIO-like, optional, default None] Buffer to write to. If None, the output is returned as a string.

columns
: [array-like, optional, default None] The subset of columns to write. Writes all columns by default.

col_space
: [str or int, list or dict of int or str, optional] The minimum width of each column in CSS length units. An int is assumed to be px units..

header
: [bool, optional] Whether to print column labels, default True.

index
[bool, optional, default True] Whether to print index (row) labels.

na_rep
[str, optional, default 'NaN'] String representation of `NaN` to use.

formatters
[list, tuple or dict of one-param. functions, optional] Formatter functions to apply to columns' elements by position or name. The result of each function must be a unicode string. List/tuple must be of length equal to the number of columns.

float_format
[one-parameter function, optional, default None] Formatter function to apply to columns' elements if they are floats. This function must return a unicode string and will be applied only to the non-`NaN` elements, with `NaN` being handled by `na_rep`.

Changed in version 1.2.0.

sparsify
[bool, optional, default True] Set to False for a DataFrame with a hierarchical index to print every multiindex key at each row.

index_names
[bool, optional, default True] Prints the names of the indexes.

justify
[str, default None] How to justify the column labels. If None uses the option from the print configuration (controlled by set_option), 'right' out of the box. Valid values are

- left
- right
- center
- justify
- justify-all
- start
- end
- inherit
- match-parent
- initial
- unset.

max_rows
[int, optional] Maximum number of rows to display in the console.

max_cols
[int, optional] Maximum number of columns to display in the console.

show_dimensions
[bool, default False] Display DataFrame dimensions (number of rows by number of columns).

decimal
[str, default '.'] Character recognized as decimal separator, e.g. ',' in Europe.

bold_rows
[bool, default True] Make the row labels bold in the output.

classes
[str or list or tuple, default None] CSS class(es) to apply to the resulting html table.

escape
[bool, default True] Convert the characters <, >, and & to HTML-safe sequences.

notebook
[{True, False}, default False] Whether the generated HTML is for IPython Notebook.

border
[int] A `border=border` attribute is included in the opening *<table>* tag. Default `pd.options.display.html.border`.

table_id
[str, optional] A css id is included in the opening *<table>* tag if specified.

render_links
[bool, default False] Convert URLs to HTML links.

encoding
[str, default "utf-8"] Set character encoding.

Returns

str or None
If buf is None, returns the result as a string. Otherwise returns None.

See also:

to_string
Convert DataFrame to a string.

Examples

```
>>> df = pd.DataFrame(data={'col1': [1, 2], 'col2': [4, 3]})
>>> html_string = '''<table border="1" class="dataframe">
...   <thead>
...     <tr style="text-align: right;">
...       <th></th>
...       <th>col1</th>
...       <th>col2</th>
...     </tr>
...   </thead>
...   <tbody>
...     <tr>
...       <th>0</th>
...       <td>1</td>
...       <td>4</td>
...     </tr>
...     <tr>
...       <th>1</th>
...       <td>2</td>
...       <td>3</td>
```

(continues on next page)

(continued from previous page)

```
...     </tr>
...   </tbody>
... </table>'''
>>> assert html_string == df.to_html()
```

AlloViz.AlloViz.Elements.Nodes.to_json

Nodes.**to_json**(*path_or_buf: FilePath | WriteBuffer[bytes] | WriteBuffer[str] | None = None*, *orient: Literal['split', 'records', 'index', 'table', 'columns', 'values'] | None = None*, *date_format: str | None = None*, *double_precision: int = 10*, *force_ascii: bool_t = True*, *date_unit: TimeUnit = 'ms'*, *default_handler: Callable[[Any], JSONSerializable] | None = None*, *lines: bool_t = False*, *compression: CompressionOptions = 'infer'*, *index: bool_t | None = None*, *indent: int | None = None*, *storage_options: StorageOptions | None = None*, *mode: Literal['a', 'w'] = 'w'*) → str | None

Convert the object to a JSON string.

Note NaN's and None will be converted to null and datetime objects will be converted to UNIX timestamps.

Parameters

path_or_buf
[str, path object, file-like object, or None, default None] String, path object (implementing os.PathLike[str]), or file-like object implementing a write() function. If None, the result is returned as a string.

orient
[str] Indication of expected JSON string format.

- Series:
  - default is 'index'
  - allowed values are: {'split', 'records', 'index', 'table'}.
- DataFrame:
  - default is 'columns'
  - allowed values are: {'split', 'records', 'index', 'columns', 'values', 'table'}.
- The format of the JSON string:
  - 'split' : dict like {'index' -> [index], 'columns' -> [columns], 'data' -> [values]}
  - 'records' : list like [{column -> value}, … , {column -> value}]
  - 'index' : dict like {index -> {column -> value}}
  - 'columns' : dict like {column -> {index -> value}}
  - 'values' : just the values array
  - 'table' : dict like {'schema': {schema}, 'data': {data}}

  Describing the data, where data component is like `orient='records'`.

date_format
[{None, 'epoch', 'iso'}] Type of date conversion. 'epoch' = epoch milliseconds,

'iso' = ISO8601. The default depends on the *orient*. For `orient='table'`, the default is 'iso'. For all other orients, the default is 'epoch'.

double_precision
[int, default 10] The number of decimal places to use when encoding floating point values. The possible maximal value is 15. Passing double_precision greater than 15 will raise a ValueError.

force_ascii
[bool, default True] Force encoded string to be ASCII.

date_unit
[str, default 'ms' (milliseconds)] The time unit to encode to, governs timestamp and ISO8601 precision. One of 's', 'ms', 'us', 'ns' for second, millisecond, microsecond, and nanosecond respectively.

default_handler
[callable, default None] Handler to call if object cannot otherwise be converted to a suitable format for JSON. Should receive a single argument which is the object to convert and return a serialisable object.

lines
[bool, default False] If 'orient' is 'records' write out line-delimited json format. Will throw ValueError if incorrect 'orient' since others are not list-like.

compression
[str or dict, default 'infer'] For on-the-fly compression of the output data. If 'infer' and 'path_or_buf' is path-like, then detect compression from the following extensions: '.gz', '.bz2', '.zip', '.xz', '.zst', '.tar', '.tar.gz', '.tar.xz' or '.tar.bz2' (otherwise no compression). Set to `None` for no compression. Can also be a dict with key `'method'` set to one of {`'zip'`, `'gzip'`, `'bz2'`, `'zstd'`, `'xz'`, `'tar'`} and other key-value pairs are forwarded to `zipfile.ZipFile`, `gzip.GzipFile`, `bz2.BZ2File`, `zstandard.ZstdCompressor`, `lzma.LZMAFile` or `tarfile.TarFile`, respectively. As an example, the following could be passed for faster compression and to create a reproducible gzip archive: `compression={'method': 'gzip', 'compresslevel': 1, 'mtime': 1}`.

New in version 1.5.0: Added support for *.tar* files.

Changed in version 1.4.0: Zstandard support.

index
[bool or None, default None] The index is only used when 'orient' is 'split', 'index', 'column', or 'table'. Of these, 'index' and 'column' do not support *index=False*.

indent
[int, optional] Length of whitespace used to indent each record.

storage_options
[dict, optional] Extra options that make sense for a particular storage connection, e.g. host, port, username, password, etc. For HTTP(S) URLs the key-value pairs are forwarded to `urllib.request.Request` as header options. For other URLs (e.g. starting with "s3://", and "gcs://") the key-value pairs are forwarded to `fsspec.open`. Please see `fsspec` and `urllib` for more details, and for more examples on storage options refer here.

New in version 1.2.0.

mode
[str, default 'w' (writing)] Specify the IO mode for output when supplying a

path_or_buf. Accepted args are 'w' (writing) and 'a' (append) only. mode='a' is only supported when lines is True and orient is 'records'.

Returns

None or str
If path_or_buf is None, returns the resulting json format as a string. Otherwise returns None.

See also:

`read_json`
Convert a JSON string to pandas object.

Notes

The behavior of `indent=0` varies from the stdlib, which does not indent the output but does insert newlines. Currently, `indent=0` and the default `indent=None` are equivalent in pandas, though this may change in a future release.

`orient='table'` contains a 'pandas_version' field under 'schema'. This stores the version of *pandas* used in the latest revision of the schema.

Examples

```
>>> from json import loads, dumps
>>> df = pd.DataFrame(
...     [["a", "b"], ["c", "d"]],
...     index=["row 1", "row 2"],
...     columns=["col 1", "col 2"],
... )
```

```
>>> result = df.to_json(orient="split")
>>> parsed = loads(result)
>>> dumps(parsed, indent=4)
{
    "columns": [
        "col 1",
        "col 2"
    ],
    "index": [
        "row 1",
        "row 2"
    ],
    "data": [
        [
            "a",
            "b"
        ],
        [
            "c",
            "d"
        ]
```

(continues on next page)

(continued from previous page)

```
    ]
}
```

Encoding/decoding a Dataframe using `'records'` formatted JSON. Note that index labels are not preserved with this encoding.

```
>>> result = df.to_json(orient="records")
>>> parsed = loads(result)
>>> dumps(parsed, indent=4)
[
    {
        "col 1": "a",
        "col 2": "b"
    },
    {
        "col 1": "c",
        "col 2": "d"
    }
]
```

Encoding/decoding a Dataframe using `'index'` formatted JSON:

```
>>> result = df.to_json(orient="index")
>>> parsed = loads(result)
>>> dumps(parsed, indent=4)
{
    "row 1": {
        "col 1": "a",
        "col 2": "b"
    },
    "row 2": {
        "col 1": "c",
        "col 2": "d"
    }
}
```

Encoding/decoding a Dataframe using `'columns'` formatted JSON:

```
>>> result = df.to_json(orient="columns")
>>> parsed = loads(result)
>>> dumps(parsed, indent=4)
{
    "col 1": {
        "row 1": "a",
        "row 2": "c"
    },
    "col 2": {
        "row 1": "b",
        "row 2": "d"
    }
}
```

Encoding/decoding a Dataframe using `'values'` formatted JSON:

```
>>> result = df.to_json(orient="values")
>>> parsed = loads(result)
>>> dumps(parsed, indent=4)
[
    [
        "a",
        "b"
    ],
    [
        "c",
        "d"
    ]
]
```

Encoding with Table Schema:

```
>>> result = df.to_json(orient="table")
>>> parsed = loads(result)
>>> dumps(parsed, indent=4)
{
    "schema": {
        "fields": [
            {
                "name": "index",
                "type": "string"
            },
            {
                "name": "col 1",
                "type": "string"
            },
            {
                "name": "col 2",
                "type": "string"
            }
        ],
        "primaryKey": [
            "index"
        ],
        "pandas_version": "1.4.0"
    },
    "data": [
        {
            "index": "row 1",
            "col 1": "a",
            "col 2": "b"
        },
        {
            "index": "row 2",
            "col 1": "c",
            "col 2": "d"
        }
    ]
}
```

AlloViz.AlloViz.Elements.Nodes.to_latex

Nodes.**to_latex**(*buf: FilePath | WriteBuffer[str] | None = None*, *columns: Sequence[Hashable] | None = None*, *header: bool_t | list[str] = True*, *index: bool_t = True*, *na_rep: str = 'NaN'*, *formatters: FormattersType | None = None*, *float_format: FloatFormatType | None = None*, *sparsify: bool_t | None = None*, *index_names: bool_t = True*, *bold_rows: bool_t = False*, *column_format: str | None = None*, *longtable: bool_t | None = None*, *escape: bool_t | None = None*, *encoding: str | None = None*, *decimal: str = '.'*, *multicolumn: bool_t | None = None*, *multicolumn_format: str | None = None*, *multirow: bool_t | None = None*, *caption: str | tuple[str, str] | None = None*, *label: str | None = None*, *position: str | None = None*) → str | None

Render object to a LaTeX tabular, longtable, or nested table.

Requires `\usepackage{{booktabs}}`. The output can be copy/pasted into a main LaTeX document or read from an external file with `\input{{table.tex}}`.

Changed in version 1.2.0: Added position argument, changed meaning of caption argument.

Changed in version 2.0.0: Refactored to use the Styler implementation via jinja2 templating.

Parameters

buf
[str, Path or StringIO-like, optional, default None] Buffer to write to. If None, the output is returned as a string.

columns
[list of label, optional] The subset of columns to write. Writes all columns by default.

header
[bool or list of str, default True] Write out the column names. If a list of strings is given, it is assumed to be aliases for the column names.

index
[bool, default True] Write row names (index).

na_rep
[str, default 'NaN'] Missing data representation.

formatters
[list of functions or dict of {{str: function}}, optional] Formatter functions to apply to columns' elements by position or name. The result of each function must be a unicode string. List must be of length equal to the number of columns.

float_format
[one-parameter function or str, optional, default None] Formatter for floating point numbers. For example `float_format="%.2f"` and `float_format="{{:0.2f}}".format` will both result in 0.1234 being formatted as 0.12.

sparsify
[bool, optional] Set to False for a DataFrame with a hierarchical index to print every multiindex key at each row. By default, the value will be read from the config module.

index_names
[bool, default True] Prints the names of the indexes.

bold_rows
[bool, default False] Make the row labels bold in the output.

column_format
[str, optional] The columns format as specified in LaTeX table format e.g. 'rcl' for 3 columns. By default, 'l' will be used for all columns except columns of numbers, which default to 'r'.

longtable
[bool, optional] Use a longtable environment instead of tabular. Requires adding a usepackage{{longtable}} to your LaTeX preamble. By default, the value will be read from the pandas config module, and set to *True* if the option `styler.latex.environment` is *"longtable"*.

Changed in version 2.0.0: The pandas option affecting this argument has changed.

escape
[bool, optional] By default, the value will be read from the pandas config module and set to *True* if the option `styler.format.escape` is *"latex"*. When set to False prevents from escaping latex special characters in column names.

Changed in version 2.0.0: The pandas option affecting this argument has changed, as has the default value to *False*.

encoding
[str, optional] A string representing the encoding to use in the output file, defaults to 'utf-8'.

decimal
[str, default '.'] Character recognized as decimal separator, e.g. ',' in Europe.

multicolumn
[bool, default True] Use multicolumn to enhance MultiIndex columns. The default will be read from the config module, and is set as the option `styler.sparse.columns`.

Changed in version 2.0.0: The pandas option affecting this argument has changed.

multicolumn_format
[str, default 'r'] The alignment for multicolumns, similar to *column_format* The default will be read from the config module, and is set as the option `styler.latex.multicol_align`.

Changed in version 2.0.0: The pandas option affecting this argument has changed, as has the default value to "r".

multirow
[bool, default True] Use multirow to enhance MultiIndex rows. Requires adding a usepackage{{multirow}} to your LaTeX preamble. Will print centered labels (instead of top-aligned) across the contained rows, separating groups via clines. The default will be read from the pandas config module, and is set as the option `styler.sparse.index`.

Changed in version 2.0.0: The pandas option affecting this argument has changed, as has the default value to *True*.

caption
[str or tuple, optional] Tuple (full_caption, short_caption), which results in \ `caption[short_caption]{{full_caption}}`; if a single string is passed, no short caption will be set.

Changed in version 1.2.0: Optionally allow caption to be a tuple `(full_caption, short_caption)`.

label
[str, optional] The LaTeX label to be placed inside `\label{{}}` in the output. This is used with `\ref{{}}` in the main `.tex` file.

position
[str, optional] The LaTeX positional argument for tables, to be placed after `\begin{{}}` in the output.

New in version 1.2.0.

Returns

str or None
If buf is None, returns the result as a string. Otherwise returns None.

See also:

`io.formats.style.Styler.to_latex`
Render a DataFrame to LaTeX with conditional formatting.

`DataFrame.to_string`
Render a DataFrame to a console-friendly tabular output.

`DataFrame.to_html`
Render a DataFrame as an HTML table.

Notes

As of v2.0.0 this method has changed to use the Styler implementation as part of `Styler.to_latex()` via `jinja2` templating. This means that `jinja2` is a requirement, and needs to be installed, for this method to function. It is advised that users switch to using Styler, since that implementation is more frequently updated and contains much more flexibility with the output.

Examples

Convert a general DataFrame to LaTeX with formatting:

```
>>> df = pd.DataFrame(dict(name=['Raphael', 'Donatello'],
...                        age=[26, 45],
...                        height=[181.23, 177.65]))
>>> print(df.to_latex(index=False,
...                   formatters={"name": str.upper},
...                   float_format="{:.1f}".format,
... ))
\begin{tabular}{lrr}
\toprule
name & age & height \\
\midrule
RAPHAEL & 26 & 181.2 \\
DONATELLO & 45 & 177.7 \\
\bottomrule
\end{tabular}
```

AlloViz.AlloViz.Elements.Nodes.to_markdown

Nodes.**to_markdown**(*buf: FilePath | WriteBuffer[str] | None = None*, *mode: str = 'wt'*, *index: bool = True*, *storage_options: StorageOptions | None = None*, ***kwargs*) → str | None

Print DataFrame in Markdown-friendly format.

Parameters

buf
[str, Path or StringIO-like, optional, default None] Buffer to write to. If None, the output is returned as a string.

mode
[str, optional] Mode in which file is opened, "wt" by default.

index
[bool, optional, default True] Add index (row) labels.

storage_options
[dict, optional] Extra options that make sense for a particular storage connection, e.g. host, port, username, password, etc. For HTTP(S) URLs the key-value pairs are forwarded to `urllib.request.Request` as header options. For other URLs (e.g. starting with "s3://", and "gcs://") the key-value pairs are forwarded to `fsspec.open`. Please see `fsspec` and `urllib` for more details, and for more examples on storage options refer here.

New in version 1.2.0.

****kwargs**
These parameters will be passed to tabulate.

Returns

str
DataFrame in Markdown-friendly format.

Notes

Requires the tabulate package.

Examples

```
>>> df = pd.DataFrame(
...     data={"animal_1": ["elk", "pig"], "animal_2": ["dog", "quetzal"]}
... )
>>> print(df.to_markdown())
|    | animal_1   | animal_2   |
|---:|:-----------|:-----------|
|  0 | elk        | dog        |
|  1 | pig        | quetzal    |
```

Output markdown with a tabulate option.

```
>>> print(df.to_markdown(tablefmt="grid"))
+----+------------+------------+
```

(continues on next page)

(continued from previous page)

```
|    | animal_1   | animal_2   |
+====+============+============+
|  0 | elk        | dog        |
+----+------------+------------+
|  1 | pig        | quetzal    |
+----+------------+------------+
```

AlloViz.AlloViz.Elements.Nodes.to_numpy

Nodes.**to_numpy**(*dtype: npt.DTypeLike | None = None*, *copy: bool = False*, *na_value: object = _NoDefault.no_default*) → np.ndarray

Convert the DataFrame to a NumPy array.

By default, the dtype of the returned array will be the common NumPy dtype of all types in the DataFrame. For example, if the dtypes are `float16` and `float32`, the results dtype will be `float32`. This may require copying data and coercing values, which may be expensive.

Parameters

dtype
: [str or numpy.dtype, optional] The dtype to pass to `numpy.asarray()`.

copy
: [bool, default False] Whether to ensure that the returned value is not a view on another array. Note that `copy=False` does not *ensure* that `to_numpy()` is no-copy. Rather, `copy=True` ensure that a copy is made, even if not strictly necessary.

na_value
: [Any, optional] The value to use for missing values. The default value depends on *dtype* and the dtypes of the DataFrame columns.

Returns

numpy.ndarray

See also:

`Series.to_numpy`
: Similar method for Series.

Examples

```
>>> pd.DataFrame({"A": [1, 2], "B": [3, 4]}).to_numpy()
array([[1, 3],
       [2, 4]])
```

With heterogeneous data, the lowest common type will have to be used.

```
>>> df = pd.DataFrame({"A": [1, 2], "B": [3.0, 4.5]})
>>> df.to_numpy()
array([[1. , 3. ],
       [2. , 4.5]])
```

For a mix of numeric and non-numeric types, the output array will have object dtype.

```
>>> df['C'] = pd.date_range('2000', periods=2)
>>> df.to_numpy()
array([[1, 3.0, Timestamp('2000-01-01 00:00:00')],
       [2, 4.5, Timestamp('2000-01-02 00:00:00')]], dtype=object)
```

AlloViz.AlloViz.Elements.Nodes.to_orc

Nodes.**to_orc**(*path: FilePath | WriteBuffer[bytes] | None = None*, *, *engine: Literal['pyarrow'] = 'pyarrow'*, *index: bool | None = None*, *engine_kwargs: dict[str, Any] | None = None*) → bytes | None

Write a DataFrame to the ORC format.

New in version 1.5.0.

Parameters

path
[str, file-like object or None, default None] If a string, it will be used as Root Directory path when writing a partitioned dataset. By file-like object, we refer to objects with a write() method, such as a file handle (e.g. via builtin open function). If path is None, a bytes object is returned.

engine
[{'pyarrow'}, default 'pyarrow'] ORC library to use. Pyarrow must be >= 7.0.0.

index
[bool, optional] If `True`, include the dataframe's index(es) in the file output. If `False`, they will not be written to the file. If `None`, similar to `infer` the dataframe's index(es) will be saved. However, instead of being saved as values, the RangeIndex will be stored as a range in the metadata so it doesn't require much space and is faster. Other indexes will be included as columns in the file output.

engine_kwargs
[dict[str, Any] or None, default None] Additional keyword arguments passed to `pyarrow.orc.write_table()`.

Returns

bytes if no path argument is provided else None

Raises

NotImplementedError
Dtype of one or more columns is category, unsigned integers, interval, period or sparse.

