ODDT Documentation

Release 0.6-128-gefc1a7b

Maciej Wojcikowski

Jul 22, 2018

Contents

Contents

CHAPTER 1

Installation

1.1 Requirements

- Python 2.7+ or 3.4+
- OpenBabel (2.3.2+) or/and RDKit (2016.03)
- Numpy (1.8+)
- Scipy (0.14+)
- Sklearn (0.18+)
- joblib (0.8+)
- pandas (0.17.1+)
- Skimage (0.10+) (optional, only for surface generation)

Note: All installation methods assume that one of toolkits is installed. For detailed installation procedure visit toolkit's website (OpenBabel, RDKit)

Most convenient way of installing ODDT is using PIP. All required python modules will be installed automatically, although toolkits, either OpenBabel (`pip install openbabel`) or RDKit need to be installed manually

```
pip install oddt
```

If you want to install cutting edge version (master branch from GitHub) of ODDT also using PIP

```
pip install git+https://github.com/oddt/oddt.git@master
```

Finally you can install ODDT straight from the source

```
wget https://github.com/oddt/oddt/archive/0.5.tar.gz
tar zxvf 0.5.tar.gz
cd oddt-0.5/
python setup.py install
```

1.2 Common installation problems

CHAPTER 2

Usage Instructions

You can use any supported toolkit united under common API (for reference see Pybel or Cinfony). All methods and software which based on Pybel/Cinfony should be drop in compatible with ODDT toolkits. In contrast to its predecessors, which were aimed to have minimalistic API, ODDT introduces extended methods and additional handles. This extensions allow to use toolkits at all its grace and some features may be backported from others to introduce missing functionalities. To name a few:

- coordinates are returned as Numpy Arrays
- atoms and residues methods of Molecule class are lazy, ie. not returning a list of pointers, rather an object which allows indexing and iterating through atoms/residues
- Bond object (similar to Atom)
- *atom_dict*, *ring_dict*, *res_dict* - comprehensive Numpy Arrays containing common information about given entity, particularly useful for high performance computing, ie. interactions, scoring etc.
- lazy Molecule (asynchronous), which is not converted to an object in reading phase, rather passed as a string and read in when underlying object is called
- pickling introduced for Pybel Molecule (internally saved to mol2 string)

2.1 Atom, residues, bonds iteration

One of the most common operation would be iterating through molecules atoms

```
mol = oddt.toolkit.readstring('smi', 'c1cccc1')
for atom in mol:
    print(atom.idx)
```

Note: mol.atoms, returns an object (`AtomStack`) which can be access via indexes or iterated

Iterating over residues is also very convenient, especially for proteins

```
for res in mol.residues:
    print(res.name)
```

Additionally residues can fetch atoms belonging to them:

```
for res in mol.residues:
    for atom in res:
        print(atom.idx)
```

Bonds are also iterable, similar to residues:

```
for bond in mol.bonds:
    print(bond.order)
    for atom in bond:
        print(atom.idx)
```

2.2 Reading molecules

Reading molecules is mostly identical to Pybel.

Reading from file

```
for mol in oddt.toolkit.readfile('smi', 'test.smi'):
    print(mol.title)
```

Reading from string

```
mol = oddt.toolkit.readstring('smi', 'c1ccccc1 benzene'):
    print(mol.title)
```

Note: You can force molecules to be read in asynchronously, aka "lazy molecules". Current default is not to produce lazy molecules due to OpenBabel's Memory Leaks in OBConverter. Main advantage of lazy molecules is using them in multiprocessing, then conversion is spreaded on all jobs.

Reading molecules from file in asynchronous manner

```
for mol in oddt.toolkit.readfile('smi', 'test.smi', lazy=True):
    pass
```

This example will execute instantaneously, since no molecules were evaluated.

2.3 Numpy Dictionaries - store your molecule as an uniform structure

Most important and handy property of Molecule in ODDT are Numpy dictionaries containing most properties of supplied molecule. Some of them are straightforward, other require some calculation, ie. atom features. Dictionaries are provided for major entities of molecule: atoms, bonds, residues and rings. It was primarily used for interactions calculations, although it is applicable for any other calculation. The main benefit is marvelous Numpy broadcasting and subsetting.

Each dictionary is defined as a format in Numpy.

2.3.1 atom_dict

Atom basic information

- '*coords*', type: `float32`, shape: (3) - atom coordinates
- '*charge*', type: `float32` - atom's charge
- '*atomicnum*', type: `int8` - atomic number
- '*atomtype*', type: `a4` - Sybyl atom's type
- '*hybridization*', type: `int8` - atoms hybrydization
- '*neighbors*', type: `float32`, shape: (4,3) - coordinates of non-H neighbors coordinates for angles (max of 4 neighbors should be enough)

Residue information for current atom

- '*resid*', type: `int16` - residue ID
- '*resnumber*', type: `int16` - residue number
- '*resname*', type: `a3` - Residue name (3 letters)
- '*isbackbone*', type: `bool` - is atom part of backbone

Atom properties

- '*isacceptor*', type: `bool` - is atom H-bond acceptor
- '*isdonor*', type: `bool` - is atom H-bond donor
- '*isdonorh*', type: `bool` - is atom H-bond donor Hydrogen
- '*ismetal*', type: `bool` - is atom a metal
- '*ishydrophobe*', type: `bool` - is atom hydrophobic
- '*isaromatic*', type: `bool` - is atom aromatic
- '*isminus*', type: `bool` - is atom negatively charged/chargable
- '*isplus*', type: `bool` - is atom positively charged/chargable
- '*ishalogen*', type: `bool` - is atom a halogen

Secondary structure

- '*isalpha*', type: `bool` - is atom a part of alpha helix
- '*isbeta*', type: `bool'` - is atom a part of beta strand

2.3.2 ring_dict

- '*centroid*', type: `float32`, shape: 3 - coordinates of ring's centroid
- '*vector*', type: `float32`, shape: 3 - normal vector for ring
- '*isalpha*', type: `bool` - is ring a part of alpha helix
- '*isbeta*', type: `bool'` - is ring a part of beta strand

2.3.3 res_dict

- '*id*', type: `int16` - residue ID
- '*resnumber*', type: `int16` - residue number
- '*resname*', type: `a3` - Residue name (3 letters)
- '*N*', type: `float32`, shape: 3 - cordinates of backbone N atom
- '*CA*', type: `float32`, shape: 3 - cordinates of backbone CA atom
- '*C*', type: `float32`, shape: 3 - cordinates of backbone C atom
- '*isalpha*', type: `bool` - is residue a part of alpha helix
- '*isbeta*', type: `bool'` - is residue a part of beta strand

Note: All aforementioned dictionaries are generated "on demand", and are cached for molecule, thus can be shared between calculations. Caching of dictionaries brings incredible performance gain, since in some applications their generation is the major time consuming task.

Get all acceptor atoms:

```
mol.atom_dict['isacceptor']
```

2.4 Interaction Fingerprints

Module, where interactions between two molecules are calculated and stored in fingerprint.

2.4.1 The most common usage

Firstly, loading files

```
protein = next(oddt.toolkit.readfile('pdb', 'protein.pdb'))
protein.protein = True
ligand = next(oddt.toolkit.readfile('sdf', 'ligand.sdf'))
```

Note: You have to mark a variable with file as protein, otherwise You won't be able to get access to e.g. 'resname; , 'resid' etc. It can be done as above.

File with more than one molecule

```
mols = list(oddt.toolkit.readfile('sdf', 'ligands.sdf'))
```

When files are loaded, You can check interactions between molecules. Let's find out, which amino acids creates hydrogen bonds

```
protein_atoms, ligand_atoms, strict = hbonds(protein, ligand)
print(protein_atoms['resname'])
```

Or check hydrophobic contacts between molecules

```
protein_atoms, ligand_atoms = hydrophobic_contacts(protein, ligand)
print(protein_atoms, ligand_atoms)
```

But instead of checking interactions one by one, You can use fingerprints module.

```
IFP = InteractionFingerprint(ligand, protein)
SIFP = SimpleInteractionFingerprint(ligand, protein)
```

Very often we're looking for similar molecules. We can easily accomplish this by e.g.

```
results = []
reference = SimpleInteractionFingerprint(ligand, protein)
for el in query:
    fp_query = SimpleInteractionFingerprint(el, protein)
    # similarity score for current query
    cur_score = dice(reference, fp_query)
    # score is the lowest, required similarity
    if cur_score > score:
        results.append(el)
return results
```

2.5 Molecular shape comparison

Three methods for molecular shape comparison are supported: USR and its two derivatives: USRCAT and Electroshape.

- **USR (Ultrafast Shape Recognition) - function usr(molecule)** Ballester PJ, Richards WG (2007). Ultrafast shape recognition to search compound databases for similar molecular shapes. Journal of computational chemistry, 28(10):1711-23. http://dx.doi.org/10.1002/jcc.20681
- **USRCAT (USR with Credo Atom Types) - function usr_cat(molecule)** Adrian M Schreyer, Tom Blundell (2012). USRCAT: real-time ultrafast shape recognition with pharmacophoric constraints. Journal of Cheminformatics, 2012 4:27. http://dx.doi.org/10.1186/1758-2946-4-27
- **Electroshape - function electroshape(molecule)** Armstrong, M. S. et al. ElectroShape: fast molecular similarity calculations incorporating shape, chirality and electrostatics. J Comput Aided Mol Des 24, 789-801 (2010). http://dx.doi.org/doi:10.1007/s10822-010-9374-0

  Aside from spatial coordinates, atoms' charges are also used as the fourth dimension to describe shape of the molecule.

To find most similar molecules from the given set, each of these methods can be used.

Loading files:

```
query = next(oddt.toolkit.readfile('sdf', 'query.sdf'))
database = list(oddt.toolkit.readfile('sdf', 'database.sdf'))
```

Example code to find similar molecules:

```
results = []
query_shape = usr(query)
for mol in database:
    mol_shape = usr(mol)
    similarity = usr_similarity(query_shape, mol_shape)
    if similarity > 0.7:
        results.append(mol)
```

To use another method, replace usr(mol) with usr_cat(mol) or electroshape(mol).

CHAPTER 3

ODDT command line interface (CLI)

There is an *oddt* command to interface with Open Drug Discovery Toolkit from terminal, without any programming knowleadge. It simply reproduces *oddt.virtualscreening.virtualscreening*. One can filter, dock and score ligands using methods implemented or compatible with ODDT. All positional arguments are treated as input ligands, whereas output must be assigned using *-O* option (following *obabel* convention). Input and output formats are defined using *-i* and *-o* accordingly. If output format is present and no output file is assigned, then molecules are printed to STDOUT.

To list all the available options issue *-h* option:

```
oddt_cli -h
```

1. Docking ligand using Autodock Vina (construct box using ligand from crystal structure) with additional RFscore v2 rescoring:

```
oddt_cli input_ligands.sdf --dock autodock_vina --receptor rec.mol2 --auto_ligand␣
↪crystal_ligand.mol2 --score rfscore_v2 -O output_ligands.sdf
```

2. Filtering ligands using Lipinski RO5 and PAINS. Afterwards dock with Autodock Vina:

```
oddt_cli input_ligands.sdf --filter ro5 --filter pains --dock autodock_vina --
↪receptor rec.mol2 --auto_ligand crystal_ligand.mol2 -O output_ligands.sdf
```

3. Dock with Autodock Vina, with precise box position and dimensions. Fix seed for reproducibility and increase exhaustiveness:

```
oddt_cli ampc/actives_final.mol2.gz --dock autodock_vina --receptor ampc/receptor.pdb␣
↪--size '(8,8,8)' --center '(1,2,0.5)' --exhaustiveness 20 --seed 1 -O ampc_docked.
↪sdf
```

4. Rescore ligands using 3 versions of RFscore and pre-trained scoring function (either pickle from ODDT or any other SF implementing *oddt.scoring.scorer* API):

```
oddt_cli docked_ligands.sdf --receptor rec.mol2 --score rfscore_v1 --score rfscore_v2␣
↪--score rfscore_v3 --score TrainedNN.pickle -O docked_ligands_rescored.sdf
```

CHAPTER 4

Development and contributions guide

1. Indicies All indicies within toolkit are 0-based, but for backward compatibility with OpenBabel there is `mol.idx` property. If you develop using ODDT you are encouraged to use 0-based indicies and/or `mol.idx0` and `mol.idx1` properties to be exact which convention you adhere to. Otherwise you can run into bags which are hard to catch, when writing toolkit independent code.

CHAPTER 5

ODDT API documentation

5.1 oddt package

5.1.1 Subpackages

oddt.docking package

Submodules

oddt.docking.AutodockVina module

class oddt.docking.AutodockVina.**autodock_vina**(*protein=None*, *auto_ligand=None*, *size=(20, 20, 20)*, *center=(0, 0, 0)*, *exhaustiveness=8*, *num_modes=9*, *energy_range=3*, *seed=None*, *prefix_dir='/tmp'*, *n_cpu=1*, *executable=None*, *autocleanup=True*, *skip_bad_mols=True*)

Bases: object

Autodock Vina docking engine, which extends it's capabilities: automatic box (auto-centering on ligand).

Parameters

protein: oddt.toolkit.Molecule object (default=None) Protein object to be used while generating descriptors.

auto_ligand: oddt.toolkit.Molecule object or string (default=None) Ligand use to center the docking box. Either ODDT molecule or a file (opened based on extesion and read to ODDT molecule). Box is centered on geometric center of molecule.

size: tuple, shape=[3] (default=(20, 20, 20)) Dimentions of docking box (in Angstroms)

center: tuple, shape=[3] (default=(0,0,0)) The center of docking box in cartesian space.

exhaustiveness: int (default=8) Exhaustiveness parameter of Autodock Vina

num_modes: int (default=9) Number of conformations generated by Autodock Vina. The maximum number of docked poses is 9 (due to Autodock Vina limitation).

energy_range: int (default=3) Energy range cutoff for Autodock Vina

seed: int or None (default=None) Random seed for Autodock Vina

prefix_dir: string (default=/tmp) Temporary directory for Autodock Vina files

executable: string or None (default=None) Autodock Vina executable location in the system. It's realy necessary if autodetection fails.

autocleanup: bool (default=True) Should the docking engine clean up after execution?

skip_bad_mols: bool (default=True) Should molecules that crash Autodock Vina be skipped.

Attributes

tmp_dir

Methods

dock(ligands[, protein])	Automated docking procedure.
predict_ligand(ligand)	Local method to score one ligand and update it's scores.
predict_ligands(ligands)	Method to score ligands lazily
score(ligands[, protein])	Automated scoring procedure.
set_protein(protein)	Change protein to dock to.

clean

clean()

dock(*ligands*, *protein=None*)
Automated docking procedure.

Parameters

ligands: iterable of oddt.toolkit.Molecule objects Ligands to dock

protein: oddt.toolkit.Molecule object or None Protein object to be used. If None, then the default one is used, else the protein is new default.

Returns

ligands [array of oddt.toolkit.Molecule objects] Array of ligands (scores are stored in mol.data method)

predict_ligand(*ligand*)
Local method to score one ligand and update it's scores.

Parameters

ligand: oddt.toolkit.Molecule object Ligand to be scored

Returns

ligand: oddt.toolkit.Molecule object Scored ligand with updated scores

predict_ligands(*ligands*)
Method to score ligands lazily

Parameters

ligands: iterable of oddt.toolkit.Molecule objects Ligands to be scored

Returns

ligand: iterator of oddt.toolkit.Molecule objects Scored ligands with updated scores

score(*ligands*, *protein=None*)
Automated scoring procedure.

Parameters

ligands: iterable of oddt.toolkit.Molecule objects Ligands to score

protein: oddt.toolkit.Molecule object or None Protein object to be used. If None, then the default one is used, else the protein is new default.

Returns

ligands [array of oddt.toolkit.Molecule objects] Array of ligands (scores are stored in mol.data method)

set_protein(*protein*)
Change protein to dock to.

Parameters

protein: oddt.toolkit.Molecule object Protein object to be used.

tmp_dir

oddt.docking.AutodockVina.**parse_vina_docking_output**(*output*)
Function parsing Autodock Vina docking output to a dictionary

Parameters

output [string] Autodock Vina standard ouptud (STDOUT).

Returns

out [dict] dicitionary containing scores computed by Autodock Vina

oddt.docking.AutodockVina.**parse_vina_scoring_output**(*output*)
Function parsing Autodock Vina scoring output to a dictionary

Parameters

output [string] Autodock Vina standard ouptud (STDOUT).

Returns

out [dict] dicitionary containing scores computed by Autodock Vina

oddt.docking.AutodockVina.**write_vina_pdbqt**(*mol*, *directory*, *flexible=True*, *name_id=None*)
Write single PDBQT molecule to a given directory. For proteins use *flexible=False* to avoid encoding torsions. Additionally an name ID can be appended to a name to avoid conflicts.

oddt.docking.internal module

ODDT's internal docking/scoring engines

oddt.docking.internal.**change_dihedral**(*coords*, *a1*, *a2*, *a3*, *a4*, *target_angle*, *rot_mask*)

oddt.docking.internal.**get_children**(*molecule*, *mother*, *restricted*)

oddt.docking.internal.**get_close_neighbors**(*molecule*, *a_idx*, *num_bonds=1*)

oddt.docking.internal.**num_rotors_pdbqt**(*lig*)

class oddt.docking.internal.**vina_docking**(*rec*, *lig=None*, *box=None*, *box_size=1.0*, *weights=None*)

Bases: object

Methods

correct_radius	
score	
score_inter	
score_intra	
score_total	
set_box	
set_coords	
set_ligand	
set_protein	
weighted_inter	
weighted_intra	
weighted_total	

correct_radius(*atom_dict*)

score(*coords=None*)

score_inter(*coords=None*)

score_intra(*coords=None*)

score_total(*coords=None*)

set_box(*box*)

set_coords(*coords*)

set_ligand(*lig*)

set_protein(*rec*)

weighted_inter(*coords=None*)

weighted_intra(*coords=None*)

weighted_total(*coords=None*)

class oddt.docking.internal.**vina_ligand**(*c0*, *num_rotors*, *engine*, *box_size=1*)

Bases: object

Methods

mutate	

mutate(*x2*, *force=False*)

Module contents

class oddt.docking.**autodock_vina**(*protein=None*, *auto_ligand=None*, *size=(20, 20, 20)*, *center=(0, 0, 0)*, *exhaustiveness=8*, *num_modes=9*, *energy_range=3*, *seed=None*, *prefix_dir='/tmp'*, *n_cpu=1*, *executable=None*, *autocleanup=True*, *skip_bad_mols=True*)

Bases: object

Autodock Vina docking engine, which extends it's capabilities: automatic box (auto-centering on ligand).

Parameters

protein: oddt.toolkit.Molecule object (default=None) Protein object to be used while generating descriptors.

auto_ligand: oddt.toolkit.Molecule object or string (default=None) Ligand use to center the docking box. Either ODDT molecule or a file (opened based on extesion and read to ODDT molecule). Box is centered on geometric center of molecule.

size: tuple, shape=[3] (default=(20, 20, 20)) Dimentions of docking box (in Angstroms)

center: tuple, shape=[3] (default=(0,0,0)) The center of docking box in cartesian space.

exhaustiveness: int (default=8) Exhaustiveness parameter of Autodock Vina

num_modes: int (default=9) Number of conformations generated by Autodock Vina. The maximum number of docked poses is 9 (due to Autodock Vina limitation).

energy_range: int (default=3) Energy range cutoff for Autodock Vina

seed: int or None (default=None) Random seed for Autodock Vina

prefix_dir: string (default=/tmp) Temporary directory for Autodock Vina files

executable: string or None (default=None) Autodock Vina executable location in the system. It's realy necessary if autodetection fails.

autocleanup: bool (default=True) Should the docking engine clean up after execution?

skip_bad_mols: bool (default=True) Should molecules that crash Autodock Vina be skipped.

Attributes

tmp_dir

Methods

dock(ligands[, protein])	Automated docking procedure.
predict_ligand(ligand)	Local method to score one ligand and update it's scores.
predict_ligands(ligands)	Method to score ligands lazily
score(ligands[, protein])	Automated scoring procedure.
set_protein(protein)	Change protein to dock to.

clean

clean()

dock(*ligands*, *protein=None*)
Automated docking procedure.

Parameters

ligands: iterable of oddt.toolkit.Molecule objects Ligands to dock

protein: oddt.toolkit.Molecule object or None Protein object to be used. If None, then the default one is used, else the protein is new default.

Returns

ligands [array of oddt.toolkit.Molecule objects] Array of ligands (scores are stored in mol.data method)

predict_ligand(*ligand*)
Local method to score one ligand and update it's scores.

Parameters

ligand: oddt.toolkit.Molecule object Ligand to be scored

Returns

ligand: oddt.toolkit.Molecule object Scored ligand with updated scores

predict_ligands(*ligands*)
Method to score ligands lazily

Parameters

ligands: iterable of oddt.toolkit.Molecule objects Ligands to be scored

Returns

ligand: iterator of oddt.toolkit.Molecule objects Scored ligands with updated scores

score(*ligands*, *protein=None*)
Automated scoring procedure.

Parameters

ligands: iterable of oddt.toolkit.Molecule objects Ligands to score

protein: oddt.toolkit.Molecule object or None Protein object to be used. If None, then the default one is used, else the protein is new default.

Returns

ligands [array of oddt.toolkit.Molecule objects] Array of ligands (scores are stored in mol.data method)

set_protein(*protein*)
Change protein to dock to.

Parameters

protein: oddt.toolkit.Molecule object Protein object to be used.

tmp_dir

oddt.scoring package

Subpackages

oddt.scoring.descriptors package

Submodules

oddt.scoring.descriptors.binana module

Internal implementation of binana software (http://nbcr.ucsd.edu/data/sw/hosted/binana/)

class oddt.scoring.descriptors.binana.**binana_descriptor**(*protein=None*)
Bases: object

Descriptor build from binana script (as used in NNScore 2.0

Parameters

protein: oddt.toolkit.Molecule object (default=None) Protein object to be used while generating descriptors.

Methods

build(ligands[, protein])	Descriptor building method
set_protein(protein)	One function to change all relevant proteins

build(*ligands*, *protein=None*)
Descriptor building method

Parameters

ligands: array-like An array of generator of oddt.toolkit.Molecule objects for which the descriptor is computed

protein: oddt.toolkit.Molecule object (default=None) Protein object to be used while generating descriptors. If none, then the default protein (from constructor) is used. Otherwise, protein becomes new global and default protein.

Returns

descs: numpy array, shape=[n_samples, 351] An array of binana descriptors, aligned with input ligands

set_protein(*protein*)
One function to change all relevant proteins

Parameters

protein: oddt.toolkit.Molecule object Protein object to be used while generating descriptors. Protein becomes new global and default protein.

Module contents

class oddt.scoring.descriptors.**close_contacts_descriptor**(*protein=None*, *cutoff=4*, *mode='atomic_nums'*, *ligand_types=None*, *protein_types=None*, *aligned_pairs=False*)

Bases: object

Close contacts descriptor which tallies atoms of type X in certain cutoff from atoms of type Y.

Parameters

protein: oddt.toolkit.Molecule or None (default=None) Default protein to use as reference

cutoff: int or list, shape=[n,] or shape=[n,2] (default=4) Cutoff for atoms in Angstroms given as an integer or a list of ranges, eg. [0, 4, 8, 12] or [[0,4],[4,8],[8,12]]. Upper bound is always inclusive, lower exclusive.

mode: string (default='atomic_nums') Method of atoms selection, as used in *atoms_by_type*

ligand_types: array List of ligand atom types to use

protein_types: array List of protein atom types to use

aligned_pairs: bool (default=False) Flag indicating should permutation of types should be done, otherwise the atoms are treated as aligned pairs.

Methods

build(ligands[, protein])	Builds descriptors for series of ligands

build(*ligands*, *protein=None*)

Builds descriptors for series of ligands

Parameters

ligands: iterable of oddt.toolkit.Molecules or oddt.toolkit.Molecule A list or iterable of ligands to build the descriptor or a single molecule.

protein: oddt.toolkit.Molecule or None (default=None) Default protein to use as reference

class oddt.scoring.descriptors.**fingerprints**(*fp='fp2'*, *toolkit='ob'*)

Bases: object

Methods

build

build(*mols*)

class oddt.scoring.descriptors.**autodock_vina_descriptor**(*protein=None*, *vina_scores=None*)

Bases: object

Methods

build	
set_protein	

build(*ligands*, *protein=None*)

set_protein(*protein*)

class oddt.scoring.descriptors.**oddt_vina_descriptor**(*protein=None*, *vina_scores=None*)

Bases: object

Methods

build	
set_protein	

build(*ligands*, *protein=None*)

set_protein(*protein*)

oddt.scoring.functions package

Submodules

oddt.scoring.functions.NNScore module

class oddt.scoring.functions.NNScore.**nnscore**(*protein=None*, *n_jobs=-1*)

Bases: *oddt.scoring.scorer*

NNScore implementation *[1]*. Based on Binana descriptors *[2]* and an ensemble of 20 best scored nerual networks with a hidden layer of 5 nodes. The NNScore predicts binding affinity (pKi/d).

Parameters

protein [oddt.toolkit.Molecule object] Receptor for the scored ligands

n_jobs: int (default=-1) Number of cores to use for scoring and training. By default (-1) all cores are allocated.

References

[1], *[2]*

Methods

fit(ligands, target, *args, **kwargs)	Trains model on supplied ligands and target values
predict(ligands, *args, **kwargs)	Predicts values (eg.

Continued on next page

Table 5 – continued from previous page

predict_ligand(ligand)	Local method to score one ligand and update it's scores.
predict_ligands(ligands)	Method to score ligands in a lazy fashion.
save(filename)	Saves scoring function to a pickle file.
score(ligands, target, *args, **kwargs)	Methods estimates the quality of prediction using model's default score (accuracy for classification or R^2 for regression)
set_protein(protein)	Proxy method to update protein in all relevant places.

gen_training_data	
load	
train	

fit (*ligands*, *target*, **args*, ***kwargs*)
Trains model on supplied ligands and target values

Parameters

ligands: array-like of ligands Molecules to featurize and feed into the model

target: array-like of shape = [n_samples] or [n_samples, n_outputs] Ground truth (correct) target values.

gen_training_data (*pdbbind_dir*, *pdbbind_versions=(2007*, *2012*, *2013*, *2014*, *2015*, *2016)*, *home_dir=None*, *use_proteins=False*)

classmethod load (*filename=None*, *pdbbind_version=2016*)
Loads scoring function from a pickle file.

Parameters

filename: string Pickle filename

Returns

sf: scorer-like object Scoring function object loaded from a pickle

predict (*ligands*, **args*, ***kwargs*)
Predicts values (eg. affinity) for supplied ligands.

Parameters

ligands: array-like of ligands Molecules to featurize and feed into the model

Returns

predicted: np.array or array of np.arrays of shape = [n_ligands] Predicted scores for ligands

predict_ligand (*ligand*)
Local method to score one ligand and update it's scores.

Parameters

ligand: oddt.toolkit.Molecule object Ligand to be scored

Returns

ligand: oddt.toolkit.Molecule object Scored ligand with updated scores

predict_ligands (*ligands*)
Method to score ligands in a lazy fashion.

Parameters

ligands: iterable of oddt.toolkit.Molecule objects Ligands to be scored

Returns

ligand: iterator of oddt.toolkit.Molecule objects Scored ligands with updated scores

save (*filename*)
Saves scoring function to a pickle file.

Parameters

filename: string Pickle filename

score (*ligands*, *target*, **args*, ***kwargs*)
Methods estimates the quality of prediction using model's default score (accuracy for classification or R^2 for regression)

Parameters

ligands: array-like of ligands Molecules to featurize and feed into the model

target: array-like of shape = [n_samples] or [n_samples, n_outputs] Ground truth (correct) target values.

Returns

s: float Quality score (accuracy or R^2) for prediction

set_protein (*protein*)
Proxy method to update protein in all relevant places.

Parameters

protein: oddt.toolkit.Molecule object New default protein

train (*home_dir=None*, *sf_pickle=None*, *pdbbind_version=2016*)

oddt.scoring.functions.PLECscore module

class oddt.scoring.functions.PLECscore.**PLECscore** (*protein=None*, *n_jobs=-1*, *version='linear'*, *depth_protein=5*, *depth_ligand=1*, *size=65536*)

Bases: *oddt.scoring.scorer*

PLECscore - a novel scoring function based on PLEC fingerprints. The underlying model can be one of:

- linear regression
- neural network (dense, 200x200x200)
- random forest (100 trees)

The scoring function is trained on PDBbind v2016 database and even with linear model outperforms other machine-learning ones in terms of Pearson correlation coefficient on "core set". For details see PLEC publication. PLECscore predicts binding affinity (pKi/d).

New in version 0.6.

Parameters

protein [oddt.toolkit.Molecule object] Receptor for the scored ligands

n_jobs: int (default=-1) Number of cores to use for scoring and training. By default (-1) all cores are allocated.

version: str (default='linear') A version of scoring function ('linear', 'nn' or 'rf') - which model should be used for the scoring function.

depth_protein: int (default=5) The depth of ECFP environments generated on the protein side of interaction. By default 6 (0 to 5) environments are generated.

depth_ligand: int (default=1) The depth of ECFP environments generated on the ligand side of interaction. By default 2 (0 to 1) environments are generated.

size: int (default=65536) The final size of a folded PLEC fingerprint. This setting is not used to limit the data encoded in PLEC fingerprint (for that tune the depths), but only the final lenght. Setting it to too low value will lead to many collisions.

Methods

fit(ligands, target, *args, **kwargs)	Trains model on supplied ligands and target values
predict(ligands, *args, **kwargs)	Predicts values (eg.
predict_ligand(ligand)	Local method to score one ligand and update it's scores.
predict_ligands(ligands)	Method to score ligands in a lazy fashion.
save(filename)	Saves scoring function to a pickle file.
score(ligands, target, *args, **kwargs)	Methods estimates the quality of prediction using model's default score (accuracy for classification or R^2 for regression)
set_protein(protein)	Proxy method to update protein in all relevant places.

gen_json	
gen_training_data	
load	
train	

fit (*ligands*, *target*, **args*, ***kwargs*)
Trains model on supplied ligands and target values

Parameters

ligands: array-like of ligands Molecules to featurize and feed into the model

target: array-like of shape = [n_samples] or [n_samples, n_outputs] Ground truth (correct) target values.

gen_json (*home_dir=None*, *pdbbind_version=2016*)

gen_training_data (*pdbbind_dir*, *pdbbind_versions=(2016,)*, *home_dir=None*, *use_proteins=True*)

classmethod load (*filename=None*, *version='linear'*, *pdbbind_version=2016*, *depth_protein=5*, *depth_ligand=1*, *size=65536*)
Loads scoring function from a pickle file.

Parameters

filename: string Pickle filename

Returns

sf: scorer-like object Scoring function object loaded from a pickle

predict (*ligands*, **args*, ***kwargs*)
Predicts values (eg. affinity) for supplied ligands.

Parameters

ligands: array-like of ligands Molecules to featurize and feed into the model

Returns

predicted: np.array or array of np.arrays of shape = [n_ligands] Predicted scores for ligands

predict_ligand (*ligand*)
Local method to score one ligand and update it's scores.

Parameters

ligand: oddt.toolkit.Molecule object Ligand to be scored

Returns

ligand: oddt.toolkit.Molecule object Scored ligand with updated scores

predict_ligands (*ligands*)
Method to score ligands in a lazy fashion.

Parameters

ligands: iterable of oddt.toolkit.Molecule objects Ligands to be scored

Returns

ligand: iterator of oddt.toolkit.Molecule objects Scored ligands with updated scores

save (*filename*)
Saves scoring function to a pickle file.

Parameters

filename: string Pickle filename

score (*ligands*, *target*, **args*, ***kwargs*)
Methods estimates the quality of prediction using model's default score (accuracy for classification or R^2 for regression)

Parameters

ligands: array-like of ligands Molecules to featurize and feed into the model

target: array-like of shape = [n_samples] or [n_samples, n_outputs] Ground truth (correct) target values.

Returns

s: float Quality score (accuracy or R^2) for prediction

set_protein (*protein*)
Proxy method to update protein in all relevant places.

Parameters

protein: oddt.toolkit.Molecule object New default protein

train (*home_dir=None*, *sf_pickle=None*, *pdbbind_version=2016*, *ignore_json=False*)

oddt.scoring.functions.RFScore module

class oddt.scoring.functions.RFScore.**rfscore**(*protein=None*, *n_jobs=-1*, *version=1*, *spr=0*, ***kwargs*)

Bases: *oddt.scoring.scorer*

Scoring function implementing RF-Score variants. It predicts the binding affinity (pKi/d) of ligand in a complex utilizng simple descriptors (close contacts of atoms <12A) with sophisticated machine-learning model (random forest). The third variand supplements those contacts with Vina partial scores. For futher details see RF-Score publications v1[Rd9e4db499696-1]_, v2[Rd9e4db499696-2]_, v3[Rd9e4db499696-3]_.

Parameters

protein [oddt.toolkit.Molecule object] Receptor for the scored ligands

n_jobs: int (default=-1) Number of cores to use for scoring and training. By default (-1) all cores are allocated.

version: int (default=1) Scoring function variant. The deault is the simplest one (v1).

spr: int (default=0) The minimum number of contacts in each pair of atom types in the training set for the column to be included in training. This is a way of removal of not frequent and empty contacts.