ValueError
engine is not pyarrow.

See also:

`read_orc`
Read a ORC file.

`DataFrame.to_parquet`
Write a parquet file.

`DataFrame.to_csv`
Write a csv file.

DataFrame.to_sql
Write to a sql table.

DataFrame.to_hdf
Write to hdf.

Notes

- Before using this function you should read the user guide about ORC and install optional dependencies.
- This function requires pyarrow library.
- For supported dtypes please refer to supported ORC features in Arrow.
- Currently timezones in datetime columns are not preserved when a dataframe is converted into ORC files.

Examples

```
>>> df = pd.DataFrame(data={'col1': [1, 2], 'col2': [4, 3]})
>>> df.to_orc('df.orc')
>>> pd.read_orc('df.orc')
   col1  col2
0     1     4
1     2     3
```

If you want to get a buffer to the orc content you can write it to io.BytesIO

```
>>> import io
>>> b = io.BytesIO(df.to_orc())
>>> b.seek(0)
0
>>> content = b.read()
```

AlloViz.AlloViz.Elements.Nodes.to_parquet

Nodes.**to_parquet**(*path: FilePath | WriteBuffer[bytes] | None = None*, *engine: Literal['auto', 'pyarrow', 'fastparquet'] = 'auto'*, *compression: str | None = 'snappy'*, *index: bool | None = None*, *partition_cols: list[str] | None = None*, *storage_options: StorageOptions | None = None*, ***kwargs*) → bytes | None

Write a DataFrame to the binary parquet format.

This function writes the dataframe as a parquet file. You can choose different parquet backends, and have the option of compression. See the user guide for more details.

Parameters

path
[str, path object, file-like object, or None, default None] String, path object (implementing `os.PathLike[str]`), or file-like object implementing a binary `write()` function. If None, the result is returned as bytes. If a string or path, it will be used as Root Directory path when writing a partitioned dataset.

Changed in version 1.2.0.

Previously this was “fname”

engine
[{‘auto’, ‘pyarrow’, ‘fastparquet’}, default ‘auto’] Parquet library to use. If ‘auto’, then the option `io.parquet.engine` is used. The default `io.parquet.engine` behavior is to try ‘pyarrow’, falling back to ‘fastparquet’ if ‘pyarrow’ is unavailable.

compression
[str or None, default ‘snappy’] Name of the compression to use. Use `None` for no compression. Supported options: ‘snappy’, ‘gzip’, ‘brotli’, ‘lz4’, ‘zstd’.

index
[bool, default None] If `True`, include the dataframe’s index(es) in the file output. If `False`, they will not be written to the file. If `None`, similar to `True` the dataframe’s index(es) will be saved. However, instead of being saved as values, the RangeIndex will be stored as a range in the metadata so it doesn’t require much space and is faster. Other indexes will be included as columns in the file output.

partition_cols
[list, optional, default None] Column names by which to partition the dataset. Columns are partitioned in the order they are given. Must be None if path is not a string.

storage_options
[dict, optional] Extra options that make sense for a particular storage connection, e.g. host, port, username, password, etc. For HTTP(S) URLs the key-value pairs are forwarded to `urllib.request.Request` as header options. For other URLs (e.g. starting with “s3://”, and “gcs://”) the key-value pairs are forwarded to `fsspec.open`. Please see `fsspec` and `urllib` for more details, and for more examples on storage options refer here.

New in version 1.2.0.

****kwargs**
Additional arguments passed to the parquet library. See pandas io for more details.

Returns

bytes if no path argument is provided else None

See also:

`read_parquet`
Read a parquet file.

`DataFrame.to_orc`
Write an orc file.

`DataFrame.to_csv`
Write a csv file.

`DataFrame.to_sql`
Write to a sql table.

`DataFrame.to_hdf`
Write to hdf.

Notes

This function requires either the fastparquet or pyarrow library.

Examples

```
>>> df = pd.DataFrame(data={'col1': [1, 2], 'col2': [3, 4]})
>>> df.to_parquet('df.parquet.gzip',
...               compression='gzip')
>>> pd.read_parquet('df.parquet.gzip')
   col1  col2
0     1     3
1     2     4
```

If you want to get a buffer to the parquet content you can use a io.BytesIO object, as long as you don't use partition_cols, which creates multiple files.

```
>>> import io
>>> f = io.BytesIO()
>>> df.to_parquet(f)
>>> f.seek(0)
0
>>> content = f.read()
```

AlloViz.AlloViz.Elements.Nodes.to_period

`Nodes.`**`to_period`**(*freq: Frequency | None = None*, *axis: Axis = 0*, *copy: bool | None = None*) → DataFrame

Convert DataFrame from DatetimeIndex to PeriodIndex.

Convert DataFrame from DatetimeIndex to PeriodIndex with desired frequency (inferred from index if not passed).

Parameters

freq
: [str, default] Frequency of the PeriodIndex.

axis
: [{0 or 'index', 1 or 'columns'}, default 0] The axis to convert (the index by default).

copy
: [bool, default True] If False then underlying input data is not copied.

Returns

DataFrame
: The DataFrame has a PeriodIndex.

Examples

```
>>> idx = pd.to_datetime(
...     [
...         "2001-03-31 00:00:00",
...         "2002-05-31 00:00:00",
...         "2003-08-31 00:00:00",
...     ]
... )
```

```
>>> idx
DatetimeIndex(['2001-03-31', '2002-05-31', '2003-08-31'],
dtype='datetime64[ns]', freq=None)
```

```
>>> idx.to_period("M")
PeriodIndex(['2001-03', '2002-05', '2003-08'], dtype='period[M]')
```

For the yearly frequency

```
>>> idx.to_period("Y")
PeriodIndex(['2001', '2002', '2003'], dtype='period[A-DEC]')
```

AlloViz.AlloViz.Elements.Nodes.to_pickle

Nodes.**to_pickle**(*path: FilePath | WriteBuffer[bytes]*, *compression: CompressionOptions = 'infer'*, *protocol: int = 5*, *storage_options: StorageOptions | None = None*) → None

Pickle (serialize) object to file.

Parameters

path
[str, path object, or file-like object] String, path object (implementing `os.PathLike[str]`), or file-like object implementing a binary `write()` function. File path where the pickled object will be stored.

compression
[str or dict, default 'infer'] For on-the-fly compression of the output data. If 'infer' and 'path' is path-like, then detect compression from the following extensions: '.gz', '.bz2', '.zip', '.xz', '.zst', '.tar', '.tar.gz', '.tar.xz' or '.tar.bz2' (otherwise no compression). Set to `None` for no compression. Can also be a dict with key `'method'` set to one of {`'zip'`, `'gzip'`, `'bz2'`, `'zstd'`, `'xz'`, `'tar'`} and other key-value pairs are forwarded to `zipfile.ZipFile`, `gzip.GzipFile`, `bz2.BZ2File`, `zstandard.ZstdCompressor`, `lzma.LZMAFile` or `tarfile.TarFile`, respectively. As an example, the following could be passed for faster compression and to create a reproducible gzip archive: `compression={'method': 'gzip', 'compresslevel': 1, 'mtime': 1}`.

New in version 1.5.0: Added support for *.tar* files.

protocol
[int] Int which indicates which protocol should be used by the pickler, default HIGHEST_PROTOCOL (see [1] paragraph 12.1.2). The possible values are 0, 1, 2, 3, 4, 5. A negative value for the protocol parameter is equivalent to setting its value to HIGHEST_PROTOCOL.

storage_options
[dict, optional] Extra options that make sense for a particular storage connection, e.g. host, port, username, password, etc. For HTTP(S) URLs the key-value pairs are forwarded to `urllib.request.Request` as header options. For other URLs (e.g. starting with "s3://", and "gcs://") the key-value pairs are forwarded to `fsspec.open`. Please see `fsspec` and `urllib` for more details, and for more examples on storage options refer here.

New in version 1.2.0.

See also:

`read_pickle`
Load pickled pandas object (or any object) from file.

`DataFrame.to_hdf`
Write DataFrame to an HDF5 file.

`DataFrame.to_sql`
Write DataFrame to a SQL database.

`DataFrame.to_parquet`
Write a DataFrame to the binary parquet format.

Examples

```
>>> original_df = pd.DataFrame({"foo": range(5), "bar": range(5, 10)})
>>> original_df
   foo  bar
0    0    5
1    1    6
2    2    7
3    3    8
4    4    9
>>> original_df.to_pickle("./dummy.pkl")
```

```
>>> unpickled_df = pd.read_pickle("./dummy.pkl")
>>> unpickled_df
   foo  bar
0    0    5
1    1    6
2    2    7
3    3    8
4    4    9
```

AlloViz.AlloViz.Elements.Nodes.to_records

`Nodes.to_records`(*index: bool = True*, *column_dtypes=None*, *index_dtypes=None*) → recarray

Convert DataFrame to a NumPy record array.

Index will be included as the first field of the record array if requested.

Parameters

index
[bool, default True] Include index in resulting record array, stored in 'index' field or using the index label, if set.

column_dtypes
[str, type, dict, default None] If a string or type, the data type to store all columns. If a dictionary, a mapping of column names and indices (zero-indexed) to specific data types.

index_dtypes
[str, type, dict, default None] If a string or type, the data type to store all index levels. If a dictionary, a mapping of index level names and indices (zero-indexed) to specific data types.

This mapping is applied only if *index=True*.

Returns

numpy.recarray
NumPy ndarray with the DataFrame labels as fields and each row of the DataFrame as entries.

See also:

`DataFrame.from_records`
Convert structured or record ndarray to DataFrame.

`numpy.recarray`
An ndarray that allows field access using attributes, analogous to typed columns in a spreadsheet.

Examples

```
>>> df = pd.DataFrame({'A': [1, 2], 'B': [0.5, 0.75]},
...                   index=['a', 'b'])
>>> df
   A     B
a  1  0.50
b  2  0.75
>>> df.to_records()
rec.array([('a', 1, 0.5 ), ('b', 2, 0.75)],
          dtype=[('index', 'O'), ('A', '<i8'), ('B', '<f8')])
```

If the DataFrame index has no label then the recarray field name is set to 'index'. If the index has a label then this is used as the field name:

```
>>> df.index = df.index.rename("I")
>>> df.to_records()
```

(continues on next page)

(continued from previous page)

```
rec.array([('a', 1, 0.5 ), ('b', 2, 0.75)],
          dtype=[('I', 'O'), ('A', '<i8'), ('B', '<f8')])
```

The index can be excluded from the record array:

```
>>> df.to_records(index=False)
rec.array([(1, 0.5 ), (2, 0.75)],
          dtype=[('A', '<i8'), ('B', '<f8')])
```

Data types can be specified for the columns:

```
>>> df.to_records(column_dtypes={"A": "int32"})
rec.array([('a', 1, 0.5 ), ('b', 2, 0.75)],
          dtype=[('I', 'O'), ('A', '<i4'), ('B', '<f8')])
```

As well as for the index:

```
>>> df.to_records(index_dtypes="<S2")
rec.array([(b'a', 1, 0.5 ), (b'b', 2, 0.75)],
          dtype=[('I', 'S2'), ('A', '<i8'), ('B', '<f8')])
```

```
>>> index_dtypes = f"<S{df.index.str.len().max()}"
>>> df.to_records(index_dtypes=index_dtypes)
rec.array([(b'a', 1, 0.5 ), (b'b', 2, 0.75)],
          dtype=[('I', 'S1'), ('A', '<i8'), ('B', '<f8')])
```

AlloViz.AlloViz.Elements.Nodes.to_sql

Nodes.**to_sql**(*name: str*, *con*, ***, *schema: str | None = None*, *if_exists: Literal['fail', 'replace', 'append'] = 'fail'*, *index: bool = True*, *index_label: Hashable | Sequence[Hashable] | None = None*, *chunksize: int | None = None*, *dtype: ExtensionDtype | str | dtype | Type[str | complex | bool | object] | dict[Hashable, ExtensionDtype | str | dtype | Type[str | complex | bool | object]] | None = None*, *method: Literal['multi'] | Callable | None = None*) → int | None

Write records stored in a DataFrame to a SQL database.

Databases supported by SQLAlchemy [1] are supported. Tables can be newly created, appended to, or overwritten.

Parameters

name
: [str] Name of SQL table.

con
: [sqlalchemy.engine.(Engine or Connection) or sqlite3.Connection] Using SQLAlchemy makes it possible to use any DB supported by that library. Legacy support is provided for sqlite3.Connection objects. The user is responsible for engine disposal and connection closure for the SQLAlchemy connectable. See here. If passing a sqlalchemy.engine.Connection which is already in a transaction, the transaction will not be committed. If passing a sqlite3.Connection, it will not be possible to roll back the record insertion.

schema
[str, optional] Specify the schema (if database flavor supports this). If None, use default schema.

if_exists
[{'fail', 'replace', 'append'}, default 'fail'] How to behave if the table already exists.

- fail: Raise a ValueError.
- replace: Drop the table before inserting new values.
- append: Insert new values to the existing table.

index
[bool, default True] Write DataFrame index as a column. Uses *index_label* as the column name in the table.

index_label
[str or sequence, default None] Column label for index column(s). If None is given (default) and *index* is True, then the index names are used. A sequence should be given if the DataFrame uses MultiIndex.

chunksize
[int, optional] Specify the number of rows in each batch to be written at a time. By default, all rows will be written at once.

dtype
[dict or scalar, optional] Specifying the datatype for columns. If a dictionary is used, the keys should be the column names and the values should be the SQLAlchemy types or strings for the sqlite3 legacy mode. If a scalar is provided, it will be applied to all columns.

method
[{None, 'multi', callable}, optional] Controls the SQL insertion clause used:

- None : Uses standard SQL `INSERT` clause (one per row).
- 'multi': Pass multiple values in a single `INSERT` clause.
- callable with signature `(pd_table, conn, keys, data_iter)`.

Details and a sample callable implementation can be found in the section insert method.

Returns

None or int
Number of rows affected by to_sql. None is returned if the callable passed into `method` does not return an integer number of rows.

The number of returned rows affected is the sum of the `rowcount` attribute of `sqlite3.Cursor` or SQLAlchemy connectable which may not reflect the exact number of written rows as stipulated in the sqlite3 or SQLAlchemy.

New in version 1.4.0.

Raises

ValueError
When the table already exists and *if_exists* is 'fail' (the default).

See also:

read_sql
Read a DataFrame from a table.

Notes

Timezone aware datetime columns will be written as `Timestamp with timezone` type with SQLAlchemy if supported by the database. Otherwise, the datetimes will be stored as timezone unaware timestamps local to the original timezone.

References

[1], [2]

Examples

Create an in-memory SQLite database.

```
>>> from sqlalchemy import create_engine
>>> engine = create_engine('sqlite://', echo=False)
```

Create a table from scratch with 3 rows.

```
>>> df = pd.DataFrame({'name' : ['User 1', 'User 2', 'User 3']})
>>> df
     name
0  User 1
1  User 2
2  User 3
```

```
>>> df.to_sql(name='users', con=engine)
3
>>> from sqlalchemy import text
>>> with engine.connect() as conn:
...    conn.execute(text("SELECT * FROM users")).fetchall()
[(0, 'User 1'), (1, 'User 2'), (2, 'User 3')]
```

An *sqlalchemy.engine.Connection* can also be passed to *con*:

```
>>> with engine.begin() as connection:
...     df1 = pd.DataFrame({'name' : ['User 4', 'User 5']})
...     df1.to_sql(name='users', con=connection, if_exists='append')
2
```

This is allowed to support operations that require that the same DBAPI connection is used for the entire operation.

```
>>> df2 = pd.DataFrame({'name' : ['User 6', 'User 7']})
>>> df2.to_sql(name='users', con=engine, if_exists='append')
2
>>> with engine.connect() as conn:
...    conn.execute(text("SELECT * FROM users")).fetchall()
[(0, 'User 1'), (1, 'User 2'), (2, 'User 3'),
```

(continues on next page)

(continued from previous page)

```
 (0, 'User 4'), (1, 'User 5'), (0, 'User 6'),
 (1, 'User 7')]
```

Overwrite the table with just `df2`.

```
>>> df2.to_sql(name='users', con=engine, if_exists='replace',
...            index_label='id')
2
>>> with engine.connect() as conn:
...     conn.execute(text("SELECT * FROM users")).fetchall()
[(0, 'User 6'), (1, 'User 7')]
```

Use `method` to define a callable insertion method to do nothing if there's a primary key conflict on a table in a PostgreSQL database.

```
>>> from sqlalchemy.dialects.postgresql import insert
>>> def insert_on_conflict_nothing(table, conn, keys, data_iter):
...     # "a" is the primary key in "conflict_table"
...     data = [dict(zip(keys, row)) for row in data_iter]
...     stmt = insert(table.table).values(data).on_conflict_do_nothing(index_
↪elements=["a"])
...     result = conn.execute(stmt)
...     return result.rowcount
>>> df_conflict.to_sql(name="conflict_table", con=conn, if_exists="append",␣
↪method=insert_on_conflict_nothing)
0
```

For MySQL, a callable to update columns `b` and `c` if there's a conflict on a primary key.

```
>>> from sqlalchemy.dialects.mysql import insert
>>> def insert_on_conflict_update(table, conn, keys, data_iter):
...     # update columns "b" and "c" on primary key conflict
...     data = [dict(zip(keys, row)) for row in data_iter]
...     stmt = (
...         insert(table.table)
...         .values(data)
...     )
...     stmt = stmt.on_duplicate_key_update(b=stmt.inserted.b, c=stmt.inserted.
↪c)
...     result = conn.execute(stmt)
...     return result.rowcount
>>> df_conflict.to_sql(name="conflict_table", con=conn, if_exists="append",␣
↪method=insert_on_conflict_update)
2
```

Specify the dtype (especially useful for integers with missing values). Notice that while pandas is forced to store the data as floating point, the database supports nullable integers. When fetching the data with Python, we get back integer scalars.

```
>>> df = pd.DataFrame({"A": [1, None, 2]})
>>> df
     A
0  1.0
```

(continues on next page)

(continued from previous page)

```
1  NaN
2  2.0
```

```
>>> from sqlalchemy.types import Integer
>>> df.to_sql(name='integers', con=engine, index=False,
...           dtype={"A": Integer()})
3
```

```
>>> with engine.connect() as conn:
...   conn.execute(text("SELECT * FROM integers")).fetchall()
[(1,), (None,), (2,)]
```

AlloViz.AlloViz.Elements.Nodes.to_stata

Nodes.**to_stata**(*path: FilePath | WriteBuffer[bytes]*, ***, *convert_dates: dict[Hashable, str] | None = None*, *write_index: bool = True*, *byteorder: ToStataByteorder | None = None*, *time_stamp: datetime.datetime | None = None*, *data_label: str | None = None*, *variable_labels: dict[Hashable, str] | None = None*, *version: int | None = 114*, *convert_strl: Sequence[Hashable] | None = None*, *compression: CompressionOptions = 'infer'*, *storage_options: StorageOptions | None = None*, *value_labels: dict[Hashable, dict[float, str]] | None = None*) → None

Export DataFrame object to Stata dta format.

Writes the DataFrame to a Stata dataset file. “dta” files contain a Stata dataset.

Parameters

path
: [str, path object, or buffer] String, path object (implementing `os.PathLike[str]`), or file-like object implementing a binary `write()` function.

convert_dates
: [dict] Dictionary mapping columns containing datetime types to stata internal format to use when writing the dates. Options are ‘tc’, ‘td’, ‘tm’, ‘tw’, ‘th’, ‘tq’, ‘ty’. Column can be either an integer or a name. Datetime columns that do not have a conversion type specified will be converted to ‘tc’. Raises NotImplementedError if a datetime column has timezone information.

write_index
: [bool] Write the index to Stata dataset.

byteorder
: [str] Can be “>”, “<”, “little”, or “big”. default is *sys.byteorder*.

time_stamp
: [datetime] A datetime to use as file creation date. Default is the current time.

data_label
: [str, optional] A label for the data set. Must be 80 characters or smaller.

variable_labels
: [dict] Dictionary containing columns as keys and variable labels as values. Each label must be 80 characters or smaller.

version

[{114, 117, 118, 119, None}, default 114] Version to use in the output dta file. Set to None to let pandas decide between 118 or 119 formats depending on the number of columns in the frame. Version 114 can be read by Stata 10 and later. Version 117 can be read by Stata 13 or later. Version 118 is supported in Stata 14 and later. Version 119 is supported in Stata 15 and later. Version 114 limits string variables to 244 characters or fewer while versions 117 and later allow strings with lengths up to 2,000,000 characters. Versions 118 and 119 support Unicode characters, and version 119 supports more than 32,767 variables.

Version 119 should usually only be used when the number of variables exceeds the capacity of dta format 118. Exporting smaller datasets in format 119 may have unintended consequences, and, as of November 2020, Stata SE cannot read version 119 files.

convert_strl

[list, optional] List of column names to convert to string columns to Stata StrL format. Only available if version is 117. Storing strings in the StrL format can produce smaller dta files if strings have more than 8 characters and values are repeated.

compression

[str or dict, default 'infer'] For on-the-fly compression of the output data. If 'infer' and 'path' is path-like, then detect compression from the following extensions: '.gz', '.bz2', '.zip', '.xz', '.zst', '.tar', '.tar.gz', '.tar.xz' or '.tar.bz2' (otherwise no compression). Set to `None` for no compression. Can also be a dict with key `'method'` set to one of {`'zip'`, `'gzip'`, `'bz2'`, `'zstd'`, `'xz'`, `'tar'`} and other key-value pairs are forwarded to `zipfile.ZipFile`, `gzip.GzipFile`, `bz2.BZ2File`, `zstandard.ZstdCompressor`, `lzma.LZMAFile` or `tarfile.TarFile`, respectively. As an example, the following could be passed for faster compression and to create a reproducible gzip archive: `compression={'method': 'gzip', 'compresslevel': 1, 'mtime': 1}`.

New in version 1.5.0: Added support for *.tar* files.

Changed in version 1.4.0: Zstandard support.

storage_options

[dict, optional] Extra options that make sense for a particular storage connection, e.g. host, port, username, password, etc. For HTTP(S) URLs the key-value pairs are forwarded to `urllib.request.Request` as header options. For other URLs (e.g. starting with "s3://", and "gcs://") the key-value pairs are forwarded to `fsspec.open`. Please see `fsspec` and `urllib` for more details, and for more examples on storage options refer here.

New in version 1.2.0.

value_labels

[dict of dicts] Dictionary containing columns as keys and dictionaries of column value to labels as values. Labels for a single variable must be 32,000 characters or smaller.

New in version 1.4.0.

Raises

NotImplementedError

- If datetimes contain timezone information
- Column dtype is not representable in Stata

ValueError

- Columns listed in convert_dates are neither datetime64[ns] or datetime.datetime
- Column listed in convert_dates is not in DataFrame
- Categorical label contains more than 32,000 characters

See also:

read_stata
Import Stata data files.

io.stata.StataWriter
Low-level writer for Stata data files.

io.stata.StataWriter117
Low-level writer for version 117 files.

Examples

```
>>> df = pd.DataFrame({'animal': ['falcon', 'parrot', 'falcon',
...                               'parrot'],
...                    'speed': [350, 18, 361, 15]})
>>> df.to_stata('animals.dta')
```

AlloViz.AlloViz.Elements.Nodes.to_string

Nodes.**to_string**(*buf: FilePath | WriteBuffer[str] | None = None*, *columns: Axes | None = None*, *col_space: int | list[int] | dict[Hashable, int] | None = None*, *header: bool | list[str] = True*, *index: bool = True*, *na_rep: str = 'NaN'*, *formatters: fmt.FormattersType | None = None*, *float_format: fmt.FloatFormatType | None = None*, *sparsify: bool | None = None*, *index_names: bool = True*, *justify: str | None = None*, *max_rows: int | None = None*, *max_cols: int | None = None*, *show_dimensions: bool = False*, *decimal: str = '.'*, *line_width: int | None = None*, *min_rows: int | None = None*, *max_colwidth: int | None = None*, *encoding: str | None = None*) → str | None

Render a DataFrame to a console-friendly tabular output.

Parameters

buf
[str, Path or StringIO-like, optional, default None] Buffer to write to. If None, the output is returned as a string.

columns
[array-like, optional, default None] The subset of columns to write. Writes all columns by default.

col_space
[int, list or dict of int, optional] The minimum width of each column. If a list of ints is given every integers corresponds with one column. If a dict is given, the key references the column, while the value defines the space to use..

header
[bool or list of str, optional] Write out the column names. If a list of columns is given, it is assumed to be aliases for the column names.

index
[bool, optional, default True] Whether to print index (row) labels.

na_rep
[str, optional, default 'NaN'] String representation of `NaN` to use.

formatters
[list, tuple or dict of one-param. functions, optional] Formatter functions to apply to columns' elements by position or name. The result of each function must be a unicode string. List/tuple must be of length equal to the number of columns.

float_format
[one-parameter function, optional, default None] Formatter function to apply to columns' elements if they are floats. This function must return a unicode string and will be applied only to the non-`NaN` elements, with `NaN` being handled by `na_rep`.

Changed in version 1.2.0.

sparsify
[bool, optional, default True] Set to False for a DataFrame with a hierarchical index to print every multiindex key at each row.

index_names
[bool, optional, default True] Prints the names of the indexes.

justify
[str, default None] How to justify the column labels. If None uses the option from the print configuration (controlled by set_option), 'right' out of the box. Valid values are

- left
- right
- center
- justify
- justify-all
- start
- end
- inherit
- match-parent
- initial
- unset.

max_rows
[int, optional] Maximum number of rows to display in the console.

max_cols
[int, optional] Maximum number of columns to display in the console.

show_dimensions
[bool, default False] Display DataFrame dimensions (number of rows by number of columns).

decimal
[str, default '.'] Character recognized as decimal separator, e.g. ',' in Europe.

line_width
[int, optional] Width to wrap a line in characters.

min_rows
[int, optional] The number of rows to display in the console in a truncated repr (when number of rows is above *max_rows*).

max_colwidth
[int, optional] Max width to truncate each column in characters. By default, no limit.

encoding
[str, default "utf-8"] Set character encoding.

Returns

str or None
If buf is None, returns the result as a string. Otherwise returns None.

See also:

to_html
Convert DataFrame to HTML.

Examples

```
>>> d = {'col1': [1, 2, 3], 'col2': [4, 5, 6]}
>>> df = pd.DataFrame(d)
>>> print(df.to_string())
   col1  col2
0     1     4
1     2     5
2     3     6
```

AlloViz.AlloViz.Elements.Nodes.to_timestamp

Nodes.**to_timestamp**(*freq: Frequency | None = None*, *how: ToTimestampHow = 'start'*, *axis: Axis = 0*, *copy: bool | None = None*) → DataFrame

Cast to DatetimeIndex of timestamps, at *beginning* of period.

Parameters

freq
[str, default frequency of PeriodIndex] Desired frequency.

how
[{'s', 'e', 'start', 'end'}] Convention for converting period to timestamp; start of period vs. end.

axis
[{0 or 'index', 1 or 'columns'}, default 0] The axis to convert (the index by default).

copy
[bool, default True] If False then underlying input data is not copied.

Returns

DataFrame
The DataFrame has a DatetimeIndex.

Examples

```
>>> idx = pd.PeriodIndex(['2023', '2024'], freq='Y')
>>> d = {'col1': [1, 2], 'col2': [3, 4]}
>>> df1 = pd.DataFrame(data=d, index=idx)
>>> df1
      col1   col2
2023     1      3
2024     2      4
```

The resulting timestamps will be at the beginning of the year in this case

```
>>> df1 = df1.to_timestamp()
>>> df1
            col1   col2
2023-01-01     1      3
2024-01-01     2      4
>>> df1.index
DatetimeIndex(['2023-01-01', '2024-01-01'], dtype='datetime64[ns]', freq=None)
```

Using *freq* which is the offset that the Timestamps will have

```
>>> df2 = pd.DataFrame(data=d, index=idx)
>>> df2 = df2.to_timestamp(freq='M')
>>> df2
            col1   col2
2023-01-31     1      3
2024-01-31     2      4
>>> df2.index
DatetimeIndex(['2023-01-31', '2024-01-31'], dtype='datetime64[ns]', freq=None)
```

AlloViz.AlloViz.Elements.Nodes.to_xarray

`Nodes.`**`to_xarray`**`()`

Return an xarray object from the pandas object.

Returns

xarray.DataArray or xarray.Dataset
Data in the pandas structure converted to Dataset if the object is a DataFrame, or a DataArray if the object is a Series.

See also:

`DataFrame.to_hdf`
Write DataFrame to an HDF5 file.

`DataFrame.to_parquet`
Write a DataFrame to the binary parquet format.

Notes

See the xarray docs

Examples

```
>>> df = pd.DataFrame([('falcon', 'bird', 389.0, 2),
...                    ('parrot', 'bird', 24.0, 2),
...                    ('lion', 'mammal', 80.5, 4),
...                    ('monkey', 'mammal', np.nan, 4)],
...                   columns=['name', 'class', 'max_speed',
...                            'num_legs'])
>>> df
     name   class  max_speed  num_legs
0  falcon    bird      389.0         2
1  parrot    bird       24.0         2
2    lion  mammal       80.5         4
3  monkey  mammal        NaN         4
```

```
>>> df.to_xarray()
<xarray.Dataset>
Dimensions:    (index: 4)
Coordinates:
  * index      (index) int64 0 1 2 3
Data variables:
    name       (index) object 'falcon' 'parrot' 'lion' 'monkey'
    class      (index) object 'bird' 'bird' 'mammal' 'mammal'
    max_speed  (index) float64 389.0 24.0 80.5 nan
    num_legs   (index) int64 2 2 4 4
```

```
>>> df['max_speed'].to_xarray()
<xarray.DataArray 'max_speed' (index: 4)>
array([389. ,  24. ,  80.5,   nan])
Coordinates:
  * index    (index) int64 0 1 2 3
```

```
>>> dates = pd.to_datetime(['2018-01-01', '2018-01-01',
...                         '2018-01-02', '2018-01-02'])
>>> df_multiindex = pd.DataFrame({'date': dates,
...                               'animal': ['falcon', 'parrot',
...                                          'falcon', 'parrot'],
...                               'speed': [350, 18, 361, 15]})
>>> df_multiindex = df_multiindex.set_index(['date', 'animal'])
```

```
>>> df_multiindex
                   speed
date       animal
2018-01-01 falcon    350
           parrot     18
2018-01-02 falcon    361
           parrot     15
```

```
>>> df_multiindex.to_xarray()
<xarray.Dataset>
Dimensions:  (date: 2, animal: 2)
Coordinates:
  * date     (date) datetime64[ns] 2018-01-01 2018-01-02
  * animal   (animal) object 'falcon' 'parrot'
Data variables:
    speed    (date, animal) int64 350 18 361 15
```

AlloViz.AlloViz.Elements.Nodes.to_xml

Nodes.**to_xml**(*path_or_buffer: FilePath | WriteBuffer[bytes] | WriteBuffer[str] | None = None*, *index: bool = True*, *root_name: str | None = 'data'*, *row_name: str | None = 'row'*, *na_rep: str | None = None*, *attr_cols: list[str] | None = None*, *elem_cols: list[str] | None = None*, *namespaces: dict[str | None, str] | None = None*, *prefix: str | None = None*, *encoding: str = 'utf-8'*, *xml_declaration: bool | None = True*, *pretty_print: bool | None = True*, *parser: XMLParsers | None = 'lxml'*, *stylesheet: FilePath | ReadBuffer[str] | ReadBuffer[bytes] | None = None*, *compression: CompressionOptions = 'infer'*, *storage_options: StorageOptions | None = None*) → str | None

Render a DataFrame to an XML document.

New in version 1.3.0.

Parameters

path_or_buffer
[str, path object, file-like object, or None, default None] String, path object (implementing `os.PathLike[str]`), or file-like object implementing a `write()` function. If None, the result is returned as a string.

index
[bool, default True] Whether to include index in XML document.

root_name
[str, default 'data'] The name of root element in XML document.

row_name
[str, default 'row'] The name of row element in XML document.

na_rep
[str, optional] Missing data representation.

attr_cols
[list-like, optional] List of columns to write as attributes in row element. Hierarchical columns will be flattened with underscore delimiting the different levels.

elem_cols
[list-like, optional] List of columns to write as children in row element. By default, all columns output as children of row element. Hierarchical columns will be flattened with underscore delimiting the different levels.

namespaces
[dict, optional] All namespaces to be defined in root element. Keys of dict should be prefix names and values of dict corresponding URIs. Default namespaces should be given empty string key. For example,

```
namespaces = {"": "https://example.com"}
```

prefix
: [str, optional] Namespace prefix to be used for every element and/or attribute in document. This should be one of the keys in `namespaces` dict.

encoding
: [str, default 'utf-8'] Encoding of the resulting document.

xml_declaration
: [bool, default True] Whether to include the XML declaration at start of document.

pretty_print
: [bool, default True] Whether output should be pretty printed with indentation and line breaks.

parser
: [{'lxml','etree'}, default 'lxml'] Parser module to use for building of tree. Only 'lxml' and 'etree' are supported. With 'lxml', the ability to use XSLT stylesheet is supported.

stylesheet
: [str, path object or file-like object, optional] A URL, file-like object, or a raw string containing an XSLT script used to transform the raw XML output. Script should use layout of elements and attributes from original output. This argument requires `lxml` to be installed. Only XSLT 1.0 scripts and not later versions is currently supported.

compression
: [str or dict, default 'infer'] For on-the-fly compression of the output data. If 'infer' and 'path_or_buffer' is path-like, then detect compression from the following extensions: '.gz', '.bz2', '.zip', '.xz', '.zst', '.tar', '.tar.gz', '.tar.xz' or '.tar.bz2' (otherwise no compression). Set to `None` for no compression. Can also be a dict with key `'method'` set to one of {`'zip'`, `'gzip'`, `'bz2'`, `'zstd'`, `'xz'`, `'tar'`} and other key-value pairs are forwarded to `zipfile.ZipFile`, `gzip.GzipFile`, `bz2.BZ2File`, `zstandard.ZstdCompressor`, `lzma.LZMAFile` or `tarfile.TarFile`, respectively. As an example, the following could be passed for faster compression and to create a reproducible gzip archive: `compression={'method': 'gzip', 'compresslevel': 1, 'mtime': 1}`.

New in version 1.5.0: Added support for *.tar* files.

Changed in version 1.4.0: Zstandard support.

storage_options
: [dict, optional] Extra options that make sense for a particular storage connection, e.g. host, port, username, password, etc. For HTTP(S) URLs the key-value pairs are forwarded to `urllib.request.Request` as header options. For other URLs (e.g. starting with "s3://", and "gcs://") the key-value pairs are forwarded to `fsspec.open`. Please see `fsspec` and `urllib` for more details, and for more examples on storage options refer here.

Returns

None or str
: If `io` is None, returns the resulting XML format as a string. Otherwise returns None.

See also:

to_json
Convert the pandas object to a JSON string.

to_html
Convert DataFrame to a html.

Examples

```
>>> df = pd.DataFrame({'shape': ['square', 'circle', 'triangle'],
...                    'degrees': [360, 360, 180],
...                    'sides': [4, np.nan, 3]})
```

```
>>> df.to_xml()
<?xml version='1.0' encoding='utf-8'?>
<data>
  <row>
    <index>0</index>
    <shape>square</shape>
    <degrees>360</degrees>
    <sides>4.0</sides>
  </row>
  <row>
    <index>1</index>
    <shape>circle</shape>
    <degrees>360</degrees>
    <sides/>
  </row>
  <row>
    <index>2</index>
    <shape>triangle</shape>
    <degrees>180</degrees>
    <sides>3.0</sides>
  </row>
</data>
```

```
>>> df.to_xml(attr_cols=[
...           'index', 'shape', 'degrees', 'sides'
...           ])
<?xml version='1.0' encoding='utf-8'?>
<data>
  <row index="0" shape="square" degrees="360" sides="4.0"/>
  <row index="1" shape="circle" degrees="360"/>
  <row index="2" shape="triangle" degrees="180" sides="3.0"/>
</data>
```

```
>>> df.to_xml(namespaces={"doc": "https://example.com"},
...           prefix="doc")
<?xml version='1.0' encoding='utf-8'?>
<doc:data xmlns:doc="https://example.com">
  <doc:row>
    <doc:index>0</doc:index>
    <doc:shape>square</doc:shape>
```

(continues on next page)

(continued from previous page)

```
    <doc:degrees>360</doc:degrees>
    <doc:sides>4.0</doc:sides>
  </doc:row>
  <doc:row>
    <doc:index>1</doc:index>
    <doc:shape>circle</doc:shape>
    <doc:degrees>360</doc:degrees>
    <doc:sides/>
  </doc:row>
  <doc:row>
    <doc:index>2</doc:index>
    <doc:shape>triangle</doc:shape>
    <doc:degrees>180</doc:degrees>
    <doc:sides>3.0</doc:sides>
  </doc:row>
</doc:data>
```

AlloViz.AlloViz.Elements.Nodes.transform

`Nodes.transform`(*func: AggFuncType*, *axis: Axis = 0*, **args*, ***kwargs*) → DataFrame

Call `func` on self producing a DataFrame with the same axis shape as self.

Parameters

func
[function, str, list-like or dict-like] Function to use for transforming the data. If a function, must either work when passed a DataFrame or when passed to DataFrame.apply. If func is both list-like and dict-like, dict-like behavior takes precedence.

Accepted combinations are:

- function
- string function name
- list-like of functions and/or function names, e.g. `[np.exp, 'sqrt']`
- dict-like of axis labels -> functions, function names or list-like of such.

axis
[{0 or 'index', 1 or 'columns'}, default 0] If 0 or 'index': apply function to each column. If 1 or 'columns': apply function to each row.

***args**
Positional arguments to pass to *func*.

****kwargs**
Keyword arguments to pass to *func*.

Returns

DataFrame
A DataFrame that must have the same length as self.

Raises

ValueError
[If the returned DataFrame has a different length than self.]

See also:

DataFrame.agg
Only perform aggregating type operations.

DataFrame.apply
Invoke function on a DataFrame.

Notes

Functions that mutate the passed object can produce unexpected behavior or errors and are not supported. See Mutating with User Defined Function (UDF) methods for more details.

Examples

```
>>> df = pd.DataFrame({'A': range(3), 'B': range(1, 4)})
>>> df
   A  B
0  0  1
1  1  2
2  2  3
>>> df.transform(lambda x: x + 1)
   A  B
0  1  2
1  2  3
2  3  4
```

Even though the resulting DataFrame must have the same length as the input DataFrame, it is possible to provide several input functions:

```
>>> s = pd.Series(range(3))
>>> s
0    0
1    1
2    2
dtype: int64
>>> s.transform([np.sqrt, np.exp])
       sqrt        exp
0  0.000000   1.000000
1  1.000000   2.718282
2  1.414214   7.389056
```

You can call transform on a GroupBy object:

```
>>> df = pd.DataFrame({
...     "Date": [
...         "2015-05-08", "2015-05-07", "2015-05-06", "2015-05-05",
...         "2015-05-08", "2015-05-07", "2015-05-06", "2015-05-05"],
...     "Data": [5, 8, 6, 1, 50, 100, 60, 120],
... })
```

(continues on next page)

(continued from previous page)

```
>>> df
         Date  Data
0  2015-05-08     5
1  2015-05-07     8
2  2015-05-06     6
3  2015-05-05     1
4  2015-05-08    50
5  2015-05-07   100
6  2015-05-06    60
7  2015-05-05   120
>>> df.groupby('Date')['Data'].transform('sum')
0     55
1    108
2     66
3    121
4     55
5    108
6     66
7    121
Name: Data, dtype: int64
```

```
>>> df = pd.DataFrame({
...     "c": [1, 1, 1, 2, 2, 2, 2],
...     "type": ["m", "n", "o", "m", "m", "n", "n"]
... })
>>> df
   c type
0  1    m
1  1    n
2  1    o
3  2    m
4  2    m
5  2    n
6  2    n
>>> df['size'] = df.groupby('c')['type'].transform(len)
>>> df
   c type size
0  1    m    3
1  1    n    3
2  1    o    3
3  2    m    4
4  2    m    4
5  2    n    4
6  2    n    4
```

AlloViz.AlloViz.Elements.Nodes.transpose

`Nodes.transpose`(**args*, *copy: bool = False*) → DataFrame

Transpose index and columns.

Reflect the DataFrame over its main diagonal by writing rows as columns and vice-versa. The property `T` is an accessor to the method *transpose()*.

Parameters

***args**
[tuple, optional] Accepted for compatibility with NumPy.

copy
[bool, default False] Whether to copy the data after transposing, even for DataFrames with a single dtype.

Note that a copy is always required for mixed dtype DataFrames, or for DataFrames with any extension types.

Returns

DataFrame
The transposed DataFrame.

See also:

numpy.transpose
Permute the dimensions of a given array.

Notes

Transposing a DataFrame with mixed dtypes will result in a homogeneous DataFrame with the *object* dtype. In such a case, a copy of the data is always made.

Examples

Square DataFrame with homogeneous dtype

```
>>> d1 = {'col1': [1, 2], 'col2': [3, 4]}
>>> df1 = pd.DataFrame(data=d1)
>>> df1
   col1  col2
0     1     3
1     2     4
```

```
>>> df1_transposed = df1.T  # or df1.transpose()
>>> df1_transposed
      0  1
col1  1  2
col2  3  4
```

When the dtype is homogeneous in the original DataFrame, we get a transposed DataFrame with the same dtype:

```
>>> df1.dtypes
col1    int64
col2    int64
dtype: object
>>> df1_transposed.dtypes
0    int64
1    int64
dtype: object
```

Non-square DataFrame with mixed dtypes

```
>>> d2 = {'name': ['Alice', 'Bob'],
...       'score': [9.5, 8],
...       'employed': [False, True],
...       'kids': [0, 0]}
>>> df2 = pd.DataFrame(data=d2)
>>> df2
    name  score  employed  kids
0  Alice    9.5     False     0
1    Bob    8.0      True     0
```

```
>>> df2_transposed = df2.T  # or df2.transpose()
>>> df2_transposed
              0     1
name      Alice   Bob
score       9.5   8.0
employed  False  True
kids          0     0
```

When the DataFrame has mixed dtypes, we get a transposed DataFrame with the *object* dtype:

```
>>> df2.dtypes
name         object
score       float64
employed       bool
kids          int64
dtype: object
>>> df2_transposed.dtypes
0    object
1    object
dtype: object
```

AlloViz.AlloViz.Elements.Nodes.truediv

Nodes.**truediv**(*other*, *axis: Axis = 'columns'*, *level=None*, *fill_value=None*)

Get Floating division of dataframe and other, element-wise (binary operator *truediv*).