References

[1], *[2]*, *[3]*

Methods

fit(ligands, target, *args, **kwargs)	Trains model on supplied ligands and target values
predict(ligands, *args, **kwargs)	Predicts values (eg.
predict_ligand(ligand)	Local method to score one ligand and update it's scores.
predict_ligands(ligands)	Method to score ligands in a lazy fashion.
save(filename)	Saves scoring function to a pickle file.
score(ligands, target, *args, **kwargs)	Methods estimates the quality of prediction using model's default score (accuracy for classification or R^2 for regression)
set_protein(protein)	Proxy method to update protein in all relevant places.

gen_training_data	
load	
train	

fit(*ligands*, *target*, **args*, ***kwargs*)

Trains model on supplied ligands and target values

Parameters

ligands: array-like of ligands Molecules to featurize and feed into the model

target: array-like of shape = [n_samples] or [n_samples, n_outputs] Ground truth (correct) target values.

gen_training_data(*pdbbind_dir*, *pdbbind_versions=(2007, 2012, 2013, 2014, 2015, 2016)*, *home_dir=None*, *use_proteins=False*)

classmethod load(*filename=None*, *version=1*, *pdbbind_version=2016*)

Loads scoring function from a pickle file.

Parameters

filename: string Pickle filename

Returns

sf: scorer-like object Scoring function object loaded from a pickle

predict(*ligands*, **args*, ***kwargs*)

Predicts values (eg. affinity) for supplied ligands.

Parameters

ligands: array-like of ligands Molecules to featurize and feed into the model

Returns

predicted: np.array or array of np.arrays of shape = [n_ligands] Predicted scores for ligands

predict_ligand(*ligand*)

Local method to score one ligand and update it's scores.

Parameters

ligand: oddt.toolkit.Molecule object Ligand to be scored

Returns

ligand: oddt.toolkit.Molecule object Scored ligand with updated scores

predict_ligands(*ligands*)

Method to score ligands in a lazy fashion.

Parameters

ligands: iterable of oddt.toolkit.Molecule objects Ligands to be scored

Returns

ligand: iterator of oddt.toolkit.Molecule objects Scored ligands with updated scores

save(*filename*)

Saves scoring function to a pickle file.

Parameters

filename: string Pickle filename

score(*ligands*, *target*, **args*, ***kwargs*)

Methods estimates the quality of prediction using model's default score (accuracy for classification or R^2 for regression)

Parameters

ligands: array-like of ligands Molecules to featurize and feed into the model

target: array-like of shape = [n_samples] or [n_samples, n_outputs] Ground truth (correct) target values.

Returns

s: float Quality score (accuracy or R^2) for prediction

set_protein(*protein*)
Proxy method to update protein in all relevant places.

Parameters

protein: oddt.toolkit.Molecule object New default protein

train(*home_dir=None*, *sf_pickle=None*, *pdbbind_version=2016*)

Module contents

class oddt.scoring.functions.**rfscore**(*protein=None*, *n_jobs=-1*, *version=1*, *spr=0*, ***kwargs*)

Bases: *oddt.scoring.scorer*

Scoring function implementing RF-Score variants. It predicts the binding affinity (pKi/d) of ligand in a complex utilizng simple descriptors (close contacts of atoms <12A) with sophisticated machine-learning model (random forest). The third variand supplements those contacts with Vina partial scores. For futher details see RF-Score publications v1[R062ccc3ea4fa-1]_, v2[R062ccc3ea4fa-2]_, v3[R062ccc3ea4fa-3]_.

Parameters

protein [oddt.toolkit.Molecule object] Receptor for the scored ligands

n_jobs: int (default=-1) Number of cores to use for scoring and training. By default (-1) all cores are allocated.

version: int (default=1) Scoring function variant. The deault is the simplest one (v1).

spr: int (default=0) The minimum number of contacts in each pair of atom types in the training set for the column to be included in training. This is a way of removal of not frequent and empty contacts.

References

[1], *[2]*, *[3]*

Methods

fit(ligands, target, *args, **kwargs)	Trains model on supplied ligands and target values
predict(ligands, *args, **kwargs)	Predicts values (eg.
predict_ligand(ligand)	Local method to score one ligand and update it's scores.
predict_ligands(ligands)	Method to score ligands in a lazy fashion.
save(filename)	Saves scoring function to a pickle file.
score(ligands, target, *args, **kwargs)	Methods estimates the quality of prediction using model's default score (accuracy for classification or R^2 for regression)
set_protein(protein)	Proxy method to update protein in all relevant places.

gen_training_data	
load	
train	

fit (*ligands*, *target*, **args*, ***kwargs*)
Trains model on supplied ligands and target values

Parameters

ligands: array-like of ligands Molecules to featurize and feed into the model

target: array-like of shape = [n_samples] or [n_samples, n_outputs] Ground truth (correct) target values.

gen_training_data (*pdbbind_dir*, *pdbbind_versions=(2007*, *2012*, *2013*, *2014*, *2015*, *2016)*, *home_dir=None*, *use_proteins=False*)

classmethod load (*filename=None*, *version=1*, *pdbbind_version=2016*)
Loads scoring function from a pickle file.

Parameters

filename: string Pickle filename

Returns

sf: scorer-like object Scoring function object loaded from a pickle

predict (*ligands*, **args*, ***kwargs*)
Predicts values (eg. affinity) for supplied ligands.

Parameters

ligands: array-like of ligands Molecules to featurize and feed into the model

Returns

predicted: np.array or array of np.arrays of shape = [n_ligands] Predicted scores for ligands

predict_ligand (*ligand*)
Local method to score one ligand and update it's scores.

Parameters

ligand: oddt.toolkit.Molecule object Ligand to be scored

Returns

ligand: oddt.toolkit.Molecule object Scored ligand with updated scores

predict_ligands (*ligands*)
Method to score ligands in a lazy fashion.

Parameters

ligands: iterable of oddt.toolkit.Molecule objects Ligands to be scored

Returns

ligand: iterator of oddt.toolkit.Molecule objects Scored ligands with updated scores

save (*filename*)
Saves scoring function to a pickle file.

Parameters

filename: string Pickle filename

score (*ligands*, *target*, **args*, ***kwargs*)
Methods estimates the quality of prediction using model's default score (accuracy for classification or R^2 for regression)

Parameters

ligands: array-like of ligands Molecules to featurize and feed into the model

target: array-like of shape = [n_samples] or [n_samples, n_outputs] Ground truth (correct) target values.

Returns

s: float Quality score (accuracy or R^2) for prediction

set_protein(*protein*)
Proxy method to update protein in all relevant places.

Parameters

protein: oddt.toolkit.Molecule object New default protein

train(*home_dir=None*, *sf_pickle=None*, *pdbbind_version=2016*)

class oddt.scoring.functions.**nnscore**(*protein=None*, *n_jobs=-1*)
Bases: *oddt.scoring.scorer*

NNScore implementation *[1]*. Based on Binana descriptors *[2]* and an ensemble of 20 best scored nerual networks with a hidden layer of 5 nodes. The NNScore predicts binding affinity (pKi/d).

Parameters

protein [oddt.toolkit.Molecule object] Receptor for the scored ligands

n_jobs: int (default=-1) Number of cores to use for scoring and training. By default (-1) all cores are allocated.

References

[1], *[2]*

Methods

fit(ligands, target, *args, **kwargs)	Trains model on supplied ligands and target values
predict(ligands, *args, **kwargs)	Predicts values (eg.
predict_ligand(ligand)	Local method to score one ligand and update it's scores.
predict_ligands(ligands)	Method to score ligands in a lazy fashion.
save(filename)	Saves scoring function to a pickle file.
score(ligands, target, *args, **kwargs)	Methods estimates the quality of prediction using model's default score (accuracy for classification or R^2 for regression)
set_protein(protein)	Proxy method to update protein in all relevant places.

gen_training_data	
load	
train	

fit(*ligands*, *target*, **args*, ***kwargs*)
Trains model on supplied ligands and target values

Parameters

ligands: array-like of ligands Molecules to featurize and feed into the model

target: array-like of shape = [n_samples] or [n_samples, n_outputs] Ground truth (correct) target values.

gen_training_data(*pdbbind_dir*, *pdbbind_versions=(2007*, *2012*, *2013*, *2014*, *2015*, *2016)*, *home_dir=None*, *use_proteins=False*)

classmethod load(*filename=None*, *pdbbind_version=2016*)

Loads scoring function from a pickle file.

Parameters

filename: string Pickle filename

Returns

sf: scorer-like object Scoring function object loaded from a pickle

predict(*ligands*, **args*, ***kwargs*)

Predicts values (eg. affinity) for supplied ligands.

Parameters

ligands: array-like of ligands Molecules to featurize and feed into the model

Returns

predicted: np.array or array of np.arrays of shape = [n_ligands] Predicted scores for ligands

predict_ligand(*ligand*)

Local method to score one ligand and update it's scores.

Parameters

ligand: oddt.toolkit.Molecule object Ligand to be scored

Returns

ligand: oddt.toolkit.Molecule object Scored ligand with updated scores

predict_ligands(*ligands*)

Method to score ligands in a lazy fashion.

Parameters

ligands: iterable of oddt.toolkit.Molecule objects Ligands to be scored

Returns

ligand: iterator of oddt.toolkit.Molecule objects Scored ligands with updated scores

save(*filename*)

Saves scoring function to a pickle file.

Parameters

filename: string Pickle filename

score(*ligands*, *target*, **args*, ***kwargs*)

Methods estimates the quality of prediction using model's default score (accuracy for classification or R^2 for regression)

Parameters

ligands: array-like of ligands Molecules to featurize and feed into the model

target: array-like of shape = [n_samples] or [n_samples, n_outputs] Ground truth (correct) target values.

Returns

s: float Quality score (accuracy or R^2) for prediction

set_protein(*protein*)
Proxy method to update protein in all relevant places.

Parameters

protein: oddt.toolkit.Molecule object New default protein

train(*home_dir=None*, *sf_pickle=None*, *pdbbind_version=2016*)

class oddt.scoring.functions.**PLECscore**(*protein=None*, *n_jobs=-1*, *version='linear'*, *depth_protein=5*, *depth_ligand=1*, *size=65536*)
Bases: *oddt.scoring.scorer*

PLECscore - a novel scoring function based on PLEC fingerprints. The underlying model can be one of:

- linear regression
- neural network (dense, 200x200x200)
- random forest (100 trees)

The scoring function is trained on PDBbind v2016 database and even with linear model outperforms other machine-learning ones in terms of Pearson correlation coefficient on "core set". For details see PLEC publication. PLECscore predicts binding affinity (pKi/d).

New in version 0.6.

Parameters

protein [oddt.toolkit.Molecule object] Receptor for the scored ligands

n_jobs: int (default=-1) Number of cores to use for scoring and training. By default (-1) all cores are allocated.

version: str (default='linear') A version of scoring function ('linear', 'nn' or 'rf') - which model should be used for the scoring function.

depth_protein: int (default=5) The depth of ECFP environments generated on the protein side of interaction. By default 6 (0 to 5) environments are generated.

depth_ligand: int (default=1) The depth of ECFP environments generated on the ligand side of interaction. By default 2 (0 to 1) environments are generated.

size: int (default=65536) The final size of a folded PLEC fingerprint. This setting is not used to limit the data encoded in PLEC fingerprint (for that tune the depths), but only the final lenght. Setting it to too low value will lead to many collisions.

Methods

fit
predict
predict_ligand
predict_ligands
save

Continued on next page

Table 10 – continued from previous page

score
set_protein

gen_json	
gen_training_data	
load	
train	

fit (*ligands*, *target*, **args*, ***kwargs*)
Trains model on supplied ligands and target values

Parameters

ligands: array-like of ligands Molecules to featurize and feed into the model

target: array-like of shape = [n_samples] or [n_samples, n_outputs] Ground truth (correct) target values.

gen_json (*home_dir=None*, *pdbbind_version=2016*)

gen_training_data (*pdbbind_dir*, *pdbbind_versions=(2016,)*, *home_dir=None*, *use_proteins=True*)

classmethod load (*filename=None*, *version='linear'*, *pdbbind_version=2016*, *depth_protein=5*, *depth_ligand=1*, *size=65536*)
Loads scoring function from a pickle file.

Parameters

filename: string Pickle filename

Returns

sf: scorer-like object Scoring function object loaded from a pickle

predict (*ligands*, **args*, ***kwargs*)
Predicts values (eg. affinity) for supplied ligands.

Parameters

ligands: array-like of ligands Molecules to featurize and feed into the model

Returns

predicted: np.array or array of np.arrays of shape = [n_ligands] Predicted scores for ligands

predict_ligand (*ligand*)
Local method to score one ligand and update it's scores.

Parameters

ligand: oddt.toolkit.Molecule object Ligand to be scored

Returns

ligand: oddt.toolkit.Molecule object Scored ligand with updated scores

predict_ligands (*ligands*)
Method to score ligands in a lazy fashion.

Parameters

ligands: iterable of oddt.toolkit.Molecule objects Ligands to be scored

Returns

ligand: iterator of oddt.toolkit.Molecule objects Scored ligands with updated scores

save (*filename*)
Saves scoring function to a pickle file.

Parameters

filename: string Pickle filename

score (*ligands*, *target*, **args*, ***kwargs*)
Methods estimates the quality of prediction using model's default score (accuracy for classification or R^2 for regression)

Parameters

ligands: array-like of ligands Molecules to featurize and feed into the model

target: array-like of shape = [n_samples] or [n_samples, n_outputs] Ground truth (correct) target values.

Returns

s: float Quality score (accuracy or R^2) for prediction

set_protein (*protein*)
Proxy method to update protein in all relevant places.

Parameters

protein: oddt.toolkit.Molecule object New default protein

train (*home_dir=None*, *sf_pickle=None*, *pdbbind_version=2016*, *ignore_json=False*)

oddt.scoring.models package

Submodules

oddt.scoring.models.classifiers module

oddt.scoring.models.classifiers.**randomforest**
alias of sklearn.ensemble.forest.RandomForestClassifier

class oddt.scoring.models.classifiers.**svm**(**args*, ***kwargs*)
Bases: oddt.scoring.models.classifiers.OddtClassifier

Methods

fit	
get_params	
predict	
predict_log_proba	
predict_proba	
score	
set_params	

fit (*descs*, *target_values*, ***kwargs*)

get_params (*deep=True*)

predict (*descs*)

predict_log_proba (*descs*)

predict_proba (*descs*)

score (*descs*, *target_values*)
Returns the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X [array-like, shape = (n_samples, n_features)] Test samples.

y [array-like, shape = (n_samples) or (n_samples, n_outputs)] True labels for X.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

score [float] Mean accuracy of self.predict(X) wrt. y.

set_params (***kwargs*)

class oddt.scoring.models.classifiers.**neuralnetwork** (**args*, ***kwargs*)
Bases: oddt.scoring.models.classifiers.OddtClassifier

Methods

fit	
get_params	
predict	
predict_log_proba	
predict_proba	
score	
set_params	

fit (*descs*, *target_values*, ***kwargs*)

get_params (*deep=True*)

predict (*descs*)

predict_log_proba (*descs*)

predict_proba (*descs*)

score (*descs*, *target_values*)
Returns the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X [array-like, shape = (n_samples, n_features)] Test samples.

y [array-like, shape = (n_samples) or (n_samples, n_outputs)] True labels for X.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

score [float] Mean accuracy of self.predict(X) wrt. y.

set_params(***kwargs*)

oddt.scoring.models.regressors module

Collection of regressors models

oddt.scoring.models.regressors.**randomforest**
alias of sklearn.ensemble.forest.RandomForestRegressor

class oddt.scoring.models.regressors.**svm**(**args*, ***kwargs*)
Bases: oddt.scoring.models.regressors.OddtRegressor

Methods

fit	
get_params	
predict	
score	
set_params	

fit(*descs*, *target_values*, ***kwargs*)

get_params(*deep=True*)

predict(*descs*)

score(*descs*, *target_values*)
Returns the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like, shape = (n_samples, n_features)] Test samples.

y [array-like, shape = (n_samples) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

set_params(***kwargs*)

oddt.scoring.models.regressors.**pls**
alias of sklearn.cross_decomposition.pls_.PLSRegression

class oddt.scoring.models.regressors.**neuralnetwork**(**args*, ***kwargs*)
Bases: oddt.scoring.models.regressors.OddtRegressor

Methods

fit	
get_params	
predict	
score	
set_params	

fit (*descs*, *target_values*, ***kwargs*)

get_params (*deep=True*)

predict (*descs*)

score (*descs*, *target_values*)

Returns the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like, shape = (n_samples, n_features)] Test samples.

y [array-like, shape = (n_samples) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

set_params (***kwargs*)

oddt.scoring.models.regressors.**mlr**

alias of sklearn.linear_model.base.LinearRegression

Module contents

Module contents

oddt.scoring.**cross_validate** (*model*, *cv_set*, *cv_target*, *n=10*, *shuffle=True*, *n_jobs=1*)

Perform cross validation of model using provided data

Parameters

model: object Model to be tested

cv_set: array-like of shape = [n_samples, n_features] Estimated target values.

cv_target: array-like of shape = [n_samples] or [n_samples, n_outputs] Estimated target values.

n: integer (default = 10) How many folds to be created from dataset

shuffle: bool (default = True) Should data be shuffled before folding.

n_jobs: integer (default = 1) How many CPUs to use during cross validation

Returns

r2: array of shape = [n] R^2 score for each of generated folds

class oddt.scoring.**ensemble_descriptor**(*descriptor_generators*)

Bases: object

Proxy class to build an ensemble of destriptors with an API as one

Parameters

models: array An array of models

Methods

build	
set_protein	

build(*mols*, **args*, ***kwargs*)

set_protein(*protein*)

class oddt.scoring.**ensemble_model**(*models*)

Bases: object

Proxy class to build an ensemble of models with an API as one

Parameters

models: array An array of models

Methods

fit	
predict	
score	

fit(*X*, *y*, **args*, ***kwargs*)

predict(*X*, **args*, ***kwargs*)

score(*X*, *y*, **args*, ***kwargs*)

class oddt.scoring.**scorer**(*model_instance*, *descriptor_generator_instance*, *score_title='score'*)

Bases: object

Scorer class is parent class for scoring functions.

Parameters

model_instance: model Medel compatible with sklearn API (fit, predict and score methods)

descriptor_generator_instance: array of descriptors Descriptor generator object

score_title: string Title of score to be used.

Methods

fit(ligands, target, *args, **kwargs)	Trains model on supplied ligands and target values
load(filename)	Loads scoring function from a pickle file.
predict(ligands, *args, **kwargs)	Predicts values (eg.
predict_ligand(ligand)	Local method to score one ligand and update it's scores.
predict_ligands(ligands)	Method to score ligands in a lazy fashion.
save(filename)	Saves scoring function to a pickle file.
score(ligands, target, *args, **kwargs)	Methods estimates the quality of prediction using model's default score (accuracy for classification or R^2 for regression)
set_protein(protein)	Proxy method to update protein in all relevant places.

fit (*ligands*, *target*, **args*, ***kwargs*)
Trains model on supplied ligands and target values

Parameters

ligands: array-like of ligands Molecules to featurize and feed into the model

target: array-like of shape = [n_samples] or [n_samples, n_outputs] Ground truth (correct) target values.

classmethod load (*filename*)
Loads scoring function from a pickle file.

Parameters

filename: string Pickle filename

Returns

sf: scorer-like object Scoring function object loaded from a pickle

predict (*ligands*, **args*, ***kwargs*)
Predicts values (eg. affinity) for supplied ligands.

Parameters

ligands: array-like of ligands Molecules to featurize and feed into the model

Returns

predicted: np.array or array of np.arrays of shape = [n_ligands] Predicted scores for ligands

predict_ligand (*ligand*)
Local method to score one ligand and update it's scores.

Parameters

ligand: oddt.toolkit.Molecule object Ligand to be scored

Returns

ligand: oddt.toolkit.Molecule object Scored ligand with updated scores

predict_ligands (*ligands*)
Method to score ligands in a lazy fashion.

Parameters

ligands: iterable of oddt.toolkit.Molecule objects Ligands to be scored

Returns

ligand: iterator of oddt.toolkit.Molecule objects Scored ligands with updated scores

save (*filename*)
Saves scoring function to a pickle file.

Parameters

filename: string Pickle filename

score (*ligands*, *target*, **args*, ***kwargs*)
Methods estimates the quality of prediction using model's default score (accuracy for classification or R^2 for regression)

Parameters

ligands: array-like of ligands Molecules to featurize and feed into the model

target: array-like of shape = [n_samples] or [n_samples, n_outputs] Ground truth (correct) target values.

Returns

s: float Quality score (accuracy or R^2) for prediction

set_protein (*protein*)
Proxy method to update protein in all relevant places.

Parameters

protein: oddt.toolkit.Molecule object New default protein

oddt.toolkits package

Subpackages

oddt.toolkits.extras package

Subpackages

oddt.toolkits.extras.rdkit package

Submodules

oddt.toolkits.extras.rdkit.fixer module

exception oddt.toolkits.extras.rdkit.fixer.**AddAtomsError**
Bases: exceptions.Exception

args

message

oddt.toolkits.extras.rdkit.fixer.**AddMissingAtoms** (*protein*, *residue*, *amap*, *template*)
Add missing atoms to protein molecule only at the residue according to template.

Parameters

protein: rdkit.Chem.rdchem.RWMol

Mol with whole protein. Note that it is modified in place.

residue: Mol with residue only

amap: list List mapping atom IDs in residue to atom IDs in whole protein (amap[i] = j means that i'th atom in residue corresponds to j'th atom in protein)

template: Residue template

Returns

——-

protein: rdkit.Chem.rdchem.RWMol Modified protein

visited_bonds: list Bonds that match the template

is_complete: bool Indicates whether all atoms in template were found in residue

oddt.toolkits.extras.rdkit.fixer.**ExtractPocketAndLigand**(*mol*, *cutoff=12.0*, *expandResidues=True*, *ligand_residue=None*, *ligand_residue_blacklist=None*, *append_residues=None*)

Function extracting a ligand (the largest HETATM residue) and the protein pocket within certain cutoff. The selection of pocket atoms can be expanded to contain whole residues. The single atom HETATM residues are attributed to pocket (metals and waters)

Parameters

mol: rdkit.Chem.rdchem.Mol

Molecule with a protein ligand complex

cutoff: float (default=12.) Distance cutoff for the pocket atoms

expandResidues: bool (default=True) Expand selection to whole residues within cutoff.

ligand_residue: string (default None) Residue name which explicitly pint to a ligand(s).

ligand_residue_blacklist: array-like, optional (default None) List of residues to ignore during ligand lookup.

append_residues: array-like, optional (default None) List of residues to append to pocket, even if they are HETATM, such as MSE, ATP, AMP, ADP, etc.

Returns

pocket: rdkit.Chem.rdchem.RWMol

Pocket constructed of protein residues/atoms around ligand

ligand: rdkit.Chem.rdchem.RWMol Largest HETATM residue contained in input molecule

oddt.toolkits.extras.rdkit.fixer.**FetchAffinityTable**(*pdbids*, *affinity_types*)

Fetch affinity data from RCSB PDB server.

Parameters

pdbids: array-like

List of PDB IDs of structres with protein-ligand complexes.

affinity_types: array-like List of types of affinity data to retrieve. Available types are: Ki, Kd, EC50, IC50, deltaG, deltaH, deltaS, Ka.

Returns

ligand_affinity: pd.DataFrame Table with protein-ligand binding affinities. Table contains following columns: structureId, ligandId, ligandFormula, ligandMolecularWeight + columns named after affinity types specified byt the user.

oddt.toolkits.extras.rdkit.fixer.**FetchStructure**(*pdbid*, *sanitize=False*, *removeHs=True*, *cache_dir=None*)

Fetch the structure in PDB format from RCSB PDB server and read it with rdkit.

Parameters

pdbid: str

PDB IDs of the structre

sanitize: bool, optional (default False) Toggles molecule sanitation

removeHs: bool, optional (default False) Indicates wheter Hs should be removed during reading

Returns

mol: Chem.rdchem.Mol Retrieved molecule

exception oddt.toolkits.extras.rdkit.fixer.**FixerError**

Bases: exceptions.Exception

args

message

oddt.toolkits.extras.rdkit.fixer.**GetAtomResidueId**(*atom*)

Return (residue number, residue name, chain id) for a given atom

oddt.toolkits.extras.rdkit.fixer.**GetResidues**(*mol*, *atom_list=None*)

Create dictrionary that maps residues to atom IDs: (res number, res name, chain id) –> [atom1 idx, atom2 idx, ...]

oddt.toolkits.extras.rdkit.fixer.**IsResidueConnected**(*mol*, *atom_ids*)

Check if residue with given atom IDs is connected to other residues in the molecule.

oddt.toolkits.extras.rdkit.fixer.**MolToTemplates**(*mol*)

Prepare set of templates for a given PDB residue.

oddt.toolkits.extras.rdkit.fixer.**PrepareComplexes**(*pdbids*, *pocket_dist_cutoff=12.0*, *affinity_types=None*, *cache_dir=None*)

Fetch structures and affinity data from RCSB PDB server and prepare ligand-pocket pairs for small molecules with known activites.

Parameters

pdbids: array-like

List of PDB IDs of structres with protein-ligand complexes.

pocket_dist_cutoff: float, optional (default 12.) Distance cutoff for the pocket atoms

affinity_types: array-like, optional (default None) List of types of affinity data to retrieve. Available types are: Ki, Kd, EC50, IC50, deltaG, deltaH, deltaS, Ka. If not specified Ki, Kd, EC50, and IC50 are used.

Returns

complexes: dict Dictionary with pocket-ligand paris, structured as follows: {'pdbid': {'ligid': (pocket_mol, ligand_mol)}. Ligands have binding affinity data stored as properties.

oddt.toolkits.extras.rdkit.fixer.**PreparePDBMol**(*mol*, *removeHs=True*, *removeHOHs=True*, *residue_whitelist=None*, *residue_blacklist=None*, *remove_incomplete=False*, *add_missing_atoms=False*, *custom_templates=None*, *replace_default_templates=False*)

Prepares protein molecule by:

- Removing Hs by hard using atomic number [default=True]
- Removes HOH [default=True]
- Assign bond orders from smiles of PDB residues (over 24k templates)
- Removes bonds to metals

Parameters

mol: rdkit.Chem.rdchem.Mol

Mol with whole protein.

removeHs: bool, optional (default True) If True, hydrogens will be forcefully removed

removeHOHs: bool, optional (default True) If True, remove waters using residue name

residue_whitelist: array-like, optional (default None) List of residues to clean. If not specified, all residues present in the structure will be used.

residue_blacklist: array-like, optional (default None) List of residues to ignore during cleaning. If not specified, all residues present in the structure will be cleaned.

remove_incomplete: bool, optional (default False) If True, remove residues that do not fully match the template

add_missing_atoms: bool (default=False) Switch to add missing atoms accordingly to template SMILES structure.

custom_templates: str or dict, optional (default None) Custom templates for residues. Can be either path to SMILES file, or dictionary mapping names to SMILES or Mol objects

replace_default_templates: bool, optional (default False) Indicates whether default default templates should be replaced by cusom ones. If False, default templates will be updated with custom ones. This argument is ignored if custom_templates is None.

Returns

new_mol: rdkit.Chem.rdchem.RWMol Modified protein

oddt.toolkits.extras.rdkit.fixer.**PreparePDBResidue**(*protein*, *residue*, *amap*, *template*)

Parameters

protein: rdkit.Chem.rdchem.RWMol

Mol with whole protein. Note that it is modified in place.

residue: Mol with residue only

amap: list List mapping atom IDs in residue to atom IDs in whole protein (amap[i] = j means that i'th atom in residue corresponds to j'th atom in protein)

template: Residue template

Returns

——-

protein: rdkit.Chem.rdchem.RWMol Modified protein

visited_bonds: list Bonds that match the template

is_complete: bool Indicates whether all atoms in template were found in residue

oddt.toolkits.extras.rdkit.fixer.**ReadTemplates**(*filename*, *resnames*)
Load templates from file for specified residues

exception oddt.toolkits.extras.rdkit.fixer.**SanitizeError**
Bases: exceptions.Exception

args

message

oddt.toolkits.extras.rdkit.fixer.**SimplifyMol**(*mol*)
Change all bonds to single and discharge/dearomatize all atoms. The molecule is modified in-place (no copy is made).

exception oddt.toolkits.extras.rdkit.fixer.**SubstructureMatchError**
Bases: exceptions.Exception

args

message

oddt.toolkits.extras.rdkit.fixer.**UFFConstrainedOptimize**(*mol*, *moving_atoms=None*, *fixed_atoms=None*, *cutoff=5.0*, *verbose=False*)
Minimize a molecule using UFF forcefield with a set of moving/fixed atoms. If both moving and fixed atoms are provided, fixed_atoms parameter will be ignored. The minimization is done in-place (without copying molecule).

Parameters

mol: rdkit.Chem.rdchem.Mol

Molecule to be minimized.

moving_atoms: array-like (default=None) Indices of freely moving atoms. If None, fixed atoms are assigned based on *fixed_atoms*. These two arguments are mutually exclusive.

fixed_atoms: array-like (default=None) Indices of fixed atoms. If None, fixed atoms are assigned based on *moving_atoms*. These two arguments are mutually exclusive.

cutoff: float (default=10.) Distance cutoff for the UFF minimization

Returns

mol: rdkit.Chem.rdchem.Mol Molecule with mimimized *moving_atoms*

Module contents

oddt.toolkits.extras.rdkit.**AtomListToSubMol**(*mol*, *amap*, *includeConformer=False*)

Parameters

mol: rdkit.Chem.rdchem.Mol

Molecule

amap: array-like List of atom indices (zero-based)

includeConformer: bool (default=True) Toogle to include atoms coordinates in sub-molecule.

Returns

submol: rdkit.Chem.rdchem.RWMol Submol determined by specified atom list

oddt.toolkits.extras.rdkit.**MolFromPDBBlock**(*molBlock*, *sanitize=True*, *removeHs=True*, *flavor=0*)

oddt.toolkits.extras.rdkit.**MolFromPDBQTBlock**(*block*, *sanitize=True*, *removeHs=True*)

Read PDBQT block to a RDKit Molecule

Parameters

block: string

Residue name which explicitly pint to a ligand(s).

sanitize: bool (default=True) Should the sanitization be performed

removeHs: bool (default=True) Should hydrogens be removed when reading molecule.

Returns

mol: rdkit.Chem.rdchem.Mol Molecule read from PDBQT

oddt.toolkits.extras.rdkit.**MolToPDBQTBlock**(*mol*, *flexible=True*, *addHs=False*, *computeCharges=False*)

Write RDKit Molecule to a PDBQT block

Parameters

mol: rdkit.Chem.rdchem.Mol

Molecule with a protein ligand complex

flexible: bool (default=True) Should the molecule encode torsions. Ligands should be flexible, proteins in turn can be rigid.

addHs: bool (default=False) The PDBQT format requires at least polar Hs on donors. By default Hs are added.

computeCharges: bool (default=False) Should the partial charges be automatically computed. If the Hs are added the charges must and will be recomputed. If there are no partial charge information, they are set to 0.0.

Returns

block: str String wit PDBQT encoded molecule

oddt.toolkits.extras.rdkit.**PDBQTAtomLines**(*mol*, *donors*, *acceptors*)
Create a list with PDBQT atom lines for each atom in molecule. Donors and acceptors are given as a list of atom indices.

oddt.toolkits.extras.rdkit.**PathFromAtomList**(*mol*, *amap*)

Module contents

Submodules

oddt.toolkits.common module

Code common to all toolkits

oddt.toolkits.common.**canonize_ring_path**(*path*)
Make a canonic path - list of consecutive atom IDXs bonded in a ring sorted in an uniform fasion.

1. Move the smallest index to position 0
2. Look for the smallest first step (delta IDX)
3. Ff -1 is smallest, inverse the path and move min IDX to position 0

Parameters

path [list of integers] A list of consecutive atom indices in a ring

Returns

canonic_path [list of integers] Sorted list of atoms

oddt.toolkits.common.**detect_secondary_structure**(*res_dict*)
Detect alpha helices and beta sheets in res_dict by phi and psi angles

oddt.toolkits.ob module

class oddt.toolkits.ob.**Atom**(*OBAtom*)
Bases: pybel.Atom

Attributes

atomicmass

atomicnum

bonds

cidx

coordidx

coords

exactmass

formalcharge

heavyvalence

heterovalence

hyb

idx DEPRECATED: RDKit is 0-based and OpenBabel is 1-based.

idx0 Note that this index is 0-based and OpenBabel's internal index in 1-based.

idx1 Note that this index is 1-based as OpenBabel's internal index.

implicitvalence

isotope

neighbors

partialcharge

residue

spin

type

valence

vector

atomicmass

atomicnum

bonds

cidx

coordidx

coords

exactmass

formalcharge

heavyvalence

heterovalence

hyb

idx
DEPRECATED: RDKit is 0-based and OpenBabel is 1-based. State which convention you desire and use *idx0* or *idx1*.

Note that this index is 1-based as OpenBabel's internal index.

idx0
Note that this index is 0-based and OpenBabel's internal index in 1-based. Changed to be compatible with RDKit

idx1
Note that this index is 1-based as OpenBabel's internal index.

implicitvalence

isotope

neighbors

partialcharge

residue

spin

type

valence

vector

class oddt.toolkits.ob.**AtomStack**(*OBMol*)
Bases: object

class oddt.toolkits.ob.**Bond**(*OBBond*)
Bases: object

Attributes

atoms

isrotor

order

atoms

isrotor

order

class oddt.toolkits.ob.**BondStack**(*OBMol*)
Bases: object

class oddt.toolkits.ob.**Fingerprint**(*fingerprint*)
Bases: pybel.Fingerprint

Attributes

bits

raw

bits

raw

class oddt.toolkits.ob.**Molecule**(*OBMol=None*, *source=None*, *protein=False*)
Bases: pybel.Molecule

Attributes

OBMol

atom_dict

atoms

bonds

canonic_order Returns np.array with canonic order of heavy atoms in the molecule

charge

charges

clone

conformers

coords

data

dim

energy

exactmass

formula

molwt

num_rotors Number of strict rotatable

protein A flag for identifing the protein molecules, for which *atom_dict* procedures may differ.

res_dict

residues

ring_dict

smiles

spin

sssr

title

unitcell

Methods

addh([only_polar])	Add hydrogens
calccharges([model])	Calculate partial charges for a molecule.
calcdesc([descnames])	Calculate descriptor values.
calcfp([fptype])	Calculate a molecular fingerprint.
convertdbonds()	Convert Dative Bonds.
draw([show, filename, update, usecoords])	Create a 2D depiction of the molecule.
localopt([forcefield, steps])	Locally optimize the coordinates.
make2D()	Generate 2D coordinates for molecule
make3D([forcefield, steps])	Generate 3D coordinates
removeh()	Remove hydrogens

clone_coords	
write	

OBMol

addh(*only_polar=False*)
Add hydrogens

atom_dict

atoms

bonds

calccharges(*model='gasteiger'*)
Calculate partial charges for a molecule. By default the Gasteiger charge model is used.