Equivalent to `dataframe / other`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *rtruediv*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *floordiv*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other
[scalar, sequence, Series, dict or DataFrame] Any single or multiple element data structure, or list-like object.

axis
[{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns. (1 or 'columns'). For Series input, axis to match Series index on.

level
[int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value
[float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame
Result of the arithmetic operation.

See also:

`DataFrame.add`
Add DataFrames.

`DataFrame.sub`
Subtract DataFrames.

`DataFrame.mul`
Multiply DataFrames.

`DataFrame.div`
Divide DataFrames (float division).

`DataFrame.truediv`
Divide DataFrames (float division).

`DataFrame.floordiv`
Divide DataFrames (integer division).

`DataFrame.mod`
Calculate modulo (remainder after division).

`DataFrame.pow`
Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...         axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a dictionary by axis.

```
>>> df.mul({'angles': 0, 'degrees': 2})
            angles  degrees
circle           0      720
triangle         0      360
rectangle        0      720
```

```
>>> df.mul({'circle': 0, 'triangle': 2, 'rectangle': 3}, axis='index')
            angles  degrees
circle           0        0
triangle         6      360
rectangle       12     1080
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

AlloViz.AlloViz.Elements.Nodes.truncate

Nodes.**truncate**(*before=None*, *after=None*, *axis: int | Literal['index', 'columns', 'rows'] | None = None*, *copy: bool | None = None*) → None

Truncate a Series or DataFrame before and after some index value.

This is a useful shorthand for boolean indexing based on index values above or below certain thresholds.

Parameters

before
[date, str, int] Truncate all rows before this index value.

after
[date, str, int] Truncate all rows after this index value.

axis
[{0 or 'index', 1 or 'columns'}, optional] Axis to truncate. Truncates the index (rows) by default. For *Series* this parameter is unused and defaults to 0.

copy
[bool, default is True,] Return a copy of the truncated section.

Returns

type of caller
The truncated Series or DataFrame.

See also:

DataFrame.loc
Select a subset of a DataFrame by label.

DataFrame.iloc
Select a subset of a DataFrame by position.

Notes

If the index being truncated contains only datetime values, *before* and *after* may be specified as strings instead of Timestamps.

Examples

```
>>> df = pd.DataFrame({'A': ['a', 'b', 'c', 'd', 'e'],
...                    'B': ['f', 'g', 'h', 'i', 'j'],
...                    'C': ['k', 'l', 'm', 'n', 'o']},
...                   index=[1, 2, 3, 4, 5])
>>> df
   A  B  C
1  a  f  k
2  b  g  l
3  c  h  m
4  d  i  n
5  e  j  o
```

```
>>> df.truncate(before=2, after=4)
   A  B  C
2  b  g  l
3  c  h  m
4  d  i  n
```

The columns of a DataFrame can be truncated.

```
>>> df.truncate(before="A", after="B", axis="columns")
   A  B
1  a  f
2  b  g
3  c  h
4  d  i
5  e  j
```

For Series, only rows can be truncated.

```
>>> df['A'].truncate(before=2, after=4)
2    b
3    c
4    d
Name: A, dtype: object
```

The index values in `truncate` can be datetimes or string dates.

```
>>> dates = pd.date_range('2016-01-01', '2016-02-01', freq='s')
>>> df = pd.DataFrame(index=dates, data={'A': 1})
>>> df.tail()
                     A
```

(continues on next page)

(continued from previous page)

```
2016-01-31 23:59:56  1
2016-01-31 23:59:57  1
2016-01-31 23:59:58  1
2016-01-31 23:59:59  1
2016-02-01 00:00:00  1
```

```
>>> df.truncate(before=pd.Timestamp('2016-01-05'),
...             after=pd.Timestamp('2016-01-10')).tail()
                     A
2016-01-09 23:59:56  1
2016-01-09 23:59:57  1
2016-01-09 23:59:58  1
2016-01-09 23:59:59  1
2016-01-10 00:00:00  1
```

Because the index is a DatetimeIndex containing only dates, we can specify *before* and *after* as strings. They will be coerced to Timestamps before truncation.

```
>>> df.truncate('2016-01-05', '2016-01-10').tail()
                     A
2016-01-09 23:59:56  1
2016-01-09 23:59:57  1
2016-01-09 23:59:58  1
2016-01-09 23:59:59  1
2016-01-10 00:00:00  1
```

Note that `truncate` assumes a 0 value for any unspecified time component (midnight). This differs from partial string slicing, which returns any partially matching dates.

```
>>> df.loc['2016-01-05':'2016-01-10', :].tail()
                     A
2016-01-10 23:59:55  1
2016-01-10 23:59:56  1
2016-01-10 23:59:57  1
2016-01-10 23:59:58  1
2016-01-10 23:59:59  1
```

AlloViz.AlloViz.Elements.Nodes.tz_convert

`Nodes.tz_convert`(*tz*, *axis: int | Literal['index', 'columns', 'rows'] = 0*, *level=None*, *copy: bool | None = None*) → None

Convert tz-aware axis to target time zone.

Parameters

tz
: [str or tzinfo object or None] Target time zone. Passing `None` will convert to UTC and remove the timezone information.

axis
: [{0 or 'index', 1 or 'columns'}, default 0] The axis to convert

level
[int, str, default None] If axis is a MultiIndex, convert a specific level. Otherwise must be None.

copy
[bool, default True] Also make a copy of the underlying data.

Returns

Series/DataFrame
Object with time zone converted axis.

Raises

TypeError
If the axis is tz-naive.

Examples

Change to another time zone:

```
>>> s = pd.Series(
...     [1],
...     index=pd.DatetimeIndex(['2018-09-15 01:30:00+02:00']),
... )
>>> s.tz_convert('Asia/Shanghai')
2018-09-15 07:30:00+08:00    1
dtype: int64
```

Pass None to convert to UTC and get a tz-naive index:

```
>>> s = pd.Series([1],
...     index=pd.DatetimeIndex(['2018-09-15 01:30:00+02:00']))
>>> s.tz_convert(None)
2018-09-14 23:30:00    1
dtype: int64
```

AlloViz.AlloViz.Elements.Nodes.tz_localize

`Nodes.tz_localize`(*tz*, *axis: Axis = 0*, *level=None*, *copy: bool_t | None = None*, *ambiguous: TimeAmbiguous = 'raise'*, *nonexistent: TimeNonexistent = 'raise'*) → Self

Localize tz-naive index of a Series or DataFrame to target time zone.

This operation localizes the Index. To localize the values in a timezone-naive Series, use `Series.dt.tz_localize()`.

Parameters

tz
[str or tzinfo or None] Time zone to localize. Passing `None` will remove the time zone information and preserve local time.

axis
[{0 or 'index', 1 or 'columns'}, default 0] The axis to localize

level
[int, str, default None] If axis ia a MultiIndex, localize a specific level. Otherwise must be None.

copy
[bool, default True] Also make a copy of the underlying data.

ambiguous
['infer', bool-ndarray, 'NaT', default 'raise'] When clocks moved backward due to DST, ambiguous times may arise. For example in Central European Time (UTC+01), when going from 03:00 DST to 02:00 non-DST, 02:30:00 local time occurs both at 00:30:00 UTC and at 01:30:00 UTC. In such a situation, the *ambiguous* parameter dictates how ambiguous times should be handled.

- 'infer' will attempt to infer fall dst-transition hours based on order
- bool-ndarray where True signifies a DST time, False designates a non-DST time (note that this flag is only applicable for ambiguous times)
- 'NaT' will return NaT where there are ambiguous times
- 'raise' will raise an AmbiguousTimeError if there are ambiguous times.

nonexistent
[str, default 'raise'] A nonexistent time does not exist in a particular timezone where clocks moved forward due to DST. Valid values are:

- 'shift_forward' will shift the nonexistent time forward to the closest existing time
- 'shift_backward' will shift the nonexistent time backward to the closest existing time
- 'NaT' will return NaT where there are nonexistent times
- timedelta objects will shift nonexistent times by the timedelta
- 'raise' will raise an NonExistentTimeError if there are nonexistent times.

Returns

Series/DataFrame
Same type as the input.

Raises

TypeError
If the TimeSeries is tz-aware and tz is not None.

Examples

Localize local times:

```
>>> s = pd.Series(
...     [1],
...     index=pd.DatetimeIndex(['2018-09-15 01:30:00']),
... )
>>> s.tz_localize('CET')
2018-09-15 01:30:00+02:00    1
dtype: int64
```

Pass None to convert to tz-naive index and preserve local time:

```
>>> s = pd.Series([1],
...     index=pd.DatetimeIndex(['2018-09-15 01:30:00+02:00']))
>>> s.tz_localize(None)
2018-09-15 01:30:00    1
dtype: int64
```

Be careful with DST changes. When there is sequential data, pandas can infer the DST time:

```
>>> s = pd.Series(range(7),
...               index=pd.DatetimeIndex(['2018-10-28 01:30:00',
...                                       '2018-10-28 02:00:00',
...                                       '2018-10-28 02:30:00',
...                                       '2018-10-28 02:00:00',
...                                       '2018-10-28 02:30:00',
...                                       '2018-10-28 03:00:00',
...                                       '2018-10-28 03:30:00']))
>>> s.tz_localize('CET', ambiguous='infer')
2018-10-28 01:30:00+02:00    0
2018-10-28 02:00:00+02:00    1
2018-10-28 02:30:00+02:00    2
2018-10-28 02:00:00+01:00    3
2018-10-28 02:30:00+01:00    4
2018-10-28 03:00:00+01:00    5
2018-10-28 03:30:00+01:00    6
dtype: int64
```

In some cases, inferring the DST is impossible. In such cases, you can pass an ndarray to the ambiguous parameter to set the DST explicitly

```
>>> s = pd.Series(range(3),
...               index=pd.DatetimeIndex(['2018-10-28 01:20:00',
...                                       '2018-10-28 02:36:00',
...                                       '2018-10-28 03:46:00']))
>>> s.tz_localize('CET', ambiguous=np.array([True, True, False]))
2018-10-28 01:20:00+02:00    0
2018-10-28 02:36:00+02:00    1
2018-10-28 03:46:00+01:00    2
dtype: int64
```

If the DST transition causes nonexistent times, you can shift these dates forward or backward with a timedelta object or *'shift_forward'* or *'shift_backward'*.

```
>>> s = pd.Series(range(2),
...               index=pd.DatetimeIndex(['2015-03-29 02:30:00',
...                                       '2015-03-29 03:30:00']))
>>> s.tz_localize('Europe/Warsaw', nonexistent='shift_forward')
2015-03-29 03:00:00+02:00    0
2015-03-29 03:30:00+02:00    1
dtype: int64
>>> s.tz_localize('Europe/Warsaw', nonexistent='shift_backward')
2015-03-29 01:59:59.999999999+01:00    0
2015-03-29 03:30:00+02:00              1
```

(continues on next page)

(continued from previous page)

```
dtype: int64
>>> s.tz_localize('Europe/Warsaw', nonexistent=pd.Timedelta('1H'))
2015-03-29 03:30:00+02:00    0
2015-03-29 03:30:00+02:00    1
dtype: int64
```

AlloViz.AlloViz.Elements.Nodes.unstack

Nodes.**unstack**(*level: IndexLabel = -1*, *fill_value=None*, *sort: bool = True*)

Pivot a level of the (necessarily hierarchical) index labels.

Returns a DataFrame having a new level of column labels whose inner-most level consists of the pivoted index labels.

If the index is not a MultiIndex, the output will be a Series (the analogue of stack when the columns are not a MultiIndex).

Parameters

level
[int, str, or list of these, default -1 (last level)] Level(s) of index to unstack, can pass level name.

fill_value
[int, str or dict] Replace NaN with this value if the unstack produces missing values.

sort
[bool, default True] Sort the level(s) in the resulting MultiIndex columns.

Returns

Series or DataFrame

See also:

DataFrame.pivot
Pivot a table based on column values.

DataFrame.stack
Pivot a level of the column labels (inverse operation from *unstack*).

Notes

Reference the user guide for more examples.

Examples

```
>>> index = pd.MultiIndex.from_tuples([('one', 'a'), ('one', 'b'),
...                                    ('two', 'a'), ('two', 'b')])
>>> s = pd.Series(np.arange(1.0, 5.0), index=index)
>>> s
one  a   1.0
     b   2.0
two  a   3.0
     b   4.0
dtype: float64
```

```
>>> s.unstack(level=-1)
     a   b
one  1.0  2.0
two  3.0  4.0
```

```
>>> s.unstack(level=0)
   one  two
a  1.0   3.0
b  2.0   4.0
```

```
>>> df = s.unstack(level=0)
>>> df.unstack()
one  a  1.0
     b  2.0
two  a  3.0
     b  4.0
dtype: float64
```

AlloViz.AlloViz.Elements.Nodes.update

Nodes.**update**(*other*, *join: UpdateJoin = 'left'*, *overwrite: bool = True*, *filter_func=None*, *errors: IgnoreRaise = 'ignore'*) → None

Modify in place using non-NA values from another DataFrame.

Aligns on indices. There is no return value.

Parameters

other
[DataFrame, or object coercible into a DataFrame] Should have at least one matching index/column label with the original DataFrame. If a Series is passed, its name attribute must be set, and that will be used as the column name to align with the original DataFrame.

join
[{'left'}, default 'left'] Only left join is implemented, keeping the index and columns of the original object.

overwrite
[bool, default True] How to handle non-NA values for overlapping keys:

- True: overwrite original DataFrame's values with values from *other*.

- False: only update values that are NA in the original DataFrame.

filter_func
[callable(1d-array) -> bool 1d-array, optional] Can choose to replace values other than NA. Return True for values that should be updated.

errors
[{'raise', 'ignore'}, default 'ignore'] If 'raise', will raise a ValueError if the DataFrame and *other* both contain non-NA data in the same place.

Returns

None
This method directly changes calling object.

Raises

ValueError

- When *errors='raise'* and there's overlapping non-NA data.
- When *errors* is not either *'ignore'* or *'raise'*

NotImplementedError

- If *join != 'left'*

See also:

`dict.update`
Similar method for dictionaries.

`DataFrame.merge`
For column(s)-on-column(s) operations.

Examples

```
>>> df = pd.DataFrame({'A': [1, 2, 3],
...                    'B': [400, 500, 600]})
>>> new_df = pd.DataFrame({'B': [4, 5, 6],
...                        'C': [7, 8, 9]})
>>> df.update(new_df)
>>> df
   A  B
0  1  4
1  2  5
2  3  6
```

The DataFrame's length does not increase as a result of the update, only values at matching index/column labels are updated.

```
>>> df = pd.DataFrame({'A': ['a', 'b', 'c'],
...                    'B': ['x', 'y', 'z']})
>>> new_df = pd.DataFrame({'B': ['d', 'e', 'f', 'g', 'h', 'i']})
>>> df.update(new_df)
>>> df
   A  B
0  a  d
```

(continues on next page)

(continued from previous page)

```
1  b  e
2  c  f
```

For Series, its name attribute must be set.

```
>>> df = pd.DataFrame({'A': ['a', 'b', 'c'],
...                    'B': ['x', 'y', 'z']})
>>> new_column = pd.Series(['d', 'e'], name='B', index=[0, 2])
>>> df.update(new_column)
>>> df
   A  B
0  a  d
1  b  y
2  c  e
>>> df = pd.DataFrame({'A': ['a', 'b', 'c'],
...                    'B': ['x', 'y', 'z']})
>>> new_df = pd.DataFrame({'B': ['d', 'e']}, index=[1, 2])
>>> df.update(new_df)
>>> df
   A  B
0  a  x
1  b  d
2  c  e
```

If *other* contains NaNs the corresponding values are not updated in the original dataframe.

```
>>> df = pd.DataFrame({'A': [1, 2, 3],
...                    'B': [400, 500, 600]})
>>> new_df = pd.DataFrame({'B': [4, np.nan, 6]})
>>> df.update(new_df)
>>> df
   A    B
0  1    4
1  2  500
2  3    6
```

AlloViz.AlloViz.Elements.Nodes.value_counts

Nodes.**value_counts**(*subset: IndexLabel | None = None*, *normalize: bool = False*, *sort: bool = True*, *ascending: bool = False*, *dropna: bool = True*) → Series

Return a Series containing the frequency of each distinct row in the Dataframe.

Parameters

subset
[label or list of labels, optional] Columns to use when counting unique combinations.

normalize
[bool, default False] Return proportions rather than frequencies.

sort
[bool, default True] Sort by frequencies when True. Sort by DataFrame column

values when False.

ascending
[bool, default False] Sort in ascending order.

dropna
[bool, default True] Don't include counts of rows that contain NA values.

New in version 1.3.0.

Returns

Series

See also:

Series.value_counts
Equivalent method on Series.

Notes

The returned Series will have a MultiIndex with one level per input column but an Index (non-multi) for a single label. By default, rows that contain any NA values are omitted from the result. By default, the resulting Series will be in descending order so that the first element is the most frequently-occurring row.

Examples

```
>>> df = pd.DataFrame({'num_legs': [2, 4, 4, 6],
...                    'num_wings': [2, 0, 0, 0]},
...                   index=['falcon', 'dog', 'cat', 'ant'])
>>> df
        num_legs  num_wings
falcon         2          2
dog            4          0
cat            4          0
ant            6          0
```

```
>>> df.value_counts()
num_legs  num_wings
4         0            2
2         2            1
6         0            1
Name: count, dtype: int64
```

```
>>> df.value_counts(sort=False)
num_legs  num_wings
2         2            1
4         0            2
6         0            1
Name: count, dtype: int64
```

```
>>> df.value_counts(ascending=True)
num_legs  num_wings
2         2            1
```

(continues on next page)

(continued from previous page)

```
6         0            1
4         0            2
Name: count, dtype: int64
```

```
>>> df.value_counts(normalize=True)
num_legs  num_wings
4         0            0.50
2         2            0.25
6         0            0.25
Name: proportion, dtype: float64
```

With *dropna* set to *False* we can also count rows with NA values.

```
>>> df = pd.DataFrame({'first_name': ['John', 'Anne', 'John', 'Beth'],
...                    'middle_name': ['Smith', pd.NA, pd.NA, 'Louise']})
>>> df
  first_name middle_name
0       John       Smith
1       Anne        <NA>
2       John        <NA>
3       Beth      Louise
```

```
>>> df.value_counts()
first_name  middle_name
Beth        Louise         1
John        Smith          1
Name: count, dtype: int64
```

```
>>> df.value_counts(dropna=False)
first_name  middle_name
Anne        NaN            1
Beth        Louise         1
John        Smith          1
            NaN            1
Name: count, dtype: int64
```

```
>>> df.value_counts("first_name")
first_name
John    2
Anne    1
Beth    1
Name: count, dtype: int64
```

AlloViz.AlloViz.Elements.Nodes.var

Nodes.**var**(*axis: Axis | None = 0*, *skipna: bool = True*, *ddof: int = 1*, *numeric_only: bool = False*, ***kwargs*)

Return unbiased variance over requested axis.

Normalized by N-1 by default. This can be changed using the ddof argument.

Parameters

axis
[{index (0), columns (1)}] For *Series* this parameter is unused and defaults to 0.

skipna
[bool, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

ddof
[int, default 1] Delta Degrees of Freedom. The divisor used in calculations is N - ddof, where N represents the number of elements.

numeric_only
[bool, default False] Include only float, int, boolean columns. Not implemented for Series.

Returns

Series or DataFrame (if level specified)

Examples

```
>>> df = pd.DataFrame({'person_id': [0, 1, 2, 3],
...                    'age': [21, 25, 62, 43],
...                    'height': [1.61, 1.87, 1.49, 2.01]}
...                   ).set_index('person_id')
>>> df
           age  height
person_id
0           21    1.61
1           25    1.87
2           62    1.49
3           43    2.01
```

```
>>> df.var()
age       352.916667
height      0.056367
dtype: float64
```

Alternatively, `ddof=0` can be set to normalize by N instead of N-1:

```
>>> df.var(ddof=0)
age       264.687500
height      0.042275
dtype: float64
```

AlloViz.AlloViz.Elements.Nodes.view

Nodes.**view**(*metric*, *num=20*, *colors=['orange', 'turquoise']*, *nv=None*)

Represent the selected metric in the structure

Retrieves the analyzed data corresponding to the present Element (depending on the class) and from it the corresponding metric column from the DataFrame. It is used to obtain the elements' colors, sizes (inv. proportional to errors, if available) and names (resnames); and the parent instance attribute is used to retrieve the structure for representation using `nglview.NGLWidget`.

Data is sorted according to the selected metric in absolute value and descending order, a `LinearSegmentedColormap` is made with the passed colors. The colormap is represented in a colorbar through `_show_cbar()` and is used to establish the elements' colors through `_get_colors()`.

Errors, if available (i.e., if more than one trajectory has been used and averages were calculated), are used to establish the elements' sizes to be inversely proportional to them (interpolated between 1 and 0.1), thus directly proportional to the "confidence" in the calculated value in a way.

Elements are shown on a representation of the selected structure or added to the passed `NGLWidget` if applicable.

Parameters

metric
[str] Metric/Name of the column in the object's df attribute to represent.

num
[int, default: 20] Number of (each of the) network elements to show on the structure.

colors
[list, default: ["orange", "turquoise"]] List of two colors to assign to the minimum and maximum values of the network to be represented, respectively. Middle value is assigned "white" and it will be the mean of the network values or 0 if the network has both negative and positive values.

nv
[`nglview.NGLWidget`, optional] A structure representation into which the shapes representing the chosen network elements will be added.

See also:

AlloViz.Protein.view

AlloViz.AlloViz.Elements.Nodes.where

Nodes.**where**(*cond*, *other=nan*, ***, *inplace: bool_t = False*, *axis: Axis | None = None*, *level: Level | None = None*) → Self | None

Replace values where the condition is False.

Parameters

cond
[bool Series/DataFrame, array-like, or callable] Where *cond* is True, keep the original value. Where False, replace with corresponding value from *other*. If *cond* is callable, it is computed on the Series/DataFrame and should return boolean Series/DataFrame or array. The callable must not change input Series/DataFrame (though pandas doesn't check it).

other
[scalar, Series/DataFrame, or callable] Entries where *cond* is False are replaced with corresponding value from *other*. If other is callable, it is computed on the Series/DataFrame and should return scalar or Series/DataFrame. The callable must not change input Series/DataFrame (though pandas doesn't check it). If not specified, entries will be filled with the corresponding NULL value (`np.nan` for numpy dtypes, `pd.NA` for extension dtypes).

inplace
[bool, default False] Whether to perform the operation in place on the data.

axis
[int, default None] Alignment axis if needed. For *Series* this parameter is unused and defaults to 0.

level
[int, default None] Alignment level if needed.

Returns

Same type as caller or None if `inplace=True`.

See also:

`DataFrame.mask()`
Return an object of same shape as self.

Notes

The where method is an application of the if-then idiom. For each element in the calling DataFrame, if `cond` is `True` the element is used; otherwise the corresponding element from the DataFrame `other` is used. If the axis of `other` does not align with axis of `cond` Series/DataFrame, the misaligned index positions will be filled with False.

The signature for `DataFrame.where()` differs from `numpy.where()`. Roughly `df1.where(m, df2)` is equivalent to `np.where(m, df1, df2)`.

For further details and examples see the `where` documentation in indexing.

The dtype of the object takes precedence. The fill value is casted to the object's dtype, if this can be done losslessly.

Examples

```
>>> s = pd.Series(range(5))
>>> s.where(s > 0)
0    NaN
1    1.0
2    2.0
3    3.0
4    4.0
dtype: float64
>>> s.mask(s > 0)
0    0.0
1    NaN
2    NaN
```

(continues on next page)

(continued from previous page)

```
3    NaN
4    NaN
dtype: float64
```

```
>>> s = pd.Series(range(5))
>>> t = pd.Series([True, False])
>>> s.where(t, 99)
0     0
1    99
2    99
3    99
4    99
dtype: int64
>>> s.mask(t, 99)
0    99
1     1
2    99
3    99
4    99
dtype: int64
```

```
>>> s.where(s > 1, 10)
0    10
1    10
2    2
3    3
4    4
dtype: int64
>>> s.mask(s > 1, 10)
0     0
1     1
2    10
3    10
4    10
dtype: int64
```

```
>>> df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])
>>> df
   A  B
0  0  1
1  2  3
2  4  5
3  6  7
4  8  9
>>> m = df % 3 == 0
>>> df.where(m, -df)
   A  B
0  0 -1
1 -2  3
2 -4 -5
3  6 -7
```

(continues on next page)

(continued from previous page)

```
4 -8  9
>>> df.where(m, -df) == np.where(m, df, -df)
      A     B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True
>>> df.where(m, -df) == df.mask(~m, -df)
      A     B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True
```

AlloViz.AlloViz.Elements.Nodes.xs

Nodes.**xs**(*key: Hashable | Sequence[Hashable]*, *axis: int | Literal['index', 'columns', 'rows'] = 0*, *level: Hashable | Sequence[Hashable] | None = None*, *drop_level: bool = True*) → None

Return cross-section from the Series/DataFrame.

This method takes a *key* argument to select data at a particular level of a MultiIndex.

Parameters

key
: [label or tuple of label] Label contained in the index, or partially in a MultiIndex.

axis
: [{0 or 'index', 1 or 'columns'}, default 0] Axis to retrieve cross-section on.

level
: [object, defaults to first n levels (n=1 or len(key))] In case of a key partially contained in a MultiIndex, indicate which levels are used. Levels can be referred by label or position.

drop_level
: [bool, default True] If False, returns object with same levels as self.

Returns

Series or DataFrame
: Cross-section from the original Series or DataFrame corresponding to the selected index levels.

See also:

`DataFrame.loc`
: Access a group of rows and columns by label(s) or a boolean array.

`DataFrame.iloc`
: Purely integer-location based indexing for selection by position.

Notes

xs can not be used to set values.

MultiIndex Slicers is a generic way to get/set values on any level or levels. It is a superset of *xs* functionality, see MultiIndex Slicers.

Examples

```
>>> d = {'num_legs': [4, 4, 2, 2],
...      'num_wings': [0, 0, 2, 2],
...      'class': ['mammal', 'mammal', 'mammal', 'bird'],
...      'animal': ['cat', 'dog', 'bat', 'penguin'],
...      'locomotion': ['walks', 'walks', 'flies', 'walks']}
>>> df = pd.DataFrame(data=d)
>>> df = df.set_index(['class', 'animal', 'locomotion'])
>>> df
                           num_legs  num_wings
class  animal  locomotion
mammal cat     walks              4          0
       dog     walks              4          0
       bat     flies              2          2
bird   penguin walks              2          2
```

Get values at specified index

```
>>> df.xs('mammal')
                    num_legs  num_wings
animal locomotion
cat    walks              4          0
dog    walks              4          0
bat    flies              2          2
```

Get values at several indexes

```
>>> df.xs(('mammal', 'dog', 'walks'))
num_legs     4
num_wings    0
Name: (mammal, dog, walks), dtype: int64
```

Get values at specified index and level

```
>>> df.xs('cat', level=1)
                    num_legs  num_wings
class  locomotion
mammal walks              4          0
```

Get values at several indexes and levels

```
>>> df.xs(('bird', 'walks'),
...        level=[0, 'locomotion'])
         num_legs  num_wings
animal
penguin         2          2
```

Get values at specified column and axis

```
>>> df.xs('num_wings', axis=1)
class   animal   locomotion
mammal  cat      walks         0
        dog      walks         0
        bat      flies         2
bird    penguin  walks         2
Name: num_wings, dtype: int64
```

AlloViz.AlloViz.Filtering

Module with classes to filter and store networks

Functions of the module (and/or combinations of them) are used for filtering the passed raw networks with the main class *Filtering*.

Functions

All(pkg, data, **kwargs)	No filtering
GPCR_Interhelix(pkg, data, **kwargs)	Retain only edges between different TMs(/ECLs/ICLs) of GPCRs
GetContacts_edges(pkg, data, ...)	Retain only edges found by GetContacts
No_Sequence_Neighbors(pkg, data, ...)	Filter out residue pairs too close in the sequence
Spatially_distant(pkg, data, ...)	Retain only edges between spatially distant residue pairs

AlloViz.AlloViz.Filtering.All

AlloViz.AlloViz.Filtering.**All**(*pkg*, *data*, ***kwargs*)

No filtering

AlloViz.AlloViz.Filtering.GPCR_Interhelix

AlloViz.AlloViz.Filtering.**GPCR_Interhelix**(*pkg*, *data*, ***kwargs*)

Retain only edges between different TMs(/ECLs/ICLs) of GPCRs

It can only be used for GPCR systems (*GPCR==True*) for which GPCRdb's generic numbering could be retrieved, and it only retains edges between residue pairs that have generic numbering and are in different transmembrane helices (and/or intra-cellular or extracellular loops) according to it.

AlloViz.AlloViz.Filtering.GetContacts_edges

AlloViz.AlloViz.Filtering.**GetContacts_edges**(*pkg*, *data*, *GetContacts_threshold*, ***kwargs*)

Retain only edges found by GetContacts

It retains only the raw edges for which *GetContacts* has been able to calculate a contact frequency value, i.e., residue pairs that are in physicochemical contact.

GetContacts data is retrieved from the passed *pkg*'s `Protein` object stored in it and thus is needed to have been calculated previously. It may (or not) have been calculated using the *GetContacts_threshold* kwarg, filtered afterwards with `AlloViz.Wrappers.GetContacts.GetContacts.filter_contacts()`, or the *GetContacts_threshold* kwarg can also be passed to the analysis to filter the *GetContacts* data in this point.

Other Parameters

****kwargs**

GetContacts_threshold kwarg can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

AlloViz.AlloViz.Filtering.No_Sequence_Neighbors

AlloViz.AlloViz.Filtering.**No_Sequence_Neighbors**(*pkg*, *data*, *Sequence_Neighbor_distance*, ***kwargs*)

Filter out residue pairs too close in the sequence

It only retains edges between residue pairs that are minimum a certain number of positions away in the protein sequence (default: 5). It can be assumed that residues that are close in the protein sequence will be considerably correlated and/or interacting, because of their intrinsic short spatial distance. Specially, it is known that a turn of an alpha-helix is completed almost every 4 residues, and that the residues i and i+4 (and i-5) in an alpha-helix interact with a backbone hydrogen bond. Thus, to take out these interactions and also the contacts due to closeness of residues i and i+5, the default value is 5 sequence positions.

Other Parameters

****kwargs**

Sequence_Neighbor_distance kwarg can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in Intercontact filtering, which defaults to 5.

AlloViz.AlloViz.Filtering.Spatially_distant

AlloViz.AlloViz.Filtering.**Spatially_distant**(*pkg*, *data*, *Interresidue_distance*, ***kwargs*)

Retain only edges between spatially distant residue pairs

It only retains edges between residue pairs whose CA atoms are minimum a certain number of angstroms away from each other in the initial PDB/structure (default 10 Å). The relationship found between these residues can be considered purely allosteric, as they are spatially distant and have no direct communication but can be found to be interacting/correlated…

Other Parameters

****kwargs**

'Interresidue_distance' kwarg can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

Classes

Filtering(pkg, filtering, name, *[, ...])	Class for network filtering

AlloViz.AlloViz.Filtering.Filtering

class AlloViz.AlloViz.Filtering.**Filtering**(*pkg*, *filtering*, *name*, *, *GetContacts_threshold=0*, *Sequence_Neighbor_distance=5*, *Interresidue_distance=10*)

Bases: `object`

Class for network filtering

Instances of this class are used to be added as attributes to instances of Wrappers' classes (as attributes of a *AlloViz.Protein* object) with the purpose of storing (un)filtered networks according to different criteria.

Raw network edges stored as a DataFrame in the passed *pkg* object are filtered with the corresponding function or combination of functions of the *Filtering* module and converted to individual NetworkX' Graphs with `_get_G()` to be stored as private attributes.

Parameters

pkg
[instance of a class from *AlloViz.Wrappers*] The object is used to retrieve the raw network edges for network analysis. It also contains/gives access to the corresponding:class:*~AlloViz.Protein* object and its information.

filtering
[str or list] Filtering scheme(s) with which to filter the list of network edges. A list of strings is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*, *Spatially_distant()*.

name
[str] Name of the filtering scheme. It will be the same name as the passed filtering option if it is a single string, or the names of the filtering options joined by "_" if a lsit has been passed. It is used to name the pkg's attribute in which the class instance is saved.

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in No_Sequence_Neighbors filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.Protein.filter
Class method to filter the network(s) raw edge weights with different criteria.

AlloViz.Wrappers.Base.Base.filter

Pkg's method to filter the network(s) raw edge weights with different criteria.

Attributes

graphs
[dict of external:ref:*Graph <graph>* objects]

Methods

analyze([elements, metrics, cores, ...])	Analyze the filtered network

AlloViz.AlloViz.Filtering.Filtering.analyze

Filtering.**analyze**(*elements='edges'*, *metrics='all'*, *cores=1*, *nodes_dict={'btw': 'networkx.algorithms.centrality.betweenness_centrality', 'cfb': 'networkx.algorithms.centrality.current_flow_betweenness_centrality'}*, *edges_dict={'btw': 'networkx.algorithms.centrality.edge_betweenness_centrality', 'cfb': 'networkx.algorithms.centrality.edge_current_flow_betweenness_centrality'}*, ***kwargs*)

Analyze the filtered network

Send the analyses of the filtered network for the passed combinations of elements-metrics. The individual Graphs saved as private attributes upon object initialization are analyzed independently with the *Analysis* module (this function calls *AlloViz.AlloViz.Analysis.analyze()*). Results are stored as new instances of classes from the *AlloViz.AlloViz.Elements* module, which extend the pandas.DataFrame class.

Parameters

elements
[str or list, {"edges", "nodes"}] Network element for which to perform the analysis.

metrics
[str or list, default: "all"] Network metrics to compute, which must be keys in the *nodes_dict* or *edges_dict* dictionaries. Default is "all" and it sends the computation for all the metrics defined in the corresponding dictionary of the selected elements in *element*.

cores
[int, default: 1] Number of cores to use for parallelization with a *multiprocess* Pool. Default value only uses 1 core with a custom `AlloViz.utils.dummypool` that performs computations synchronously.

Other Parameters

nodes_dict, edges_dict
[dict] Optional kwarg(s) of the dictionary(ies) that maps network metrics custom names (e.g., betweenness centrality, "btw") with their corresponding NetworkX function (e.g., "networkx.algorithms.centrality.betweenness_centrality"). Functions strings must be written as if they were absolute imports, and must return a dictionary of edges or nodes, depending on the element dictionary in which they are. The keys of the dictionaries will be used to name the columns of the analyzed data that the functions produce. Defaults are *nodes_dict* and *edges_dict*.

****kwargs**
Other optional keyword arguments that will be passed to the NetworkX analysis function(s) that is(are) used on the method call in case they need extra parameters.

Exceptions

`NoNetworkException`

AlloViz.AlloViz.info

Module containing variables and elements with AlloViz information

The main member is the *wrappers* dictionary, which contains all the available network construction methods in AlloViz as keys and the kind of information they use as their respective values.

Module Attributes

wrappers	Dictionary with all the available network construction methods and their information
dihedrals_atoms	Dictionary with MDAnalysis' selection functions-ready dihedral-participating atomnames

AlloViz.AlloViz.info.wrappers

AlloViz.AlloViz.info.wrappers = {'AlloViz_Backbone_Dihs': ('Dihedral angles', 'AlloViz', 'MI', 'All backbone dihedrals (Phi and psi)'), 'AlloViz_Chi1': ('Dihedral angles', 'AlloViz', 'MI', 'Chi1'), 'AlloViz_Chi2': ('Dihedral angles', 'AlloViz', 'MI', 'Chi2'), 'AlloViz_Chi3': ('Dihedral angles', 'AlloViz', 'MI', 'Chi3'), 'AlloViz_Chi4': ('Dihedral angles', 'AlloViz', 'MI', 'Chi4'), 'AlloViz_Dihs': ('Dihedral angles', 'AlloViz', 'MI', 'All dihedrals'), 'AlloViz_Phi': ('Dihedral angles', 'AlloViz', 'MI', 'Phi'), 'AlloViz_Psi': ('Dihedral angles', 'AlloViz', 'MI', 'Psi'), 'AlloViz_Sidechain_Dihs': ('Dihedral angles', 'AlloViz', 'MI', 'All side-chain dihedrals'), 'CARDS_Disorder_Backbone_Dihs': ('Dihedral angles', 'CARDS', 'Pure-disorder MI', 'All backbone dihedrals (Phi and psi)'), 'CARDS_Disorder_Chi1': ('Dihedral angles', 'CARDS', 'Pure-disorder MI', 'Chi1'), 'CARDS_Disorder_Chi2': ('Dihedral angles', 'CARDS', 'Pure-disorder MI', 'Chi2'), 'CARDS_Disorder_Chi3': ('Dihedral angles', 'CARDS', 'Pure-disorder MI', 'Chi3'), 'CARDS_Disorder_Chi4': ('Dihedral angles', 'CARDS', 'Pure-disorder MI', 'Chi4'), 'CARDS_Disorder_Dihs': ('Dihedral angles', 'CARDS', 'Pure-disorder MI', 'All dihedrals'), 'CARDS_Disorder_Phi': ('Dihedral angles', 'CARDS', 'Pure-disorder MI', 'Phi'), 'CARDS_Disorder_Psi': ('Dihedral angles', 'CARDS', 'Pure-disorder MI', 'Psi'), 'CARDS_Disorder_Sidechain_Dihs': ('Dihedral angles', 'CARDS', 'Pure-disorder MI', 'All side-chain dihedrals'), 'CARDS_Disorder_mediated_Backbone_Dihs': ('Dihedral angles', 'CARDS', 'Disorder-mediated MI', 'All backbone dihedrals (Phi and psi)'), 'CARDS_Disorder_mediated_Chi1': ('Dihedral angles', 'CARDS', 'Disorder-mediated MI', 'Chi1'), 'CARDS_Disorder_mediated_Chi2': ('Dihedral angles', 'CARDS', 'Disorder-mediated MI', 'Chi2'), 'CARDS_Disorder_mediated_Chi3': ('Dihedral angles', 'CARDS', 'Disorder-mediated MI', 'Chi3'), 'CARDS_Disorder_mediated_Chi4': ('Dihedral angles', 'CARDS', 'Disorder-mediated MI', 'Chi4'), 'CARDS_Disorder_mediated_Dihs': ('Dihedral angles', 'CARDS', 'Disorder-mediated MI', 'All dihedrals'), 'CARDS_Disorder_mediated_Phi': ('Dihedral angles', 'CARDS', 'Disorder-mediated MI', 'Phi'), 'CARDS_Disorder_mediated_Psi': ('Dihedral angles', 'CARDS', 'Disorder-mediated MI', 'Psi'), 'CARDS_Disorder_mediated_Sidechain_Dihs': ('Dihedral angles', 'CARDS', 'Disorder-mediated MI', 'All side-chain dihedrals'), 'CARDS_Holistic_Backbone_Dihs': ('Dihedral angles', 'CARDS', 'Holistic MI', 'All backbone dihedrals (Phi and psi)'), 'CARDS_Holistic_Chi1': ('Dihedral angles', 'CARDS', 'Holistic MI', 'Chi1'), 'CARDS_Holistic_Chi2': ('Dihedral angles', 'CARDS', 'Holistic MI', 'Chi2'), 'CARDS_Holistic_Chi3': ('Dihedral angles', 'CARDS', 'Holistic MI', 'Chi3'), 'CARDS_Holistic_Chi4': ('Dihedral angles', 'CARDS', 'Holistic MI', 'Chi4'), 'CARDS_Holistic_Dihs': ('Dihedral angles', 'CARDS', 'Holistic MI', 'All dihedrals'), 'CARDS_Holistic_Phi': ('Dihedral angles', 'CARDS', 'Holistic MI', 'Phi'), 'CARDS_Holistic_Psi': ('Dihedral angles', 'CARDS', 'Holistic MI', 'Psi'), 'CARDS_Holistic_Sidechain_Dihs': ('Dihedral angles', 'CARDS', 'Holistic MI', 'All side-chain dihedrals'), 'CARDS_MI_Backbone_Dihs': ('Dihedral angles', 'CARDS', 'MI', 'All backbone dihedrals (Phi and psi)'), 'CARDS_MI_Chi1': ('Dihedral angles', 'CARDS', 'MI', 'Chi1'), 'CARDS_MI_Chi2': ('Dihedral angles', 'CARDS', 'MI', 'Chi2'), 'CARDS_MI_Chi3': ('Dihedral angles', 'CARDS', 'MI', 'Chi3'), 'CARDS_MI_Chi4': ('Dihedral angles', 'CARDS', 'MI', 'Chi4'), 'CARDS_MI_Dihs': ('Dihedral angles', 'CARDS', 'MI', 'All dihedrals'), 'CARDS_MI_Phi': ('Dihedral angles', 'CARDS', 'MI', 'Phi'), 'CARDS_MI_Psi': ('Dihedral angles', 'CARDS', 'MI', 'Psi'), 'CARDS_MI_Sidechain_Dihs': ('Dihedral angles', 'CARDS', 'MI', 'All side-chain dihedrals'), 'GetContacts': ('Contacts', 'GetContacts', 'None', 'Contacts occurrence'), 'MDEntropy_AlphaAngle': ('Dihedral angles', 'MDEntropy', 'MI', 'Alpha angle'), 'MDEntropy_Backbone_Dihs': ('Dihedral angles', 'MDEntropy', 'MI', 'All backbone dihedrals (Phi and psi)'), 'MDEntropy_Contacts': ('Contacts', 'MDEntropy', 'MI', 'Contacts occurrence'), 'MDEntropy_Phi': ('Dihedral angles', 'MDEntropy', 'MI', 'Phi'), 'MDEntropy_Psi': ('Dihedral angles', 'MDEntropy', 'MI', 'Psi'), 'MDTASK': ("Atoms' displacements", 'MD-TASK', "Pearson's", 'Carbon '), 'PyInteraph2_Atomic_Contacts_Occurrence': ('Contacts', 'PyInteraph2', 'None', 'Atomic contacts occurrence'), 'PyInteraph2_Atomic_Contacts_Strength': ('Contacts', 'PyInteraph2', 'None', 'Atomic contacts strength'), 'PyInteraph2_Contacts': ('Contacts', 'PyInteraph2', 'None', 'Contacts occurrence'), 'PyInteraph2_Energy': ('Interaction energy', 'PyInteraph2', 'None', 'Whole residue'), 'correlationplus_Backbone_Dihs': ('Dihedral angles', 'correlationplus', "Pearson's", 'All backbone dihedrals (Phi and psi)'), 'correlationplus_CA_LMI': ("Atoms' displacements", 'correlationplus', 'LMI', 'Carbon '), 'correlationplus_CA_Pear': ("Atoms' displacements", 'correlationplus',

Dictionary with all the available network construction methods and their information

All network construction methods available in AlloViz are the keys of the dictionary and each of the values is a 4-member tuple. In order, the tuple contains the kind of information it uses for network construction, the main package name (not the same as the AlloViz accession name of the wrapper), the correlaton metric they use if applicable, and the atom/angle whose information it uses.

AlloViz.AlloViz.info.dihedrals_atoms

AlloViz.AlloViz.info.**dihedrals_atoms = {'chi1': {'ca_name': 'CA', 'cb_name': 'CB', 'cg_name': 'CG CG1 OG OG1 SG', 'n_name': 'N'}, 'chi2': {'ca_name': 'CB', 'cb_name': 'CG CG1', 'cg_name': 'CD OD1 ND1 CD1 SD', 'n_name': 'CA'}, 'chi3': {'ca_name': 'CG', 'cb_name': 'CD SD', 'cg_name': 'NE OE1 CE', 'n_name': 'CB'}, 'chi4': {'ca_name': 'CD', 'cb_name': 'CE NE', 'cg_name': 'CZ NZ', 'n_name': 'CG'}, 'chi5': {'ca_name': 'NE', 'cb_name': 'CZ', 'cg_name': 'NH1', 'n_name': 'CD'}, 'omega': {'c_name': 'C', 'ca_name': 'CA', 'n_name': 'N'}, 'phi': {'c_name': 'C', 'ca_name': 'CA', 'n_name': 'N'}, 'psi': {'c_name': 'C', 'ca_name': 'CA', 'n_name': 'N'}}**

Dictionary with MDAnalysis' selection functions-ready dihedral-participating atomnames

Each entry from the dictionary corresponds to a dihedral and is a dictionary itself ready to be passed to MDAnalysis' selection functions (phi_selection, etc) as kwargs. Chi2 and onward can be selected with the chi1_selection function passing custom atom names.

Info retrieved from https://docs.mdanalysis.org/stable/documentation_pages/core/topologyattrs.html#MDAnalysis.core.topologyattrs.Atomnames.chi1_selection and http://www.mlb.co.jp/linux/science/garlic/doc/commands/dihedrals.html.

AlloViz.AlloViz.trajutils

Module containing functions for trajectory processing

Functions in this module are used by *AlloViz.Protein* for input file processing or by other functions herein to process and prepare the passed input files for analysis with the rest of AlloViz's functionalities.

Functions

download_GPCRmd_files(GPCRmdid, path)	Download the files of a GPCRmd dynid stored in the database
download_SCoV2MD_files(scov2mdid, path)	Download the files of a SARSCoV2-MD dynid stored in the database
get_GPCRdb_numbering(protein)	Retrieve the GPCRdb-assigned generic numbering of a GPCR structure
get_bonded_cys(pdbf)	Identify disulfide bond-forming cysteines' sulphur atoms
process_input(GPCR, pdb, protein_sel, pdbf, ...)	Process the input structure and trajectory(ies) files
standardize_resnames(protein, **kwargs)	Change the residue names of a protein to standard 3-letter codes

AlloViz.AlloViz.trajutils.download_GPCRmd_files

AlloViz.AlloViz.trajutils.**download_GPCRmd_files**(*GPCRmdid*, *path*)

Download the files of a GPCRmd dynid stored in the database

The website of the corresponding GPCRmd ID is scanned to retrieve the downloadable files and they are downloaded in parallel in the passed path with the same name that they have in the database.

Parameters

GPCRmdid
[int] Dynid from the GPCRmd database.

path
[str] Path, relative or absolute, in which to download the files.

Notes

The force-field parameters file is downloaded as a tar file and is extracted and also transformed in-place to avoid downstream problems with the gRINN construction method.

AlloViz.AlloViz.trajutils.download_SCoV2MD_files

AlloViz.AlloViz.trajutils.**download_SCoV2MD_files**(*scov2mdid*, *path*)

Download the files of a SARSCoV2-MD dynid stored in the database

The website of the corresponding ID is scanned to retrieve the downloadable files and they are downloaded in parallel in the passed path with the same name that they have in the database.

Parameters

scov2mdid
[int] Dynid

path
[str] Path, relative or absolute, in which to download the files.

AlloViz.AlloViz.trajutils.get_GPCRdb_numbering

AlloViz.AlloViz.trajutils.**get_GPCRdb_numbering**(*protein*)

Retrieve the GPCRdb-assigned generic numbering of a GPCR structure

A PDB file with GPCR generic numbering of residues in the B-factor column of the file is retrieved through the GPCRdb API and the GPCRdb-scheme generic numbers are transfered to the *protein* CA atoms' *tempfactors*.

GPCRdb assigns Ballesteros-Weinstein generic numbers to the N atoms' B-factors, and the GPCRdb-scheme ones to the CA atoms'. GPCRdb "bulges" are marked by repeating the generic number but making it negative (negative B-factor instead of a three-decimal B-factor) and must be transformed by *-b + 0.001*. Some residual *1.00* B-factors remain and must be taken care of.

Parameters

protein
[AtomGroup] Atoms of the protein for which to retrieve generic numbers and that will be transformed (its CA atoms' *tempfactors*) in-place.

Notes

GPCRdb provides a PyMOL file to expedite the visualization of the two schemes of generic numbers as text tags over the structure, which is where the information about what is stored in which atom's B-factor column is taken from:

```
print "GPCRdb script labeling generic numbers on the annotated pdb structure\nKey␣
↪bindings for labels:\nF1 - show generic numbers\nF2 - show Ballesteros-Weinstein␣
↪numbers\nF3 - clear labels"
label n. CA or n. N
cmd.set_key('F1', 'label n. CA & (b >-8.1 and  b <8.1), str("%1.2f" %b).replace(".",
↪"x") if b > 0 else str("%1.3f" %(-b + 0.001)).replace(".", "x")')
cmd.set_key('F2', 'label n. N & (b > 0 and  b <8.1), "%1.2f" %b')
cmd.set_key('F3', 'label n. CA or n. N')
```

AlloViz.AlloViz.trajutils.get_bonded_cys

AlloViz.AlloViz.trajutils.**get_bonded_cys**(*pdbf*)

Identify disulfide bond-forming cysteines' sulphur atoms

Returns a list of the residue indices of cysteines whose sulphur atoms form a disulfide bond using ParmEd. These are used downstream by the PyInteraph2_Contacts network construction method to delete them from the results, as the program does not detect the bonds by itself and it results in contact frequencies of '1' that are uninteresting for allosteric communication.

Parameters

pdbf
[str] Filename of the already-processed protein PDB structure to use for detection.

AlloViz.AlloViz.trajutils.process_input

AlloViz.AlloViz.trajutils.**process_input**(*GPCR*, *pdb*, *protein_sel*, *pdbf*, *psf*, *psff*, *trajs*, *trajsf*, ***kwargs*)

Process the input structure and trajectory(ies) files

Only the passed selection is used from the whole input structure file. Its residue names are standardized to the usual 3-letter residue codes (*AlloViz.AlloViz.trajutils.standardize_resnames()*) and, if the structure is a GPCR, the generic numbers are retrieved from GPCRdb (*AlloViz.AlloViz.trajutils.get_GPCRdb_numbering()*). PDB files of the selection and its CA atoms are written, and also a PSF file with the selection if applicable. The trajectory(ies) of the selection and its CA atoms in xtc format are also generated.

Parameters

GPCR
[bool or int] Use *True* if the structure is a GPCR, or the ID of a GPCRmd database dynamics entry if it was used originally to download the files from it.

pdb
[str] Filename of the PDB structure to read, process and use.

protein_sel
[str] MDAnalysis atom selection string to select the protein structure from the Universe (e.g., in case simulations in biological conditions are used, to avoid selecting extra chains, water molecules, ions…).

pdbf
[str] Complete relative filename of the processed PDB structure to save.

psf
[str] Filename of the .psf file corresponding to the pdb used to read, process and use.

psff
[str] Complete relative filename of the processed PSF structure to save.

trajs
[list] Filename(s) of the MD trajectory (or trajectories) to read and use. File format must be recognized by MDAnalysis (e.g., xtc).

trajsf
[dict] Complete relative filename(s) of the processed MD trajectory (or trajectories) to save.

****kwargs**
special_res can be passed as an optional kwarg, and it should be a dictionary containing a mapping of special residue 3/4-letter code(s) present in the structure to the corresponding standard 1-letter code(s).

AlloViz.AlloViz.trajutils.standardize_resnames

`AlloViz.AlloViz.trajutils.`**`standardize_resnames`**(*protein*, ***kwargs*)

Change the residue names of a protein to standard 3-letter codes

Change the residue names of the passed `Universe` to standard 3-letter names and also put them in the same segment (the first segment of the Universe by default) to avoid problems with packages not recognizing non-standard residue names or producing unexpected results if residues are in various segments.

`Bio.SeqUtils` is used to change the codes and also retrieve the standard 3-to-1 and 1-to-3 residue code mapping, which is also extended with `MDAnalysis.lib.util.inverse_aa_codes`.

Parameters

protein
[`AtomGroup`] Atoms of the protein for which to standardize residue names.

****kwargs**
special_res can be passed as an optional kwarg, and it should be a dictionary containing a mapping of special residue 3/4-letter code(s) present in the structure to the corresponding standard 1-letter code(s).

AlloViz.AlloViz.utils

Module containing helper functions and variables used by many other modules

Module Attributes

pkgsl	List of the correct names of available network construction methods
metricsl	List of the available network metrics that can be calculated in analysis
filteringsl	List of the available network filterings that can be performed for analysis
pool	Pool variable to share among modules

AlloViz.AlloViz.utils.pkgsl

AlloViz.AlloViz.utils.**pkgsl = ['CARDS_Holistic_Phi', 'CARDS_MI_Chi3', 'correlationplus_Phi', 'pytraj_CB', 'CARDS_Disorder_mediated_Psi', 'CARDS_Disorder_Phi', 'correlationplus_CA_LMI', 'MDEntropy_Contacts', 'CARDS_Disorder_Chi3', 'CARDS_MI_Sidechain_Dihs', 'pytraj_CA', 'CARDS_Holistic_Dihs', 'PyInteraph2_Atomic_Contacts_Strength', 'CARDS_MI_Chi2', 'dynetan', 'CARDS_Holistic_Chi3', 'CARDS_Disorder_Chi2', 'CARDS_Disorder_Psi', 'CARDS_Disorder_mediated_Sidechain_Dihs', 'MDEntropy_Backbone_Dihs', 'CARDS_Disorder_mediated_Dihs', 'MDEntropy_Phi', 'AlloViz_Chi1', 'CARDS_Disorder_mediated_Backbone_Dihs', 'correlationplus_Psi', 'PyInteraph2_Contacts', 'AlloViz_Backbone_Dihs', 'AlloViz_Sidechain_Dihs', 'CARDS_Disorder_Backbone_Dihs', 'AlloViz_Chi3', 'MDEntropy_Psi', 'AlloViz_Chi2', 'CARDS_MI_Chi1', 'CARDS_Disorder_mediated_Phi', 'CARDS_MI_Chi4', 'CARDS_Disorder_Dihs', 'PyInteraph2_Atomic_Contacts_Occurrence', 'CARDS_MI_Psi', 'AlloViz_Phi', 'correlationplus_CA_Pear', 'CARDS_Disorder_mediated_Chi3', 'CARDS_Disorder_Sidechain_Dihs', 'MDTASK', 'CARDS_Holistic_Backbone_Dihs', 'MDEntropy_AlphaAngle', 'CARDS_Holistic_Psi', 'PyInteraph2_Energy', 'CARDS_MI_Dihs', 'correlationplus_Backbone_Dihs', 'CARDS_Holistic_Chi2', 'CARDS_Holistic_Sidechain_Dihs', 'CARDS_Holistic_Chi4', 'GetContacts', 'CARDS_MI_Phi', 'AlloViz_Dihs', 'CARDS_Disorder_Chi4', 'CARDS_Disorder_Chi1', 'AlloViz_Chi4', 'CARDS_Disorder_mediated_Chi2', 'CARDS_Disorder_mediated_Chi4', 'CARDS_MI_Backbone_Dihs', 'CARDS_Disorder_mediated_Chi1', 'AlloViz_Psi', 'CARDS_Holistic_Chi1']**

List of the correct names of available network construction methods

Imported from *AlloViz.AlloViz.info.wrappers*

AlloViz.AlloViz.utils.metricsl

AlloViz.AlloViz.utils.**metricsl = ['cfb', 'btw']**

List of the available network metrics that can be calculated in analysis

To be used when the metrics="all" parameter is used in a function

AlloViz.AlloViz.utils.filteringsl

AlloViz.AlloViz.utils.**filteringsl = ['All', 'GetContacts_edges', 'No_Sequence_Neighbors', 'GPCR_Interhelix', 'Spatially_distant']**

List of the available network filterings that can be performed for analysis

To be used when the filterby="all" parameter is used in a function

AlloViz.AlloViz.utils.pool

AlloViz.AlloViz.utils.**pool = <AlloViz.AlloViz.utils.dummypool object>**

Pool variable to share among modules

Defining a *pool* variable inside this module allows for other modules to modify it and share it between modules, even when pickling due to the use of a multiprocess.Pool

Functions

get_pool()	Function to retrieve shared pool variable
make_list(obj, if_all[, apply])	Process the passed object and return a list of strings
pkgname(pkg[, fail])	Return the case-sensitive, correct name of an AlloViz network construction method
rgetattr(obj, *attrs)	Recursive version of the built-in getattr
rhasattr(obj, *attrs)	Recursive version of the built-in hasattr

AlloViz.AlloViz.utils.get_pool

AlloViz.AlloViz.utils.**get_pool**()

Function to retrieve shared pool variable

This function retrieves this module's pool variable from the main namespace and returns it to use it in whatever namespace it is called from.

AlloViz.AlloViz.utils.make_list

AlloViz.AlloViz.utils.**make_list**(*obj*, *if_all*, *apply=<function <lambda>>*)

Process the passed object and return a list of strings

Check if the object is a string, a list, or "all" and return the a list with the appropriate values (if it is the case, after applying the *apply* function to them). Make the string a list of length 1, return the unedited list, or return the passed *if_all* object if *obj* is '"all"'. If it is the case, the individual string(s) of the returned list are processed with the passed *apply* function (default does nothing).

Parameters

obj
[str or list] String or list to process.

if_all
Object to return if *obj* == *"all"*.

apply
[func, optional] Function to apply to the list's strings before returning. Default does nothing.

AlloViz.AlloViz.utils.pkgname

AlloViz.AlloViz.utils.**pkgname**(*pkg*, *fail=True*)

Return the case-sensitive, correct name of an AlloViz network construction method

Check if the passed string matches any of the available AlloViz network construction methods detailed in *AlloViz.AlloViz.info.wrappers* and return the correctly written AlloViz accession name, else raise an Exception or return False.

Parameters

pkg
[str] Name for which to retrieve the correct AlloViz accession name.

fail
[bool, default=True] Raise an Exception if an AlloViz accession name cannot be retrieved or simply return False.

AlloViz.AlloViz.utils.rgetattr

AlloViz.AlloViz.utils.**rgetattr**(*obj*, **attrs*)

Recursive version of the built-in getattr

It recursively checks if the successive strings passed are attributes of the object or the object's attribute, or the object's attribute's attribute... to finally return the final attribute or else return False.

Parameters

obj
Object in which to check if the first attribute passed exists and retrieve it.

attrs
[str] Strings to recursively use to retrieve attributes.

See also:

AlloViz.AlloViz.utils.rhasattr

Examples

```
>>> import AlloViz
>>> rgetattr = AlloViz.AlloViz.utils.rgetattr
>>> rgetattr
<function AlloViz.AlloViz.utils.rgetattr(obj, *attrs)>
>>> rgetattr(AlloViz, "AlloViz", "utils", "rgetattr")
<function AlloViz.AlloViz.utils.rgetattr(obj, *attrs)>
>>> rgetattr(AlloViz.AlloViz.utils, "rgetattr")
<function AlloViz.AlloViz.utils.rgetattr(obj, *attrs)>
>>> rgetattr(AlloViz, "AlloViz", "utils", "is_attr")
False
```

AlloViz.AlloViz.utils.rhasattr

AlloViz.AlloViz.utils.**rhasattr**(*obj*, **attrs*)

Recursive version of the built-in hasattr

It recursively checks if the successive strings passed are attributes of the object or the object's attribute, or the object's attribute's attribute... It exploits *rgetattr()* and the fact that it already returns False if any of the strings passed for the recursive search doesn't exist as attribute.

Parameters

obj
Object in which to check if the first attribute passed exists.

attrs
[str] Strings to recursively use to retrieve attributes.

Examples