Parameters

model [str (default="gasteiger")] Method for generating partial charges. Supported models: * gasteiger * mmff94 * others supported by OpenBabel (*obabel -L charges*)

calcdesc(*descnames=[]*)
Calculate descriptor values.

Optional parameter: descnames – a list of names of descriptors

If descnames is not specified, all available descriptors are calculated. See the descs variable for a list of available descriptors.

calcfp(*fptype='FP2'*)
Calculate a molecular fingerprint.

Optional parameters:

fptype – the fingerprint type (default is "FP2"). See the fps variable for a list of of available fingerprint types.

canonic_order
Returns np.array with canonic order of heavy atoms in the molecule

charge

charges

clone

clone_coords(*source*)

conformers

convertdbonds()
Convert Dative Bonds.

coords

data

dim

draw(*show=True*, *filename=None*, *update=False*, *usecoords=False*)
Create a 2D depiction of the molecule.

Optional parameters: show – display on screen (default is True) filename – write to file (default is None) update – update the coordinates of the atoms to those

determined by the structure diagram generator (default is False)

usecoords – don't calculate 2D coordinates, just use the current coordinates (default is False)

Tkinter and Python Imaging Library are required for image display.

energy

exactmass

formula

localopt(*forcefield='mmff94'*, *steps=500*)
Locally optimize the coordinates.

Optional parameters:

forcefield – default is "mmff94". See the forcefields variable for a list of available forcefields.

steps – default is 500

If the molecule does not have any coordinates, make3D() is called before the optimization. Note that the molecule needs to have explicit hydrogens. If not, call addh().

make2D()
Generate 2D coordinates for molecule

make3D(*forcefield='mmff94'*, *steps=50*)
Generate 3D coordinates

molwt

num_rotors
Number of strict rotatable

protein
A flag for identifing the protein molecules, for which *atom_dict* procedures may differ.

removeh()
Remove hydrogens

res_dict

residues

ring_dict

smiles

spin

sssr

title

unitcell

write(*format='smi'*, *filename=None*, *overwrite=False*, *opt=None*, *size=None*)
Write the molecule to a file or return a string.

Optional parameters:

format – see the informats variable for a list of available output formats (default is "smi")

filename – default is None overwite – if the output file already exists, should it

be overwritten? (default is False)

opt – a dictionary of format specific options For format options with no parameters, specify the value as None.

If a filename is specified, the result is written to a file. Otherwise, a string is returned containing the result.

To write multiple molecules to the same file you should use the Outputfile class.

class oddt.toolkits.ob.**MoleculeData**(*obmol*)
Bases: pybel.MoleculeData

Methods

clear	
has_key	
items	
iteritems	
keys	
to_dict	
update	
values	

clear()

has_key(*key*)

items()

iteritems()

keys()

to_dict()

update(*dictionary*)

values()

class oddt.toolkits.ob.**Outputfile**(*format*, *filename*, *overwrite=False*, *opt=None*)
Bases: pybel.Outputfile

Methods

close()	Close the Outputfile to further writing.
write(molecule)	Write a molecule to the output file.

close()
Close the Outputfile to further writing.

write(*molecule*)
Write a molecule to the output file.

Required parameters: molecule

class oddt.toolkits.ob.**Residue**(*OBResidue*)
Bases: object

Represent a Pybel residue.

Required parameter: OBResidue – an Open Babel OBResidue

Attributes: atoms, idx, name.

(refer to the Open Babel library documentation for more info).

The original Open Babel atom can be accessed using the attribute: OBResidue

Attributes

atoms List of Atoms in the Residue

chain Resdiue chain ID

idx DEPRECATED: Use *idx0* instead.

idx0 Internal index (0-based) of the Residue

name Residue name

number Residue number

atoms
List of Atoms in the Residue

chain
Resdiue chain ID

idx
DEPRECATED: Use *idx0* instead.

Internal index (0-based) of the Residue

idx0
Internal index (0-based) of the Residue

name
Residue name

number
Residue number

class oddt.toolkits.ob.**ResidueStack**(*OBMol*)
Bases: object

class oddt.toolkits.ob.**Smarts**(*smartspattern*)
Bases: pybel.Smarts

Initialise with a SMARTS pattern.

Methods

findall(molecule[, unique])	Find all matches of the SMARTS pattern to a particular molecule
match(molecule)	Checks if there is any match.

findall(*molecule*, *unique=True*)
Find all matches of the SMARTS pattern to a particular molecule

match(*molecule*)
Checks if there is any match. Returns True or False

oddt.toolkits.ob.**diverse_conformers_generator**(*mol*, *n_conf=10*, *method='confab'*, *seed=None*, ***kwargs*)
Produce diverse conformers using current conformer as starting point. Returns a generator. Each conformer is a copy of original molecule object.

New in version 0.6.

Parameters

mol [oddt.toolkit.Molecule object] Molecule for which generating conformers

n_conf [int (default=10)] Targer number of conformers

method [string (default='confab')] Method for generating conformers. Supported methods: * confab * ga

seed [None or int (default=None)] Random seed

mutability [int (default=5)] The inverse of probability of mutation. By default 5, which translates to 1/5 (20%) chance of mutation. This setting only works with genetic algorithm method ("ga").

convergence [int (default=5)] The number of generations with unchanged fitness, should the algorothm converge. This setting only works with genetic algorithm method ("ga").

rmsd [float (default=0.5)] The conformers are pruned unless their RMSD is higher than this cutoff. This setting only works with Confab method ("confab").

nconf [int (default=10000)] The number of initial conformers to generate before energy pruning. This setting only works with Confab method ("confab").

energy_gap [float (default=5000.)] Energy gap from the lowest energy conformer to the highest possible. This setting only works with Confab method ("confab").

Returns

mols [list of oddt.toolkit.Molecule objects] Molecules with diverse conformers

`oddt.toolkits.ob.`**`readfile`**(*format*, *filename*, *opt=None*, *lazy=False*)

oddt.toolkits.rdk module

rdkit - A Cinfony module for accessing the RDKit from CPython

Global variables: Chem and AllChem - the underlying RDKit Python bindings informats - a dictionary of supported input formats outformats - a dictionary of supported output formats descs - a list of supported descriptors fps - a list of supported fingerprint types forcefields - a list of supported forcefields

`class` `oddt.toolkits.rdk.`**`Atom`**(*Atom*)
Bases: `object`

Represent an rdkit Atom.

Required parameters: Atom – an RDKit Atom

Attributes: atomicnum, coords, formalcharge

The original RDKit Atom can be accessed using the attribute: Atom

Attributes

atomicnum

bonds

coords

formalcharge

idx DEPRECATED: RDKit is 0-based and OpenBabel is 1-based.

idx0 Note that this index is 0-based as RDKit's

idx1 Note that this index is 1-based and RDKit's internal index in 0-based.

neighbors

partialcharge

atomicnum

bonds

coords

formalcharge

idx
DEPRECATED: RDKit is 0-based and OpenBabel is 1-based. State which convention you desire and use *idx0* or *idx1*.

Note that this index is 1-based and RDKit's internal index in 0-based. Changed to be compatible with OpenBabel

idx0
Note that this index is 0-based as RDKit's

idx1
Note that this index is 1-based and RDKit's internal index in 0-based. Changed to be compatible with OpenBabel

neighbors

partialcharge

class oddt.toolkits.rdk.**AtomStack**(*Mol*)
Bases: object

class oddt.toolkits.rdk.**Bond**(*Bond*)
Bases: object

Attributes

atoms

isrotor

order

atoms

isrotor

order

class oddt.toolkits.rdk.**BondStack**(*Mol*)
Bases: object

class oddt.toolkits.rdk.**Fingerprint**(*fingerprint*)
Bases: object

A Molecular Fingerprint.

Required parameters: fingerprint – a vector calculated by one of the fingerprint methods

Attributes: fp – the underlying fingerprint object bits – a list of bits set in the Fingerprint

Methods: The "|" operator can be used to calculate the Tanimoto coeff. For example, given two Fingerprints 'a', and 'b', the Tanimoto coefficient is given by:

tanimoto = a | b

Attributes

raw

raw

class oddt.toolkits.rdk.**Molecule**(*Mol=None*, *source=None*, *protein=False*)

Bases: object

Trap RDKit molecules which are 'None'

Attributes

Mol

atom_dict

atoms

bonds

canonic_order Returns np.array with canonic order of heavy atoms in the molecule

charges

clone

coords

data

formula

molwt

num_rotors

protein A flag for identifing the protein molecules, for which *atom_dict* procedures may differ.

res_dict

residues

ring_dict

smiles

sssr

title

Methods

addh([only_polar])	Add hydrogens.
calccharges([model])	Calculate partial charges for a molecule.
calcdesc([descnames])	Calculate descriptor values.
calcfp([fptype, opt])	Calculate a molecular fingerprint.
localopt([forcefield, steps])	Locally optimize the coordinates.
make2D()	Generate 2D coordinates for molecule
make3D([forcefield, steps])	Generate 3D coordinates.
removeh(**kwargs)	Remove hydrogens.
write([format, filename, overwrite, size])	Write the molecule to a file or return a string.

clone_coords	

Mol

addh(*only_polar=False*, ***kwargs*)
Add hydrogens.

atom_dict

atoms

bonds

calccharges(*model='gasteiger'*)
Calculate partial charges for a molecule. By default the Gasteiger charge model is used.

Parameters

model [str (default="gasteiger")] Method for generating partial charges. Supported models: * gasteiger * mmff94

calcdesc(*descnames=None*)
Calculate descriptor values.

Optional parameter: descnames – a list of names of descriptors

If descnames is not specified, all available descriptors are calculated. See the descs variable for a list of available descriptors.

calcfp(*fptype='rdkit'*, *opt=None*)
Calculate a molecular fingerprint.

Optional parameters:

fptype – the fingerprint type (default is "rdkit"). See the fps variable for a list of of available fingerprint types.

opt – a dictionary of options for fingerprints. Currently only used for radius and bitInfo in Morgan fingerprints.

canonic_order
Returns np.array with canonic order of heavy atoms in the molecule

charges

clone

clone_coords(*source*)

coords

data

formula

localopt(*forcefield='uff'*, *steps=500*)
Locally optimize the coordinates.

Optional parameters:

forcefield – default is "uff". See the forcefields variable for a list of available forcefields.

steps – default is 500

If the molecule does not have any coordinates, make3D() is called before the optimization.

make2D()
Generate 2D coordinates for molecule

make3D(*forcefield='mmff94'*, *steps=50*)
Generate 3D coordinates.

Optional parameters:

forcefield – default is "uff". See the forcefields variable for a list of available forcefields.

steps – default is 50

Once coordinates are generated, a quick local optimization is carried out with 50 steps and the UFF forcefield. Call localopt() if you want to improve the coordinates further.

molwt

num_rotors

protein
A flag for identifing the protein molecules, for which *atom_dict* procedures may differ.

removeh(***kwargs*)
Remove hydrogens.

res_dict

residues

ring_dict

smiles

sssr

title

write(*format='smi'*, *filename=None*, *overwrite=False*, *size=None*, ***kwargs*)
Write the molecule to a file or return a string.

Optional parameters:

format – see the informats variable for a list of available output formats (default is "smi")

filename – default is None overwite – if the output file already exists, should it

be overwritten? (default is False)

If a filename is specified, the result is written to a file. Otherwise, a string is returned containing the result.

To write multiple molecules to the same file you should use the Outputfile class.

class oddt.toolkits.rdk.**MoleculeData**(*Mol*)
Bases: object

Store molecule data in a dictionary-type object

Required parameters: Mol – an RDKit Mol

Methods and accessor methods are like those of a dictionary except that the data is retrieved on-the-fly from the underlying Mol.

Example: >>> mol = next(readfile("sdf", 'head.sdf')) >>> data = mol.data >>> print(data) {'Comment': 'CORINA 2.61 0041 25.10.2001', 'NSC': '1'} >>> print(len(data), data.keys(), data.has_key("NSC")) 2 ['Comment', 'NSC'] True >>> print(data['Comment']) CORINA 2.61 0041 25.10.2001 >>> data['Comment'] = 'This is a new comment' >>> for k,v in data.items(): ... print(k, "–>", v) Comment –> This is a new comment NSC –> 1 >>> del data['NSC'] >>> print(len(data), data.keys(), data.has_key("NSC")) 1 ['Comment'] False

Methods

clear	
has_key	
items	
iteritems	
keys	
to_dict	
update	
values	

clear()

has_key(*key*)

items()

iteritems()

keys()

to_dict()

update(*dictionary*)

values()

class oddt.toolkits.rdk.**Outputfile**(*format*, *filename*, *overwrite=False*)
Bases: object

Represent a file to which *output* is to be sent.

Required parameters:

format - see the outformats variable for a list of available output formats

filename

Optional parameters:

overwite – if the output file already exists, should it be overwritten? (default is False)

Methods: write(molecule) close()

Methods

close()	Close the Outputfile to further writing.
write(molecule)	Write a molecule to the output file.

close()
Close the Outputfile to further writing.

write(*molecule*)
Write a molecule to the output file.

Required parameters: molecule

class oddt.toolkits.rdk.**Residue**(*ParentMol*, *atom_path*, *idx=0*)
Bases: object

Represent a RDKit residue.

Required parameter: ParentMol – Parent molecule (Mol) object path – atoms path of a residue

Attributes: atoms, idx, name.

(refer to the Open Babel library documentation for more info).

The Mol object constucted of residues' atoms can be accessed using the attribute: Residue

Attributes

atoms List of Atoms in the Residue

chain Resdiue chain ID

idx DEPRECATED: Use *idx0* instead.

idx0 Internal index (0-based) of the Residue

name Residue name

number Residue number

atoms
List of Atoms in the Residue

chain
Resdiue chain ID

idx
DEPRECATED: Use *idx0* instead.

Internal index (0-based) of the Residue

idx0
Internal index (0-based) of the Residue

name
Residue name

number
Residue number

class oddt.toolkits.rdk.**ResidueStack**(*Mol*, *paths*)
Bases: object

class oddt.toolkits.rdk.**Smarts**(*smartspattern*)
Bases: object

Initialise with a SMARTS pattern.

Methods

findall(molecule[, unique])	Find all matches of the SMARTS pattern to a particular molecule.
match(molecule)	Find all matches of the SMARTS pattern to a particular molecule.

findall(*molecule*, *unique=True*)
Find all matches of the SMARTS pattern to a particular molecule.

Required parameters: molecule

match (*molecule*)
Find all matches of the SMARTS pattern to a particular molecule.

Required parameters: molecule

oddt.toolkits.rdk.**base_feature_factory = <rdkit.Chem.rdMolChemicalFeatures.MolChemicalFeatu**
Global feature factory based on BaseFeatures.fdef

oddt.toolkits.rdk.**descs = ['fr_C_O_noCOO', 'PEOE_VSA3', 'Chi4v', 'fr_Ar_COO', 'fr_SH', 'Chi**
A list of supported descriptors

oddt.toolkits.rdk.**diverse_conformers_generator**(*mol*, *n_conf=10*, *method='etkdg'*, *seed=None*, *rmsd=0.5*)
Produce diverse conformers using current conformer as starting point. Each conformer is a copy of original molecule object.

New in version 0.6.

Parameters

mol [oddt.toolkit.Molecule object] Molecule for which generating conformers

n_conf [int (default=10)] Targer number of conformers

method [string (default='etkdg')] Method for generating conformers. Supported methods: "etkdg", "etdg", "kdg", "dg".

seed [None or int (default=None)] Random seed

rmsd [float (default=0.5)] The minimum RMSD that separates conformers to be ratained (otherwise, they will be pruned).

Returns

mols [list of oddt.toolkit.Molecule objects] Molecules with diverse conformers

oddt.toolkits.rdk.**forcefields = ['mmff94', 'uff']**
A list of supported forcefields

oddt.toolkits.rdk.**fps = ['rdkit', 'layered', 'maccs', 'atompairs', 'torsions', 'morgan']**
A list of supported fingerprint types

oddt.toolkits.rdk.**informats = {'inchi': 'InChI', 'mol': 'MDL MOL file', 'mol2': 'Tripos**
A dictionary of supported input formats

oddt.toolkits.rdk.**outformats = {'can': 'Canonical SMILES', 'inchi': 'InChI', 'inchikey':**
A dictionary of supported output formats

oddt.toolkits.rdk.**readfile**(*format*, *filename*, *lazy=False*, *opt=None*, ***kwargs*)
Iterate over the molecules in a file.

Required parameters:

format - see the informats variable for a list of available input formats

filename

You can access the first molecule in a file using the next() method of the iterator:

mol = next(readfile("smi", "myfile.smi"))

You can make a list of the molecules in a file using: mols = list(readfile("smi", "myfile.smi"))

You can iterate over the molecules in a file as shown in the following code snippet: >>> atomtotal = 0 >>> for mol in readfile("sdf", "head.sdf"): ... atomtotal += len(mol.atoms) ... >>> print(atomtotal) 43

oddt.toolkits.rdk.**readstring**(*format*, *string*, ***kwargs*)

Read in a molecule from a string.

Required parameters:

format - see the informats variable for a list of available input formats

string

Example: >>> input = "C1=CC=CS1" >>> mymol = readstring("smi", input) >>> len(mymol.atoms) 5

Module contents

5.1.2 Submodules

5.1.3 oddt.datasets module

Datasets wrapped in convenient models

class oddt.datasets.**CASF**(*home*)

Load CASF dataset as described in Li, Y. et al. Comparative Assessment of Scoring Functions on an Updated Benchmark: 2. Evaluation Methods and General Results. J. Chem. Inf. Model. 54, 1717-1736. (2014) http://dx.doi.org/10.1021/ci500081m

Parameters

home: string Path to CASF dataset main directory

Methods

precomputed_score([scoring_function])	Load precomputed results of scoring power test for various scoring functions.
precomputed_screening([scoring_function, ...])	Load precomputed results of screening power test for various scoring functions

precomputed_score(*scoring_function=None*)

Load precomputed results of scoring power test for various scoring functions.

Parameters

scoring_function: string (default=None) Name of the scoring function to get results If None, all results are returned.

precomputed_screening(*scoring_function=None*, *cluster_id=None*)

Load precomputed results of screening power test for various scoring functions

Parameters

scoring_function: string (default=None) Name of the scoring function to get results If None, all results are returned

cluster_id: int (default=None) Number of the protein cluster to get results If None, all results are returned

class oddt.datasets.**dude**(*home*)
Bases: object

A wrapper for DUD-E (A Database of Useful Decoys: Enhanced) http://dude.docking.org/

Parameters

home [str] Path to files from dud-e

class oddt.datasets.**pdbbind**(*home*, *version=None*, *default_set=None*, *opt=None*)
Bases: object

Attributes

activities

ids

activities

ids

5.1.4 oddt.fingerprints module

Module checks interactions between two molecules and creates interacion fingerprints.

oddt.fingerprints.**InteractionFingerprint**(*ligand*, *protein*, *strict=True*)
Interaction fingerprint accomplished by converting the molecular interaction of ligand-protein into bit array according to the residue of choice and the interaction. For every residue (One row = one residue) there are eight bits which represent eight type of interactions:

- (Column 0) hydrophobic contacts
- (Column 1) aromatic face to face
- (Column 2) aromatic edge to face
- (Column 3) hydrogen bond (protein as hydrogen bond donor)
- (Column 4) hydrogen bond (protein as hydrogen bond acceptor)
- (Column 5) salt bridges (protein positively charged)
- (Column 6) salt bridges (protein negatively charged)
- (Column 7) salt bridges (ionic bond with metal ion)

Parameters

ligand, protein [oddt.toolkit.Molecule object] Molecules, which are analysed in order to find interactions.

strict [bool (deafult = True)] If False, do not include condition, which informs whether atoms form 'strict' H-bond (pass all angular cutoffs).

Returns

InteractionFingerprint [numpy array] Vector of calculated IFP (size = no residues * 8 type of interaction)

oddt.fingerprints.**SimpleInteractionFingerprint**(*ligand*, *protein*, *strict=True*)
Based on http://dx.doi.org/10.1016/j.csbj.2014.05.004. Every IFP consists of 8 bits per amino acid (One row = one amino acid) and present eight type of interaction:

- (Column 0) hydrophobic contacts

- (Column 1) aromatic face to face
- (Column 2) aromatic edge to face
- (Column 3) hydrogen bond (protein as hydrogen bond donor)
- (Column 4) hydrogen bond (protein as hydrogen bond acceptor)
- (Column 5) salt bridges (protein positively charged)
- (Column 6) salt bridges (protein negatively charged)
- (Column 7) salt bridges (ionic bond with metal ion)

Returns matrix, which is sorted according to this pattern : 'ALA', 'ARG', 'ASN', 'ASP', 'CYS', 'GLN', 'GLU', 'GLY', 'HIS', 'ILE', 'LEU', 'LYS', 'MET', 'PHE', 'PRO', 'SER', 'THR', 'TRP', 'TYR', 'VAL', ''. The '' means cofactor. Index of amino acid in pattern coresponds to row in returned matrix.

Parameters

ligand, protein [oddt.toolkit.Molecule object] Molecules, which are analysed in order to find interactions.

strict [bool (deafult = True)] If False, do not include condition, which informs whether atoms form 'strict' H-bond (pass all angular cutoffs).

Returns

InteractionFingerprint [numpy array] Vector of calculated IFP (size = 168)

oddt.fingerprints.**SPLIF**(*ligand*, *protein*, *depth=1*, *size=4096*, *distance_cutoff=4.5*)

Calculates structural protein-ligand interaction fingerprint (SPLIF), based on http://pubs.acs.org/doi/abs/10.1021/ci500319f.

Parameters

ligand, protein [oddt.toolkit.Molecule object] Molecules, which are analysed in order to find interactions.

depth [int (deafult = 1)] The depth of the fingerprint, i.e. the number of bonds in Morgan algorithm. Note: For ECFP2: depth = 1, ECFP4: depth = 2, etc.

size: int (default = 4096) SPLIF is folded to given size.

distance_cutoff: float (default=4.5) Cutoff distance for close contacts.

Returns

SPLIF [numpy array] Calculated SPLIF.shape = (no. of atoms,). Every row consists of three elements:

row[0] = index of hashed atoms row[1].shape = (7, 3) -> ligand's atom coords and 6 his neigbor's row[2].shape = (7, 3) -> protein's atom coords and 6 his neigbor's

oddt.fingerprints.**similarity_SPLIF**(*reference*, *query*, *rmsd_cutoff=1.0*)

Calculates similarity between structural interaction fingerprints, based on doi:http://pubs.acs.org/doi/abs/10.1021/ci500319f.

Parameters

reference, query: numpy.array SPLIFs, which are compared in order to determine similarity.

rmsd_cutoff [int (default = 1)] Specific treshold for which, bits are considered as fully matching.

Returns

SimilarityScore [float] Similarity between given fingerprints.

oddt.fingerprints.**ECFP**(*mol*, *depth=2*, *size=4096*, *count_bits=True*, *sparse=True*, *use_pharm_features=False*)

Extended connectivity fingerprints (ECFP) with an option to include atom features (FCPF). Depth of a fingerprint is counted as bond-steps, thus the depth for ECFP2 = 1, ECPF4 = 2, ECFP6 = 3, etc.

Reference: Rogers D, Hahn M. Extended-connectivity fingerprints. J Chem Inf Model. 2010;50: 742-754. http://dx.doi.org/10.1021/ci100050t

Parameters

mol [oddt.toolkit.Molecule object] Input molecule for the FP calculations

depth [int (deafult = 2)] The depth of the fingerprint, i.e. the number of bonds in Morgan algorithm. Note: For ECFP2: depth = 1, ECFP4: depth = 2, etc.

size [int (default = 4096)] Final size of fingerprint to which it is folded.

count_bits [bool (default = True)] Should the bits be counted or unique. In dense representation it translates to integer array (count_bits=True) or boolean array if False.

sparse [bool (default=True)] Should fingerprints be dense (contain all bits) or sparse (just the on bits).

use_pharm_features [bool (default=False)] Switch to use pharmacophoric features as atom representation instead of explicit atomic numbers etc.

Returns

fingerprint [numpy array] Calsulated FP of fixed size (dense) or on bits indices (sparse). Dtype is either integer or boolean.

oddt.fingerprints.**PLEC**(*ligand*, *protein*, *depth_ligand=2*, *depth_protein=4*, *distance_cutoff=4.5*, *size=16384*, *count_bits=True*, *sparse=True*, *ignore_hoh=True*)

Protein ligand extended connectivity fingerprint. For every pair of atoms in contact, compute ECFP and then hash every single, corresponding depth.

Parameters

ligand, protein [oddt.toolkit.Molecule object] Molecules, which are analysed in order to find interactions.

depth_ligand, depth_protein [int (deafult = (2, 4))] The depth of the fingerprint, i.e. the number of bonds in Morgan algorithm. Note: For ECFP2: depth = 1, ECFP4: depth = 2, etc.

size: int (default = 16384) SPLIF is folded to given size.

distance_cutoff: float (default=4.5) Cutoff distance for close contacts.

sparse [bool (default = True)] Should fingerprints be dense (contain all bits) or sparse (just the on bits).

count_bits [bool (default = True)] Should the bits be counted or unique. In dense representation it translates to integer array (count_bits=True) or boolean array if False.

ignore_hoh [bool (default = True)] Should the water molecules be ignored. This is based on the name of the residue ('HOH').

Returns

PLEC [numpy array] fp (size = atoms in contacts * max(depth_protein, depth_ligand))

oddt.fingerprints.**dice**(*a*, *b*, *sparse=False*)

Calculates the Dice coefficient, the ratio of the bits in common to the arithmetic mean of the number of 'on' bits in the two fingerprints. Supports integer and boolean fingerprints.

Parameters

a, b [numpy array] Interaction fingerprints, which are compared in order to determine similarity.

sparse [bool (default=False)] Type of FPs to use. Defaults to dense form.

Returns

score [float] Similarity between a, b.

oddt.fingerprints.**tanimoto**(*a*, *b*, *sparse=False*)

Tanimoto coefficient, supports boolean fingerprints. Integer fingerprints are casted to boolean.

Parameters

a, b [numpy array] Interaction fingerprints, which are compared in order to determine similarity.

sparse [bool (default=False)] Type of FPs to use. Defaults to dense form.

Returns

score [float] Similarity between a, b.

5.1.5 oddt.interactions module

Module calculates interactions between two molecules (proein-protein, protein-ligand, small-small). Currently following interacions are implemented:

- hydrogen bonds
- halogen bonds
- pi stacking (parallel and perpendicular)
- salt bridges
- hydrophobic contacts
- pi-cation
- metal coordination
- pi-metal

oddt.interactions.**close_contacts**(*x*, *y*, *cutoff*, *x_column='coords'*, *y_column='coords'*, *cutoff_low=0.0*)

Returns pairs of atoms which are within close contac distance cutoff. The cutoff is semi-inclusive, i.e (cutoff_low, cutoff].

Parameters

x, y [atom_dict-type numpy array] Atom dictionaries generated by oddt.toolkit.Molecule objects.

cutoff [float] Cutoff distance for close contacts

x_column, ycolumn [string, (default='coords')] Column containing coordinates of atoms (or pseudo-atoms, i.e. ring centroids)

cutoff_low [float (default=0.)] Lower bound of contacts to find (exclusive). Zero by default. .. versionadded:: 0.6

Returns

x_, y_ [atom_dict-type numpy array] Aligned pairs of atoms in close contact for further processing.

oddt.interactions.**hbond_acceptor_donor**(*mol1*, *mol2*, *cutoff=3.5*, *base_angle=120*, *tolerance=30*)

Returns pairs of acceptor-donor atoms, which meet H-bond criteria

Parameters

mol1, mol2 [oddt.toolkit.Molecule object] Molecules to compute H-bond acceptor and H-bond donor pairs

cutoff [float, (default=3.5)] Distance cutoff for A-D pairs

base_angle [int, (default=120)] Base angle determining allowed direction of hydrogen bond formation, which is devided by the number of neighbors of acceptor atom to establish final directional angle

tolerance [int, (default=30)] Range (+/- tolerance) from perfect direction (base_angle/n_neighbors) in which H-bonds are considered as strict.

Returns

a, d [atom_dict-type numpy array] Aligned arrays of atoms forming H-bond, firstly acceptors, secondly donors.

strict [numpy array, dtype=bool] Boolean array align with atom pairs, informing whether atoms form 'strict' H-bond (pass all angular cutoffs). If false, only distance cutoff is met, therefore the bond is 'crude'.

oddt.interactions.**hbonds**(*mol1*, *mol2*, **args*, ***kwargs*)

Calculates H-bonds between molecules

Parameters

mol1, mol2 [oddt.toolkit.Molecule object] Molecules to compute H-bond acceptor and H-bond donor pairs

cutoff [float, (default=3.5)] Distance cutoff for A-D pairs

base_angle [int, (default=120)] Base angle determining allowed direction of hydrogen bond formation, which is devided by the number of neighbors of acceptor atom to establish final directional angle

tolerance [int, (default=30)] Range (+/- tolerance) from perfect direction (base_angle/n_neighbors) in which H-bonds are considered as strict.

Returns

mol1_atoms, mol2_atoms [atom_dict-type numpy array] Aligned arrays of atoms forming H-bond

strict [numpy array, dtype=bool] Boolean array align with atom pairs, informing whether atoms form 'strict' H-bond (pass all angular cutoffs). If false, only distance cutoff is met, therefore the bond is 'crude'.

oddt.interactions.**halogenbond_acceptor_halogen**(*mol1*, *mol2*, *base_angle_acceptor=120*, *base_angle_halogen=180*, *tolerance=30*, *cutoff=4*)

Returns pairs of acceptor-halogen atoms, which meet halogen bond criteria

Parameters

mol1, mol2 [oddt.toolkit.Molecule object] Molecules to compute halogen bond acceptor and halogen pairs

cutoff [float, (default=4)] Distance cutoff for A-H pairs

base_angle_acceptor [int, (default=120)] Base angle determining allowed direction of halogen bond formation, which is devided by the number of neighbors of acceptor atom to establish final directional angle

base_angle_halogen [int (default=180)] Ideal base angle between halogen bond and halogen-neighbor bond

tolerance [int, (default=30)] Range (+/- tolerance) from perfect direction (base_angle/n_neighbors) in which halogen bonds are considered as strict.

Returns

a, h [atom_dict-type numpy array] Aligned arrays of atoms forming halogen bond, firstly acceptors, secondly halogens

strict [numpy array, dtype=bool] Boolean array align with atom pairs, informing whether atoms form 'strict' halogen bond (pass all angular cutoffs). If false, only distance cutoff is met, therefore the bond is 'crude'.

oddt.interactions.**halogenbonds**(*mol1*, *mol2*, ***kwargs*)

Calculates halogen bonds between molecules

Parameters

mol1, mol2 [oddt.toolkit.Molecule object] Molecules to compute halogen bond acceptor and halogen pairs

cutoff [float, (default=4)] Distance cutoff for A-H pairs

base_angle_acceptor [int, (default=120)] Base angle determining allowed direction of halogen bond formation, which is devided by the number of neighbors of acceptor atom to establish final directional angle

base_angle_halogen [int (default=180)] Ideal base angle between halogen bond and halogen-neighbor bond

tolerance [int, (default=30)] Range (+/- tolerance) from perfect direction (base_angle/n_neighbors) in which halogen bonds are considered as strict.

Returns

mol1_atoms, mol2_atoms [atom_dict-type numpy array] Aligned arrays of atoms forming halogen bond

strict [numpy array, dtype=bool] Boolean array align with atom pairs, informing whether atoms form 'strict' halogen bond (pass all angular cutoffs). If false, only distance cutoff is met, therefore the bond is 'crude'.

oddt.interactions.**pi_stacking**(*mol1*, *mol2*, *cutoff=5*, *tolerance=30*)

Returns pairs of rings, which meet pi stacking criteria

Parameters

mol1, mol2 [oddt.toolkit.Molecule object] Molecules to compute ring pairs

cutoff [float, (default=5)] Distance cutoff for Pi-stacking pairs

tolerance [int, (default=30)] Range (+/- tolerance) from perfect direction (parallel or perpendicular) in which pi-stackings are considered as strict.

Returns

r1, r2 [ring_dict-type numpy array] Aligned arrays of rings forming pi-stacking

strict_parallel [numpy array, dtype=bool] Boolean array align with ring pairs, informing whether rings form 'strict' parallel pi-stacking. If false, only distance cutoff is met, therefore the stacking is 'crude'.

strict_perpendicular [numpy array, dtype=bool] Boolean array align with ring pairs, informing whether rings form 'strict' perpendicular pi-stacking (T-shaped, T-face, etc.). If false, only distance cutoff is met, therefore the stacking is 'crude'.

oddt.interactions.**salt_bridge_plus_minus**(*mol1*, *mol2*, *cutoff=4*)

Returns pairs of plus-mins atoms, which meet salt bridge criteria

Parameters

mol1, mol2 [oddt.toolkit.Molecule object] Molecules to compute plus and minus pairs

cutoff [float, (default=4)] Distance cutoff for A-H pairs

Returns

plus, minus [atom_dict-type numpy array] Aligned arrays of atoms forming salt bridge, firstly plus, secondly minus

oddt.interactions.**salt_bridges**(*mol1*, *mol2*, **args*, ***kwargs*)

Calculates salt bridges between molecules

Parameters

mol1, mol2 [oddt.toolkit.Molecule object] Molecules to compute plus and minus pairs

cutoff [float, (default=4)] Distance cutoff for plus-minus pairs

Returns

mol1_atoms, mol2_atoms [atom_dict-type numpy array] Aligned arrays of atoms forming salt bridges

oddt.interactions.**hydrophobic_contacts**(*mol1*, *mol2*, *cutoff=4*)

Calculates hydrophobic contacts between molecules

Parameters

mol1, mol2 [oddt.toolkit.Molecule object] Molecules to compute hydrophobe pairs

cutoff [float, (default=4)] Distance cutoff for hydrophobe pairs

Returns

mol1_atoms, mol2_atoms [atom_dict-type numpy array] Aligned arrays of atoms forming hydrophobic contacts

oddt.interactions.**pi_cation**(*mol1*, *mol2*, *cutoff=5*, *tolerance=30*)

Returns pairs of ring-cation atoms, which meet pi-cation criteria

Parameters

mol1, mol2 [oddt.toolkit.Molecule object] Molecules to compute ring-cation pairs

cutoff [float, (default=5)] Distance cutoff for Pi-cation pairs

tolerance [int, (default=30)] Range (+/- tolerance) from perfect direction (perpendicular) in which pi-cation are considered as strict.

Returns

r1 [ring_dict-type numpy array] Aligned rings forming pi-stacking

plus2 [atom_dict-type numpy array] Aligned cations forming pi-cation

strict_parallel [numpy array, dtype=bool] Boolean array align with ring-cation pairs, informing whether they form 'strict' pi-cation. If false, only distance cutoff is met, therefore the interaction is 'crude'.

oddt.interactions.**acceptor_metal**(*mol1*, *mol2*, *base_angle=120*, *tolerance=30*, *cutoff=4*)

Returns pairs of acceptor-metal atoms, which meet metal coordination criteria Note: This function is directional (mol1 holds acceptors, mol2 holds metals)

Parameters

mol1, mol2 [oddt.toolkit.Molecule object] Molecules to compute acceptor and metal pairs

cutoff [float, (default=4)] Distance cutoff for A-M pairs

base_angle [int, (default=120)] Base angle determining allowed direction of metal coordination, which is devided by the number of neighbors of acceptor atom to establish final directional angle

tolerance [int, (default=30)] Range (+/- tolerance) from perfect direction (base_angle/n_neighbors) in metal coordination are considered as strict.

Returns

a, d [atom_dict-type numpy array] Aligned arrays of atoms forming metal coordination, firstly acceptors, secondly metals.

strict [numpy array, dtype=bool] Boolean array align with atom pairs, informing whether atoms form 'strict' metal coordination (pass all angular cutoffs). If false, only distance cutoff is met, therefore the interaction is 'crude'.

oddt.interactions.**pi_metal**(*mol1*, *mol2*, *cutoff=5*, *tolerance=30*)

Returns pairs of ring-metal atoms, which meet pi-metal criteria

Parameters

mol1, mol2 [oddt.toolkit.Molecule object] Molecules to compute ring-metal pairs

cutoff [float, (default=5)] Distance cutoff for Pi-metal pairs

tolerance [int, (default=30)] Range (+/- tolerance) from perfect direction (perpendicular) in which pi-metal are considered as strict.

Returns

r1 [ring_dict-type numpy array] Aligned rings forming pi-metal

m [atom_dict-type numpy array] Aligned metals forming pi-metal

strict_parallel [numpy array, dtype=bool] Boolean array align with ring-metal pairs, informing whether they form 'strict' pi-metal. If false, only distance cutoff is met, therefore the interaction is 'crude'.

5.1.6 oddt.metrics module

Metrics for estimating performance of drug discovery methods implemented in ODDT

oddt.metrics.**roc**(*y_true*, *y_score*, *pos_label=None*, *sample_weight=None*, *drop_intermediate=True*)

Compute Receiver operating characteristic (ROC)

Note: this implementation is restricted to the binary classification task.

Read more in the User Guide.

Parameters

y_true [array, shape = [n_samples]] True binary labels in range {0, 1} or {-1, 1}. If labels are not binary, pos_label should be explicitly given.

y_score [array, shape = [n_samples]] Target scores, can either be probability estimates of the positive class, confidence values, or non-thresholded measure of decisions (as returned by "decision_function" on some classifiers).

pos_label [int or str, default=None] Label considered as positive and others are considered negative.

sample_weight [array-like of shape = [n_samples], optional] Sample weights.

drop_intermediate [boolean, optional (default=True)] Whether to drop some suboptimal thresholds which would not appear on a plotted ROC curve. This is useful in order to create lighter ROC curves.

New in version 0.17: parameter *drop_intermediate*.

Returns

fpr [array, shape = [>2]] Increasing false positive rates such that element i is the false positive rate of predictions with score >= thresholds[i].

tpr [array, shape = [>2]] Increasing true positive rates such that element i is the true positive rate of predictions with score >= thresholds[i].

thresholds [array, shape = [n_thresholds]] Decreasing thresholds on the decision function used to compute fpr and tpr. *thresholds[0]* represents no instances being predicted and is arbitrarily set to *max(y_score) + 1*.

See also:

roc_auc_score Compute the area under the ROC curve

Notes

Since the thresholds are sorted from low to high values, they are reversed upon returning them to ensure they correspond to both `fpr` and `tpr`, which are sorted in reversed order during their calculation.

References

[1]

Examples

```
>>> import numpy as np
>>> from sklearn import metrics
>>> y = np.array([1, 1, 2, 2])
>>> scores = np.array([0.1, 0.4, 0.35, 0.8])
>>> fpr, tpr, thresholds = metrics.roc_curve(y, scores, pos_label=2)
>>> fpr
array([ 0. ,  0.5,  0.5,  1. ])
>>> tpr
array([ 0.5,  0.5,  1. ,  1. ])
>>> thresholds
array([ 0.8 ,  0.4 ,  0.35,  0.1 ])
```

oddt.metrics.**auc**(*x*, *y*, *reorder=False*)

Compute Area Under the Curve (AUC) using the trapezoidal rule

This is a general function, given points on a curve. For computing the area under the ROC-curve, see `roc_auc_score()`. For an alternative way to summarize a precision-recall curve, see `average_precision_score()`.

Parameters

x [array, shape = [n]] x coordinates.

y [array, shape = [n]] y coordinates.

reorder [boolean, optional (default=False)] If True, assume that the curve is ascending in the case of ties, as for an ROC curve. If the curve is non-ascending, the result will be wrong.

Returns

auc [float]

See also:

roc_auc_score Compute the area under the ROC curve

average_precision_score Compute average precision from prediction scores

precision_recall_curve Compute precision-recall pairs for different probability thresholds

Examples

```
>>> import numpy as np
>>> from sklearn import metrics
>>> y = np.array([1, 1, 2, 2])
>>> pred = np.array([0.1, 0.4, 0.35, 0.8])
>>> fpr, tpr, thresholds = metrics.roc_curve(y, pred, pos_label=2)
>>> metrics.auc(fpr, tpr)
0.75
```

oddt.metrics.**roc_auc**(*y_true*, *y_score*, *pos_label=None*, *ascending_score=True*)

Computes ROC AUC score

Parameters

y_true [array, shape=[n_samples]] True binary labels, in range {0,1} or {-1,1}. If positive label is different than 1, it must be explicitly defined.

y_score [array, shape=[n_samples]] Scores for tested series of samples

pos_label: int Positive label of samples (if other than 1)

ascending_score: bool (default=True) Indicates if your score is ascendig. Ascending score icreases with deacreasing activity. In other words it ascends on ranking list (where actives are on top).

Returns

roc_auc [float] ROC AUC in range 0:1

oddt.metrics.**roc_log_auc**(*y_true*, *y_score*, *pos_label=None*, *ascending_score=True*, *log_min=0.001*, *log_max=1.0*)

Computes area under semi-log ROC.

Parameters

y_true [array, shape=[n_samples]] True binary labels, in range {0,1} or {-1,1}. If positive label is different than 1, it must be explicitly defined.

y_score [array, shape=[n_samples]] Scores for tested series of samples

pos_label: int Positive label of samples (if other than 1)

ascending_score: bool (default=True) Indicates if your score is ascendig. Ascending score icreases with deacreasing activity. In other words it ascends on ranking list (where actives are on top).

log_min [float (default=0.001)] Minimum value for estimating AUC. Lower values will be clipped for numerical stability.

log_max [float (default=1.)] Maximum value for estimating AUC. Higher values will be ignored.

Returns

auc [float] semi-log ROC AUC

oddt.metrics.**enrichment_factor**(*y_true*, *y_score*, *percentage=1*, *pos_label=None*, *kind='fold'*)

Computes enrichment factor for given percentage, i.e. EF_1% is enrichment factor for first percent of given samples. This function assumes that results are already sorted and samples with best predictions are first.

Parameters

y_true [array, shape=[n_samples]] True binary labels, in range {0,1} or {-1,1}. If positive label is different than 1, it must be explicitly defined.

y_score [array, shape=[n_samples]] Scores for tested series of samples

percentage [int or float] The percentage for which EF is being calculated

pos_label: int Positive label of samples (if other than 1)

kind: 'fold' or 'percentage' (default='fold') Two kinds of enrichment factor: fold and percentage. Fold shows the increase over random distribution (1 is random, the higher EF the better enrichment). Percentage returns the fraction of positive labels within the top x% of dataset.

Returns

ef [float] Enrichment Factor for given percenage in range 0:1

oddt.metrics.**random_roc_log_auc**(*log_min=0.001*, *log_max=1.0*)

Computes area under semi-log ROC for random distribution.

Parameters

log_min [float (default=0.001)] Minimum logarithm value for estimating AUC

log_max [float (default=1.)] Maximum logarithm value for estimating AUC.

Returns

auc [float] semi-log ROC AUC for random distribution

oddt.metrics.**rmse**(*y_true*, *y_pred*)

Compute Root Mean Squared Error (RMSE)

Parameters

y_true [array-like of shape = [n_samples] or [n_samples, n_outputs]] Ground truth (correct) target values.

y_pred [array-like of shape = [n_samples] or [n_samples, n_outputs]] Estimated target values.

Returns

rmse [float] A positive floating point value (the best value is 0.0).

oddt.metrics.**rie**(*y_true*, *y_score*, *alpha=20*, *pos_label=None*)

Computes Robust Initial Enhancement *[1]*. This function assumes that results are already sorted and samples with best predictions are first.

Parameters

y_true [array, shape=[n_samples]] True binary labels, in range {0,1} or {-1,1}. If positive label is different than 1, it must be explicitly defined.

y_score [array, shape=[n_samples]] Scores for tested series of samples

alpha: float Alpha. 1/Alpha should be proportional to the percentage in EF.

pos_label: int Positive label of samples (if other than 1)

Returns

rie_score [float] Robust Initial Enhancement

References

[1]

oddt.metrics.**bedroc**(*y_true*, *y_score*, *alpha=20.0*, *pos_label=None*)

Computes Boltzmann-Enhanced Discrimination of Receiver Operating Characteristic *[1]*. This function assumes that results are already sorted and samples with best predictions are first.

Parameters

y_true [array, shape=[n_samples]] True binary labels, in range {0,1} or {-1,1}. If positive label is different than 1, it must be explicitly defined.

y_score [array, shape=[n_samples]] Scores for tested series of samples

alpha: float Alpha. 1/Alpha should be proportional to the percentage in EF.

pos_label: int Positive label of samples (if other than 1)

Returns

bedroc_score [float] Boltzmann-Enhanced Discrimination of Receiver Operating Characteristic

References

[1]

5.1.7 oddt.pandas module

Pandas extension for chemical analysis

class oddt.pandas.**ChemDataFrame**(*data=None*, *index=None*, *columns=None*, *dtype=None*, *copy=False*)

Bases: pandas.core.frame.DataFrame

Chemical DataFrame object, which contains molecules column of *oddt.toolkit.Molecule* objects. Rich display of moleucles (2D) is available in iPython Notebook. Additional *to_sdf* and *to_mol2* methods make writing to molecular formats easy.

New in version 0.3.

Notes

Thanks to: http://blog.snapdragon.cc/2015/05/05/subclass-pandas-dataframe-to-save-custom-attributes/

Attributes

T Transpose index and columns.

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

axes Return a list representing the axes of the DataFrame.

blocks Internal property, property synonym for as_blocks()

columns The column labels of the DataFrame.

dtypes Return the dtypes in the DataFrame.

empty Indicator whether DataFrame is empty.

ftypes Return the ftypes (indication of sparse/dense and dtype) in DataFrame.

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.

is_copy

ix A primarily label-location based indexer, with integer position fallback.

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 Property returning a Styler object containing methods for building a styled HTML representation fo the DataFrame.

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])	Addition of dataframe and other, element-wise (binary operator *add*).
add_prefix(prefix)	Prefix labels with string *prefix*.
add_suffix(suffix)	Suffix labels with string *suffix*.

Continued on next page

Table 19 – continued from previous page

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 for each axis Index
all([axis, bool_only, skipna, level])	Return whether all elements are True, potentially over an axis.
any([axis, bool_only, skipna, level])	Return whether any element is True over requested axis.
append(other[, ignore_index, ...])	Append rows of *other* to the end of this frame, returning a new object.
apply(func[, axis, broadcast, raw, reduce, ...])	Apply a function along an axis of the DataFrame.
applymap(func)	Apply a function to a Dataframe elementwise.
as_blocks([copy])	Convert the frame to a dict of dtype -> Constructor Types that each has a homogeneous dtype.
as_matrix([columns])	Convert the frame to its Numpy-array representation.
asfreq(freq[, method, how, normalize, ...])	Convert TimeSeries to specified frequency.
asof(where[, subset])	The last row without any NaN is taken (or the last row without NaN considering only the subset of columns in the case of a DataFrame)
assign(**kwargs)	Assign new columns to a DataFrame, returning a new object (a copy) with the new columns added to the original ones.
astype(**kwargs)	Cast a pandas object to a specified dtype `dtype`.
at_time(time[, asof])	Select values at particular time of day (e.g.
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])	Synonym for `DataFrame.fillna(method='bfill')`
bool()	Return the bool of a single element PandasObject.
boxplot([column, by, ax, fontsize, rot, ...])	Make a box plot from DataFrame columns.
clip([lower, upper, axis, inplace])	Trim values at input threshold(s).
clip_lower(threshold[, axis, inplace])	Return copy of the input with values below a threshold truncated.
clip_upper(threshold[, axis, inplace])	Return copy of input with values above given value(s) truncated.
combine(other, func[, fill_value, overwrite])	Add two DataFrame objects and do not propagate NaN values, so if for a (column, time) one frame is missing a value, it will default to the other frame's value (which might be NaN as well)
combine_first(other)	Combine two DataFrame objects and default to non-null values in frame calling the method.
compound([axis, skipna, level])	Return the compound percentage of the values for the requested axis
consolidate([inplace])	Compute NDFrame with "consolidated" internals (data of each dtype grouped together in a single ndarray).
convert_objects([convert_dates, ...])	Attempt to infer better dtype for object columns.
copy([deep])	Make a copy of this object's indices and data.

Continued on next page

Table 19 – continued from previous page

corr([method, min_periods])	Compute pairwise correlation of columns, excluding NA/null values
corrwith(other[, axis, drop])	Compute pairwise correlation between rows or columns of two DataFrame objects.
count([axis, level, numeric_only])	Count non-NA cells for each column or row.
cov([min_periods])	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])	Generates descriptive statistics that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding `NaN` values.
diff([periods, axis])	First discrete difference of element.
div(other[, axis, level, fill_value])	Floating division of dataframe and other, element-wise (binary operator *truediv*).
divide(other[, axis, level, fill_value])	Floating division of dataframe and other, element-wise (binary operator *truediv*).
dot(other)	Matrix multiplication with DataFrame or Series objects.
drop([labels, axis, index, columns, level, ...])	Drop specified labels from rows or columns.
drop_duplicates([subset, keep, inplace])	Return DataFrame with duplicate rows removed, optionally only considering certain columns
dropna([axis, how, thresh, subset, inplace])	Remove missing values.
duplicated([subset, keep])	Return boolean Series denoting duplicate rows, optionally only considering certain columns
eq(other[, axis, level])	Wrapper for flexible comparison methods eq
equals(other)	Determines if two NDFrame objects contain the same elements.
eval(expr[, inplace])	Evaluate a string describing operations on DataFrame columns.
ewm([com, span, halflife, alpha, ...])	Provides exponential weighted functions
expanding([min_periods, center, axis])	Provides expanding transformations.
ffill([axis, inplace, limit, downcast])	Synonym for `DataFrame.fillna(method='ffill')`
fillna([value, method, axis, inplace, ...])	Fill NA/NaN values using the specified method
filter([items, like, regex, axis])	Subset rows or columns of dataframe according to labels in the specified index.
first(offset)	Convenience method for subsetting initial periods of time series data based on a date offset.
first_valid_index()	Return index for first non-NA/null value.
floordiv(other[, axis, level, fill_value])	Integer division of dataframe and other, element-wise (binary operator *floordiv*).
from_csv(path[, header, sep, index_col, ...])	Read CSV file.
from_dict(data[, orient, dtype, columns])	Construct DataFrame from dict of array-like or dicts.
from_items(items[, columns, orient])	Construct a dataframe from a list of tuples

Continued on next page

Table 19 – continued from previous page

from_records(data[, index, exclude, ...])	Convert structured or record ndarray to DataFrame
ge(other[, axis, level])	Wrapper for flexible comparison methods ge
get(key[, default])	Get item from object for given key (DataFrame column, Panel slice, etc.).
get_dtype_counts()	Return counts of unique dtypes in this object.
get_ftype_counts()	Return counts of unique ftypes in this object.
get_value(index, col[, takeable])	Quickly retrieve single value at passed column and index
get_values()	Return an ndarray after converting sparse values to dense.
groupby([by, axis, level, as_index, sort, ...])	Group series using mapper (dict or key function, apply given function to group, return result as series) or by a series of columns.
gt(other[, axis, level])	Wrapper for flexible comparison methods gt
head([n])	Return the first *n* rows.
hist([column, by, grid, xlabelsize, xrot, ...])	Make a histogram of the DataFrame's.
idxmax([axis, skipna])	Return index of first occurrence of maximum over requested axis.
idxmin([axis, skipna])	Return index of first occurrence of minimum over requested axis.
infer_objects()	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, ...])	Interpolate values according to different methods.
isin(values)	Return boolean DataFrame showing whether each element in the DataFrame is contained in values.
isna()	Detect missing values.
isnull()	Detect missing values.
items()	Iterator over (column name, Series) pairs.
iteritems()	Iterator over (column name, Series) pairs.
iterrows()	Iterate over DataFrame rows as (index, Series) pairs.
itertuples([index, name])	Iterate over DataFrame rows as namedtuples, with index value as first element of the tuple.
join(other[, on, how, lsuffix, rsuffix, sort])	Join columns with other DataFrame either on index or on a key column.
keys()	Get the 'info axis' (see Indexing for more)
kurt([axis, skipna, level, numeric_only])	Return unbiased kurtosis over requested axis using Fisher's definition of kurtosis (kurtosis of normal == 0.0).
kurtosis([axis, skipna, level, numeric_only])	Return unbiased kurtosis over requested axis using Fisher's definition of kurtosis (kurtosis of normal == 0.0).
last(offset)	Convenience method for subsetting final periods of time series data based on a date offset.
last_valid_index()	Return index for last non-NA/null value.
le(other[, axis, level])	Wrapper for flexible comparison methods le
lookup(row_labels, col_labels)	Label-based "fancy indexing" function for DataFrame.
lt(other[, axis, level])	Wrapper for flexible comparison methods lt

Continued on next page

Table 19 – continued from previous page

mad([axis, skipna, level])	Return the mean absolute deviation of the values for the requested axis
mask(cond[, other, inplace, axis, level, ...])	Return an object of same shape as self and whose corresponding entries are from self where *cond* is False and otherwise are from *other*.
max([axis, skipna, level, numeric_only])	This method returns the maximum of the values in the object.
mean([axis, skipna, level, numeric_only])	Return the mean of the values for the requested axis
median([axis, skipna, level, numeric_only])	Return the median of the values for the requested axis
melt([id_vars, value_vars, var_name, ...])	"Unpivots" a DataFrame from wide format to long format, optionally leaving identifier variables set.
memory_usage([index, deep])	Return the memory usage of each column in bytes.
merge(right[, how, on, left_on, right_on, ...])	Merge DataFrame objects by performing a database-style join operation by columns or indexes.
min([axis, skipna, level, numeric_only])	This method returns the minimum of the values in the object.
mod(other[, axis, level, fill_value])	Modulo of dataframe and other, element-wise (binary operator *mod*).
mode([axis, numeric_only])	Gets the mode(s) of each element along the axis selected.
mul(other[, axis, level, fill_value])	Multiplication of dataframe and other, element-wise (binary operator *mul*).
multiply(other[, axis, level, fill_value])	Multiplication of dataframe and other, element-wise (binary operator *mul*).
ne(other[, axis, level])	Wrapper for flexible comparison methods ne
nlargest(n, columns[, keep])	Return the first *n* rows ordered by *columns* in descending order.
notna()	Detect existing (non-missing) values.
notnull()	Detect existing (non-missing) values.
nsmallest(n, columns[, keep])	Get the rows of a DataFrame sorted by the *n* smallest values of *columns*.
nunique([axis, dropna])	Return Series with number of distinct observations over requested axis.
pct_change([periods, fill_method, limit, freq])	Percentage change between the current and a prior element.
pipe(func, *args, **kwargs)	Apply func(self, *args, **kwargs)
pivot([index, columns, 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 pandas.plotting._core.FramePlotMethods
pop(item)	Return item and drop from frame.
pow(other[, axis, level, fill_value])	Exponential power of dataframe and other, element-wise (binary operator *pow*).
prod([axis, skipna, level, numeric_only, ...])	Return the product of the values for the requested axis
product([axis, skipna, level, numeric_only, ...])	Return the product of the values for the requested axis

Continued on next page

Table 19 – continued from previous page

quantile([q, axis, numeric_only, interpolation])	Return values at the given quantile over requested axis, a la numpy.percentile.
query(expr[, inplace])	Query the columns of a frame with a boolean expression.
radd(other[, axis, level, fill_value])	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])	Floating division of dataframe and other, element-wise (binary operator *rtruediv*).
reindex(**kwargs)	Conform DataFrame to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index.
reindex_axis(labels[, axis, method, level, ...])	Conform input object to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index.
reindex_like(other[, method, copy, limit, ...])	Return an object with matching indices to myself.
rename(**kwargs)	Alter axes labels.
rename_axis(mapper[, axis, copy, inplace])	Alter the name of 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[, how, axis, fill_method, ...])	Convenience method for frequency conversion and resampling of time series.
reset_index([level, drop, inplace, ...])	For DataFrame with multi-level index, return new DataFrame with labeling information in the columns under the index names, defaulting to 'level_0', 'level_1', etc.
rfloordiv(other[, axis, level, fill_value])	Integer division of dataframe and other, element-wise (binary operator *rfloordiv*).
rmod(other[, axis, level, fill_value])	Modulo of dataframe and other, element-wise (binary operator *rmod*).
rmul(other[, axis, level, fill_value])	Multiplication of dataframe and other, element-wise (binary operator *rmul*).
rolling(window[, min_periods, center, ...])	Provides rolling window calculations.
round([decimals])	Round a DataFrame to a variable number of decimal places.
rpow(other[, axis, level, fill_value])	Exponential power of dataframe and other, element-wise (binary operator *rpow*).
rsub(other[, axis, level, fill_value])	Subtraction of dataframe and other, element-wise (binary operator *rsub*).
rtruediv(other[, axis, level, fill_value])	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(crit[, axis])	Return data corresponding to axis labels matching criteria
select_dtypes([include, exclude])	Return a subset of the DataFrame's columns based on the column dtypes.
sem([axis, skipna, level, ddof, numeric_only])	Return unbiased standard error of the mean over requested axis.
set_axis(labels[, axis, inplace])	Assign desired index to given axis.

Continued on next page

Table 19 – continued from previous page

set_index(keys[, drop, append, inplace, ...])	Set the DataFrame index (row labels) using one or more existing columns.
set_value(index, col, value[, takeable])	Put single value at passed column and index
shift([periods, freq, axis])	Shift index by desired number of periods with an optional time freq
skew([axis, skipna, level, numeric_only])	Return unbiased skew over requested axis Normalized by N-1
slice_shift([periods, axis])	Equivalent to *shift* without copying data.
sort_index([axis, level, ascending, ...])	Sort object by labels (along an axis)
sort_values(by[, axis, ascending, inplace, ...])	Sort by the values along either axis
sortlevel([level, axis, ascending, inplace, ...])	Sort multilevel index by chosen axis and primary level.
squeeze([axis])	Squeeze length 1 dimensions.
stack([level, dropna])	Stack the prescribed level(s) from columns to index.
std([axis, skipna, level, ddof, numeric_only])	Return sample standard deviation over requested axis.
sub(other[, axis, level, fill_value])	Subtraction of dataframe and other, element-wise (binary operator *sub*).
subtract(other[, axis, level, fill_value])	Subtraction of dataframe and other, element-wise (binary operator *sub*).
sum([axis, skipna, level, numeric_only, ...])	Return the sum of the values for 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 on a particular axis
tail([n])	Return the last *n* rows.
take(indices[, axis, convert, is_copy])	Return the elements in the given *positional* indices along an axis.
to_clipboard([excel, sep])	Copy object to the system clipboard.
to_csv(*args, **kwargs)	Write DataFrame to a comma-separated values (csv) file
to_dense()	Return dense representation of NDFrame (as opposed to sparse)
to_dict([orient, into])	Convert the DataFrame to a dictionary.
to_excel(*args, **kwargs)	Write DataFrame to an excel sheet
to_feather(fname)	write out the binary feather-format for DataFrames
to_gbq(destination_table, project_id[, ...])	Write a DataFrame to a Google BigQuery table.
to_hdf(path_or_buf, key, **kwargs)	Write the contained data to an HDF5 file using HDFStore.
to_html(*args, **kwargs)	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, col_space, header, ...])	Render an object to a tabular environment table.
to_mol2([filepath_or_buffer, ...])	Write DataFrame to Mol2 file.
to_msgpack([path_or_buf, encoding])	msgpack (serialize) object to input file path
to_panel()	Transform long (stacked) format (DataFrame) into wide (3D, Panel) format.
to_parquet(fname[, engine, compression])	Write a DataFrame to the binary parquet format.
to_period([freq, axis, copy])	Convert DataFrame from DatetimeIndex to PeriodIndex with desired frequency (inferred from index if not passed)
to_pickle(path[, compression, protocol])	Pickle (serialize) object to file.
to_records([index, convert_datetime64])	Convert DataFrame to a NumPy record array.

Continued on next page

Table 19 – continued from previous page

to_sdf([filepath_or_buffer, ...])	Write DataFrame to SDF file.
to_sparse([fill_value, kind])	Convert to SparseDataFrame
to_sql(name, con[, schema, if_exists, ...])	Write records stored in a DataFrame to a SQL database.
to_stata(fname[, convert_dates, ...])	Export Stata binary dta files.
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.
transform(func, *args, **kwargs)	Call function producing a like-indexed NDFrame and return a NDFrame with the transformed values
transpose(*args, **kwargs)	Transpose index and columns.
truediv(other[, axis, level, fill_value])	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.
tshift([periods, freq, axis])	Shift the time index, using the index's frequency if available.
tz_convert(tz[, axis, level, copy])	Convert tz-aware axis to target time zone.
tz_localize(tz[, axis, level, copy, ambiguous])	Localize tz-naive TimeSeries to target time zone.
unstack([level, fill_value])	Pivot a level of the (necessarily hierarchical) index labels, returning a DataFrame having a new level of column labels whose inner-most level consists of the pivoted index labels.
update(other[, join, overwrite, ...])	Modify in place using non-NA values from another DataFrame.
var([axis, skipna, level, ddof, numeric_only])	Return unbiased variance over requested axis.
where(cond[, other, inplace, axis, level, ...])	Return an object of same shape as self and whose corresponding entries are from self where *cond* is True and otherwise are from *other*.
xs(key[, axis, level, drop_level])	Returns a cross-section (row(s) or column(s)) from the Series/DataFrame.

T

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

copy [bool, default False] If True, the underlying data is copied. Otherwise (default), no copy is made if possible.

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

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
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
```

abs()

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
```

add(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

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.

Parameters

other [Series, DataFrame, or constant]

axis [{0, 1, 'index', 'columns'}] For Series input, axis to match Series index on

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

fill_value [None or float value, 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

result [DataFrame]

See also:

`DataFrame.radd`

Notes

Mismatched indices will be unioned together

Examples

```
>>> a = pd.DataFrame([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'],
...                  columns=['one'])
>>> a
   one
a  1.0
```

(continues on next page)

(continued from previous page)

```
b  1.0
c  1.0
d  NaN
>>> b = pd.DataFrame(dict(one=[1, np.nan, 1, np.nan],
...                       two=[np.nan, 2, np.nan, 2]),
...                  index=['a', 'b', 'd', 'e'])
>>> b
   one  two
a  1.0  NaN
b  NaN  2.0
d  1.0  NaN
e  NaN  2.0
>>> a.add(b, fill_value=0)
   one  two
a  2.0  NaN
b  1.0  2.0
c  1.0  NaN
d  1.0  NaN
e  NaN  2.0
```

add_prefix(*prefix*)

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.

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
```

add_suffix(*suffix*)

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.

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
```

(continues on next page)

(continued from previous page)

```
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
```

agg (*func*, *axis=0*, **args*, ***kwargs*)

Aggregate using one or more operations over the specified axis.

New in version 0.20.0.

Parameters

func [function, string, dictionary, or list of string/functions] Function to use for aggregating the data. If a function, must either work when passed a DataFrame or when passed to DataFrame.apply. For a DataFrame, can pass a dict, if the keys are DataFrame column names.

Accepted combinations are:

- string function name.
- function.
- list of functions.
- dict of column names -> functions (or list of functions).

axis [{0 or 'index', 1 or 'columns'}, default 0]

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

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

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

Returns

aggregated [DataFrame]

See also:

DataFrame.apply Perform any type of operations.

DataFrame.transform Perform transformation type operations.

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

pandas.core.resample.Resampler Perform operations over resampled bins.

pandas.core.window.Rolling Perform operations over rolling window.

pandas.core.window.Expanding Perform operations over expanding window.

pandas.core.window.EWM Perform operation over exponential weighted window.

Notes

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

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

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.

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
max   NaN  8.0
min   1.0  2.0
sum  12.0  NaN
```

Aggregate over the columns.

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

aggregate(*func*, *axis=0*, **args*, ***kwargs*)

Aggregate using one or more operations over the specified axis.

New in version 0.20.0.

Parameters

func [function, string, dictionary, or list of string/functions] Function to use for aggregating the data. If a function, must either work when passed a DataFrame or when passed to DataFrame.apply. For a DataFrame, can pass a dict, if the keys are DataFrame column names.

Accepted combinations are:

- string function name.
- function.
- list of functions.
- dict of column names -> functions (or list of functions).

axis [{0 or 'index', 1 or 'columns'}, default 0]

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

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

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

Returns

aggregated [DataFrame]

See also:

`DataFrame.apply` Perform any type of operations.

`DataFrame.transform` Perform transformation type operations.

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

`pandas.core.resample.Resampler` Perform operations over resampled bins.

`pandas.core.window.Rolling` Perform operations over rolling window.

`pandas.core.window.Expanding` Perform operations over expanding window.

`pandas.core.window.EWM` Perform operation over exponential weighted window.

Notes

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

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

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.

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
max   NaN  8.0
min   1.0  2.0
sum  12.0  NaN
```

Aggregate over the columns.

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

align(*other*, *join='outer'*, *axis=None*, *level=None*, *copy=True*, *fill_value=None*, *method=None*, *limit=None*, *fill_axis=0*, *broadcast_axis=None*)
Align two objects on their axes with the specified join method for each axis Index

Parameters

other [DataFrame or Series]

join [{'outer', 'inner', 'left', 'right'}, default 'outer']

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 [boolean, 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 [str, default None]

limit [int, default None]

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

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

Returns

(left, right) [(DataFrame, type of other)] Aligned objects

all(*axis=0*, *bool_only=None*, *skipna=True*, *level=None*, ***kwargs*)
Return whether all elements are True, potentially over an axis.

Returns True if all elements within a series or along a Dataframe axis are non-zero, not-empty or not-False.

Parameters

axis [{0 or 'index', 1 or 'columns', None}, default 0] Indicate which axis or axes should be reduced.

- 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.

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.

bool_only [boolean, default None] Include only boolean columns. If None, will attempt to use everything, then use only boolean data. Not implemented for Series.

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

Returns

all [Series or DataFrame (if level specified)]

See also:

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

`pandas.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
```

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 column-wise values all return True.

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

Specify `axis='columns'` to check if row-wise values 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
```

any (*axis=0*, *bool_only=None*, *skipna=True*, *level=None*, ***kwargs*)
Return whether any element is True over requested axis.

Unlike `DataFrame.all()`, this performs an *or* operation. If any of the values along the specified axis is True, this will return True.

Parameters

axis [{0 or 'index', 1 or 'columns', None}, default 0] Indicate which axis or axes should be reduced.

- 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.

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.

bool_only [boolean, default None] Include only boolean columns. If None, will attempt to use everything, then use only boolean data. Not implemented for Series.

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

Returns

any [Series or DataFrame (if level specified)]

See also:

pandas.DataFrame.all Return whether all elements are True.