```
>>> import AlloViz
>>> rgetattr = AlloViz.AlloViz.utils.rgetattr
>>> rgetattr(AlloViz, "AlloViz", "utils", "rgetattr")
True
>>> rgetattr(AlloViz.AlloViz.utils, "rgetattr")
True
>>> rgetattr(AlloViz, "AlloViz", "utils", "is_attr")
False
```

Classes

dummypool()	Class to mimic a process Pool when only using 1 core (synchronous)

AlloViz.AlloViz.utils.dummypool

class AlloViz.AlloViz.utils.**dummypool**

Bases: object

Class to mimic a process Pool when only using 1 core (synchronous)

This class aims to be able to be used with the same syntax as a multiprocess(ing) Pool managed through apply_async. Instead of running the tasks sent with the method asynchronously, if the pool is an instance of this class they will run synchronously in each call to the method.

Notes

Using a multiprocess(ing) Pool initialized with 1 core would have the same effect in terms of resource consumption, but this way the tasks are run in the main namespace instead of on a pickled copy, which is useful for debugging, i.e., getting the stdout and stderr immediately.

Methods

apply_async(function[, args, callback])	Execute the function with the passed args
close()	Empty function
join()	Empty function

AlloViz.AlloViz.utils.dummypool.apply_async

dummypool.**apply_async**(*function*, *args=[]*, *callback=None*)

Execute the function with the passed args

It executes the function with the passed args synchronously, and optionally the specified callback function to the resulting output as well.

Parameters

function
[func]

args
[list]

callback
[func, optional]

AlloViz.AlloViz.utils.dummypool.close

dummypool.**close**()

Empty function

AlloViz.AlloViz.utils.dummypool.join

dummypool.**join**()

Empty function

AlloViz.Wrappers

Wrappers of AlloViz's network construction methods/packages

Each module contains the class(es) that wrap the code of a package used by AlloViz as a network construction method. They are used to send their calculations through the AlloViz code without needing to execute them independently, and process their results to store them in AlloViz objects. They are child classes of *Base* module's *Base* class and other classes therein (by multiple inheritance).

Modules

AlloViz.Wrappers.AlloViz_w	AlloViz's own network construction method wrapper
AlloViz.Wrappers.Base	Base module for package/software wrapping for AlloViz
AlloViz.Wrappers.CARDS_w	CARDS network construction method wrapper
AlloViz.Wrappers.GSAtools	GSAtools wrapper
AlloViz.Wrappers.GetContacts(protein, d)	GetContacts' contact frequencies
AlloViz.Wrappers.MDEntropy_w	MDEntropy wrapper
AlloViz.Wrappers.MDTASK(protein, d)	MD-TASK's Pearson's correlation of CA atoms
AlloViz.Wrappers.PyInteraph2_w	PyInteraph2 wrapper
AlloViz.Wrappers.correlationplus_w	correlationplus wrapper
AlloViz.Wrappers.dynetan_w	dynetan wrapper
AlloViz.Wrappers.g_correlation_w	g_correlation wrapper
AlloViz.Wrappers.pytraj_w	pytraj wrapper

AlloViz.Wrappers.AlloViz_w

AlloViz's own network construction method wrapper

It calculates the Mutual Information (MI) generalized correlation of the residues' dihedral angles and also their combinations.

Classes

AlloViz_Backbone_Dihs(protein, d)	AlloViz network construction method's of the combination of the backbone dihedrals' MIs by averaging
AlloViz_Base(protein, d)	AlloViz network construction method base class
AlloViz_Chi1(protein, d)	AlloViz network construction method's MI of Chi1 side-chain dihedral
AlloViz_Chi2(protein, d)	AlloViz network construction method's MI of Chi2 side-chain dihedral
AlloViz_Chi3(protein, d)	AlloViz network construction method's MI of Chi3 side-chain dihedral
AlloViz_Chi4(protein, d)	AlloViz network construction method's MI of Chi4 side-chain dihedral
AlloViz_Dihs(protein, d)	AlloViz network construction method's of the combination of all backbone and side-chain dihedrals' MIs by averaging
AlloViz_Phi(protein, d)	AlloViz network construction method's MI of Phi backbone dihedral
AlloViz_Psi(protein, d)	AlloViz network construction method's MI of Psi backbone dihedral
AlloViz_Sidechain_Dihs(protein, d)	AlloViz network construction method's of the combination of the side-chain dihedrals' MIs by averaging

AlloViz.Wrappers.AlloViz_w.AlloViz_Backbone_Dihs

class AlloViz.Wrappers.AlloViz_w.**AlloViz_Backbone_Dihs**(*protein*, *d*)

Bases: *Combined_Dihs_Avg*, *AlloViz_Base*

AlloViz network construction method's of the combination of the backbone dihedrals' MIs by averaging

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.AlloViz_w.AlloViz_Backbone_Dihs.filter

AlloViz_Backbone_Dihs.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings

[str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of

strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.AlloViz_w.AlloViz_Base

class AlloViz.Wrappers.AlloViz_w.**AlloViz_Base**(*protein*, *d*)

Bases: *Multicore*

AlloViz network construction method base class

`_computation()` uses `MDAnalysis.analysis.dihedrals` to extract data and NPEET_LNC for MI calculation.

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.AlloViz_w.AlloViz_Base.filter

AlloViz_Base.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
[str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
: Filtering class.

AlloViz.Protein.calculate
: Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
: Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
: Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.AlloViz_w.AlloViz_Chi1

class AlloViz.Wrappers.AlloViz_w.**AlloViz_Chi1**(*protein*, *d*)

Bases: *AlloViz_Base*

AlloViz network construction method's MI of Chi1 side-chain dihedral

GLY and ALA residues don't have the Chi1 side-chain dihedral.

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.AlloViz_w.AlloViz_Chi1.filter

AlloViz_Chi1.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
: [str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of

strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.AlloViz_w.AlloViz_Chi2

class AlloViz.Wrappers.AlloViz_w.**AlloViz_Chi2**(*protein*, *d*)

Bases: *AlloViz_Base*

AlloViz network construction method's MI of Chi2 side-chain dihedral

In addition to GLY and ALA, CYS, SER, THR and VAL residues don't have the Chi2 dihedral.

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.AlloViz_w.AlloViz_Chi2.filter

AlloViz_Chi2.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
[str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
: Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
: Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
: Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.AlloViz_w.AlloViz_Chi3

class AlloViz.Wrappers.AlloViz_w.**AlloViz_Chi3**(*protein*, *d*)

Bases: *AlloViz_Base*

AlloViz network construction method's MI of Chi3 side-chain dihedral

Chi3 side-chain dihedral is only available for ARG, GLN, GLU, LYS and MET residues.

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.AlloViz_w.AlloViz_Chi3.filter

AlloViz_Chi3.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
: [str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.AlloViz_w.AlloViz_Chi4

class AlloViz.Wrappers.AlloViz_w.**AlloViz_Chi4**(*protein*, *d*)

Bases: *AlloViz_Base*

AlloViz network construction method's MI of Chi4 side-chain dihedral

Chi4 side-chain dihedral is only available for ARG and LYS residues.

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.AlloViz_w.AlloViz_Chi4.filter

AlloViz_Chi4.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
[str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.AlloViz_w.AlloViz_Dihs

class AlloViz.Wrappers.AlloViz_w.**AlloViz_Dihs**(*protein*, *d*)

Bases: *Combined_Dihs_Avg*, *AlloViz_Base*

AlloViz network construction method's of the combination of all backbone and side-chain dihedrals' MIs by averaging

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.AlloViz_w.AlloViz_Dihs.filter

AlloViz_Dihs.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
[str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence

positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.AlloViz_w.AlloViz_Phi

class AlloViz.Wrappers.AlloViz_w.**AlloViz_Phi**(*protein*, *d*)

Bases: *AlloViz_Base*

AlloViz network construction method's MI of Phi backbone dihedral

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.AlloViz_w.AlloViz_Phi.filter

AlloViz_Phi.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
[str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.AlloViz_w.AlloViz_Psi

class AlloViz.Wrappers.AlloViz_w.**AlloViz_Psi**(*protein*, *d*)

Bases: *AlloViz_Base*

AlloViz network construction method's MI of Psi backbone dihedral

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.AlloViz_w.AlloViz_Psi.filter

AlloViz_Psi.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
[str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.AlloViz_w.AlloViz_Sidechain_Dihs

class AlloViz.Wrappers.AlloViz_w.**AlloViz_Sidechain_Dihs**(*protein*, *d*)

Bases: *Combined_Dihs_Avg*, *AlloViz_Base*

AlloViz network construction method's of the combination of the side-chain dihedrals' MIs by averaging

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.AlloViz_w.AlloViz_Sidechain_Dihs.filter

AlloViz_Sidechain_Dihs.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
: [str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
: [float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
: [int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
: [int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
: Filtering class.

AlloViz.Protein.calculate
: Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
: Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
: Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.Base

Base module for package/software wrapping for AlloViz

Main *Base* class stores the information about the *Protein* object and defines the general private methods that most child classes use to launch, manage and store the network calculations. It exploits the '__new__' special method to establish the object's attributes and that can be extended in the child classes and '__init__' is used to launch calculations, which can be extended or overriden by child classes.

In this module there are additional classes that override or extend *Base*'s private methods and that are used to be combined to create child classes with multiple inheritance for specific uses.

Functions

lazy_import(key, val)	Return a lazily-imported module to store in a variable

AlloViz.Wrappers.Base.lazy_import

AlloViz.Wrappers.Base.**lazy_import**(*key*, *val*)

Return a lazily-imported module to store in a variable

It uses lazyasd.

Parameters

key
[str] The variable name with which the module is going to be accessed. In *import numpy as np* "key" would be *np*.

val
[str] Absolute import of the module.

Classes

Base(protein, d)	Base class for network calculation
Combined_Dihs(protein, d)	Class for combination of dihedral angle data
Combined_Dihs_Avg(protein, d)	Class for combination of dihedral angle data by averaging
Multicore(protein, d)	Class for multi-core packages calculations

AlloViz.Wrappers.Base.Base

class AlloViz.Wrappers.Base.**Base**(*protein*, *d*)

Bases: object

Base class for network calculation

This class uses the '__new__' special method to establish the object's attributes, which can be extended and/or modified in the child classes' '__new__'. The '__init__' special method is then used to launch calculations, and it can also be extended or overriden by child classes.

It also defines the private methods `_calculate()` to launch calculations, which exploits the child classes' custom '_computation' private method so that the different packages can be run with them using the standardized information stored in the objects attributes; and `_save_pq()` to save the results of the calculations in parquet format. Both can also be extended or overriden by child classes.

filter() allows to filter the calculated raw edges for posterior analysis.

Parameters

protein
[*Protein* object] Protein object from which to extract the information for calculation.

d
[dict] Dictionary containing all the elements of the protein's *__dict__* and also any keyword arguments passed for calculation. This is needed for successful parallelization, as pickling objects of Base's child classes means pickling the Protein object and the attributes that are added to it during its *__init__* are lost.

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.Base.Base.filter

Base.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
[str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.Base.Combined_Dihs

class AlloViz.Wrappers.Base.**Combined_Dihs**(*protein*, *d*)

Bases: *Base*

Class for combination of dihedral angle data

This class' child classes are used to combine the information from multiple dihedral angles by overriding the *_calculate* and *_save_pq* private methods. It checks that the calculations of the desired dihedral angles for the current trajectory number (*xtc*) are available, or else raises an error, and saves the data with the child class' *_save_pq* specific private method.

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.Base.Combined_Dihs.filter

Combined_Dihs.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
[str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.Base.Combined_Dihs_Avg

class AlloViz.Wrappers.Base.**Combined_Dihs_Avg**(*protein*, *d*)

Bases: *Combined_Dihs*

Class for combination of dihedral angle data by averaging

This class' child classes are used to combine the information from multiple dihedral angles by averaging, with its specific `_save_pq()` private method.

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.Base.Combined_Dihs_Avg.filter

Combined_Dihs_Avg.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings

[str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold

[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.Base.Multicore

class AlloViz.Wrappers.Base.**Multicore**(*protein*, *d*)

Bases: *Base*

Class for multi-core packages calculations

This class defines additional attributes 'taskcpus' and '_empties' in '__new__' for packages that are able to perform multi-core calculations by themselves (besides AlloViz's parallelization of the calculations for each trajectory file). 'taskcpus' is the number of cores the package should use per trajectory and '_empties' the number of empty jobs that should be sent to the same Pool that the task/_computation is being sent to to "occupy" the number of cores that the package is going to use in reality, as sending it to the Pool would only take 1 of its workers. It extends the `_calculate()` private method to do it.

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.Base.Multicore.filter

Multicore.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
[str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.CARDS_w

CARDS network construction method wrapper

It calculates the Mutual Information (MI) generalized correlation of the residues' dihedral angles, and also the MI of their disorder, the MI between their values and their disorder (pure disorder or disorder-mediated) and the holistic MI (MI of the values plus of the disorder), and also their combinations.

Classes

CARDS(protein, d)	CARDS network construction method base class for calculations
CARDS_Disorder_Backbone_Dihs(protein, d)	CARDS network construction method's of the combination of the backbone dihedrals' Disorder MIs by averaging.
CARDS_Disorder_Chi1(protein, d)	CARDS network construction method's Disorder MI of Chi1 dihedral
CARDS_Disorder_Chi2(protein, d)	CARDS network construction method's Disorder MI of Chi2 dihedral
CARDS_Disorder_Chi3(protein, d)	CARDS network construction method's Disorder MI of Chi3 dihedral
CARDS_Disorder_Chi4(protein, d)	CARDS network construction method's Disorder MI of Chi4 dihedral
CARDS_Disorder_Dihs(protein, d)	CARDS network construction method's of the combination of all backbone and side-chain dihedrals' Disorder MIs by averaging.
CARDS_Disorder_Phi(protein, d)	CARDS network construction method's Disorder MI of Phi dihedral
CARDS_Disorder_Psi(protein, d)	CARDS network construction method's Disorder MI of Psi dihedral
CARDS_Disorder_Sidechain_Dihs(protein, d)	CARDS network construction method's of the combination of the side-chain dihedrals' Disorder MIs by averaging.
CARDS_Disorder_mediated_Backbone_Dihs(protein, d)	CARDS network construction method's of the combination of the backbone dihedrals' Disorder_mediated MIs by averaging.
CARDS_Disorder_mediated_Chi1(protein, d)	CARDS network construction method's Disorder_mediated MI of Chi1 dihedral
CARDS_Disorder_mediated_Chi2(protein, d)	CARDS network construction method's Disorder_mediated MI of Chi2 dihedral

continues on next page

Table 4 – continued from previous page

CARDS_Disorder_mediated_Chi3(protein, d)	CARDS network construction method's Disorder_mediated MI of Chi3 dihedral
CARDS_Disorder_mediated_Chi4(protein, d)	CARDS network construction method's Disorder_mediated MI of Chi4 dihedral
CARDS_Disorder_mediated_Dihs(protein, d)	CARDS network construction method's of the combination of all backbone and side-chain dihedrals' Disorder_mediated MIs by averaging.
CARDS_Disorder_mediated_Phi(protein, d)	CARDS network construction method's Disorder_mediated MI of Phi dihedral
CARDS_Disorder_mediated_Psi(protein, d)	CARDS network construction method's Disorder_mediated MI of Psi dihedral
CARDS_Disorder_mediated_Sidechain_Dihs(...)	CARDS network construction method's of the combination of the side-chain dihedrals' Disorder_mediated MIs by averaging.
CARDS_Holistic_Backbone_Dihs(protein, d)	CARDS network construction method's of the combination of the backbone dihedrals' Holistic MIs by averaging.
CARDS_Holistic_Chi1(protein, d)	CARDS network construction method's Holistic MI of Chi1 dihedral
CARDS_Holistic_Chi2(protein, d)	CARDS network construction method's Holistic MI of Chi2 dihedral
CARDS_Holistic_Chi3(protein, d)	CARDS network construction method's Holistic MI of Chi3 dihedral
CARDS_Holistic_Chi4(protein, d)	CARDS network construction method's Holistic MI of Chi4 dihedral
CARDS_Holistic_Dihs(protein, d)	CARDS network construction method's of the combination of all backbone and side-chain dihedrals' Holistic MIs by averaging.
CARDS_Holistic_Phi(protein, d)	CARDS network construction method's Holistic MI of Phi dihedral
CARDS_Holistic_Psi(protein, d)	CARDS network construction method's Holistic MI of Psi dihedral
CARDS_Holistic_Sidechain_Dihs(protein, d)	CARDS network construction method's of the combination of the side-chain dihedrals' Holistic MIs by averaging.
CARDS_MI_Backbone_Dihs(protein, d)	CARDS network construction method's of the combination of the backbone dihedrals' MIs by averaging.
CARDS_MI_Chi1(protein, d)	CARDS network construction method's MI of Chi1 dihedral
CARDS_MI_Chi2(protein, d)	CARDS network construction method's MI of Chi2 dihedral
CARDS_MI_Chi3(protein, d)	CARDS network construction method's MI of Chi3 dihedral
CARDS_MI_Chi4(protein, d)	CARDS network construction method's MI of Chi4 dihedral
CARDS_MI_Dihs(protein, d)	CARDS network construction method's of the combination of all backbone and side-chain dihedrals' MIs by averaging.
CARDS_MI_Phi(protein, d)	CARDS network construction method's MI of Phi dihedral
CARDS_MI_Psi(protein, d)	CARDS network construction method's MI of Psi dihedral

continues on next page

Table 4 – continued from previous page

CARDS_MI_Sidechain_Dihs(protein, d)	CARDS network construction method's of the combination of the side-chain dihedrals' MIs by averaging.

AlloViz.Wrappers.CARDS_w.CARDS

class AlloViz.Wrappers.CARDS_w.**CARDS**(*protein*, *d*)

Bases: *Multicore*

CARDS network construction method base class for calculations

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.CARDS_w.CARDS.filter

CARDS.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
[str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.CARDS_w.CARDS_Disorder_Backbone_Dihs

class AlloViz.Wrappers.CARDS_w.**CARDS_Disorder_Backbone_Dihs**(*protein*, *d*)

Bases: *Combined_Dihs_Avg*

CARDS network construction method's of the combination of the backbone dihedrals' Disorder MIs by averaging.

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.CARDS_w.CARDS_Disorder_Backbone_Dihs.filter

CARDS_Disorder_Backbone_Dihs.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
[str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of

strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.CARDS_w.CARDS_Disorder_Chi1

class AlloViz.Wrappers.CARDS_w.**CARDS_Disorder_Chi1**(*protein*, *d*)

Bases: *Base*

CARDS network construction method's Disorder MI of Chi1 dihedral

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.CARDS_w.CARDS_Disorder_Chi1.filter

CARDS_Disorder_Chi1.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
[str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
: Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
: Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
: Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.CARDS_w.CARDS_Disorder_Chi2

class AlloViz.Wrappers.CARDS_w.**CARDS_Disorder_Chi2**(*protein*, *d*)

Bases: *Base*

CARDS network construction method's Disorder MI of Chi2 dihedral

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.CARDS_w.CARDS_Disorder_Chi2.filter

CARDS_Disorder_Chi2.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
: [str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.CARDS_w.CARDS_Disorder_Chi3

class AlloViz.Wrappers.CARDS_w.**CARDS_Disorder_Chi3**(*protein*, *d*)

Bases: *Base*

CARDS network construction method's Disorder MI of Chi3 dihedral

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.CARDS_w.CARDS_Disorder_Chi3.filter

CARDS_Disorder_Chi3.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
[str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.CARDS_w.CARDS_Disorder_Chi4

class AlloViz.Wrappers.CARDS_w.**CARDS_Disorder_Chi4**(*protein*, *d*)

Bases: *Base*

CARDS network construction method's Disorder MI of Chi4 dihedral

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.CARDS_w.CARDS_Disorder_Chi4.filter

CARDS_Disorder_Chi4.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
[str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.CARDS_w.CARDS_Disorder_Dihs

class AlloViz.Wrappers.CARDS_w.**CARDS_Disorder_Dihs**(*protein*, *d*)

Bases: *Combined_Dihs_Avg*

CARDS network construction method's of the combination of all backbone and side-chain dihedrals' Disorder MIs by averaging.

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.CARDS_w.CARDS_Disorder_Dihs.filter

CARDS_Disorder_Dihs.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
[str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.CARDS_w.CARDS_Disorder_Phi

class AlloViz.Wrappers.CARDS_w.**CARDS_Disorder_Phi**(*protein*, *d*)

Bases: *Base*

CARDS network construction method's Disorder MI of Phi dihedral

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.CARDS_w.CARDS_Disorder_Phi.filter

CARDS_Disorder_Phi.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
[str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
: Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
: Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
: Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.CARDS_w.CARDS_Disorder_Psi

class AlloViz.Wrappers.CARDS_w.**CARDS_Disorder_Psi**(*protein*, *d*)

Bases: *Base*

CARDS network construction method's Disorder MI of Psi dihedral

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.CARDS_w.CARDS_Disorder_Psi.filter

CARDS_Disorder_Psi.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
: [str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.CARDS_w.CARDS_Disorder_Sidechain_Dihs

class AlloViz.Wrappers.CARDS_w.**CARDS_Disorder_Sidechain_Dihs**(*protein*, *d*)

Bases: *Combined_Dihs_Avg*

CARDS network construction method's of the combination of the side-chain dihedrals' Disorder MIs by averaging.

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.CARDS_w.CARDS_Disorder_Sidechain_Dihs.filter

CARDS_Disorder_Sidechain_Dihs.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
[str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.CARDS_w.CARDS_Disorder_mediated_Backbone_Dihs

class AlloViz.Wrappers.CARDS_w.**CARDS_Disorder_mediated_Backbone_Dihs**(*protein*, *d*)

Bases: *Combined_Dihs_Avg*

CARDS network construction method's of the combination of the backbone dihedrals' Disorder_mediated MIs by averaging.

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.CARDS_w.CARDS_Disorder_mediated_Backbone_Dihs.filter

CARDS_Disorder_mediated_Backbone_Dihs.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
[str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence

positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.CARDS_w.CARDS_Disorder_mediated_Chi1

class AlloViz.Wrappers.CARDS_w.**CARDS_Disorder_mediated_Chi1**(*protein*, *d*)

Bases: *Base*

CARDS network construction method's Disorder_mediated MI of Chi1 dihedral

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.CARDS_w.CARDS_Disorder_mediated_Chi1.filter

CARDS_Disorder_mediated_Chi1.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
[str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.CARDS_w.CARDS_Disorder_mediated_Chi2

class AlloViz.Wrappers.CARDS_w.**CARDS_Disorder_mediated_Chi2**(*protein*, *d*)

Bases: *Base*

CARDS network construction method's Disorder_mediated MI of Chi2 dihedral

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.CARDS_w.CARDS_Disorder_mediated_Chi2.filter

CARDS_Disorder_mediated_Chi2.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
[str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.CARDS_w.CARDS_Disorder_mediated_Chi3

class AlloViz.Wrappers.CARDS_w.**CARDS_Disorder_mediated_Chi3**(*protein*, *d*)

Bases: *Base*

CARDS network construction method's Disorder_mediated MI of Chi3 dihedral

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.CARDS_w.CARDS_Disorder_mediated_Chi3.filter

CARDS_Disorder_mediated_Chi3.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
: [str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
: [float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
: [int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
: [int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
: Filtering class.

AlloViz.Protein.calculate
: Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
: Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
: Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.CARDS_w.CARDS_Disorder_mediated_Chi4

class AlloViz.Wrappers.CARDS_w.**CARDS_Disorder_mediated_Chi4**(*protein*, *d*)

Bases: *Base*

CARDS network construction method's Disorder_mediated MI of Chi4 dihedral

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.CARDS_w.CARDS_Disorder_mediated_Chi4.filter

CARDS_Disorder_mediated_Chi4.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
[str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
: Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
: Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
: Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.CARDS_w.CARDS_Disorder_mediated_Dihs

class AlloViz.Wrappers.CARDS_w.**CARDS_Disorder_mediated_Dihs**(*protein*, *d*)

Bases: *Combined_Dihs_Avg*

CARDS network construction method's of the combination of all backbone and side-chain dihedrals' Disorder_mediated MIs by averaging.

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.CARDS_w.CARDS_Disorder_mediated_Dihs.filter

CARDS_Disorder_mediated_Dihs.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
: [str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.CARDS_w.CARDS_Disorder_mediated_Phi

class AlloViz.Wrappers.CARDS_w.**CARDS_Disorder_mediated_Phi**(*protein*, *d*)

Bases: *Base*

CARDS network construction method's Disorder_mediated MI of Phi dihedral

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.CARDS_w.CARDS_Disorder_mediated_Phi.filter

CARDS_Disorder_mediated_Phi.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
[str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.CARDS_w.CARDS_Disorder_mediated_Psi

class AlloViz.Wrappers.CARDS_w.**CARDS_Disorder_mediated_Psi**(*protein*, *d*)

Bases: *Base*

CARDS network construction method's Disorder_mediated MI of Psi dihedral

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.CARDS_w.CARDS_Disorder_mediated_Psi.filter

CARDS_Disorder_mediated_Psi.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
[str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.CARDS_w.CARDS_Disorder_mediated_Sidechain_Dihs

class AlloViz.Wrappers.CARDS_w.**CARDS_Disorder_mediated_Sidechain_Dihs**(*protein*, *d*)

Bases: *Combined_Dihs_Avg*

CARDS network construction method's of the combination of the side-chain dihedrals' Disorder_mediated MIs by averaging.

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.CARDS_w.CARDS_Disorder_mediated_Sidechain_Dihs.filter

CARDS_Disorder_mediated_Sidechain_Dihs.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
[str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.CARDS_w.CARDS_Holistic_Backbone_Dihs

class AlloViz.Wrappers.CARDS_w.**CARDS_Holistic_Backbone_Dihs**(*protein*, *d*)

Bases: *Combined_Dihs_Avg*

CARDS network construction method's of the combination of the backbone dihedrals' Holistic MIs by averaging.

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.CARDS_w.CARDS_Holistic_Backbone_Dihs.filter

CARDS_Holistic_Backbone_Dihs.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
[str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
: Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
: Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
: Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.CARDS_w.CARDS_Holistic_Chi1

class AlloViz.Wrappers.CARDS_w.**CARDS_Holistic_Chi1**(*protein*, *d*)

Bases: *Base*

CARDS network construction method's Holistic MI of Chi1 dihedral

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.CARDS_w.CARDS_Holistic_Chi1.filter

CARDS_Holistic_Chi1.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
: [str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.CARDS_w.CARDS_Holistic_Chi2

class AlloViz.Wrappers.CARDS_w.**CARDS_Holistic_Chi2**(*protein*, *d*)

Bases: *Base*

CARDS network construction method's Holistic MI of Chi2 dihedral

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.CARDS_w.CARDS_Holistic_Chi2.filter

CARDS_Holistic_Chi2.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
[str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.CARDS_w.CARDS_Holistic_Chi3

class AlloViz.Wrappers.CARDS_w.**CARDS_Holistic_Chi3**(*protein*, *d*)

Bases: *Base*

CARDS network construction method's Holistic MI of Chi3 dihedral

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.CARDS_w.CARDS_Holistic_Chi3.filter

CARDS_Holistic_Chi3.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
[str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.CARDS_w.CARDS_Holistic_Chi4

class AlloViz.Wrappers.CARDS_w.**CARDS_Holistic_Chi4**(*protein*, *d*)

Bases: *Base*

CARDS network construction method's Holistic MI of Chi4 dihedral

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.CARDS_w.CARDS_Holistic_Chi4.filter

CARDS_Holistic_Chi4.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
: [str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
: [float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
: [int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
: [int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
: Filtering class.

AlloViz.Protein.calculate
: Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
: Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
: Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.CARDS_w.CARDS_Holistic_Dihs

class AlloViz.Wrappers.CARDS_w.**CARDS_Holistic_Dihs**(*protein*, *d*)

Bases: *Combined_Dihs_Avg*

CARDS network construction method's of the combination of all backbone and side-chain dihedrals' Holistic MIs by averaging.

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.CARDS_w.CARDS_Holistic_Dihs.filter

CARDS_Holistic_Dihs.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
[str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
: Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
: Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
: Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.CARDS_w.CARDS_Holistic_Phi

class AlloViz.Wrappers.CARDS_w.**CARDS_Holistic_Phi**(*protein*, *d*)

Bases: *Base*

CARDS network construction method's Holistic MI of Phi dihedral

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.CARDS_w.CARDS_Holistic_Phi.filter

CARDS_Holistic_Phi.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
: [str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.CARDS_w.CARDS_Holistic_Psi

class AlloViz.Wrappers.CARDS_w.**CARDS_Holistic_Psi**(*protein*, *d*)

Bases: *Base*

CARDS network construction method's Holistic MI of Psi dihedral

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.CARDS_w.CARDS_Holistic_Psi.filter

CARDS_Holistic_Psi.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
[str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.CARDS_w.CARDS_Holistic_Sidechain_Dihs

class AlloViz.Wrappers.CARDS_w.**CARDS_Holistic_Sidechain_Dihs**(*protein*, *d*)

Bases: *Combined_Dihs_Avg*

CARDS network construction method's of the combination of the side-chain dihedrals' Holistic MIs by averaging.

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.CARDS_w.CARDS_Holistic_Sidechain_Dihs.filter

CARDS_Holistic_Sidechain_Dihs.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
[str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence

positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.CARDS_w.CARDS_MI_Backbone_Dihs

class AlloViz.Wrappers.CARDS_w.**CARDS_MI_Backbone_Dihs**(*protein*, *d*)

Bases: *Combined_Dihs_Avg*

CARDS network construction method's of the combination of the backbone dihedrals' MIs by averaging.

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.CARDS_w.CARDS_MI_Backbone_Dihs.filter

CARDS_MI_Backbone_Dihs.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
[str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.CARDS_w.CARDS_MI_Chi1

class AlloViz.Wrappers.CARDS_w.**CARDS_MI_Chi1**(*protein*, *d*)

Bases: *Base*

CARDS network construction method's MI of Chi1 dihedral

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.CARDS_w.CARDS_MI_Chi1.filter

CARDS_MI_Chi1.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
[str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.CARDS_w.CARDS_MI_Chi2

class AlloViz.Wrappers.CARDS_w.**CARDS_MI_Chi2**(*protein*, *d*)

Bases: *Base*

CARDS network construction method's MI of Chi2 dihedral

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.CARDS_w.CARDS_MI_Chi2.filter

CARDS_MI_Chi2.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
: [str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
: [float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
: [int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
: [int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
: Filtering class.

AlloViz.Protein.calculate
: Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
: Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
: Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.CARDS_w.CARDS_MI_Chi3

class AlloViz.Wrappers.CARDS_w.**CARDS_MI_Chi3**(*protein*, *d*)

Bases: *Base*

CARDS network construction method's MI of Chi3 dihedral

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.CARDS_w.CARDS_MI_Chi3.filter

CARDS_MI_Chi3.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
[str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
: Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
: Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
: Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.CARDS_w.CARDS_MI_Chi4

class AlloViz.Wrappers.CARDS_w.**CARDS_MI_Chi4**(*protein*, *d*)

Bases: *Base*

CARDS network construction method's MI of Chi4 dihedral

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.CARDS_w.CARDS_MI_Chi4.filter

CARDS_MI_Chi4.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
: [str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.CARDS_w.CARDS_MI_Dihs

class AlloViz.Wrappers.CARDS_w.**CARDS_MI_Dihs**(*protein*, *d*)

Bases: *Combined_Dihs_Avg*

CARDS network construction method's of the combination of all backbone and side-chain dihedrals' MIs by averaging.

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.CARDS_w.CARDS_MI_Dihs.filter

CARDS_MI_Dihs.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
[str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.CARDS_w.CARDS_MI_Phi

class AlloViz.Wrappers.CARDS_w.**CARDS_MI_Phi**(*protein*, *d*)

Bases: *Base*

CARDS network construction method's MI of Phi dihedral

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.CARDS_w.CARDS_MI_Phi.filter

CARDS_MI_Phi.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
[str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.CARDS_w.CARDS_MI_Psi

class AlloViz.Wrappers.CARDS_w.**CARDS_MI_Psi**(*protein*, *d*)

Bases: *Base*

CARDS network construction method's MI of Psi dihedral

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.CARDS_w.CARDS_MI_Psi.filter

CARDS_MI_Psi.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
[str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.CARDS_w.CARDS_MI_Sidechain_Dihs

class AlloViz.Wrappers.CARDS_w.**CARDS_MI_Sidechain_Dihs**(*protein*, *d*)

Bases: *Combined_Dihs_Avg*

CARDS network construction method's of the combination of the side-chain dihedrals' MIs by averaging.

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.CARDS_w.CARDS_MI_Sidechain_Dihs.filter

CARDS_MI_Sidechain_Dihs.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
[str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
: Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
: Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
: Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.GSAtools

GSAtools wrapper

It calculates Mutual Information (MI).

Classes

GSAtools(protein, d)	GSAtools' MI

AlloViz.Wrappers.GSAtools.GSAtools

class AlloViz.Wrappers.GSAtools.**GSAtools**(*protein*, *d*)

Bases: *Base*

GSAtools' MI

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.GSAtools.GSAtools.filter

GSAtools.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
[str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.GetContacts

class AlloViz.Wrappers.**GetContacts**(*protein*, *d*)

Bases: *Multicore*

GetContacts' contact frequencies

Methods

AlloViz.Wrappers.MDEntropy_w

MDEntropy wrapper

It calculates the Mutual Information (MI) of the combination of the backbone's dihedral angles, of the Alpha Angles (torsion angle defined by the i-1, i, i+1 and i+2 residues' CA atoms), and of the Contacts.

Classes

MDEntropy_AlphaAngle(protein, d)	MDEntropy's MI of the Alpha Angles
MDEntropy_Backbone_Dihs(protein, d)	MDEntropy's MI of the backbone's dihedrals
MDEntropy_Base(protein, d)	MDEntropy base class
MDEntropy_Contacts(protein, d)	MDEntropy's MI of Contacts
MDEntropy_Phi(protein, d)	MDEntropy's MI of the phi backbone dihedral
MDEntropy_Psi(protein, d)	MDEntropy's MI of the psi backbone dihedral

AlloViz.Wrappers.MDEntropy_w.MDEntropy_AlphaAngle

class AlloViz.Wrappers.MDEntropy_w.**MDEntropy_AlphaAngle**(*protein*, *d*)

Bases: *MDEntropy_Base*

MDEntropy's MI of the Alpha Angles

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.MDEntropy_w.MDEntropy_AlphaAngle.filter

MDEntropy_AlphaAngle.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
[str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.MDEntropy_w.MDEntropy_Backbone_Dihs

class AlloViz.Wrappers.MDEntropy_w.**MDEntropy_Backbone_Dihs**(*protein*, *d*)

Bases: *MDEntropy_Base*

MDEntropy's MI of the backbone's dihedrals

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.MDEntropy_w.MDEntropy_Backbone_Dihs.filter

MDEntropy_Backbone_Dihs.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
[str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.MDEntropy_w.MDEntropy_Base

class AlloViz.Wrappers.MDEntropy_w.**MDEntropy_Base**(*protein*, *d*)

Bases: *Multicore*

MDEntropy base class

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.MDEntropy_w.MDEntropy_Base.filter

MDEntropy_Base.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
: [str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
: [float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
: [int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
: [int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
: Filtering class.

AlloViz.Protein.calculate
: Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
: Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
: Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.MDEntropy_w.MDEntropy_Contacts

class AlloViz.Wrappers.MDEntropy_w.**MDEntropy_Contacts**(*protein*, *d*)

Bases: *MDEntropy_Base*

MDEntropy's MI of Contacts

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.MDEntropy_w.MDEntropy_Contacts.filter

MDEntropy_Contacts.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
[str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
: Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
: Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
: Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.MDEntropy_w.MDEntropy_Phi

class AlloViz.Wrappers.MDEntropy_w.**MDEntropy_Phi**(*protein*, *d*)

Bases: *MDEntropy_Backbone_Dihs*

MDEntropy's MI of the phi backbone dihedral

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.MDEntropy_w.MDEntropy_Phi.filter

MDEntropy_Phi.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
: [str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.MDEntropy_w.MDEntropy_Psi

class AlloViz.Wrappers.MDEntropy_w.**MDEntropy_Psi**(*protein*, *d*)

Bases: *MDEntropy_Backbone_Dihs*

MDEntropy's MI of the psi backbone dihedral

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.MDEntropy_w.MDEntropy_Psi.filter

MDEntropy_Psi.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
[str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.MDTASK

class AlloViz.Wrappers.**MDTASK**(*protein*, *d*)

Bases: *Multicore*

MD-TASK's Pearson's correlation of CA atoms

Methods

AlloViz.Wrappers.PyInteraph2_w

PyInteraph2 wrapper

It calculates interaction energies and contact frequencies.

Classes

PyInteraph2_Atomic_Contacts_Occurrence(...)	PyInteraph2's atomic contacts occurrence
PyInteraph2_Atomic_Contacts_Strength(protein, d)	PyInteraph2's atomic contacts strength
PyInteraph2_Base(protein, d)	PyInteraph2 base class
PyInteraph2_Contacts(protein, d)	PyInteraph2's side-chain contact frequencies
PyInteraph2_Energy(protein, d)	PyInteraph2's interaction energies

AlloViz.Wrappers.PyInteraph2_w.PyInteraph2_Atomic_Contacts_Occurrence

class AlloViz.Wrappers.PyInteraph2_w.**PyInteraph2_Atomic_Contacts_Occurrence**(*protein*, *d*)

Bases: *PyInteraph2_Contacts*

PyInteraph2's atomic contacts occurrence

Atom pair distance-based residue pairs' contacts occurrence/"persistence"/frequency.

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.PyInteraph2_w.PyInteraph2_Atomic_Contacts_Occurrence.filter

PyInteraph2_Atomic_Contacts_Occurrence.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
[str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.PyInteraph2_w.PyInteraph2_Atomic_Contacts_Strength

class AlloViz.Wrappers.PyInteraph2_w.**PyInteraph2_Atomic_Contacts_Strength**(*protein*, *d*)

Bases: *PyInteraph2_Base*

PyInteraph2's atomic contacts strength

Atom pair distance-based residue pairs' contacts strength, based on the work by Brinda and Vishveshwara, 2005 [1].

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.PyInteraph2_w.PyInteraph2_Atomic_Contacts_Strength.filter

PyInteraph2_Atomic_Contacts_Strength.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings

[str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold

[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.PyInteraph2_w.PyInteraph2_Base

class AlloViz.Wrappers.PyInteraph2_w.**PyInteraph2_Base**(*protein*, *d*)

Bases: *Base*

PyInteraph2 base class

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.PyInteraph2_w.PyInteraph2_Base.filter

PyInteraph2_Base.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
[str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.PyInteraph2_w.PyInteraph2_Contacts

class AlloViz.Wrappers.PyInteraph2_w.**PyInteraph2_Contacts**(*protein*, *d*)

Bases: *PyInteraph2_Base*

PyInteraph2's side-chain contact frequencies

Contact frequencies based on the fulfillment of a distance threshold (5 angstroms) requirement of each residue pair's COMs' distance

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.PyInteraph2_w.PyInteraph2_Contacts.filter

PyInteraph2_Contacts.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
[str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.PyInteraph2_w.PyInteraph2_Energy

class AlloViz.Wrappers.PyInteraph2_w.**PyInteraph2_Energy**(*protein*, *d*)

Bases: *PyInteraph2_Base*

PyInteraph2's interaction energies

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.PyInteraph2_w.PyInteraph2_Energy.filter

PyInteraph2_Energy.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
[str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.correlationplus_w

correlationplus wrapper

It calculates the Pearson's correlation and the Linear Mutual Information (LMI) of the residues' CA atoms, and also the Pearson's correlation of the residues' backbone dihedral angles (Phi, Psi) and their average.

Classes

correlationplus_Backbone_Dihs(protein, d)	correlationplus' combination of the backbone dihedrals' Pearson's correlations by averaging
correlationplus_CA_LMI(protein, d)	correlationplus' LMI of CA atoms
correlationplus_CA_Pear(protein, d)	correlationplus' Pearson's correlation of CA atoms
correlationplus_Phi(protein, d)	correlationplus' Pearson's correlation of Phi backbone dihedral
correlationplus_Psi(protein, d)	correlationplus' Pearson's correlation of Psi backbone dihedral

AlloViz.Wrappers.correlationplus_w.correlationplus_Backbone_Dihs

class AlloViz.Wrappers.correlationplus_w.**correlationplus_Backbone_Dihs**(*protein*, *d*)

Bases: *Combined_Dihs_Avg*, *correlationplus_CA_Pear*

correlationplus' combination of the backbone dihedrals' Pearson's correlations by averaging

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.correlationplus_w.correlationplus_Backbone_Dihs.filter

correlationplus_Backbone_Dihs.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
[str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.correlationplus_w.correlationplus_CA_LMI

class AlloViz.Wrappers.correlationplus_w.**correlationplus_CA_LMI**(*protein*, *d*)

Bases: *correlationplus_CA_Pear*

correlationplus' LMI of CA atoms

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.correlationplus_w.correlationplus_CA_LMI.filter

correlationplus_CA_LMI.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
[str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.correlationplus_w.correlationplus_CA_Pear

class AlloViz.Wrappers.correlationplus_w.**correlationplus_CA_Pear**(*protein*, *d*)

Bases: *Base*

correlationplus' Pearson's correlation of CA atoms

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.correlationplus_w.correlationplus_CA_Pear.filter

correlationplus_CA_Pear.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
: [str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
: [float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
: [int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
: [int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
: Filtering class.

AlloViz.Protein.calculate
: Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
: Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
: Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.correlationplus_w.correlationplus_Phi

class AlloViz.Wrappers.correlationplus_w.**correlationplus_Phi**(*protein*, *d*)

Bases: *correlationplus_Psi*

correlationplus' Pearson's correlation of Phi backbone dihedral

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.correlationplus_w.correlationplus_Phi.filter

correlationplus_Phi.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
[str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
: Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
: Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
: Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.correlationplus_w.correlationplus_Psi

class AlloViz.Wrappers.correlationplus_w.**correlationplus_Psi**(*protein*, *d*)

Bases: *Base*

correlationplus' Pearson's correlation of Psi backbone dihedral

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.correlationplus_w.correlationplus_Psi.filter

correlationplus_Psi.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
: [str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.dynetan_w

dynetan wrapper

It calculates the Mutual Information of the residues' movement.

Classes

dynetan(protein, d)	dyentan's MI of the residues

AlloViz.Wrappers.dynetan_w.dynetan

class AlloViz.Wrappers.dynetan_w.**dynetan**(*protein*, *d*)

Bases: *Multicore*

dyentan's MI of the residues

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.dynetan_w.dynetan.filter

dynetan.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
[str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.g_correlation_w

g_correlation wrapper

It calculates the Mutual Information (MI) and Linear MI (LMI) of the residues' CA atoms. Only for local installations.

Classes

g_correlation_CA_LMI(protein, d)	g_correlation's LMI of CA atoms
g_correlation_CA_MI(protein, d)	g_correlation's MI of CA atoms

AlloViz.Wrappers.g_correlation_w.g_correlation_CA_LMI

class AlloViz.Wrappers.g_correlation_w.**g_correlation_CA_LMI**(*protein*, *d*)

Bases: *g_correlation_CA_MI*

g_correlation's LMI of CA atoms

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.g_correlation_w.g_correlation_CA_LMI.filter

g_correlation_CA_LMI.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
[str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.g_correlation_w.g_correlation_CA_MI

class AlloViz.Wrappers.g_correlation_w.**g_correlation_CA_MI**(*protein*, *d*)

Bases: *Base*

g_correlation's MI of CA atoms

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.g_correlation_w.g_correlation_CA_MI.filter

g_correlation_CA_MI.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
[str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.pytraj_w

pytraj wrapper

It calculates the Pearson's correlation of the residues' CA and CB atoms.

Classes

pytraj_CA(protein, d)	pytraj's Pearson's correlation of CA atoms
pytraj_CB(protein, d)	pytraj's Pearson's correlation of CB atoms

AlloViz.Wrappers.pytraj_w.pytraj_CA

class AlloViz.Wrappers.pytraj_w.**pytraj_CA**(*protein*, *d*)

Bases: *Base*

pytraj's Pearson's correlation of CA atoms

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.pytraj_w.pytraj_CA.filter

pytraj_CA.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
[str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

AlloViz.Wrappers.pytraj_w.pytraj_CB

class AlloViz.Wrappers.pytraj_w.**pytraj_CB**(*protein*, *d*)

Bases: *pytraj_CA*

pytraj's Pearson's correlation of CB atoms

Methods

filter([filterings])	Filter network edges

AlloViz.Wrappers.pytraj_w.pytraj_CB.filter

pytraj_CB.**filter**(*filterings='all'*, ***kwargs*)

Filter network edges

Filter the networks according to the selected criteria to perform analyses on (all or) a subset of the edges. It calls *AlloViz.Wrappers.Base.Base.filter()* and results are stored in instances of the *AlloViz.AlloViz.Filtering.Filtering* class. The different filtering options are detailed in the *Filtering* module.

Parameters

filterings
[str or list of strs and/or lists, default: "all"] Filtering scheme(s) with which to filter the list of network edges before analysis. It can be a string, or a list of strings and/or lists: a list of lists (also with or without strings) is used to filter with a combination of criteria. All available (and combinable) filtering options are functions in the *Filtering* module: *All()*, *GetContacts_edges()*, *No_Sequence_Neighbors()*, *GPCR_Interhelix()*. The default "all" performs all the available filtering schemes (no combinations).

Other Parameters

GetContacts_threshold
[float] Optional kwarg that can be passed to specify the minimum contact frequency (0-1, default 0) threshold, which will be used to filter out contacts with a frequency (average) lower than it before analysis.

Sequence_Neighbor_distance
[int] Optional kwarg that can be passed to specify the minimum number of sequence positions/distance between residues of a pair to retain in *No_Sequence_Neighbors* filtering, which defaults to 5.

Interresidue_distance
[int or float] Optional kwarg that can be passed to specify the minimum number of angstroms that the CA atoms of residue pairs should have between each other in the initial PDB/structure (default 10 Å) to be considered spatially distant.

See also:

AlloViz.AlloViz.Filtering.Filtering
Filtering class.

AlloViz.Protein.calculate
Class method to calculate the network(s) raw edge weights with different network construction methods.

AlloViz.Protein.analyze
Class method to analyze the calculated raw edge weights with graph theory-based methods.

AlloViz.Protein.view
Class method to visualize the network on the protein structure.

Examples