Examples

Series

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

```
>>> pd.Series([True, False]).any()
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
       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)
```

append(*other*, *ignore_index=False*, *verify_integrity=False*, *sort=None*)

Append rows of *other* to the end of this frame, returning a new object. Columns not in this frame are added as new columns.

Parameters

other [DataFrame or Series/dict-like object, or list of these] The data to append.

ignore_index [boolean, default False] If True, do not use the index labels.

verify_integrity [boolean, default False] If True, raise ValueError on creating index with duplicates.

sort [boolean, default None] Sort columns if the columns of *self* and *other* are not aligned. The default sorting is deprecated and will change to not-sorting in a future version of pandas. Explicitly pass `sort=True` to silence the warning and sort. Explicitly pass `sort=False` to silence the warning and not sort.

New in version 0.23.0.

Returns

appended [DataFrame]

See also:

`pandas.concat` General function to concatenate DataFrame, Series or Panel objects

Notes

If a list of dict/series is passed and the keys are all contained in the DataFrame's index, the order of the columns in the resulting DataFrame will be unchanged.

Iteratively appending rows to a DataFrame can be more computationally intensive than a single concatenate. A better solution is to append those rows to a list and then concatenate the list with the original DataFrame all at once.

Examples

```
>>> df = pd.DataFrame([[1, 2], [3, 4]], columns=list('AB'))
>>> df
   A  B
0  1  2
1  3  4
>>> df2 = pd.DataFrame([[5, 6], [7, 8]], columns=list('AB'))
>>> df.append(df2)
   A  B
0  1  2
1  3  4
0  5  6
1  7  8
```

With *ignore_index* set to True:

```
>>> df.append(df2, ignore_index=True)
   A  B
0  1  2
1  3  4
2  5  6
3  7  8
```

The following, while not recommended methods for generating DataFrames, show two ways to generate a DataFrame from multiple data sources.

Less efficient:

```
>>> df = pd.DataFrame(columns=['A'])
>>> for i in range(5):
...     df = df.append({'A': i}, ignore_index=True)
```

(continues on next page)

(continued from previous page)

```
>>> df
   A
0  0
1  1
2  2
3  3
4  4
```

More efficient:

```
>>> pd.concat([pd.DataFrame([i], columns=['A']) for i in range(5)],
...           ignore_index=True)
   A
0  0
1  1
2  2
3  3
4  4
```

apply (*func*, *axis=0*, *broadcast=None*, *raw=False*, *reduce=None*, *result_type=None*, *args=()*, ***kwds*)

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.

broadcast [bool, optional] Only relevant for aggregation functions:

- `False` or `None` : returns a Series whose length is the length of the index or the number of columns (based on the *axis* parameter)
- `True` : results will be broadcast to the original shape of the frame, the original index and columns will be retained.

Deprecated since version 0.23.0: This argument will be removed in a future version, replaced by result_type='broadcast'.

raw [bool, default False]

- `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.

reduce [bool or None, default None] Try to apply reduction procedures. If the DataFrame is empty, *apply* will use *reduce* to determine whether the result should be a Series or a DataFrame. If `reduce=None` (the default), *apply*'s return value will be guessed by calling *func* on an empty Series (note: while guessing, exceptions raised by *func* will be ignored). If `reduce=True` a Series will always be returned, and if `reduce=False` a DataFrame will always be returned.

Deprecated since version 0.23.0: This argument will be removed in a future version, replaced by `result_type='reduce'`.

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.

New in version 0.23.0.

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

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

Returns

applied [Series or DataFrame]

See also:

`DataFrame.applymap` For elementwise operations

`DataFrame.aggregate` only perform aggregating type operations

`DataFrame.transform` only perform transformating type operations

Notes

In the current implementation apply calls *func* twice on the first column/row to decide whether it can take a fast or slow code path. This can lead to unexpected behavior if *func* has side-effects, as they will take effect twice for the first column/row.

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
```

Retuning 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
```

applymap (*func*)

Apply a function to a Dataframe elementwise.

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.

Returns

DataFrame Transformed DataFrame.

See also:

DataFrame.apply Apply a function along input axis of DataFrame

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.applymap(lambda x: len(str(x)))
   0  1
0  3  4
1  5  5
```

Note that a vectorized version of *func* often exists, which will be much faster. You could square each number elementwise.

```
>>> df.applymap(lambda x: x**2)
           0          1
0   1.000000   4.494400
1  11.262736  20.857489
```

But it's better to avoid applymap in that case.

```
>>> df ** 2
           0          1
0   1.000000   4.494400
1  11.262736  20.857489
```

as_blocks(*copy=True*)

Convert the frame to a dict of dtype -> Constructor Types that each has a homogeneous dtype.

Deprecated since version 0.21.0.

NOTE: the dtypes of the blocks WILL BE PRESERVED HERE (unlike in as_matrix)

Parameters

copy [boolean, default True]

Returns

values [a dict of dtype -> Constructor Types]

as_matrix(*columns=None*)

Convert the frame to its Numpy-array representation.

Deprecated since version 0.23.0: Use `DataFrame.values()` instead.

Parameters

columns: list, optional, default:None If None, return all columns, otherwise, returns specified columns.

Returns

values [ndarray] If the caller is heterogeneous and contains booleans or objects, the result will be of dtype=object. See Notes.

See also:

`pandas.DataFrame.values`

Notes

Return is NOT a Numpy-matrix, rather, a Numpy-array.

The dtype will be a lower-common-denominator dtype (implicit upcasting); that is to say if the dtypes (even of numeric types) are mixed, the one that accommodates all will be chosen. Use this with care if you are not dealing with the blocks.

e.g. If the dtypes are float16 and float32, dtype will be upcast to float32. If dtypes are int32 and uint8, dtype will be upcase to int32. By numpy.find_common_type convention, mixing int64 and uint64 will result in a flot64 dtype.

This method is provided for backwards compatibility. Generally, it is recommended to use '.values'.

asfreq(*freq*, *method=None*, *how=None*, *normalize=False*, *fill_value=None*)
Convert TimeSeries to specified frequency.

Optionally provide filling method to pad/backfill missing values.

Returns the original data conformed to a new index with the specified frequency. `resample` is more appropriate if an operation, such as summarization, is necessary to represent the data at the new frequency.

Parameters

freq [DateOffset object, 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).

New in version 0.20.0.

Returns

converted [type of caller]

See also:

reindex

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
```

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

The last row without any NaN is taken (or the last row without NaN considering only the subset of columns in the case of a DataFrame)

New in version 0.19.0: For DataFrame

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

Parameters

where [date or array of dates]

subset [string or list of strings, default None] if not None use these columns for NaN propagation

Returns

where is scalar

- value or NaN if input is Series
- Series if input is DataFrame

where is Index: same shape object as input

See also:

`merge_asof`

Notes

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

assign (***kwargs*)

Assign new columns to a DataFrame, returning a new object (a copy) with the new columns added to the original ones. Existing columns that are re-assigned will be overwritten.

Parameters

kwargs [keyword, value pairs] keywords are the column names. 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

df [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. For Python 3.6 and above, later items in '**kwargs' may refer to newly created or modified columns in 'df'; items are computed and assigned into 'df' in order. For Python 3.5 and below, the order of keyword arguments is not specified, you cannot refer to newly created or modified columns. All items are computed first, and then assigned in alphabetical order.

Changed in version 0.23.0: Keyword argument order is maintained for Python 3.6 and later.

Examples

```
>>> df = pd.DataFrame({'A': range(1, 11), 'B': np.random.randn(10)})
```

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

```
>>> df.assign(ln_A = lambda x: np.log(x.A))
    A         B      ln_A
0   1  0.426905  0.000000
1   2 -0.780949  0.693147
2   3 -0.418711  1.098612
3   4 -0.269708  1.386294
4   5 -0.274002  1.609438
5   6 -0.500792  1.791759
6   7  1.649697  1.945910
7   8 -1.495604  2.079442
8   9  0.549296  2.197225
9  10 -0.758542  2.302585
```

Where the value already exists and is inserted:

```
>>> newcol = np.log(df['A'])
>>> df.assign(ln_A=newcol)
    A         B      ln_A
0   1  0.426905  0.000000
1   2 -0.780949  0.693147
2   3 -0.418711  1.098612
3   4 -0.269708  1.386294
4   5 -0.274002  1.609438
5   6 -0.500792  1.791759
6   7  1.649697  1.945910
7   8 -1.495604  2.079442
8   9  0.549296  2.197225
9  10 -0.758542  2.302585
```

Where the keyword arguments depend on each other

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

```
>>> df.assign(B=df.A, C=lambda x:x['A']+ x['B'])
   A  B  C
0  1  1  2
1  2  2  4
2  3  3  6
```

astype(***kwargs*)

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

Parameters

dtype [data type, or dict of column name -> data type] Use a numpy.dtype or Python type to cast entire pandas object to the same type. Alternatively, use {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

New in version 0.20.0.

raise_on_error [raise on invalid input] Deprecated since version 0.20.0: Use `errors` instead

kwargs [keyword arguments to pass on to the constructor]

Returns

casted [type of caller]

See also:

pandas.to_datetime Convert argument to datetime.

pandas.to_timedelta Convert argument to timedelta.

pandas.to_numeric Convert argument to a numeric type.

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

Examples

```
>>> 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, int64): [1, 2]
```

Convert to ordered categorical type with custom ordering:

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

Note that using `copy=False` and changing data on a new pandas object may propagate changes:

```
>>> s1 = pd.Series([1,2])
>>> s2 = s1.astype('int64', copy=False)
>>> s2[0] = 10
>>> s1  # note that s1[0] has changed too
0    10
1     2
dtype: int64
```

at

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

Similar to `loc`, in that both provide label-based lookups. Use `at` if you only need to get or set a single value in a DataFrame or Series.

Raises

KeyError When label does not exist in DataFrame

See also:

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

`DataFrame.loc` Access a group of rows and columns by label(s)

`Series.at` Access a single value using a label

Examples

```
>>> df = pd.DataFrame([[0, 2, 3], [0, 4, 1], [10, 20, 30]],
...                   index=[4, 5, 6], columns=['A', 'B', 'C'])
>>> df
    A   B   C
4   0   2   3
5   0   4   1
6  10  20  30
```

Get value at specified row/column pair

```
>>> df.at[4, 'B']
2
```

Set value at specified row/column pair

```
>>> df.at[4, 'B'] = 10
>>> df.at[4, 'B']
10
```

Get value within a Series

```
>>> df.loc[5].at['B']
4
```

`at_time`(*time*, *asof=False*)

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

Parameters

time [datetime.time or string]

Returns

values_at_time [type of caller]

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
```

axes

Return a list representing the axes of the DataFrame.

It has the row axis labels and column axis labels as the only members. They are returned in that order.

Examples

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df.axes
[RangeIndex(start=0, stop=2, step=1), Index(['coll', 'col2'],
dtype='object')]
```

between_time(*start_time*, *end_time*, *include_start=True*, *include_end=True*)

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 string]

end_time [datetime.time or string]

include_start [boolean, default True]

include_end [boolean, default True]

Returns

values_between_time [type of caller]

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
```

bfill (*axis=None*, *inplace=False*, *limit=None*, *downcast=None*)
Synonym for `DataFrame.fillna(method='bfill')`

blocks
Internal property, property synonym for as_blocks()

Deprecated since version 0.21.0.

bool ()
Return the bool of a single element PandasObject.

This must be a boolean scalar value, either True or False. Raise a ValueError if the PandasObject does not have exactly 1 element, or that element is not boolean

boxplot (*column=None*, *by=None*, *ax=None*, *fontsize=None*, *rot=0*, *grid=True*, *figsize=None*, *layout=None*, *return_type=None*, ***kwds*)
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. The position of the whiskers is set by default to *1.5 * IQR (IQR = Q3 - Q1)* from the edges of the box. Outlier points are those past the end of the whiskers.

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

grid [boolean, 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 columns and 5 rows, 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.

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

Returns

result : 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 nametuple with strucure (ax, lines)

For data grouped with `by`:

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

See also:

`Series.plot.hist` Make a histogram.

`matplotlib.pyplot.boxplot` Matplotlib equivalent plot.

Notes

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:

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

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:

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

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`):

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._subplots.AxesSubplot'>
```

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'>
```

clip (*lower=None*, *upper=None*, *axis=None*, *inplace=False*, **args*, ***kwargs*)

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.

upper [float or array_like, default None] Maximum threshold value. All values above this threshold will be set to it.

axis [int or string axis name, optional] Align object with lower and upper along the given axis.

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

New in version 0.21.0.

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

Returns

Series or DataFrame Same type as calling object with the values outside the clip boundaries replaced

See also:

clip_lower Clip values below specified threshold(s).

clip_upper Clip values above specified threshold(s).

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
```

clip_lower(*threshold*, *axis=None*, *inplace=False*)

Return copy of the input with values below a threshold truncated.

Parameters

threshold [numeric or array-like] Minimum value allowed. All values below threshold will be set to this value.

- float : every value is compared to *threshold*.
- array-like : The shape of *threshold* should match the object it's compared to. When *self* is a Series, *threshold* should be the length. When *self* is a DataFrame, *threshold* should 2-D and the same shape as *self* for `axis=None`, or 1-D and the same length as the axis being compared.

axis [{0 or 'index', 1 or 'columns'}, default 0] Align *self* with *threshold* along the given axis.

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

New in version 0.21.0.

Returns

clipped [same type as input]

See also:

`Series.clip` Return copy of input with values below and above thresholds truncated.

`Series.clip_upper` Return copy of input with values above threshold truncated.

Examples

Series single threshold clipping:

```
>>> s = pd.Series([5, 6, 7, 8, 9])
>>> s.clip_lower(8)
0    8
1    8
2    8
3    8
4    9
dtype: int64
```

Series clipping element-wise using an array of thresholds. *threshold* should be the same length as the Series.

```
>>> elemwise_thresholds = [4, 8, 7, 2, 5]
>>> s.clip_lower(elemwise_thresholds)
0    5
1    8
2    7
3    8
4    9
dtype: int64
```

DataFrames can be compared to a scalar.

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

```
>>> df.clip_lower(3)
   A  B
0  3  3
1  3  4
2  5  6
```

Or to an array of values. By default, *threshold* should be the same shape as the DataFrame.

```
>>> df.clip_lower(np.array([[3, 4], [2, 2], [6, 2]]))
   A  B
0  3  4
1  3  4
2  6  6
```

Control how *threshold* is broadcast with *axis*. In this case *threshold* should be the same length as the axis specified by *axis*.

```
>>> df.clip_lower(np.array([3, 3, 5]), axis='index')
   A  B
0  3  3
1  3  4
2  5  6
```

```
>>> df.clip_lower(np.array([4, 5]), axis='columns')
   A  B
0  4  5
1  4  5
2  5  6
```

clip_upper (*threshold*, *axis=None*, *inplace=False*)

Return copy of input with values above given value(s) truncated.

Parameters

threshold [float or array_like]

axis [int or string axis name, optional] Align object with threshold along the given axis.

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

New in version 0.21.0.

Returns

clipped [same type as input]

See also:

clip

columns

The column labels of the DataFrame.

combine (*other*, *func*, *fill_value=None*, *overwrite=True*)

Add two DataFrame objects and do not propagate NaN values, so if for a (column, time) one frame is missing a value, it will default to the other frame's value (which might be NaN as well)

Parameters

other [DataFrame]

func [function] Function that takes two series as inputs and return a Series or a scalar

fill_value [scalar value]

overwrite [boolean, default True] If True then overwrite values for common keys in the calling frame

Returns

result [DataFrame]

See also:

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

Examples

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

combine_first(*other*)

Combine two DataFrame objects and default to non-null values in frame calling the method. Result index columns will be the union of the respective indexes and columns

Parameters

other [DataFrame]

Returns

combined [DataFrame]

See also:

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

Examples

df1's values prioritized, use values from df2 to fill holes:

```
>>> df1 = pd.DataFrame([[1, np.nan]])
>>> df2 = pd.DataFrame([[3, 4]])
>>> df1.combine_first(df2)
   0    1
0  1  4.0
```

compound(*axis=None*, *skipna=None*, *level=None*)

Return the compound percentage of the values for the requested axis

Parameters

axis [{index (0), columns (1)}]

skipna [boolean, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

Returns

compounded [Series or DataFrame (if level specified)]

consolidate(*inplace=False*)

Compute NDFrame with "consolidated" internals (data of each dtype grouped together in a single ndarray).

Deprecated since version 0.20.0: Consolidate will be an internal implementation only.

convert_objects(*convert_dates=True*, *convert_numeric=False*, *convert_timedeltas=True*, *copy=True*)

Attempt to infer better dtype for object columns.

Deprecated since version 0.21.0.

Parameters

convert_dates [boolean, default True] If True, convert to date where possible. If 'coerce', force conversion, with unconvertible values becoming NaT.

convert_numeric [boolean, default False] If True, attempt to coerce to numbers (including strings), with unconvertible values becoming NaN.

convert_timedeltas [boolean, default True] If True, convert to timedelta where possible. If 'coerce', force conversion, with unconvertible values becoming NaT.

copy [boolean, default True] If True, return a copy even if no copy is necessary (e.g. no conversion was done). Note: This is meant for internal use, and should not be confused with inplace.

Returns

converted [same as input object]

See also:

pandas.to_datetime Convert argument to datetime.

pandas.to_timedelta Convert argument to timedelta.

pandas.to_numeric Return a fixed frequency timedelta index, with day as the default.

copy(*deep=True*)

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

copy [Series, DataFrame or Panel] 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.

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[0] = 3
>>> shallow[1] = 4
>>> s
a    3
b    4
dtype: int64
```

(continues on next page)

(continued from previous page)

```
>>> 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
```

corr(*method='pearson'*, *min_periods=1*)

Compute pairwise correlation of columns, excluding NA/null values

Parameters

method [{'pearson', 'kendall', 'spearman'}]

- pearson : standard correlation coefficient
- kendall : Kendall Tau correlation coefficient
- spearman : Spearman rank correlation

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

Returns

y [DataFrame]

corrwith(*other*, *axis=0*, *drop=False*)

Compute pairwise correlation between rows or columns of two DataFrame objects.

Parameters

other [DataFrame, Series]

axis [{0 or 'index', 1 or 'columns'}, default 0] 0 or 'index' to compute column-wise, 1 or 'columns' for row-wise

drop [boolean, default False] Drop missing indices from result, default returns union of all

Returns

correls [Series]

count(*axis=0*, *level=None*, *numeric_only=False*)

Count non-NA cells for each column or row.

The values *None*, *NaN*, *NaT*, and optionally *numpy.inf* (depending on *pandas.options.mode.use_inf_as_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**.

level [int or str, optional] If the axis is a *MultiIndex* (hierarchical), count along a particular *level*, collapsing into a *DataFrame*. A *str* specifies the level name.

numeric_only [boolean, default False] Include only *float*, *int* or *boolean* data.

Returns

Series or DataFrame For each column/row the number of non-NA/null entries. If *level* is specified returns a *DataFrame*.

See also:

`Series.count` number of non-NA elements in a Series

`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", None, "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    None  21.0    True
3    John  33.0    True
4    Myla  26.0   False
```

Notice the uncounted NA values:

```
>>> df.count()
Person    4
Age       4
Single    5
dtype: int64
```

Counts for each **row**:

```
>>> df.count(axis='columns')
0    3
1    2
2    2
3    3
4    3
dtype: int64
```

Counts for one level of a *MultiIndex*:

```
>>> df.set_index(["Person", "Single"]).count(level="Person")
        Age
Person
John      2
Myla      1
```

cov(*min_periods=None*)

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.

Returns

DataFrame The covariance matrix of the series of the DataFrame.

See also:

pandas.Series.cov compute covariance with another Series

pandas.core.window.EWM.cov expoential weighted sample covariance

pandas.core.window.Expanding.cov expanding sample covariance

pandas.core.window.Rolling.cov rolling sample covariance

Notes

Returns the covariance matrix of the DataFrame's time series. The covariance is normalized by N-1.

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()
```

(continues on next page)

(continued from previous page)

```
          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
>>> 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
```

cummax (*axis=None*, *skipna=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'.

skipna [boolean, 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

cummax [Series or DataFrame]

See also:

pandas.core.window.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
```

cummin (*axis=None*, *skipna=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'.

skipna [boolean, 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

cummin [Series or DataFrame]

See also:

pandas.core.window.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
```

(continues on next page)

(continued from previous page)

```
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
```

cumprod(*axis=None*, *skipna=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'.

skipna [boolean, 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

cumprod [Series or DataFrame]

See also:

pandas.core.window.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
```

(continues on next page)

(continued from previous page)

```
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
```

cumsum(*axis=None*, *skipna=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'.

skipna [boolean, 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

cumsum [Series or DataFrame]

See also:

pandas.core.window.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
```

describe(*percentiles=None*, *include=None*, *exclude=None*)

Generates descriptive statistics 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(include=['O'])`). To exclude pandas categorical columns, use `'category'`
- None (default) : The result will exclude nothing.

Returns

summary: Series/DataFrame of summary statistics

See also:

`DataFrame.count`, `DataFrame.max`, `DataFrame.min`, `DataFrame.mean`, `DataFrame.std`, `DataFrame.select_dtypes`

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
```

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
unique                      2
top       2010-01-01 00:00:00
freq                        2
first     2000-01-01 00:00:00
last      2010-01-01 00:00:00
dtype: object
```

Describing a `DataFrame`. By default only numeric fields are returned.

```
>>> df = pd.DataFrame({ 'object': ['a', 'b', 'c'],
...                     'numeric': [1, 2, 3],
...                     'categorical': pd.Categorical(['d','e','f'])
...                   })
>>> df.describe()
       numeric
count      3.0
mean       2.0
```

(continues on next page)

(continued from previous page)

```
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      c
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=[np.object])
       object
count       3
unique      3
top         c
freq        1
```

Including only categorical columns from a `DataFrame` description.

```
>>> df.describe(include=['category'])
       categorical
count             3
unique            3
top               f
freq              1
```

Excluding numeric columns from a `DataFrame` description.

```
>>> df.describe(exclude=[np.number])
       categorical object
count             3      3
unique            3      3
top               f      c
freq              1      1
```

Excluding object columns from a `DataFrame` description.

```
>>> df.describe(exclude=[np.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
```

diff (*periods=1*, *axis=0*)

First discrete difference of element.

Calculates the difference of a DataFrame element compared with another element in the DataFrame (default is the element in the same column of the 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).

New in version 0.16.1..

Returns

diffed [DataFrame]

See also:

Series.diff First discrete difference for a Series.

DataFrame.pct_change Percent change over given number of periods.

DataFrame.shift Shift index by desired number of periods with an optional time freq.

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   0.0
1 NaN -1.0   3.0
2 NaN -1.0   7.0
3 NaN -1.0  13.0
4 NaN  0.0  20.0
5 NaN  2.0  28.0
```

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
```

div(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
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.

Parameters

other [Series, DataFrame, or constant]

axis [{0, 1, 'index', 'columns'}] For Series input, axis to match Series index on

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

fill_value [None or float value, 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

result [DataFrame]

See also:

`DataFrame.rtruediv`

Notes

Mismatched indices will be unioned together

Examples

None

divide(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
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.

Parameters

other [Series, DataFrame, or constant]

axis [{0, 1, 'index', 'columns'}] For Series input, axis to match Series index on

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

fill_value [None or float value, 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

result [DataFrame]

See also:

`DataFrame.rtruediv`

Notes

Mismatched indices will be unioned together

Examples

None

dot (*other*)
Matrix multiplication with DataFrame or Series objects. Can also be called using *self @ other* in Python >= 3.5.

Parameters

other [DataFrame or Series]

Returns

dot_product [DataFrame or Series]

drop (*labels=None*, *axis=0*, *index=None*, *columns=None*, *level=None*, *inplace=False*, *errors='raise'*)
Drop specified labels from rows or columns.

Remove rows or columns by specifying label names and corresponding axis, or by specifying directly index or column names. When using a multi-index, labels on different levels can be removed by specifying the level.

Parameters

labels [single label or list-like] Index or column labels to drop.

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, columns [single label or list-like] Alternative to specifying axis (`labels, axis=1` is equivalent to `columns=labels`).

New in version 0.21.0.

level [int or level name, optional] For MultiIndex, level from which the labels will be removed.

inplace [bool, default False] If True, do operation inplace and return None.

errors [{'ignore', 'raise'}, default 'raise'] If 'ignore', suppress error and only existing labels are dropped.

Returns

dropped [pandas.DataFrame]

Raises

KeyError If none of the labels are 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=[['lama', 'cow', 'falcon'],
...                              ['speed', 'weight', 'length']],
...                      labels=[[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
lama    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
```

```
>>> df.drop(index='cow', columns='small')
                big
lama    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
lama    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
```

drop_duplicates(*subset=None*, *keep='first'*, *inplace=False*)

Return DataFrame with duplicate rows removed, optionally only considering certain columns

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']

- `first` : Drop duplicates except for the first occurrence.
- `last` : Drop duplicates except for the last occurrence.
- False : Drop all duplicates.

inplace [boolean, default False] Whether to drop duplicates in place or to return a copy

Returns

deduplicated [DataFrame]

dropna(*axis=0*, *how='any'*, *thresh=None*, *subset=None*, *inplace=False*)

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.

Deprecated since version 0.23.0:: Pass tuple or list to drop on multiple

axes.

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.

subset [array-like, 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] If True, do operation inplace and return None.

Returns

DataFrame DataFrame with NA entries dropped from it.

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', 'born'])
       name        toy       born
1    Batman  Batmobile 1940-04-25
```

Keep the DataFrame with valid entries in the same variable.

```
>>> df.dropna(inplace=True)
>>> df
     name        toy       born
1  Batman  Batmobile 1940-04-25
```

dtypes

Return the dtypes in the DataFrame.

This returns a Series with the data type of each column. The result's index is the original DataFrame's columns. Columns with mixed types are stored with the `object` dtype. See the User Guide for more.

Returns

pandas.Series The data type of each column.

See also:

pandas.DataFrame.ftypes dtype and sparsity information.

Examples

```
>>> df = pd.DataFrame({'float': [1.0],
...                    'int': [1],
...                    'datetime': [pd.Timestamp('20180310')],
...                    'string': ['foo']})
>>> df.dtypes
float              float64
int                  int64
datetime    datetime64[ns]
string              object
dtype: object
```

duplicated(*subset=None*, *keep='first'*)

Return boolean Series denoting duplicate rows, optionally only considering certain columns

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']

- `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

duplicated [Series]

empty

Indicator whether DataFrame is empty.

True if DataFrame is entirely empty (no items), meaning any of the axes are of length 0.

Returns

bool If DataFrame is empty, return True, if not return False.

See also:

`pandas.Series.dropna`, `pandas.DataFrame.dropna`

Notes

If DataFrame contains only NaNs, it is still not considered empty. See the example below.

Examples

An example of an actual empty DataFrame. Notice the index is empty:

```
>>> df_empty = pd.DataFrame({'A' : []})
>>> df_empty
Empty DataFrame
Columns: [A]
Index: []
>>> df_empty.empty
True
```

If we only have NaNs in our DataFrame, it is not considered empty! We will need to drop the NaNs to make the DataFrame empty:

```
>>> df = pd.DataFrame({'A' : [np.nan]})
>>> df
    A
0 NaN
>>> df.empty
False
>>> df.dropna().empty
True
```

eq(*other*, *axis='columns'*, *level=None*)

Wrapper for flexible comparison methods eq

equals(*other*)

Determines if two NDFrame objects contain the same elements. NaNs in the same location are considered equal.

eval(*expr*, *inplace=False*, ***kwargs*)

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.

New in version 0.18.0..

kwargs [dict] See the documentation for `eval()` for complete details on the keyword arguments accepted by `query()`.

Returns

ndarray, scalar, or pandas object The result of the evaluation.

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.

`pandas.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
```

(continues on next page)

(continued from previous page)

```
>>> df
   A   B
0  1  10
1  2   8
2  3   6
3  4   4
4  5   2
```

Use `inplace=True` to modify the original DataFrame.

```
>>> df.eval('C = A + B', inplace=True)
>>> df
   A   B   C
0  1  10  11
1  2   8  10
2  3   6   9
3  4   4   8
4  5   2   7
```

ewm(*com=None*, *span=None*, *halflife=None*, *alpha=None*, *min_periods=0*, *adjust=True*, *ignore_na=False*, *axis=0*)
Provides exponential weighted functions

New in version 0.18.0.

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, optional] Specify decay in terms of half-life, $\alpha = 1 - exp(log(0.5)/halflife),$ for $halflife > 0$

alpha [float, optional] Specify smoothing factor α directly, $0 < \alpha \leq 1$

New in version 0.18.0.

min_periods [int, default 0] Minimum number of observations in window required to have a value (otherwise result is NA).

adjust [boolean, default True] Divide by decaying adjustment factor in beginning periods to account for imbalance in relative weightings (viewing EWMA as a moving average)

ignore_na [boolean, default False] Ignore missing values when calculating weights; specify True to reproduce pre-0.15.0 behavior

Returns

a Window sub-classed for the particular operation

See also:

rolling Provides rolling window calculations

expanding Provides expanding transformations.

Notes

Exactly one of center of mass, span, half-life, and alpha must be provided. Allowed values and relationship between the parameters are specified in the parameter descriptions above; see the link at the end of this section for a detailed explanation.

When adjust is True (default), weighted averages are calculated using weights (1-alpha)**(n-1), (1-alpha)**(n-2), . . . , 1-alpha, 1.

When adjust is False, weighted averages are calculated recursively as: weighted_average[0] = arg[0]; weighted_average[i] = (1-alpha)*weighted_average[i-1] + alpha*arg[i].

When ignore_na is False (default), weights are based on absolute positions. For example, the weights of x and y used in calculating the final weighted average of [x, None, y] are (1-alpha)**2 and 1 (if adjust is True), and (1-alpha)**2 and alpha (if adjust is False).

When ignore_na is True (reproducing pre-0.15.0 behavior), weights are based on relative positions. For example, the weights of x and y used in calculating the final weighted average of [x, None, y] are 1-alpha and 1 (if adjust is True), and 1-alpha and alpha (if adjust is False).

More details can be found at http://pandas.pydata.org/pandas-docs/stable/computation.html#exponentially-weighted-windows

Examples

```
>>> df = DataFrame({'B': [0, 1, 2, np.nan, 4]})
     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
```

expanding(*min_periods=1*, *center=False*, *axis=0*)

Provides expanding transformations.

New in version 0.18.0.

Parameters

min_periods [int, default 1] Minimum number of observations in window required to have a value (otherwise result is NA).

center [boolean, default False] Set the labels at the center of the window.

axis [int or string, default 0]

Returns

a Window sub-classed for the particular operation

See also:

rolling Provides rolling window calculations

ewm Provides exponential weighted functions

Notes

By default, the result is set to the right edge of the window. This can be changed to the center of the window by setting `center=True`.

Examples

```
>>> df = DataFrame({'B': [0, 1, 2, np.nan, 4]})
     B
0  0.0
1  1.0
2  2.0
3  NaN
4  4.0
```

```
>>> df.expanding(2).sum()
     B
0  NaN
1  1.0
2  3.0
3  3.0
4  7.0
```

ffill(*axis=None*, *inplace=False*, *limit=None*, *downcast=None*)
Synonym for `DataFrame.fillna(method='ffill')`

fillna(*value=None*, *method=None*, *axis=None*, *inplace=False*, *limit=None*, *downcast=None*, ***kwargs*)
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', '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

axis [{0 or 'index', 1 or 'columns'}]

inplace [boolean, 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

filled [DataFrame]

See also:

interpolate Fill NaN values using interpolation.

reindex, *asfreq*

Examples

```
>>> df = pd.DataFrame([[np.nan, 2, np.nan, 0],
...                    [3, 4, np.nan, 1],
...                    [np.nan, np.nan, np.nan, 5],
...                    [np.nan, 3, np.nan, 4]],
...                    columns=list('ABCD'))
>>> df
     A    B   C  D
0  NaN  2.0 NaN  0
1  3.0  4.0 NaN  1
2  NaN  NaN NaN  5
3  NaN  3.0 NaN  4
```

Replace all NaN elements with 0s.

```
>>> df.fillna(0)
    A   B   C   D
0   0.0 2.0 0.0 0
1   3.0 4.0 0.0 1
2   0.0 0.0 0.0 5
3   0.0 3.0 0.0 4
```

We can also propagate non-null values forward or backward.

```
>>> df.fillna(method='ffill')
    A   B   C   D
0   NaN 2.0 NaN 0
1   3.0 4.0 NaN 1
2   3.0 4.0 NaN 5
3   3.0 3.0 NaN 4
```

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
1   3.0 4.0 2.0 1
2   0.0 1.0 2.0 5
3   0.0 3.0 2.0 4
```

Only replace the first NaN element.

```
>>> df.fillna(value=values, limit=1)
    A   B   C   D
0   0.0 2.0 2.0 0
1   3.0 4.0 NaN 1
2   NaN 1.0 NaN 5
3   NaN 3.0 NaN 4
```

filter(*items=None*, *like=None*, *regex=None*, *axis=None*)

Subset rows or columns of dataframe according to labels in the specified index.

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] List of info axis to restrict to (must not all be present)

like [string] Keep info axis where "arg in col == True"

regex [string (regular expression)] Keep info axis with re.search(regex, col) == True

axis [int or string axis name] The axis to filter on. By default this is the info axis, 'index' for Series, 'columns' for DataFrame

Returns

same type as input object

See also:

`pandas.DataFrame.loc`

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
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
```

first(*offset*)

Convenience method for subsetting initial periods of time series data based on a date offset.

Parameters

offset [string, DateOffset, dateutil.relativedelta]

Returns

subset [type of 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 calender days were returned, not the first 3 days observed in the dataset, and therefore data for 2018-04-13 was not returned.

first_valid_index()

Return index for first non-NA/null value.

Returns

scalar [type of index]

Notes

If all elements are non-NA/null, returns None. Also returns None for empty NDFrame.

floordiv(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

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.

Parameters

other [Series, DataFrame, or constant]

axis [{0, 1, 'index', 'columns'}] For Series input, axis to match Series index on

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

fill_value [None or float value, 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

result [DataFrame]

See also:

`DataFrame.rfloordiv`

Notes

Mismatched indices will be unioned together

Examples

None

classmethod from_csv(*path*, *header=0*, *sep=','*, *index_col=0*, *parse_dates=True*, *encoding=None*, *tupleize_cols=None*, *infer_datetime_format=False*)

Read CSV file.

Deprecated since version 0.21.0: Use `pandas.read_csv()` instead.

It is preferable to use the more powerful `pandas.read_csv()` for most general purposes, but `from_csv` makes for an easy roundtrip to and from a file (the exact counterpart of `to_csv`), especially with a DataFrame of time series data.

This method only differs from the preferred `pandas.read_csv()` in some defaults:

- *index_col* is `0` instead of `None` (take first column as index by default)
- *parse_dates* is `True` instead of `False` (try parsing the index as datetime by default)

So a `pd.DataFrame.from_csv(path)` can be replaced by `pd.read_csv(path, index_col=0, parse_dates=True)`.

Parameters

path [string file path or file handle / StringIO]

header [int, default 0] Row to use as header (skip prior rows)

sep [string, default ','] Field delimiter

index_col [int or sequence, default 0] Column to use for index. If a sequence is given, a MultiIndex is used. Different default from read_table

parse_dates [boolean, default True] Parse dates. Different default from read_table

tupleize_cols [boolean, default False] write multi_index columns as a list of tuples (if True) or new (expanded format) if False)

infer_datetime_format: boolean, default False If True and *parse_dates* is True for a column, try to infer the datetime format based on the first datetime string. If the format can be inferred, there often will be a large parsing speed-up.

Returns

y [DataFrame]

See also:

`pandas.read_csv`

classmethod from_dict(*data*, *orient='columns'*, *dtype=None*, *columns=None*)

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'}, 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'.

dtype [dtype, default None] Data type to force, otherwise infer.

columns [list, default None] Column labels to use when `orient='index'`. Raises a ValueError if used with `orient='columns'`.

New in version 0.23.0.

Returns

pandas.DataFrame

See also:

DataFrame.from_records DataFrame from ndarray (structured dtype), list of tuples, dict, or DataFrame

DataFrame DataFrame object creation using constructor

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
```

classmethod from_items(*items*, *columns=None*, *orient='columns'*)

Construct a dataframe from a list of tuples

Deprecated since version 0.23.0: *from_items* is deprecated and will be removed in a future version. Use `DataFrame.from_dict(dict(items))` instead. `DataFrame.from_dict(OrderedDict(items))` may be used to preserve the key order.

Convert (key, value) pairs to DataFrame. The keys will be the axis index (usually the columns, but depends on the specified orientation). The values should be arrays or Series.

Parameters

items [sequence of (key, value) pairs] Values should be arrays or Series.

columns [sequence of column labels, optional] Must be passed if orient='index'.

orient [{'columns', 'index'}, default 'columns'] The "orientation" of the data. If the keys of the input correspond to column labels, pass 'columns' (default). Otherwise if the keys correspond to the index, pass 'index'.

Returns

frame [DataFrame]

classmethod from_records(*data*, *index=None*, *exclude=None*, *columns=None*, *coerce_float=False*, *nrows=None*)

Convert structured or record ndarray to DataFrame

Parameters

data [ndarray (structured dtype), list of tuples, dict, or DataFrame]

index [string, 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 [boolean, default False] Attempt to convert values of non-string, non-numeric objects (like decimal.Decimal) to floating point, useful for SQL result sets

Returns

df [DataFrame]

ftypes
Return the ftypes (indication of sparse/dense and dtype) in DataFrame.

This returns a Series with the data type of each column. The result's index is the original DataFrame's columns. Columns with mixed types are stored with the `object` dtype. See the User Guide for more.

Returns

pandas.Series The data type and indication of sparse/dense of each column.

See also:

pandas.DataFrame.dtypes Series with just dtype information.

pandas.SparseDataFrame Container for sparse tabular data.

Notes

Sparse data should have the same dtypes as its dense representation.

Examples

```
>>> import numpy as np
>>> arr = np.random.RandomState(0).randn(100, 4)
>>> arr[arr < .8] = np.nan
>>> pd.DataFrame(arr).ftypes
0    float64:dense
1    float64:dense
2    float64:dense
3    float64:dense
dtype: object
```

```
>>> pd.SparseDataFrame(arr).ftypes
0    float64:sparse
1    float64:sparse
2    float64:sparse
3    float64:sparse
dtype: object
```

ge(*other*, *axis='columns'*, *level=None*)
Wrapper for flexible comparison methods ge

get(*key*, *default=None*)
Get item from object for given key (DataFrame column, Panel slice, etc.). Returns default value if not found.

Parameters

key [object]

Returns

value [type of items contained in object]

get_dtype_counts()
Return counts of unique dtypes in this object.

Returns

dtype [Series] Series with the count of columns with each dtype.

See also:

dtypes Return the dtypes in this object.

Examples

```
>>> a = [['a', 1, 1.0], ['b', 2, 2.0], ['c', 3, 3.0]]
>>> df = pd.DataFrame(a, columns=['str', 'int', 'float'])
>>> df
  str  int  float
0   a    1    1.0
1   b    2    2.0
2   c    3    3.0
```

```
>>> df.get_dtype_counts()
float64    1
int64      1
object     1
dtype: int64
```

get_ftype_counts()

Return counts of unique ftypes in this object.

Deprecated since version 0.23.0.

This is useful for SparseDataFrame or for DataFrames containing sparse arrays.

Returns

dtype [Series] Series with the count of columns with each type and sparsity (dense/sparse)

See also:

ftypes Return ftypes (indication of sparse/dense and dtype) in this object.

Examples

```
>>> a = [['a', 1, 1.0], ['b', 2, 2.0], ['c', 3, 3.0]]
>>> df = pd.DataFrame(a, columns=['str', 'int', 'float'])
>>> df
  str  int  float
0   a    1    1.0
1   b    2    2.0
2   c    3    3.0
```

```
>>> df.get_ftype_counts()
float64:dense    1
int64:dense      1
object:dense     1
dtype: int64
```

get_value(*index*, *col*, *takeable=False*)

Quickly retrieve single value at passed column and index

Deprecated since version 0.21.0: Use .at[] or .iat[] accessors instead.

Parameters

index [row label]

col [column label]

takeable [interpret the index/col as indexers, default False]

Returns

value [scalar value]

get_values()

Return an ndarray after converting sparse values to dense.

This is the same as `.values` for non-sparse data. For sparse data contained in a *pandas.SparseArray*, the data are first converted to a dense representation.

Returns

numpy.ndarray Numpy representation of DataFrame

See also:

values Numpy representation of DataFrame.

pandas.SparseArray Container for sparse data.

Examples

```
>>> df = pd.DataFrame({'a': [1, 2], 'b': [True, False],
...                    'c': [1.0, 2.0]})
>>> df
   a      b    c
0  1   True  1.0
1  2  False  2.0
```

```
>>> df.get_values()
array([[1, True, 1.0], [2, False, 2.0]], dtype=object)
```

```
>>> df = pd.DataFrame({"a": pd.SparseArray([1, None, None]),
...                    "c": [1.0, 2.0, 3.0]})
>>> df
     a    c
0  1.0  1.0
1  NaN  2.0
2  NaN  3.0
```

```
>>> df.get_values()
array([[ 1.,  1.],
       [nan,  2.],
       [nan,  3.]])
```

groupby(*by=None*, *axis=0*, *level=None*, *as_index=True*, *sort=True*, *group_keys=True*, *squeeze=False*, *observed=False*, ***kwargs*)

Group series using mapper (dict or key function, apply given function to group, return result as series) or by a series of columns.

Parameters

by [mapping, function, label, or list of labels] 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 an ndarray is passed, the values are used as-is 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 a (single) key.

axis [int, default 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

as_index [boolean, default True] For aggregated output, return object with group labels as the index. Only relevant for DataFrame input. as_index=False is effectively "SQL-style" grouped output

sort [boolean, 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.

group_keys [boolean, default True] When calling apply, add group keys to index to identify pieces

squeeze [boolean, default False] reduce the dimensionality of the return type if possible, otherwise return a consistent type

observed [boolean, 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.

New in version 0.23.0.

Returns

GroupBy object

See also:

resample Convenience method for frequency conversion and resampling of time series.

Notes

See the user guide for more.

Examples

DataFrame results

```
>>> data.groupby(func, axis=0).mean()
>>> data.groupby(['col1', 'col2'])['col3'].mean()
```

DataFrame with hierarchical index

```
>>> data.groupby(['col1', 'col2']).mean()
```

gt(*other*, *axis='columns'*, *level=None*)

Wrapper for flexible comparison methods gt

head(*n=5*)

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.

Parameters

n [int, default 5] Number of rows to select.

Returns

obj_head [type of caller] The first *n* rows of the caller object.

See also:

pandas.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
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
```

hist(*column=None*, *by=None*, *grid=True*, *xlabelsize=None*, *xrot=None*, *ylabelsize=None*, *yrot=None*, *ax=None*, *sharex=False*, *sharey=False*, *figsize=None*, *layout=None*, *bins=10*, ***kwds*)

Make a histogram of the DataFrame's.

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 [string or sequence] 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 [boolean, 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 [boolean, 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 [boolean, default False] In case subplots=True, share y axis and set some y axis labels to invisible.

figsize [tuple] 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 [integer 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.

****kwds** All other plotting keyword arguments to be passed to `matplotlib.pyplot.hist()`.

Returns

axes [matplotlib.AxesSubplot or numpy.ndarray of them]

See also:

`matplotlib.pyplot.hist` Plot a histogram using matplotlib.

Examples

`iat`

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

Similar to `iloc`, in that both provide integer-based lookups. Use `iat` if you only need to get or set a single value in a DataFrame or Series.

Raises

IndexError When integer position is out of bounds

See also:

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

DataFrame.loc Access a group of rows and columns by label(s)

DataFrame.iloc Access a group of rows and columns by integer position(s)

Examples

```
>>> df = pd.DataFrame([[0, 2, 3], [0, 4, 1], [10, 20, 30]],
...                   columns=['A', 'B', 'C'])
>>> df
    A   B   C
0   0   2   3
1   0   4   1
2  10  20  30
```

Get value at specified row/column pair

```
>>> df.iat[1, 2]
1
```

Set value at specified row/column pair

```
>>> df.iat[1, 2] = 10
>>> df.iat[1, 2]
10
```

Get value within a series

```
>>> df.loc[0].iat[1]
2
```

idxmax (*axis=0*, *skipna=True*)

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] 0 or 'index' for row-wise, 1 or 'columns' for column-wise

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

Returns

idxmax [Series]

Raises

ValueError

- If the row/column is empty

See also:

`Series.idxmax`

Notes

This method is the DataFrame version of `ndarray.argmax`.

idxmin(*axis=0*, *skipna=True*)
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] 0 or 'index' for row-wise, 1 or 'columns' for column-wise

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

Returns

idxmin [Series]

Raises

ValueError

- If the row/column is empty

See also:

`Series.idxmin`

Notes

This method is the DataFrame version of `ndarray.argmin`.

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

`.iloc[]` is primarily integer position based (from `0` to `length-1` of the axis), but may also be used with a boolean array.

Allowed inputs are:

- An integer, e.g. `5`.
- A list or array of integers, e.g. `[4, 3, 0]`.
- A slice object with ints, e.g. `1:7`.
- A boolean array.
- A `callable` function with one argument (the calling Series, DataFrame or Panel) and that returns valid output for indexing (one of the above)

`.iloc` will raise `IndexError` if a requested indexer is out-of-bounds, except *slice* indexers which allow out-of-bounds indexing (this conforms with python/numpy *slice* semantics).

See more at Selection by Position

index
The index (row labels) of the DataFrame.

infer_objects()
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.

New in version 0.21.0.

Returns

converted [same type as input object]

See also:

pandas.to_datetime Convert argument to datetime.

pandas.to_timedelta Convert argument to timedelta.

pandas.to_numeric Convert argument to numeric typeR

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
```

info(*verbose=None*, *buf=None*, *max_cols=None*, *memory_usage=None*, *null_counts=None*)

Print a concise summary of a DataFrame.

This method prints information about a DataFrame including the index dtype and column dtypes, 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.

null_counts [bool, optional] Whether to show the non-null counts. By default, this is shown only if the frame 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):
int_col      5 non-null int64
text_col     5 non-null object
float_col    5 non-null float64
dtypes: float64(1), int64(1), object(1)
memory usage: 200.0+ 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: 200.0+ 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_1    1000000 non-null object
column_2    1000000 non-null object
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_1    1000000 non-null object
column_2    1000000 non-null object
column_3    1000000 non-null object
dtypes: object(3)
memory usage: 188.8 MB
```

insert (*loc*, *column*, *value*, *allow_duplicates=False*)

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 [string, number, or hashable object] label of the inserted column

value [int, Series, or array-like]

allow_duplicates [bool, optional]

interpolate (*method='linear'*, *axis=0*, *limit=None*, *inplace=False*, *limit_direction='forward'*, *limit_area=None*, *downcast=None*, ***kwargs*)

Interpolate values according to different methods.

Please note that only `method='linear'` is supported for DataFrames/Series with a MultiIndex.

Parameters

method [{'linear', 'time', 'index', 'values', 'nearest', 'zero',]

'slinear', 'quadratic', 'cubic', 'barycentric', 'krogh', 'polynomial', 'spline', 'piecewise_polynomial', 'from_derivatives', 'pchip', 'akima'}

- 'linear': ignore the index and treat the values as equally spaced. This is the only method supported on MultiIndexes. default
- 'time': interpolation works on daily and higher resolution data to interpolate given length of interval
- 'index', 'values': use the actual numerical values of the index
- 'nearest', 'zero', 'slinear', 'quadratic', 'cubic', 'barycentric', 'polynomial' is passed to `scipy.interpolate.interp1d`. Both 'polynomial' and 'spline' require that you also specify an *order* (int), e.g. df.interpolate(method='polynomial', order=4). These use the actual numerical values of the index.
- 'krogh', 'piecewise_polynomial', 'spline', 'pchip' and 'akima' are all wrappers around the scipy interpolation methods of similar names. These use the actual numerical values of the index. For more information on their behavior, see the scipy documentation and tutorial documentation
- 'from_derivatives' refers to BPoly.from_derivatives which replaces 'piecewise_polynomial' interpolation method in scipy 0.18

New in version 0.18.1: Added support for the 'akima' method Added interpolate method 'from_derivatives' which replaces 'piecewise_polynomial' in scipy 0.18; backwards-compatible with scipy < 0.18

axis [{0, 1}, default 0]

- 0: fill column-by-column
- 1: fill row-by-row

limit [int, default None.] Maximum number of consecutive NaNs to fill. Must be greater than 0.

limit_direction [{'forward', 'backward', 'both'}, default 'forward']

limit_area [{'inside', 'outside'}, default None]

- None: (default) no fill restriction
- 'inside' Only fill NaNs surrounded by valid values (interpolate).
- 'outside' Only fill NaNs outside valid values (extrapolate).

If limit is specified, consecutive NaNs will be filled in this direction.

New in version 0.21.0.

inplace [bool, default False] Update the NDFrame in place if possible.

downcast [optional, 'infer' or None, defaults to None] Downcast dtypes if possible.

kwargs [keyword arguments to pass on to the interpolating function.]

Returns

Series or DataFrame of same shape interpolated at the NaNs

See also:

reindex, *replace*, *fillna*

Examples

Filling in NaNs

```
>>> s = pd.Series([0, 1, np.nan, 3])
>>> s.interpolate()
0    0
1    1
2    2
3    3
dtype: float64
```

is_copy

isin(*values*)

Return boolean DataFrame showing whether each element in the DataFrame is contained in values.

Parameters

values [iterable, Series, DataFrame or dictionary] 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 dictionary, 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 of booleans

Examples

When `values` is a list:

```
>>> df = pd.DataFrame({'A': [1, 2, 3], 'B': ['a', 'b', 'f']})
>>> df.isin([1, 3, 12, 'a'])
       A      B
0   True   True
1  False  False
2   True  False
```

When `values` is a dict:

```
>>> df = pd.DataFrame({'A': [1, 2, 3], 'B': [1, 4, 7]})
>>> df.isin({'A': [1, 3], 'B': [4, 7, 12]})
       A      B
0   True  False  # Note that B didn't match the 1 here.
1  False   True
2   True   True
```

When `values` is a Series or DataFrame:

```
>>> df = pd.DataFrame({'A': [1, 2, 3], 'B': ['a', 'b', 'f']})
>>> other = DataFrame({'A': [1, 3, 3, 2], 'B': ['e', 'f', 'f', 'e']})
>>> df.isin(other)
       A      B
0   True  False
1  False  False  # Column A in `other` has a 3, but not at index 1.
2   True   True
```

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 not 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({'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
```

isnull()

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 not 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({'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
```

items()
Iterator over (column name, Series) pairs.

See also:

iterrows Iterate over DataFrame rows as (index, Series) pairs.

itertuples Iterate over DataFrame rows as namedtuples of the values.

iteritems()
Iterator over (column name, Series) pairs.

See also:

iterrows Iterate over DataFrame rows as (index, Series) pairs.

itertuples Iterate over DataFrame rows as namedtuples of the values.

iterrows()
Iterate over DataFrame rows as (index, Series) pairs.

Returns

it [generator] A generator that iterates over the rows of the frame.

See also:

itertuples Iterate over DataFrame rows as namedtuples of the values.

iteritems 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). For example,

```
>>> 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
```

   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.

itertuples(*index=True*, *name='Pandas'*)
Iterate over DataFrame rows as namedtuples, with index value as first element of the tuple.

Parameters

index [boolean, default True] If True, return the index as the first element of the tuple.

name [string, default "Pandas"] The name of the returned namedtuples or None to return regular tuples.

See also:

iterrows Iterate over DataFrame rows as (index, Series) pairs.

iteritems 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. With a large number of columns (>255), regular tuples are returned.

Examples

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [0.1, 0.2]},
                      index=['a', 'b'])
>>> df
   col1  col2
a     1   0.1
b     2   0.2
>>> for row in df.itertuples():
...     print(row)
...
Pandas(Index='a', col1=1, col2=0.10000000000000001)
Pandas(Index='b', col1=2, col2=0.20000000000000001)
```

ix

A primarily label-location based indexer, with integer position fallback.

Warning: Starting in 0.20.0, the .ix indexer is deprecated, in favor of the more strict .iloc and .loc indexers.

`.ix[]` supports mixed integer and label based access. It is primarily label based, but will fall back to integer positional access unless the corresponding axis is of integer type.

`.ix` is the most general indexer and will support any of the inputs in `.loc` and `.iloc`. `.ix` also supports floating point label schemes. `.ix` is exceptionally useful when dealing with mixed positional and label based hierarchical indexes.

However, when an axis is integer based, ONLY label based access and not positional access is supported. Thus, in such cases, it's usually better to be explicit and use `.iloc` or `.loc`.

See more at Advanced Indexing.

join(*other*, *on=None*, *how='left'*, *lsuffix=''*, *rsuffix=''*, *sort=False*)

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 with name field set, or list of DataFrame] 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 [name, tuple/list of names, or array-like] 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'}, 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 frame's index
- outer: form union of calling frame's index (or column if on is specified) with other frame's index, and sort it lexicographically
- inner: form intersection of calling frame's index (or column if on is specified) with other frame's index, preserving the order of the calling's one

lsuffix [string] Suffix to use from left frame's overlapping columns

rsuffix [string] Suffix to use from right frame's overlapping columns

sort [boolean, 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)

Returns

joined [DataFrame]

See also:

`DataFrame.merge` For column(s)-on-columns(s) operations

Notes

on, lsuffix, and rsuffix options are not supported when passing a list of DataFrame objects

Support for specifying index levels as the *on* parameter was added in version 0.23.0

Examples

```
>>> caller = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3', 'K4', 'K5'],
...                        'A': ['A0', 'A1', 'A2', 'A3', 'A4', 'A5']})
```

```
>>> caller
    A key
0  A0  K0
1  A1  K1
2  A2  K2
3  A3  K3
4  A4  K4
5  A5  K5
```

```
>>> other = pd.DataFrame({'key': ['K0', 'K1', 'K2'],
...                       'B': ['B0', 'B1', 'B2']})
```

```
>>> other
    B key
0  B0  K0
1  B1  K1
2  B2  K2
```

Join DataFrames using their indexes.

```
>>> caller.join(other, lsuffix='_caller', rsuffix='_other')
```

```
>>>     A key_caller    B key_other
    0  A0          K0   B0         K0
    1  A1          K1   B1         K1
    2  A2          K2   B2         K2
    3  A3          K3  NaN        NaN
    4  A4          K4  NaN        NaN
    5  A5          K5  NaN        NaN
```

If we want to join using the key columns, we need to set key to be the index in both caller and other. The joined DataFrame will have key as its index.

```
>>> caller.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 the caller. This method preserves the original caller's index in the result.

```
>>> caller.join(other.set_index('key'), on='key')
```

```
>>>     A key    B
    0  A0  K0   B0
    1  A1  K1   B1
    2  A2  K2   B2
    3  A3  K3  NaN
    4  A4  K4  NaN
    5  A5  K5  NaN
```

keys()

Get the 'info axis' (see Indexing for more)

This is index for Series, columns for DataFrame and major_axis for Panel.

kurt(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

Return unbiased kurtosis over requested axis using Fisher's definition of kurtosis (kurtosis of normal == 0.0). Normalized by N-1

Parameters

axis [{index (0), columns (1)}]

skipna [boolean, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

Returns

kurt [Series or DataFrame (if level specified)]

kurtosis (*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

Return unbiased kurtosis over requested axis using Fisher's definition of kurtosis (kurtosis of normal == 0.0). Normalized by N-1

Parameters

axis [{index (0), columns (1)}]

skipna [boolean, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

Returns

kurt [Series or DataFrame (if level specified)]

last (*offset*)

Convenience method for subsetting final periods of time series data based on a date offset.

Parameters

offset [string, DateOffset, dateutil.relativedelta]

Returns

subset [type of 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

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
```

(continues on next page)

(continued from previous page)

```
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 calender days were returned, not the last 3 observed days in the dataset, and therefore data for 2018-04-11 was not returned.

last_valid_index()
Return index for last non-NA/null value.

Returns

scalar [type of index]

Notes

If all elements are non-NA/null, returns None. Also returns None for empty NDFrame.

le(*other*, *axis='columns'*, *level=None*)
Wrapper for flexible comparison methods le

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

`.loc[]` is primarily label based, but may also be used with a boolean array.

Allowed inputs are:

- A single label, e.g. `5` or `'a'`, (note that `5` is interpreted as a *label* of the index, and **never** as an integer position along the index).
- A list or array of labels, e.g. `['a', 'b', 'c']`.
- A slice object with labels, e.g. `'a':'f'`.

> **Warning:** Note that contrary to usual python slices, **both** the start and the stop are included

- A boolean array of the same length as the axis being sliced, e.g. `[True, False, True]`.
- A `callable` function with one argument (the calling Series, DataFrame or Panel) and that returns valid output for indexing (one of the above)

See more at Selection by Label

Raises

KeyError: when any items are not found

See also:

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

DataFrame.iloc Access group of rows and columns by integer position(s)

DataFrame.xs Returns a cross-section (row(s) or column(s)) from the Series/DataFrame.

Series.loc Access group of values using labels

Examples

Getting values

```
>>> df = pd.DataFrame([[1, 2], [4, 5], [7, 8]],
...       index=['cobra', 'viper', 'sidewinder'],
...       columns=['max_speed', 'shield'])
>>> df
            max_speed  shield
cobra               1       2
viper               4       5
sidewinder          7       8
```

Single label. Note this returns the row as a Series.

```
>>> df.loc['viper']
max_speed    4
shield       5
Name: viper, dtype: int64
```

List of labels. Note using `[[]]` returns a DataFrame.

```
>>> df.loc[['viper', 'sidewinder']]
            max_speed  shield
viper               4       5
sidewinder          7       8
```

Single label for row and column

```
>>> df.loc['cobra', 'shield']
2
```

Slice with labels for row and single label for column. As mentioned above, note that both the start and stop of the slice are included.

```
>>> df.loc['cobra':'viper', 'max_speed']
cobra    1
viper    4
Name: max_speed, dtype: int64
```

Boolean list with the same length as the row axis

```
>>> df.loc[[False, False, True]]
            max_speed  shield
sidewinder          7       8
```

Conditional that returns a boolean Series

```
>>> df.loc[df['shield'] > 6]
            max_speed  shield
sidewinder          7       8
```

Conditional that returns a boolean Series with column labels specified

```
>>> df.loc[df['shield'] > 6, ['max_speed']]
            max_speed
sidewinder          7
```

Callable that returns a boolean Series

```
>>> df.loc[lambda df: df['shield'] == 8]
            max_speed  shield
sidewinder          7       8
```

Setting values

Set value for all items matching the list of labels

```
>>> df.loc[['viper', 'sidewinder'], ['shield']] = 50
>>> df
            max_speed  shield
cobra               1       2
viper               4      50
sidewinder          7      50
```

Set value for an entire row

```
>>> df.loc['cobra'] = 10
>>> df
            max_speed  shield
cobra              10      10
viper               4      50
sidewinder          7      50
```

Set value for an entire column

```
>>> df.loc[:, 'max_speed'] = 30
>>> df
            max_speed  shield
cobra              30      10
viper              30      50
sidewinder         30      50
```

Set value for rows matching callable condition

```
>>> df.loc[df['shield'] > 35] = 0
>>> df
            max_speed  shield
cobra              30      10
viper               0       0
sidewinder          0       0
```

Getting values on a DataFrame with an index that has integer labels

Another example using integers for the index

```
>>> df = pd.DataFrame([[1, 2], [4, 5], [7, 8]],
...       index=[7, 8, 9], columns=['max_speed', 'shield'])
>>> df
   max_speed  shield
7          1       2
```

(continues on next page)

(continued from previous page)

```
8           4        5
9           7        8
```

Slice with integer labels for rows. As mentioned above, note that both the start and stop of the slice are included.

```
>>> df.loc[7:9]
   max_speed  shield
7          1       2
8          4       5
9          7       8
```

Getting values with a MultiIndex

A number of examples using a DataFrame with a MultiIndex

```
>>> tuples = [
...     ('cobra', 'mark i'), ('cobra', 'mark ii'),
...     ('sidewinder', 'mark i'), ('sidewinder', 'mark ii'),
...     ('viper', 'mark ii'), ('viper', 'mark iii')
... ]
>>> index = pd.MultiIndex.from_tuples(tuples)
>>> values = [[12, 2], [0, 4], [10, 20],
...           [1, 4], [7, 1], [16, 36]]
>>> df = pd.DataFrame(values, columns=['max_speed', 'shield'], index=index)
>>> df
                     max_speed  shield
cobra      mark i           12       2
           mark ii           0       4
sidewinder mark i           10      20
           mark ii           1       4
viper      mark ii           7       1
           mark iii         16      36
```

Single label. Note this returns a DataFrame with a single index.

```
>>> df.loc['cobra']
         max_speed  shield
mark i          12       2
mark ii          0       4
```

Single index tuple. Note this returns a Series.

```
>>> df.loc[('cobra', 'mark ii')]
max_speed    0
shield       4
Name: (cobra, mark ii), dtype: int64
```

Single label for row and column. Similar to passing in a tuple, this returns a Series.

```
>>> df.loc['cobra', 'mark i']
max_speed    12
shield        2
Name: (cobra, mark i), dtype: int64
```

Single tuple. Note using `[[]]` returns a DataFrame.

```
>>> df.loc[[('cobra', 'mark ii')]]
               max_speed  shield
cobra mark ii          0       4
```

Single tuple for the index with a single label for the column

```
>>> df.loc[('cobra', 'mark i'), 'shield']
2
```

Slice from index tuple to single label

```
>>> df.loc[('cobra', 'mark i'):'viper']
                     max_speed  shield
cobra      mark i           12       2
           mark ii           0       4
sidewinder mark i           10      20
           mark ii           1       4
viper      mark ii           7       1
           mark iii         16      36
```

Slice from index tuple to index tuple

```
>>> df.loc[('cobra', 'mark i'):('viper', 'mark ii')]
                   max_speed  shield
cobra      mark i          12       2
           mark ii          0       4
sidewinder mark i          10      20
           mark ii          1       4
viper      mark ii          7       1
```

lookup (*row_labels*, *col_labels*)

Label-based "fancy indexing" function for DataFrame. Given equal-length arrays of row and column labels, return an array of the values corresponding to each (row, col) pair.

Parameters

row_labels [sequence] The row labels to use for lookup

col_labels [sequence] The column labels to use for lookup

Notes

Akin to:

```
result = []
for row, col in zip(row_labels, col_labels):
    result.append(df.get_value(row, col))
```

Examples

values [ndarray] The found values

lt (*other*, *axis='columns'*, *level=None*)

Wrapper for flexible comparison methods lt

mad(*axis=None*, *skipna=None*, *level=None*)
Return the mean absolute deviation of the values for the requested axis

Parameters

axis [{index (0), columns (1)}]

skipna [boolean, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

Returns

mad [Series or DataFrame (if level specified)]

mask(*cond*, *other=nan*, *inplace=False*, *axis=None*, *level=None*, *errors='raise'*, *try_cast=False*, *raise_on_error=None*)
Return an object of same shape as self and whose corresponding entries are from self where *cond* is False and otherwise are from *other*.

Parameters

cond [boolean NDFrame, 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 NDFrame and should return boolean NDFrame or array. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1: A callable can be used as cond.

other [scalar, NDFrame, or callable] Entries where *cond* is True are replaced with corresponding value from *other*. If other is callable, it is computed on the NDFrame and should return scalar or NDFrame. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1: A callable can be used as other.

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

axis [alignment axis if needed, default None]

level [alignment level if needed, default None]

errors [str, {'raise', 'ignore'}, default 'raise']

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

Note that currently this parameter won't affect the results and will always coerce to a suitable dtype.

try_cast [boolean, default False] try to cast the result back to the input type (if possible),

raise_on_error [boolean, default True] Whether to raise on invalid data types (e.g. trying to where on strings)

Deprecated since version 0.21.0.

Returns

wh [same type as caller]

See also:

`DataFrame.where()`

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.

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.

Examples

```
>>> s = pd.Series(range(5))
>>> s.where(s > 0)
0    NaN
1    1.0
2    2.0
3    3.0
4    4.0
```

```
>>> s.mask(s > 0)
0    0.0
1    NaN
2    NaN
3    NaN
4    NaN
```

```
>>> s.where(s > 1, 10)
0    10.0
1    10.0
2    2.0
3    3.0
4    4.0
```

```
>>> df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])
>>> 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
```

(continues on next page)

(continued from previous page)

```
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
```

max (*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

This method returns the maximum of the values in the object. 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)}]

skipna [boolean, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

Returns

max [Series or DataFrame (if level specified)]

mean (*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

Return the mean of the values for the requested axis

Parameters

axis [{index (0), columns (1)}]

skipna [boolean, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

Returns

mean [Series or DataFrame (if level specified)]

median (*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

Return the median of the values for the requested axis

Parameters

axis [{index (0), columns (1)}]

skipna [boolean, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

Returns

median [Series or DataFrame (if level specified)]

melt (*id_vars=None*, *value_vars=None*, *var_name=None*, *value_name='value'*, *col_level=None*)

"Unpivots" a DataFrame from wide format to long format, optionally leaving identifier variables 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'.

New in version 0.20.0.

Parameters

frame [DataFrame]

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 string, optional] If columns are a MultiIndex then use this level to melt.