```
>>> opioidGPCR = AlloViz.Protein(GPCR=169)
>>> opioidGPCR.calculate(["getcontacts", "dynetan"], cores=6, taskcpus=2)
>>> opioidGPCR.dynetan.filter(["GetContacts_edges", ["GetContacts_edges",
↪"GPCR_Interhelix"]])
>>> opioidGPCR.dynetan.GetContacts_edges_GPCR_Interhelix
<AlloViz.AlloViz.Filtering.Filtering at 0x7f892c3c0fa0>
```

BIBLIOGRAPHY

[1] https://docs.python.org/3/library/pickle.html.

[1] https://docs.sqlalchemy.org

[2] https://www.python.org/dev/peps/pep-0249/

[1] https://docs.python.org/3/library/pickle.html.

[1] https://docs.sqlalchemy.org

[2] https://www.python.org/dev/peps/pep-0249/

[1] https://docs.python.org/3/library/pickle.html.

[1] https://docs.sqlalchemy.org

[2] https://www.python.org/dev/peps/pep-0249/

[1] Brinda,K. V and Vishveshwara,S. (2005) A network representation of protein structures: implications for protein stability. Biophys. J., 89, 4159–70.

PYTHON MODULE INDEX

a

INDEX

A

B

C

D

E

F

G

H

I

J

K

L

M

N

P

Q

R

S

T

U

V

W

X