See also:

melt, *pivot_table*, `DataFrame.pivot`

Examples

```
>>> import pandas as pd
>>> 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
```

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
```

memory_usage(*index=True*, *deep=False*)

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 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

sizes [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.

pandas.Categorical Memory-efficient array for string values with many repeated values.

DataFrame.info Concise summary of a DataFrame.

Examples

```
>>> dtypes = ['int64', 'float64', 'complex128', 'object', 'bool']
>>> data = dict([(t, np.ones(shape=5000).astype(t))
...              for t in dtypes])
>>> df = pd.DataFrame(data)
>>> df.head()
   int64  float64  complex128 object  bool
0      1      1.0      (1+0j)      1  True
1      1      1.0      (1+0j)      1  True
2      1      1.0      (1+0j)      1  True
3      1      1.0      (1+0j)      1  True
4      1      1.0      (1+0j)      1  True
```

```
>>> df.memory_usage()
Index            80
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            80
int64         40000
float64       40000
complex128    80000
object       160000
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)
5168
```

merge(*right*, *how='inner'*, *on=None*, *left_on=None*, *right_on=None*, *left_index=False*, *right_index=False*, *sort=False*, *suffixes=('_x', '_y')*, *copy=True*, *indicator=False*, *validate=None*)

Merge DataFrame objects by performing a database-style join operation by 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.

Parameters

right [DataFrame]

how [{'left', 'right', 'outer', 'inner'}, default 'inner']

- 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

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 [boolean, 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 [boolean, default False] Use the index from the right DataFrame as the join key. Same caveats as left_index

sort [boolean, 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 [2-length sequence (tuple, list, ...)] Suffix to apply to overlapping column names in the left and right side, respectively

copy [boolean, default True] If False, do not copy data unnecessarily

indicator [boolean or string, default False] If True, adds a column to output DataFrame called "_merge" with information on the source of each row. If string, column with information on source of each row will be added to output DataFrame, and column will be named value of string. Information column is Categorical-type and takes on a value of "left_only" for observations whose merge key only appears in 'left' DataFrame, "right_only" for observations whose merge key only appears in 'right' DataFrame, and "both" if the observation's merge key is found in both.

validate [string, default None] 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.

New in version 0.21.0.

Returns

merged [DataFrame] The output type will the be same as 'left', if it is a subclass of DataFrame.

See also:

`merge_ordered`, `merge_asof`, `DataFrame.join`

Notes

Support for specifying index levels as the *on*, *left_on*, and *right_on* parameters was added in version 0.23.0

Examples

```
>>> A              >>> B
    lkey value         rkey value
0   foo  1         0   foo  5
1   bar  2         1   bar  6
2   baz  3         2   qux  7
3   foo  4         3   bar  8
```

```
>>> A.merge(B, left_on='lkey', right_on='rkey', how='outer')
   lkey  value_x  rkey  value_y
0  foo   1        foo   5
1  foo   4        foo   5
2  bar   2        bar   6
3  bar   2        bar   8
4  baz   3        NaN   NaN
5  NaN   NaN      qux   7
```

min (*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

This method returns the minimum of the values in the object. 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)}]

skipna [boolean, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

Returns

min [Series or DataFrame (if level specified)]

mod (*other*, *axis='columns'*, *level=None*, *fill_value=None*)

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.

Parameters

other [Series, DataFrame, or constant]

axis [{0, 1, 'index', 'columns'}] For Series input, axis to match Series index on

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

fill_value [None or float value, 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

result [DataFrame]

See also:

`DataFrame.rmod`

Notes

Mismatched indices will be unioned together

Examples

None

mode (*axis=0*, *numeric_only=False*)

Gets the mode(s) of each element along the axis selected. Adds a row for each mode per label, fills in gaps with nan.

Note that there could be multiple values returned for the selected axis (when more than one item share the maximum frequency), which is the reason why a dataframe is returned. If you want to impute missing values with the mode in a dataframe `df`, you can just do this: `df.fillna(df.mode().iloc[0])`

Parameters

axis [{0 or 'index', 1 or 'columns'}, default 0]

- 0 or 'index' : get mode of each column
- 1 or 'columns' : get mode of each row

numeric_only [boolean, default False] if True, only apply to numeric columns

Returns

modes [DataFrame (sorted)]

Examples

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

mul(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
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.

Parameters

other [Series, DataFrame, or constant]

axis [{0, 1, 'index', 'columns'}] For Series input, axis to match Series index on

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

fill_value [None or float value, 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

result [DataFrame]

See also:

`DataFrame.rmul`

Notes

Mismatched indices will be unioned together

Examples

None

multiply(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
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.

Parameters

other [Series, DataFrame, or constant]

axis [{0, 1, 'index', 'columns'}] For Series input, axis to match Series index on

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

fill_value [None or float value, 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

result [DataFrame]

See also:

`DataFrame.rmul`

Notes

Mismatched indices will be unioned together

Examples

None

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

Return 1 if Series. Otherwise return 2 if DataFrame.

See also:

`ndarray.ndim`

Examples

```
>>> s = pd.Series({'a': 1, 'b': 2, 'c': 3})
>>> s.ndim
1
```

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df.ndim
2
```

ne (*other*, *axis='columns'*, *level=None*)
Wrapper for flexible comparison methods ne

nlargest (*n*, *columns*, *keep='first'*)
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'}, default 'first'] Where there are duplicate values:

- *first* : prioritize the first occurrence(s)
- *last* : prioritize the last occurrence(s)

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({'a': [1, 10, 8, 10, -1],
...                    'b': list('abdce'),
...                    'c': [1.0, 2.0, np.nan, 3.0, 4.0]})
>>> df
    a  b    c
0   1  a  1.0
1  10  b  2.0
2   8  d  NaN
3  10  c  3.0
4  -1  e  4.0
```

In the following example, we will use `nlargest` to select the three rows having the largest values in column "a".

```
>>> df.nlargest(3, 'a')
    a  b    c
1  10  b  2.0
3  10  c  3.0
2   8  d  NaN
```

When using `keep='last'`, ties are resolved in reverse order:

```
>>> df.nlargest(3, 'a', keep='last')
    a  b    c
3  10  c  3.0
1  10  b  2.0
2   8  d  NaN
```

To order by the largest values in column "a" and then "c", we can specify multiple columns like in the next example.

```
>>> df.nlargest(3, ['a', 'c'])
    a  b    c
3  10  c  3.0
1  10  b  2.0
2   8  d  NaN
```

Attempting to use `nlargest` on non-numeric dtypes will raise a `TypeError`:

```
>>> df.nlargest(3, 'b')
Traceback (most recent call last):
TypeError: Column 'b' has dtype object, cannot use method 'nlargest'
```

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({'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
```

`notnull`()
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({'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
```

`nsmallest`(*n*, *columns*, *keep='first'*)

Get the rows of a DataFrame sorted by the *n* smallest values of *columns*.

Parameters

n [int] Number of items to retrieve

columns [list or str] Column name or names to order by

keep [{'first', 'last'}, default 'first'] Where there are duplicate values: - `first` : take the first occurrence. - `last` : take the last occurrence.

Returns

DataFrame

Examples

```
>>> df = pd.DataFrame({'a': [1, 10, 8, 11, -1],
...                    'b': list('abdce'),
...                    'c': [1.0, 2.0, np.nan, 3.0, 4.0]})
>>> df.nsmallest(3, 'a')
   a  b   c
4 -1  e   4
0  1  a   1
2  8  d NaN
```

nunique (*axis=0*, *dropna=True*)

Return Series with number of distinct observations over requested axis.

New in version 0.20.0.

Parameters

axis [{0 or 'index', 1 or 'columns'}, default 0]

dropna [boolean, default True] Don't include NaN in the counts.

Returns

nunique [Series]

Examples

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

```
>>> df.nunique(axis=1)
0    1
1    2
2    2
```

pct_change (*periods=1*, *fill_method='pad'*, *limit=None*, *freq=None*, ***kwargs*)

Percentage change between the current and a prior element.

Computes the percentage change from the immediately previous row by default. This is useful in comparing the percentage of change in a time series of elements.

Parameters

periods [int, default 1] Periods to shift for forming percent change.

fill_method [str, default 'pad'] How to handle NAs before computing percent changes.

limit [int, default None] The number of consecutive NAs to fill before stopping.

freq [DateOffset, timedelta, or offset alias string, 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

chg [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
2     NaN
3    85.0
dtype: float64
```

```
>>> s.pct_change(fill_method='ffill')
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')
      2016      2015      2014
GOOG   NaN -0.151997 -0.086016
APPL   NaN  0.337604  0.012002
```

pipe (*func*, **args*, ***kwargs*)

Apply func(self, *args, **kwargs)

Parameters

func [function] function to apply to the NDFrame. `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 NDFrame.

args [iterable, optional] positional arguments passed into `func`.

kwargs [mapping, optional] a dictionary of keyword arguments passed into `func`.

Returns

object [the return type of `func`.]

See also:

`pandas.DataFrame.apply`, `pandas.DataFrame.applymap`, `pandas.Series.map`

Notes

Use `.pipe` when chaining together functions that expect Series, DataFrames or GroupBy objects. Instead of writing

```
>>> f(g(h(df), arg1=a), arg2=b, arg3=c)
```

You can write

```
>>> (df.pipe(h)
...     .pipe(g, arg1=a)
...     .pipe(f, arg2=b, arg3=c)
... )
```

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 `f` takes its data as `arg2`:

```
>>> (df.pipe(h)
...     .pipe(g, arg1=a)
...     .pipe((f, 'arg2'), arg1=a, arg3=c)
...  )
```

pivot (*index=None*, *columns=None*, *values=None*)

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

index [string or object, optional] Column to use to make new frame's index. If None, uses existing index.

columns [string or object] Column to use to make new frame's columns.

values [string, 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.

Changed in version 0.23.0: Also accept list of column names.

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.

Notes

For finer-tuned control, see hierarchical indexing documentation along with the related stack/unstack methods.

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
```

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
```

(continues on next page)

(continued from previous page)

```
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
```

pivot_table(*values=None*, *index=None*, *columns=None*, *aggfunc='mean'*, *fill_value=None*, *margins=False*, *dropna=True*, *margins_name='All'*)

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 [column to aggregate, optional]

index [column, Grouper, array, or list of the previous] If an array is passed, it must be the same length as the data. The list can contain any of the other types (except list). Keys to group by on the pivot table index. If an array is passed, it is being used as the same manner as column values.

columns [column, Grouper, array, or list of the previous] If an array is passed, it must be the same length as the data. The list can contain any of the other types (except list). Keys to group by on the pivot table column. If an array is passed, it is being used as the same manner as column values.

aggfunc [function, list of functions, dict, default numpy.mean] If list of functions passed, the resulting pivot table will have hierarchical columns whose top level are the function names (inferred from the function objects themselves) If dict is passed, the key is column to aggregate and value is function or list of functions

fill_value [scalar, default None] Value to replace missing values with

margins [boolean, default False] Add all row / columns (e.g. for subtotal / grand totals)

dropna [boolean, default True] Do not include columns whose entries are all NaN

margins_name [string, default 'All'] Name of the row / column that will contain the totals when margins is True.

Returns

table [DataFrame]

See also:

DataFrame.pivot pivot without aggregation that can handle non-numeric data

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",
```

(continues on next page)

(continued from previous page)

```
...                            "small", "large", "small", "small",
...                            "large"],
...                     "D": [1, 2, 2, 3, 3, 4, 5, 6, 7]})
>>> df
     A    B      C  D
0  foo  one  small  1
1  foo  one  large  2
2  foo  one  large  2
3  foo  two  small  3
4  foo  two  small  3
5  bar  one  large  4
6  bar  one  small  5
7  bar  two  small  6
8  bar  two  large  7
```

```
>>> table = pivot_table(df, values='D', index=['A', 'B'],
...                     columns=['C'], aggfunc=np.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
```

```
>>> table = pivot_table(df, values='D', index=['A', 'B'],
...                     columns=['C'], aggfunc=np.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
```

```
>>> table = pivot_table(df, values=['D', 'E'], index=['A', 'C'],
...                     aggfunc={'D': np.mean,
...                              'E': [min, max, np.mean]})
>>> table
                  D   E
               mean max median min
A   C
bar large  5.500000  16   14.5  13
    small  5.500000  15   14.5  14
foo large  2.000000  10    9.5   9
    small  2.333333  12   11.0   8
```

plot
alias of `pandas.plotting._core.FramePlotMethods`

pop (*item*)
Return item and drop from frame. Raise KeyError if not found.

Parameters

item [str] Column label to be popped

Returns

popped [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
```

pow(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

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.

Parameters

other [Series, DataFrame, or constant]

axis [{0, 1, 'index', 'columns'}] For Series input, axis to match Series index on

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

fill_value [None or float value, 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

result [DataFrame]

See also:

`DataFrame.rpow`

Notes

Mismatched indices will be unioned together

Examples

None

prod(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, *min_count=0*, ***kwargs*)
Return the product of the values for the requested axis

Parameters

axis [{index (0), columns (1)}]

skipna [boolean, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. 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.

New in version 0.22.0: Added with the default being 0. This means the sum of an all-NA or empty Series is 0, and the product of an all-NA or empty Series is 1.

Returns

prod [Series or DataFrame (if level specified)]

Examples

By default, the product of an empty or all-NA Series is `1`

```
>>> pd.Series([]).prod()
1.0
```

This can be controlled with the `min_count` parameter

```
>>> pd.Series([]).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
```

product(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, *min_count=0*, ***kwargs*)
Return the product of the values for the requested axis

Parameters

axis [{index (0), columns (1)}]

skipna [boolean, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. 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.

New in version 0.22.0: Added with the default being 0. This means the sum of an all-NA or empty Series is 0, and the product of an all-NA or empty Series is 1.

Returns

prod [Series or DataFrame (if level specified)]

Examples

By default, the product of an empty or all-NA Series is `1`

```
>>> pd.Series([]).prod()
1.0
```

This can be controlled with the `min_count` parameter

```
>>> pd.Series([]).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
```

quantile(*q=0.5*, *axis=0*, *numeric_only=True*, *interpolation='linear'*)

Return values at the given quantile over requested axis, a la numpy.percentile.

Parameters

q [float or array-like, default 0.5 (50% quantile)] 0 <= q <= 1, the quantile(s) to compute

axis [{0, 1, 'index', 'columns'} (default 0)] 0 or 'index' for row-wise, 1 or 'columns' for column-wise

numeric_only [boolean, default True] If False, the quantile of datetime and timedelta data will be computed as well

interpolation [{'linear', 'lower', 'higher', 'midpoint', 'nearest'}] New in version 0.18.0.

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.

Returns

quantiles [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:

`pandas.core.window.Rolling.quantile`

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
dtype: float64
>>> df.quantile([.1, .5])
       a     b
0.1  1.3   3.7
0.5  2.5  55.0
```

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
```

query(*expr*, *inplace=False*, ***kwargs*)

Query the columns of a frame with a boolean expression.

Parameters

expr [string] The query string to evaluate. You can refer to variables in the environment by prefixing them with an '@' character like `@a + b`.

inplace [bool] Whether the query should modify the data in place or return a modified copy

New in version 0.18.0.

kwargs [dict] See the documentation for `pandas.eval()` for complete details on the keyword arguments accepted by `DataFrame.query()`.

Returns

q [DataFrame]

See also:

`pandas.eval`, `DataFrame.eval`

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 `pandas.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.

Examples

```
>>> from numpy.random import randn
>>> from pandas import DataFrame
>>> df = pd.DataFrame(randn(10, 2), columns=list('ab'))
>>> df.query('a > b')
>>> df[df.a > df.b]  # same result as the previous expression
```

radd(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

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.

Parameters

other [Series, DataFrame, or constant]

axis [{0, 1, 'index', 'columns'}] For Series input, axis to match Series index on

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

fill_value [None or float value, 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

result [DataFrame]

See also:

`DataFrame.add`

Notes

Mismatched indices will be unioned together

Examples

```
>>> a = pd.DataFrame([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'],
...                  columns=['one'])
>>> a
   one
a  1.0
b  1.0
c  1.0
d  NaN
>>> b = pd.DataFrame(dict(one=[1, np.nan, 1, np.nan],
...                       two=[np.nan, 2, np.nan, 2]),
...                  index=['a', 'b', 'd', 'e'])
>>> b
   one  two
a  1.0  NaN
b  NaN  2.0
d  1.0  NaN
e  NaN  2.0
>>> a.add(b, fill_value=0)
   one  two
a  2.0  NaN
b  1.0  2.0
c  1.0  NaN
d  1.0  NaN
e  NaN  2.0
```

rank (*axis=0*, *method='average'*, *numeric_only=None*, *na_option='keep'*, *ascending=True*, *pct=False*)

Compute numerical data ranks (1 through n) along axis. 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

method [{'average', 'min', 'max', 'first', 'dense'}]

- average: average rank of group
- min: lowest rank in group
- max: highest rank in group
- first: ranks assigned in order they appear in the array
- dense: like 'min', but rank always increases by 1 between groups

numeric_only [boolean, default None] Include only float, int, boolean data. Valid only for DataFrame or Panel objects

na_option [{'keep', 'top', 'bottom'}]

- keep: leave NA values where they are
- top: smallest rank if ascending
- bottom: smallest rank if descending

ascending [boolean, default True] False for ranks by high (1) to low (N)

pct [boolean, default False] Computes percentage rank of data

Returns

ranks [same type as caller]

rdiv(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

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.

Parameters

other [Series, DataFrame, or constant]

axis [{0, 1, 'index', 'columns'}] For Series input, axis to match Series index on

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

fill_value [None or float value, 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

result [DataFrame]

See also:

`DataFrame.truediv`

Notes

Mismatched indices will be unioned together

Examples

None

reindex(***kwargs*)

Conform DataFrame to new index with optional filling logic, placing 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, columns [array-like, optional (should be specified using keywords)] New labels / index to conform to. 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'}, optional] method to use for filling holes in reindexed DataFrame. Please note: this is only applicable to DataFrames/Series with a monotonically increasing/decreasing index.

- 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 [boolean, 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.

New in version 0.21.0: (list-like tolerance)

Returns

reindexed [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
```

(continues on next page)

(continued from previous page)

```
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
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
```

(continues on next page)

(continued from previous page)

```
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
2010-01-02     101
2010-01-03     NaN
2010-01-04     100
2010-01-05      89
2010-01-06      88
```

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
2010-01-02     101
2010-01-03     NaN
2010-01-04     100
2010-01-05      89
2010-01-06      88
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 backpropagate 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
2009-12-30     100
2009-12-31     100
2010-01-01     100
2010-01-02     101
2010-01-03     NaN
2010-01-04     100
2010-01-05      89
2010-01-06      88
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.

reindex_axis(*labels*, *axis=0*, *method=None*, *level=None*, *copy=True*, *limit=None*, *fill_value=nan*)
Conform input object to new index with optional filling logic, placing 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] New labels / index to conform to. Preferably an Index object to avoid duplicating data

axis [{0 or 'index', 1 or 'columns'}]

method [{None, 'backfill'/'bfill', 'pad'/'ffill', 'nearest'}, optional] Method to use for filling holes in reindexed DataFrame:

- 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 [boolean, 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

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.

New in version 0.21.0: (list-like tolerance)

Returns

reindexed [DataFrame]

See also:

reindex, *reindex_like*

Examples

```
>>> df.reindex_axis(['A', 'B', 'C'], axis=1)
```

reindex_like(*other*, *method=None*, *copy=True*, *limit=None*, *tolerance=None*)
Return an object with matching indices to myself.

Parameters

other [Object]

method [string or None]

copy [boolean, default True]

limit [int, default None] Maximum number of consecutive labels to fill for inexact matches.

tolerance [optional] Maximum distance between labels of the other object and this object for inexact matches. Can be list-like.

New in version 0.21.0: (list-like tolerance)

Returns

reindexed [same as input]

Notes

Like calling s.reindex(index=other.index, columns=other.columns, method=...)

rename(***kwargs*)

Alter axes 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, index, columns [dict-like or function, optional] dict-like or functions transformations to apply to that axis' values. Use either `mapper` and `axis` to specify the axis to target with `mapper`, or `index` and `columns`.

axis [int or str, optional] Axis to target with `mapper`. Can be either the axis name ('index', 'columns') or number (0, 1). The default is 'index'.

copy [boolean, default True] Also copy underlying data

inplace [boolean, default False] Whether to return a new DataFrame. 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.

Returns

renamed [DataFrame]

See also:

`pandas.DataFrame.rename_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.

```
>>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
>>> df.rename(index=str, columns={"A": "a", "B": "c"})
   a  c
0  1  4
1  2  5
2  3  6
```

```
>>> df.rename(index=str, columns={"A": "a", "C": "c"})
   a  B
0  1  4
1  2  5
2  3  6
```

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
```

rename_axis (*mapper*, *axis=0*, *copy=True*, *inplace=False*)

Alter the name of the index or columns.

Parameters

mapper [scalar, list-like, optional] Value to set as the axis name attribute.

axis [{0 or 'index', 1 or 'columns'}, default 0] The index or the name of the axis.

copy [boolean, default True] Also copy underlying data.

inplace [boolean, default False] Modifies the object directly, instead of creating a new Series or DataFrame.

Returns

renamed [Series, DataFrame, or None] The same type as the caller or None if *inplace* is True.

See also:

pandas.Series.rename Alter Series index labels or name

pandas.DataFrame.rename Alter DataFrame index labels or name

pandas.Index.rename Set new names on index

Notes

Prior to version 0.21.0, `rename_axis` could also be used to change the axis *labels* by passing a mapping or scalar. This behavior is deprecated and will be removed in a future version. Use `rename` instead.

Examples

Series

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

DataFrame

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

```
>>> df.rename_axis("bar", axis="columns")
bar  A  B
0    1  4
1    2  5
2    3  6
```

reorder_levels(*order*, *axis=0*)

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 [int] Where to reorder levels.

Returns

type of caller (new object)

replace(*to_replace=None*, *value=None*, *inplace=False*, *limit=None*, *regex=False*, *method='pad'*)

Replace values given in *to_replace* with *value*.

Values of the 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 *value* parameter should be *None*.
  - 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 *value* parameter should be `None` 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 [boolean, default False] If True, in place. Note: this will modify any other views on this object (e.g. a column from a DataFrame). Returns the caller if this is True.

limit [int, default None] Maximum size gap to forward or backward fill.

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', *None*}] The method to use when for replacement, when *to_replace* is a scalar, list or tuple and *value* is `None`.

Changed in version 0.23.0: Added to DataFrame.

Returns

DataFrame Object after replacement.

Raises

AssertionError

- If *regex* is not a `bool` and *to_replace* is not `None`.

TypeError

- 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:

`DataFrame.fillna` Fill NA values

`DataFrame.where` Replace values based on boolean condition

`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([0, 1, 2, 3, 4])
>>> s.replace(0, 5)
0    5
1    1
2    2
3    3
4    4
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
```

(continues on next page)

(continued from previous page)

```
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    0
1    3
2    3
3    3
4    4
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
```

(continues on next page)

(continued from previous page)

```
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
```

Note that when replacing multiple `bool` or `datetime64` objects, the data types in the *to_replace* parameter must match the data type of the value being replaced:

```
>>> df = pd.DataFrame({'A': [True, False, True],
...                    'B': [False, True, False]})
>>> df.replace({'a string': 'new value', True: False})  # raises
Traceback (most recent call last):
    ...
TypeError: Cannot compare types 'ndarray(dtype=bool)' and 'str'
```

This raises a `TypeError` because one of the `dict` keys is not of the correct type for replacement.

Compare the behavior of `s.replace({'a': None})` and `s.replace('a', None)` to understand the pecularities 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=None` 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. The command `s.replace('a', None)` is actually equivalent to `s.replace(to_replace='a', value=None, method='pad')`:

```
>>> s.replace('a', None)
0    10
1    10
2    10
3     b
4     b
dtype: object
```

resample(*rule*, *how=None*, *axis=0*, *fill_method=None*, *closed=None*, *label=None*, *convention='start'*, *kind=None*, *loffset=None*, *limit=None*, *base=0*, *on=None*, *level=None*)

Convenience method for frequency conversion and resampling of time series. Object must have a datetime-like index (DatetimeIndex, PeriodIndex, or TimedeltaIndex), or pass datetime-like values to the on or level keyword.

Parameters

rule [string] the offset string or object representing target conversion

axis [int, optional, default 0]

closed [{'right', 'left'}] 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'}] 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'}] For PeriodIndex only, controls whether to use the start or end of *rule*

kind: {'timestamp', 'period'}, optional 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.

loffset [timedelta] Adjust the resampled time labels

base [int, default 0] For frequencies that evenly subdivide 1 day, the "origin" of the aggregated intervals. For example, for '5min' frequency, base could range from 0 through 4. Defaults to 0

on [string, optional] For a DataFrame, column to use instead of index for resampling. Column must be datetime-like.

New in version 0.19.0.

level [string or int, optional] For a MultiIndex, level (name or number) to use for resampling. Level must be datetime-like.

New in version 0.19.0.

Returns

Resampler object

See also:

groupby Group by mapping, function, label, or list of labels.

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
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 `pad` method.

```
>>> series.resample('30S').pad()[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
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(array_like):
...     return np.sum(array_like)+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*.

```
>>> s = pd.Series([1, 2], index=pd.period_range('2012-01-01',
                                                freq='A',
                                                periods=2))
>>> s
2012    1
```

(continues on next page)

(continued from previous page)

```
2013    2
Freq: A-DEC, dtype: int64
```

Resample by month using 'start' *convention*. Values are assigned to the first month of the period.

```
>>> s.resample('M', convention='start').asfreq().head()
2012-01    1.0
2012-02    NaN
2012-03    NaN
2012-04    NaN
2012-05    NaN
Freq: M, dtype: float64
```

Resample by month using 'end' *convention*. Values are assigned to the last month of the period.

```
>>> s.resample('M', convention='end').asfreq()
2012-12    1.0
2013-01    NaN
2013-02    NaN
2013-03    NaN
2013-04    NaN
2013-05    NaN
2013-06    NaN
2013-07    NaN
2013-08    NaN
2013-09    NaN
2013-10    NaN
2013-11    NaN
2013-12    2.0
Freq: M, dtype: float64
```

For DataFrame objects, the keyword `on` can be used to specify the column instead of the index for resampling.

```
>>> df = pd.DataFrame(data=9*[range(4)], columns=['a', 'b', 'c', 'd'])
>>> df['time'] = pd.date_range('1/1/2000', periods=9, freq='T')
>>> df.resample('3T', on='time').sum()
                     a  b  c  d
time
2000-01-01 00:00:00  0  3  6  9
2000-01-01 00:03:00  0  3  6  9
2000-01-01 00:06:00  0  3  6  9
```

For a DataFrame with MultiIndex, the keyword `level` can be used to specify on level the resampling needs to take place.

```
>>> time = pd.date_range('1/1/2000', periods=5, freq='T')
>>> df2 = pd.DataFrame(data=10*[range(4)],
                       columns=['a', 'b', 'c', 'd'],
                       index=pd.MultiIndex.from_product([time, [1, 2]])
                       )
>>> df2.resample('3T', level=0).sum()
                     a  b   c   d
2000-01-01 00:00:00  0  6  12  18
2000-01-01 00:03:00  0  4   8  12
```

reset_index(*level=None*, *drop=False*, *inplace=False*, *col_level=0*, *col_fill=''*)

For DataFrame with multi-level index, return new DataFrame with labeling information in the columns under the index names, defaulting to 'level_0', 'level_1', etc. if any are None. For a standard index, the index name will be used (if set), otherwise a default 'index' or 'level_0' (if 'index' is already taken) will be used.

Parameters

level [int, str, tuple, or list, default None] Only remove the given levels from the index. Removes all levels by default

drop [boolean, default False] Do not try to insert index into dataframe columns. This resets the index to the default integer index.

inplace [boolean, default False] Modify the DataFrame in place (do not create a new object)

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.

Returns

resetted [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
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
```

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
```

rfloordiv(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

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.

Parameters

other [Series, DataFrame, or constant]

axis [{0, 1, 'index', 'columns'}] For Series input, axis to match Series index on

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

fill_value [None or float value, 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

result [DataFrame]

See also:

`DataFrame.floordiv`

Notes

Mismatched indices will be unioned together

Examples

None

rmod(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

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.

Parameters

other [Series, DataFrame, or constant]

axis [{0, 1, 'index', 'columns'}] For Series input, axis to match Series index on

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

fill_value [None or float value, 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

result [DataFrame]

See also:

`DataFrame.mod`

Notes

Mismatched indices will be unioned together

Examples

None

rmul(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
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.

Parameters

other [Series, DataFrame, or constant]

axis [{0, 1, 'index', 'columns'}] For Series input, axis to match Series index on

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

fill_value [None or float value, 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

result [DataFrame]

See also:

`DataFrame.mul`

Notes

Mismatched indices will be unioned together

Examples

None

rolling(*window*, *min_periods=None*, *center=False*, *win_type=None*, *on=None*, *axis=0*, *closed=None*)

Provides rolling window calculations.

New in version 0.18.0.

Parameters

window [int, or offset] Size of the moving window. This is the number of observations used for calculating the statistic. Each window will be a fixed size.

If its an offset then this will be 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. This is new in 0.19.0

min_periods [int, default None] Minimum number of observations in window required to have a value (otherwise result is NA). For a window that is specified by an offset, this will default to 1.

center [boolean, default False] Set the labels at the center of the window.

win_type [string, default None] Provide a window type. If `None`, all points are evenly weighted. See the notes below for further information.

on [string, optional] For a DataFrame, column on which to calculate the rolling window, rather than the index

closed [string, default None] Make the interval closed on the 'right', 'left', 'both' or 'neither' endpoints. For offset-based windows, it defaults to 'right'. For fixed windows, defaults to 'both'. Remaining cases not implemented for fixed windows.

New in version 0.20.0.

axis [int or string, default 0]

Returns

a Window or Rolling sub-classed for the particular operation

See also:

expanding Provides expanding transformations.

ewm Provides exponential weighted functions

Notes

By default, the result is set to the right edge of the window. This can be changed to the center of the window by setting `center=True`.

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

The recognized win_types are:

- `boxcar`
- `triang`
- `blackman`
- `hamming`
- `bartlett`
- `parzen`

- `bohman`
- `blackmanharris`
- `nuttall`
- `barthann`
- `kaiser` (needs beta)
- `gaussian` (needs std)
- `general_gaussian` (needs power, width)
- `slepian` (needs width).

If `win_type=None` all points are evenly weighted. To learn more about different window types see scipy.signal window functions.

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
```

Rolling sum with a window length of 2, using the 'triang' window type.

```
>>> df.rolling(2, win_type='triang').sum()
     B
0  NaN
1  1.0
2  2.5
3  NaN
4  NaN
```

Rolling sum with a window length of 2, min_periods defaults to the window length.

```
>>> df.rolling(2).sum()
     B
0  NaN
1  1.0
2  3.0
3  NaN
4  NaN
```

Same as above, but explicitly set the min_periods

```
>>> df.rolling(2, min_periods=1).sum()
     B
0  0.0
1  1.0
2  3.0
3  2.0
4  4.0
```

A ragged (meaning not-a-regular frequency), time-indexed DataFrame

```
>>> df = 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
                     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
```

Contrasting to an integer rolling window, this will roll a variable length window corresponding to the time period. The default for min_periods is 1.

```
>>> df.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
```

round(*decimals=0*, **args*, ***kwargs*)

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.

Returns

DataFrame object

See also:

`numpy.around`, `Series.round`

Examples

```
>>> df = pd.DataFrame(np.random.random([3, 3]),
...     columns=['A', 'B', 'C'], index=['first', 'second', 'third'])
>>> df
               A         B         C
first   0.028208  0.992815  0.173891
second  0.038683  0.645646  0.577595
third   0.877076  0.149370  0.491027
```

(continues on next page)

(continued from previous page)

```
>>> df.round(2)
           A     B     C
first   0.03  0.99  0.17
second  0.04  0.65  0.58
third   0.88  0.15  0.49
>>> df.round({'A': 1, 'C': 2})
          A         B     C
first   0.0  0.992815  0.17
second  0.0  0.645646  0.58
third   0.9  0.149370  0.49
>>> decimals = pd.Series([1, 0, 2], index=['A', 'B', 'C'])
>>> df.round(decimals)
          A  B     C
first   0.0  1  0.17
second  0.0  1  0.58
third   0.9  0  0.49
```

rpow(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

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.

Parameters

other [Series, DataFrame, or constant]

axis [{0, 1, 'index', 'columns'}] For Series input, axis to match Series index on

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

fill_value [None or float value, 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

result [DataFrame]

See also:

`DataFrame.pow`

Notes

Mismatched indices will be unioned together

Examples

None

rsub(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

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.

Parameters

other [Series, DataFrame, or constant]

axis [{0, 1, 'index', 'columns'}] For Series input, axis to match Series index on

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

fill_value [None or float value, 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

result [DataFrame]

See also:

`DataFrame.sub`

Notes

Mismatched indices will be unioned together

Examples

```
>>> a = pd.DataFrame([2, 1, 1, np.nan], index=['a', 'b', 'c', 'd'],
...                  columns=['one'])
>>> a
   one
a  2.0
b  1.0
c  1.0
d  NaN
>>> b = pd.DataFrame(dict(one=[1, np.nan, 1, np.nan],
...                       two=[3, 2, np.nan, 2]),
...                  index=['a', 'b', 'd', 'e'])
>>> b
   one  two
a  1.0  3.0
b  NaN  2.0
d  1.0  NaN
e  NaN  2.0
>>> a.sub(b, fill_value=0)
   one  two
a  1.0  -3.0
b  1.0  -2.0
c  1.0  NaN
d  -1.0  NaN
e  NaN  -2.0
```

rtruediv(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

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.

Parameters

other [Series, DataFrame, or constant]

axis [{0, 1, 'index', 'columns'}] For Series input, axis to match Series index on

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

fill_value [None or float value, 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

result [DataFrame]

See also:

`DataFrame.truediv`

Notes

Mismatched indices will be unioned together

Examples

None

sample(*n=None*, *frac=None*, *replace=False*, *weights=None*, *random_state=None*, *axis=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 [boolean, optional] Sample with or without replacement. Default = False.

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. inf and -inf values not allowed.

random_state [int or numpy.random.RandomState, optional] Seed for the random number generator (if int), or numpy RandomState object.

axis [int or string, optional] Axis to sample. Accepts axis number or name. Default is stat axis for given data type (0 for Series and DataFrames, 1 for Panels).

Returns

A new object of same type as caller.

Examples

Generate an example `Series` and `DataFrame`:

```
>>> s = pd.Series(np.random.randn(50))
>>> s.head()
0   -0.038497
1    1.820773
2   -0.972766
3   -1.598270
4   -1.095526
dtype: float64
>>> df = pd.DataFrame(np.random.randn(50, 4), columns=list('ABCD'))
>>> df.head()
          A         B         C         D
0  0.016443 -2.318952 -0.566372 -1.028078
1 -1.051921  0.438836  0.658280 -0.175797
2 -1.243569 -0.364626 -0.215065  0.057736
3  1.768216  0.404512 -0.385604 -1.457834
4  1.072446 -1.137172  0.314194 -0.046661
```

Next extract a random sample from both of these objects...

3 random elements from the `Series`:

```
>>> s.sample(n=3)
27   -0.994689
55   -1.049016
67   -0.224565
dtype: float64
```

And a random 10% of the `DataFrame` with replacement:

```
>>> df.sample(frac=0.1, replace=True)
           A         B         C         D
35  1.981780  0.142106  1.817165 -0.290805
49 -1.336199 -0.448634 -0.789640  0.217116
40  0.823173 -0.078816  1.009536  1.015108
15  1.421154 -0.055301 -1.922594 -0.019696
6  -0.148339  0.832938  1.787600 -1.383767
```

You can use *random state* for reproducibility:

```
>>> df.sample(random_state=1)
A         B         C         D
37 -2.027662  0.103611  0.237496 -0.165867
43 -0.259323 -0.583426  1.516140 -0.479118
12 -1.686325 -0.579510  0.985195 -0.460286
8   1.167946  0.429082  1.215742 -1.636041
9   1.197475 -0.864188  1.554031 -1.505264
```

select (*crit*, *axis=0*)

Return data corresponding to axis labels matching criteria

Deprecated since version 0.21.0: Use df.loc[df.index.map(crit)] to select via labels

Parameters

crit [function] To be called on each index (label). Should return True or False

axis [int]

Returns

selection [type of caller]

select_dtypes(*include=None*, *exclude=None*)

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

subset [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.

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'` (new in 0.20.0) 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
```

(continues on next page)

(continued from previous page)

```
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=['int'])
       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
```

sem(*axis=None*, *skipna=None*, *level=None*, *ddof=1*, *numeric_only=None*, ***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)}]

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

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 [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

Returns

sem [Series or DataFrame (if level specified)]

set_axis(*labels*, *axis=0*, *inplace=None*)

Assign desired index to given axis.

Indexes for column or row labels can be changed by assigning a list-like or Index.

Changed in version 0.21.0: The signature is now *labels* and *axis*, consistent with the rest of pandas API. Previously, the *axis* and *labels* arguments were respectively the first and second positional arguments.

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, and 1 identifies the columns.

inplace [boolean, default None] Whether to return a new %(klass)s instance.

> **Warning:** `inplace=None` currently falls back to to True, but in a future version, will default to False. Use inplace=True explicitly rather than relying on the default.

Returns

renamed [%(klass)s or None] An object of same type as caller if inplace=False, None otherwise.

See also:

`pandas.DataFrame.rename_axis` Alter the name of the index or columns.

Examples

Series

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

```
>>> s.set_axis(['a', 'b', 'c'], axis=0, inplace=False)
a    1
b    2
c    3
dtype: int64
```

The original object is not modified.

```
>>> s
0    1
1    2
2    3
dtype: int64
```

DataFrame

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

Change the row labels.

```
>>> df.set_axis(['a', 'b', 'c'], axis='index', inplace=False)
   A  B
a  1  4
b  2  5
c  3  6
```

Change the column labels.

```
>>> df.set_axis(['I', 'II'], axis='columns', inplace=False)
   I  II
0  1   4
1  2   5
2  3   6
```

Now, update the labels inplace.

```
>>> df.set_axis(['i', 'ii'], axis='columns', inplace=True)
>>> df
   i  ii
0  1   4
1  2   5
2  3   6
```

set_index (*keys*, *drop=True*, *append=False*, *inplace=False*, *verify_integrity=False*)

Set the DataFrame index (row labels) using one or more existing columns. By default yields a new object.

Parameters

keys [column label or list of column labels / arrays]

drop [boolean, default True] Delete columns to be used as the new index

append [boolean, default False] Whether to append columns to existing index

inplace [boolean, default False] Modify the DataFrame in place (do not create a new object)

verify_integrity [boolean, 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 [DataFrame]

Examples

```
>>> df = pd.DataFrame({'month': [1, 4, 7, 10],
...                    'year': [2012, 2014, 2013, 2014],
...                    'sale':[55, 40, 84, 31]})
   month  sale  year
0  1      55    2012
1  4      40    2014
2  7      84    2013
3  10     31    2014
```

Set the index to become the ‘month’ column:

```
>>> df.set_index('month')
       sale  year
month
1      55    2012
4      40    2014
7      84    2013
10     31    2014
```

Create a multi-index 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 multi-index using a set of values and a column:

```
>>> df.set_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
```

set_value(*index*, *col*, *value*, *takeable=False*)

Put single value at passed column and index

Deprecated since version 0.21.0: Use .at[] or .iat[] accessors instead.

Parameters

index [row label]

col [column label]

value [scalar value]

takeable [interpret the index/col as indexers, default False]

Returns

frame [DataFrame] If label pair is contained, will be reference to calling DataFrame, otherwise a new object

shape

Return a tuple representing the dimensionality of the DataFrame.

See also:

`ndarray.shape`

Examples

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df.shape
(2, 2)
```

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4],
...                    'col3': [5, 6]})
>>> df.shape
(2, 3)
```

shift (*periods=1*, *freq=None*, *axis=0*)

Shift index by desired number of periods with an optional time freq

Parameters

periods [int] Number of periods to move, can be positive or negative

freq [DateOffset, timedelta, or time rule string, optional] Increment to use from the tseries module or time rule (e.g. 'EOM'). See Notes.

axis [{0 or 'index', 1 or 'columns'}]

Returns

shifted [DataFrame]

Notes

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.

size

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

Return the number of rows if Series. Otherwise return the number of rows times number of columns if DataFrame.

See also:

`ndarray.size`

Examples

```
>>> s = pd.Series({'a': 1, 'b': 2, 'c': 3})
>>> s.size
3
```

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df.size
4
```

skew (*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

Return unbiased skew over requested axis Normalized by N-1

Parameters

axis [{index (0), columns (1)}]

skipna [boolean, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

Returns

skew [Series or DataFrame (if level specified)]

slice_shift (*periods=1*, *axis=0*)
Equivalent to *shift* without copying data. The shifted data will not include the dropped periods and the shifted axis will be smaller than the original.

Parameters

periods [int] Number of periods to move, can be positive or negative

Returns

shifted [same type as caller]

Notes

While the *slice_shift* is faster than *shift*, you may pay for it later during alignment.

sort_index (*axis=0*, *level=None*, *ascending=True*, *inplace=False*, *kind='quicksort'*, *na_position='last'*, *sort_remaining=True*, *by=None*)
Sort object by labels (along an axis)

Parameters

axis [index, columns to direct sorting]

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 [boolean, default True] Sort ascending vs. descending

inplace [bool, default False] if True, perform operation in-place

kind [{'quicksort', 'mergesort', 'heapsort'}, default 'quicksort'] Choice of sorting algorithm. See also ndarray.np.sort for more information. *mergesort* is the only stable algorithm. For DataFrames, this option is only applied when sorting on a single column or label.

na_position [{'first', 'last'}, default 'last'] *first* puts NaNs at the beginning, *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

Returns

sorted_obj [DataFrame]

sort_values (*by*, *axis=0*, *ascending=True*, *inplace=False*, *kind='quicksort'*, *na_position='last'*)
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

Changed in version 0.23.0: Allow specifying index or column level names.

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'}, default 'quicksort'] Choice of sorting algorithm. See also ndarray.np.sort for more information. *mergesort* is the only stable algorithm. For DataFrames, this option is only applied when sorting on a single column or label.

na_position [{'first', 'last'}, default 'last'] *first* puts NaNs at the beginning, *last* puts NaNs at the end

Returns

sorted_obj [DataFrame]

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],
... })
>>> df
    col1 col2 col3
0   A    2    0
1   A    1    1
2   B    9    9
3   NaN  8    4
4   D    7    2
5   C    4    3
```

Sort by col1

```
>>> df.sort_values(by=['col1'])
    col1 col2 col3
0   A    2    0
1   A    1    1
2   B    9    9
5   C    4    3
4   D    7    2
3   NaN  8    4
```

Sort by multiple columns

```
>>> df.sort_values(by=['col1', 'col2'])
    col1 col2 col3
1   A    1    1
0   A    2    0
2   B    9    9
5   C    4    3
4   D    7    2
3   NaN  8    4
```

Sort Descending

```
>>> df.sort_values(by='col1', ascending=False)
    col1 col2 col3
4   D    7    2
5   C    4    3
2   B    9    9
0   A    2    0
1   A    1    1
3   NaN  8    4
```

Putting NAs first

```
>>> df.sort_values(by='col1', ascending=False, na_position='first')
    col1 col2 col3
3   NaN  8    4
4   D    7    2
5   C    4    3
2   B    9    9
0   A    2    0
1   A    1    1
```

sortlevel(*level=0*, *axis=0*, *ascending=True*, *inplace=False*, *sort_remaining=True*)

Sort multilevel index by chosen axis and primary level. Data will be lexicographically sorted by the chosen level followed by the other levels (in order).

Deprecated since version 0.20.0: Use `DataFrame.sort_index()`

Parameters

level [int]

axis [{0 or 'index', 1 or 'columns'}, default 0]

ascending [boolean, default True]

inplace [boolean, default False] Sort the DataFrame without creating a new instance

sort_remaining [boolean, default True] Sort by the other levels too.

Returns

sorted [DataFrame]

See also:

`DataFrame.sort_index`

squeeze(*axis=None*)

Squeeze length 1 dimensions.

Parameters

axis [None, integer or string axis name, optional] The axis to squeeze if 1-sized.

New in version 0.20.0.

Returns

scalar if 1-sized, else original object

stack(*level=-1*, *dropna=True*)

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.

The new index levels are sorted.

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.

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 re-organised from being side by side on a horizontal position (the columns of the dataframe) to being stacked vertically on top of of each other (in the index of the dataframe).

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()
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()
            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()
        height  weight
cat kg     NaN     1.0
    m      2.0     NaN
dog kg     NaN     3.0
    m      4.0     NaN
```

Prescribing the level(s) to be stacked

The first parameter controls which level or levels are stacked:

```
>>> df_multi_level_cols2.stack(0)
             kg    m
cat height  NaN  2.0
    weight  1.0  NaN
dog height  NaN  4.0
    weight  3.0  NaN
>>> df_multi_level_cols2.stack([0, 1])
cat  height  m     2.0
     weight  kg    1.0
```

(continues on next page)

(continued from previous page)

```
dog  height  m     4.0
     weight  kg    3.0
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)
        height  weight
cat kg     NaN     NaN
    m      1.0     NaN
dog kg     NaN     2.0
    m      3.0     NaN
>>> df_multi_level_cols3.stack(dropna=True)
        height  weight
cat m      1.0     NaN
dog kg     NaN     2.0
    m      3.0     NaN
```

std(*axis=None*, *skipna=None*, *level=None*, *ddof=1*, *numeric_only=None*, ***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)}]

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

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 [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

Returns

std [Series or DataFrame (if level specified)]

style
Property returning a Styler object containing methods for building a styled HTML representation fo the DataFrame.

See also:

`pandas.io.formats.style.Styler`

sub(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

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.

Parameters

other [Series, DataFrame, or constant]

axis [{0, 1, 'index', 'columns'}] For Series input, axis to match Series index on

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

fill_value [None or float value, 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

result [DataFrame]

See also:

`DataFrame.rsub`

Notes

Mismatched indices will be unioned together

Examples

```
>>> a = pd.DataFrame([2, 1, 1, np.nan], index=['a', 'b', 'c', 'd'],
...                  columns=['one'])
>>> a
   one
a  2.0
b  1.0
c  1.0
d  NaN
>>> b = pd.DataFrame(dict(one=[1, np.nan, 1, np.nan],
...                       two=[3, 2, np.nan, 2]),
...                  index=['a', 'b', 'd', 'e'])
>>> b
   one  two
a  1.0  3.0
b  NaN  2.0
d  1.0  NaN
e  NaN  2.0
>>> a.sub(b, fill_value=0)
   one  two
a  1.0 -3.0
b  1.0 -2.0
```

(continues on next page)

(continued from previous page)

```
c  1.0  NaN
d  -1.0  NaN
e  NaN  -2.0
```

subtract(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

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.

Parameters

other [Series, DataFrame, or constant]

axis [{0, 1, 'index', 'columns'}] For Series input, axis to match Series index on

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

fill_value [None or float value, 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

result [DataFrame]

See also:

`DataFrame.rsub`

Notes

Mismatched indices will be unioned together

Examples

```
>>> a = pd.DataFrame([2, 1, 1, np.nan], index=['a', 'b', 'c', 'd'],
...                  columns=['one'])
>>> a
   one
a  2.0
b  1.0
c  1.0
d  NaN
>>> b = pd.DataFrame(dict(one=[1, np.nan, 1, np.nan],
...                       two=[3, 2, np.nan, 2]),
...                  index=['a', 'b', 'd', 'e'])
>>> b
   one  two
a  1.0  3.0
b  NaN  2.0
d  1.0  NaN
e  NaN  2.0
>>> a.sub(b, fill_value=0)
```

(continues on next page)

(continued from previous page)

```
   one  two
a  1.0  -3.0
b  1.0  -2.0
c  1.0  NaN
d  -1.0  NaN
e  NaN  -2.0
```

sum(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, *min_count=0*, ***kwargs*)
Return the sum of the values for the requested axis

Parameters

axis [{index (0), columns (1)}]

skipna [boolean, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. 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.

New in version 0.22.0: Added with the default being 0. This means the sum of an all-NA or empty Series is 0, and the product of an all-NA or empty Series is 1.

Returns

sum [Series or DataFrame (if level specified)]

Examples

By default, the sum of an empty or all-NA Series is `0`.

```
>>> pd.Series([]).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([]).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
```

swapaxes(*axis1*, *axis2*, *copy=True*)
Interchange axes and swap values axes appropriately

Returns

y [same as input]

swaplevel (*i=-2*, *j=-1*, *axis=0*)

Swap levels i and j in a MultiIndex on a particular axis

Parameters

i, j [int, string (can be mixed)] Level of index to be swapped. Can pass level name as string.

Returns

swapped [type of caller (new object)]

.. versionchanged:: 0.18.1 The indexes `i` and `j` are now optional, and default to the two innermost levels of the index.

tail (*n=5*)

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.

Parameters

n [int, default 5] Number of rows to select.

Returns

type of caller The last *n* rows of the caller object.

See also:

pandas.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
```

take(*indices*, *axis=0*, *convert=None*, *is_copy=True*, ***kwargs*)

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.

convert [bool, default True] Whether to convert negative indices into positive ones. For example, `-1` would map to the `len(axis) - 1`. The conversions are similar to the behavior of indexing a regular Python list.

Deprecated since version 0.21.0: In the future, negative indices will always be converted.

is_copy [bool, default True] Whether to return a copy of the original object or not.

****kwargs** For compatibility with `numpy.take()`. Has no effect on the output.

Returns

taken [type of 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
```

to_clipboard(*excel=True*, *sep=None*, ***kwargs*)

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]

- True, use the provided separator, writing in a csv format for allowing easy pasting into excel.
- 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_table.

Notes

Requirements for your platform.

- Linux : *xclip*, or *xsel* (with *gtk* or *PyQt4* modules)
- Windows : none
- OS X : none

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 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
```

to_csv(**args*, ***kwargs*)

Write DataFrame to a comma-separated values (csv) file

Parameters

path_or_buf [string or file handle, default None] File path or object, if None is provided the result is returned as a string.

sep [character, default ‘,’] Field delimiter for the output file.

na_rep [string, default ‘’] Missing data representation

float_format [string, default None] Format string for floating point numbers

columns [sequence, optional] Columns to write

header [boolean or list of string, 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 [boolean, default True] Write row names (index)

index_label [string 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 DataFrame uses MultiIndex. If False do not print fields for index names. Use index_label=False for easier importing in R

mode [str] Python write mode, default ‘w’

encoding [string, optional] A string representing the encoding to use in the output file, defaults to ‘ascii’ on Python 2 and ‘utf-8’ on Python 3.

compression [string, optional] A string representing the compression to use in the output file. Allowed values are ‘gzip’, ‘bz2’, ‘zip’, ‘xz’. This input is only used when the first argument is a filename.

line_terminator [string, default `'\n'`] The newline character or character sequence to use in the output file

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 [string (length 1), default '"'] character used to quote fields

doublequote [boolean, default True] Control quoting of *quotechar* inside a field

escapechar [string (length 1), default None] character used to escape *sep* and *quotechar* when appropriate

chunksize [int or None] rows to write at a time

tupleize_cols [boolean, default False] Deprecated since version 0.21.0: This argument will be removed and will always write each row of the multi-index as a separate row in the CSV file.

Write MultiIndex columns as a list of tuples (if True) or in the new, expanded format, where each MultiIndex column is a row in the CSV (if False).

date_format [string, default None] Format string for datetime objects

decimal: string, default '.' Character recognized as decimal separator. E.g. use ',' for European data

to_dense()

Return dense representation of NDFrame (as opposed to sparse)

to_dict(*orient='dict'*, *into=<type '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', '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]}
- 'records' : list like [{column -> value}, ... , {column -> value}]
- 'index' : dict like {index -> {column -> value}}

Abbreviations are allowed. *s* indicates *series* and *sp* indicates *split*.

into [class, default dict] The collections.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.

New in version 0.21.0.

Returns

result [collections.Mapping like {column -> {index -> value}}]

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=['a', 'b'])
>>> df
   col1  col2
a     1   0.50
b     2   0.75
>>> df.to_dict()
{'col1': {'a': 1, 'b': 2}, 'col2': {'a': 0.5, 'b': 0.75}}
```

You can specify the return orientation.

```
>>> df.to_dict('series')
{'col1': a    1
         b    2
         Name: col1, dtype: int64,
 'col2': a    0.50
         b    0.75
         Name: col2, dtype: float64}
```

```
>>> df.to_dict('split')
{'index': ['a', 'b'], 'columns': ['col1', 'col2'],
 'data': [[1.0, 0.5], [2.0, 0.75]]}
```

```
>>> df.to_dict('records')
[{'col1': 1.0, 'col2': 0.5}, {'col1': 2.0, 'col2': 0.75}]
```

```
>>> df.to_dict('index')
{'a': {'col1': 1.0, 'col2': 0.5}, 'b': {'col1': 2.0, 'col2': 0.75}}
```

You can also specify the mapping type.

```
>>> from collections import OrderedDict, defaultdict
>>> df.to_dict(into=OrderedDict)
OrderedDict([('col1', OrderedDict([('a', 1), ('b', 2)])),
             ('col2', OrderedDict([('a', 0.5), ('b', 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': 1.0, 'col2': 0.5}),
 defaultdict(<class 'list'>, {'col1': 2.0, 'col2': 0.75})]
```

to_excel(**args*, ***kwargs*)

Write DataFrame to an excel sheet

Parameters

excel_writer [string or ExcelWriter object] File path or existing ExcelWriter

sheet_name [string, default 'Sheet1'] Name of sheet which will contain DataFrame

na_rep [string, default ''] Missing data representation

float_format [string, default None] Format string for floating point numbers

columns [sequence, optional] Columns to write

header [boolean or list of string, 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 [boolean, default True] Write row names (index)

index_label [string or sequence, 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 DataFrame uses MultiIndex.

startrow : upper left cell row to dump data frame

startcol : upper left cell column to dump data frame

engine [string, default None] write engine to use - you can also set this via the options `io.excel.xlsx.writer`, `io.excel.xls.writer`, and `io.excel.xlsm.writer`.

merge_cells [boolean, default True] Write MultiIndex and Hierarchical Rows as merged cells.

encoding: string, default None encoding of the resulting excel file. Only necessary for xlwt, other writers support unicode natively.

inf_rep [string, default 'inf'] Representation for infinity (there is no native representation for infinity in Excel)

freeze_panes [tuple of integer (length 2), default None] Specifies the one-based bottommost row and rightmost column that is to be frozen

New in version 0.20.0.

Notes

If passing an existing ExcelWriter object, then the sheet will be added to the existing workbook. This can be used to save different DataFrames to one workbook:

```
>>> writer = pd.ExcelWriter('output.xlsx')
>>> df1.to_excel(writer,'Sheet1')
>>> df2.to_excel(writer,'Sheet2')
>>> writer.save()
```

For compatibility with to_csv, to_excel serializes lists and dicts to strings before writing.

to_feather(*fname*)
write out the binary feather-format for DataFrames

New in version 0.20.0.

Parameters

fname [str] string file path

to_gbq(*destination_table*, *project_id*, *chunksize=None*, *verbose=None*, *reauth=False*, *if_exists='fail'*, *private_key=None*, *auth_local_webserver=False*, *table_schema=None*)
Write a DataFrame to a Google BigQuery table.

This function requires the pandas-gbq package.

Authentication to the Google BigQuery service is via OAuth 2.0.

- If `private_key` is provided, the library loads the JSON service account credentials and uses those to authenticate.
- If no `private_key` is provided, the library tries application default credentials.
- If application default credentials are not found or cannot be used with BigQuery, the library authenticates with user account credentials. In this case, you will be asked to grant permissions for product name 'pandas GBQ'.

Parameters

destination_table [str] Name of table to be written, in the form 'dataset.tablename'.

project_id [str] Google BigQuery Account project ID.

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 reauthenticate 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, do nothing.

`'replace'` If table exists, drop it, recreate it, and insert data.

`'append'` If table exists, insert data. Create if does not exist.

private_key [str, optional] Service account private key in JSON format. Can be file path or string contents. This is useful for remote server authentication (eg. Jupyter/IPython notebook on remote host).

auth_local_webserver [bool, default False] Use the local webserver flow instead of the console flow when getting user credentials.

New in version 0.2.0 of pandas-gbq.

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.

verbose [boolean, deprecated] *Deprecated in Pandas-GBQ 0.4.0.* Use the logging module to adjust verbosity instead.

See also:

`pandas_gbq.to_gbq` This function in the pandas-gbq library.

`pandas.read_gbq` Read a DataFrame from Google BigQuery.

`to_hdf`(*path_or_buf*, *key*, ***kwargs*)

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.

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.

format [{'fixed', 'table'}, 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.

append [bool, default False] For Table formats, append the input data to the existing.

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 io.hdf5-query-data-columns. Applicable only to format='table'.

complevel [{0-9}, optional] Specifies a compression level for data. A value of 0 disables compression.

complib [{'zlib', 'lzo', 'bzip2', 'blosc'}, default 'zlib'] Specifies the compression library to be used. As of v0.20.2 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.

fletcher32 [bool, default False] If applying compression use the fletcher32 checksum.

dropna [bool, default False] If true, ALL nan rows will not be written to store.

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.

See also:

`DataFrame.read_hdf` Read from HDF file.

`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
```

Deleting file with data:

```
>>> import os
>>> os.remove('data.h5')
```

to_html (**args*, ***kwargs*)

Render a DataFrame as an HTML table.

to_html-specific options:

bold_rows [boolean, 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 [boolean, default True] Convert the characters <, >, and & to HTML-safe sequences.

max_rows [int, optional] Maximum number of rows to show before truncating. If None, show all.

max_cols [int, optional] Maximum number of columns to show before truncating. If None, show all.

decimal [string, default '.'] Character recognized as decimal separator, e.g. ',' in Europe

New in version 0.18.0.

border [int] A `border=border` attribute is included in the opening <*table*> tag. Default `pd.options.html.border`.

New in version 0.19.0.

table_id [str, optional] A css id is included in the opening <*table*> tag if specified.

New in version 0.23.0.

Parameters

buf [StringIO-like, optional] buffer to write to

columns [sequence, optional] the subset of columns to write; default None writes all columns

col_space [int, optional] the minimum width of each column

header [bool, optional] whether to print column labels, default True

index [bool, optional] whether to print index (row) labels, default True

na_rep [string, optional] string representation of NAN to use, default 'NaN'

formatters [list or dict of one-parameter functions, optional] formatter functions to apply to columns' elements by position or name, default None. 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, optional] formatter function to apply to columns' elements if they are floats, default None. The result of this function must be a unicode string.

sparsify [bool, optional] Set to False for a DataFrame with a hierarchical index to print every multiindex key at each row, default True

index_names [bool, optional] Prints the names of the indexes, default True

line_width [int, optional] Width to wrap a line in characters, default no wrap

table_id [str, optional] id for the <table> element create by to_html

New in version 0.23.0.

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

Returns

formatted [string (or unicode, depending on data and options)]

to_json(*path_or_buf=None*, *orient=None*, *date_format=None*, *double_precision=10*, *force_ascii=True*, *date_unit='ms'*, *default_handler=None*, *lines=False*, *compression=None*, *index=True*)

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 [string or file handle, optional] File path or object. If not specified, the result is returned as a string.

orient [string] Indication of expected JSON string format.

- Series
  - default is 'index'
  - allowed values are: {'split','records','index'}
- DataFrame
  - default is 'columns'
  - allowed values are: {'split','records','index','columns','values'}
- 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, and the data component is like `orient='records'`.

    Changed in version 0.20.0.

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.

force_ascii [boolean, default True] Force encoded string to be ASCII.

date_unit [string, 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 [boolean, default False] If 'orient' is 'records' write out line delimited json format. Will throw ValueError if incorrect 'orient' since others are not list like.

New in version 0.19.0.

compression [{None, 'gzip', 'bz2', 'zip', 'xz'}] A string representing the compression to use in the output file, only used when the first argument is a filename.

New in version 0.21.0.

index [boolean, default True] Whether to include the index values in the JSON string. Not including the index (`index=False`) is only supported when orient is 'split' or 'table'.

New in version 0.23.0.

See also:

`pandas.read_json`

Examples

```
>>> df = pd.DataFrame([['a', 'b'], ['c', 'd']],
...                   index=['row 1', 'row 2'],
...                   columns=['col 1', 'col 2'])
>>> df.to_json(orient='split')
'{"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.

```
>>> df.to_json(orient='records')
'[{"col 1":"a","col 2":"b"},{"col 1":"c","col 2":"d"}]'
```

Encoding/decoding a Dataframe using `'index'` formatted JSON:

```
>>> df.to_json(orient='index')
'{"row 1":{"col 1":"a","col 2":"b"},"row 2":{"col 1":"c","col 2":"d"}}'
```

Encoding/decoding a Dataframe using `'columns'` formatted JSON:

```
>>> df.to_json(orient='columns')
'{"col 1":{"row 1":"a","row 2":"c"},"col 2":{"row 1":"b","row 2":"d"}}'
```

Encoding/decoding a Dataframe using `'values'` formatted JSON:

```
>>> df.to_json(orient='values')
'[["a","b"],["c","d"]]'
```

Encoding with Table Schema

```
>>> df.to_json(orient='table')
'{"schema": {"fields": [{"name": "index", "type": "string"},
                        {"name": "col 1", "type": "string"},
                        {"name": "col 2", "type": "string"}],
             "primaryKey": "index",
             "pandas_version": "0.20.0"},
  "data": [{"index": "row 1", "col 1": "a", "col 2": "b"},
           {"index": "row 2", "col 1": "c", "col 2": "d"}]}'
```

to_latex(*buf=None*, *columns=None*, *col_space=None*, *header=True*, *index=True*, *na_rep='NaN'*, *formatters=None*, *float_format=None*, *sparsify=None*, *index_names=True*, *bold_rows=False*, *column_format=None*, *longtable=None*, *escape=None*, *encoding=None*, *decimal='.'*, *multicolumn=None*, *multicolumn_format=None*, *multirow=None*)

Render an object to a tabular environment table. You can splice this into a LaTeX document. Requires \usepackage{booktabs}.

Changed in version 0.20.2: Added to Series

to_latex-specific options:

bold_rows [boolean, default False] Make the row labels bold in the output

column_format [str, default None] The columns format as specified in LaTeX table format e.g 'rcl' for 3 columns

longtable [boolean, default will be read from the pandas config module] Default: False. Use a longtable environment instead of tabular. Requires adding a \usepackage{longtable} to your LaTeX preamble.

escape [boolean, default will be read from the pandas config module] Default: True. When set to False prevents from escaping latex special characters in column names.

encoding [str, default None] A string representing the encoding to use in the output file, defaults to 'ascii' on Python 2 and 'utf-8' on Python 3.

decimal [string, default '.'] Character recognized as decimal separator, e.g. ',' in Europe.

New in version 0.18.0.

multicolumn [boolean, default True] Use multicolumn to enhance MultiIndex columns. The default will be read from the config module.

New in version 0.20.0.

multicolumn_format [str, default 'l'] The alignment for multicolumns, similar to *column_format* The default will be read from the config module.

New in version 0.20.0.

multirow [boolean, default False] 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.

New in version 0.20.0.

to_mol2 (*filepath_or_buffer=None*, *update_properties=True*, *molecule_column='mol'*, *columns=None*)

Write DataFrame to Mol2 file.

New in version 0.3.

Parameters

filepath_or_buffer [string or None] File path

update_properties [bool, optional (default=True)] Switch to update properties from the DataFrames to the molecules while writting.

molecule_column [string or None, optional (default='mol')] Name of molecule column. If None the molecules will be skipped.

columns [list or None, optional (default=None)] A list of columns to write to file. If None then all available fields are written.

to_msgpack (*path_or_buf=None*, *encoding='utf-8'*, ***kwargs*)

msgpack (serialize) object to input file path

THIS IS AN EXPERIMENTAL LIBRARY and the storage format may not be stable until a future release.

Parameters

path [string File path, buffer-like, or None] if None, return generated string

append [boolean whether to append to an existing msgpack] (default is False)

compress [type of compressor (zlib or blosc), default to None (no] compression)

to_panel()
Transform long (stacked) format (DataFrame) into wide (3D, Panel) format.

Deprecated since version 0.20.0.

Currently the index of the DataFrame must be a 2-level MultiIndex. This may be generalized later

Returns

panel [Panel]

to_parquet(*fname*, *engine='auto'*, *compression='snappy'*, ***kwargs*)
Write a DataFrame to the binary parquet format.

New in version 0.21.0.

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

fname [str] String file path.

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 [{'snappy', 'gzip', 'brotli', None}, default 'snappy'] Name of the compression to use. Use `None` for no compression.

****kwargs** Additional arguments passed to the parquet library. See pandas io for more details.

See also:

read_parquet Read 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

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
```

to_period(*freq=None*, *axis=0*, *copy=True*)
Convert DataFrame from DatetimeIndex to PeriodIndex with desired frequency (inferred from index if not passed)

Parameters

freq [string, default]

axis [{0 or 'index', 1 or 'columns'}, default 0] The axis to convert (the index by default)

copy [boolean, default True] If False then underlying input data is not copied

Returns

ts [TimeSeries with PeriodIndex]

to_pickle(*path*, *compression='infer'*, *protocol=2*)
Pickle (serialize) object to file.

Parameters

path [str] File path where the pickled object will be stored.

compression [{'infer', 'gzip', 'bz2', 'zip', 'xz', None}, default 'infer'] A string representing the compression to use in the output file. By default, infers from the file extension in specified path.

New in version 0.20.0.

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 for this parameter depend on the version of Python. For Python 2.x, possible values are 0, 1, 2. For Python>=3.0, 3 is a valid value. For Python >= 3.4, 4 is a valid value. A negative value for the protocol parameter is equivalent to setting its value to HIGHEST_PROTOCOL.

New in version 0.21.0.

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
```

(continues on next page)

(continued from previous page)

```
2    2    7
3    3    8
4    4    9
```

```
>>> import os
>>> os.remove("./dummy.pkl")
```

to_records(*index=True*, *convert_datetime64=None*)

Convert DataFrame to a NumPy record array.

Index will be put in the 'index' field of the record array if requested.

Parameters

index [boolean, default True] Include index in resulting record array, stored in 'index' field.

convert_datetime64 [boolean, default None] Deprecated since version 0.23.0.

Whether to convert the index to datetime.datetime if it is a DatetimeIndex.

Returns

y [numpy.recarray]

See also:

DataFrame.from_records convert structured or record ndarray to DataFrame.

numpy.recarray 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')])
```

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')])
```

By default, timestamps are converted to *datetime.datetime*:

```
>>> df.index = pd.date_range('2018-01-01 09:00', periods=2, freq='min')
>>> df
                     A     B
2018-01-01 09:00:00  1  0.50
2018-01-01 09:01:00  2  0.75
```

(continues on next page)

(continued from previous page)

```
>>> df.to_records()
rec.array([(datetime.datetime(2018, 1, 1, 9, 0), 1, 0.5 ),
           (datetime.datetime(2018, 1, 1, 9, 1), 2, 0.75)],
          dtype=[('index', 'O'), ('A', '<i8'), ('B', '<f8')])
```

The timestamp conversion can be disabled so NumPy's datetime64 data type is used instead:

```
>>> df.to_records(convert_datetime64=False)
rec.array([('2018-01-01T09:00:00.000000000', 1, 0.5 ),
           ('2018-01-01T09:01:00.000000000', 2, 0.75)],
          dtype=[('index', '<M8[ns]'), ('A', '<i8'), ('B', '<f8')])
```

to_sdf(*filepath_or_buffer=None*, *update_properties=True*, *molecule_column=None*, *columns=None*)
Write DataFrame to SDF file.

New in version 0.3.

Parameters

filepath_or_buffer [string or None] File path

update_properties [bool, optional (default=True)] Switch to update properties from the DataFrames to the molecules while writting.

molecule_column [string or None, optional (default='mol')] Name of molecule column. If None the molecules will be skipped.

columns [list or None, optional (default=None)] A list of columns to write to file. If None then all available fields are written.

to_sparse(*fill_value=None*, *kind='block'*)
Convert to SparseDataFrame

Parameters

fill_value [float, default NaN]

kind [{'block', 'integer'}]

Returns

y [SparseDataFrame]

to_sql(*name*, *con*, *schema=None*, *if_exists='fail'*, *index=True*, *index_label=None*, *chunksize=None*, *dtype=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 [string] Name of SQL table.

con [sqlalchemy.engine.Engine or sqlite3.Connection] Using SQLAlchemy makes it possible to use any DB supported by that library. Legacy support is provided for sqlite3.Connection objects.

schema [string, 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 [boolean, default True] Write DataFrame index as a column. Uses *index_label* as the column name in the table.

index_label [string 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] Rows will be written in batches of this size at a time. By default, all rows will be written at once.

dtype [dict, optional] Specifying the datatype for columns. The keys should be the column names and the values should be the SQLAlchemy types or strings for the sqlite3 legacy mode.

Raises

ValueError When the table already exists and *if_exists* is 'fail' (the default).

See also:

`pandas.read_sql` read a DataFrame from a table

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('users', con=engine)
>>> engine.execute("SELECT * FROM users").fetchall()
[(0, 'User 1'), (1, 'User 2'), (2, 'User 3')]
```

```
>>> df1 = pd.DataFrame({'name' : ['User 4', 'User 5']})
>>> df1.to_sql('users', con=engine, if_exists='append')
>>> engine.execute("SELECT * FROM users").fetchall()
[(0, 'User 1'), (1, 'User 2'), (2, 'User 3'),
 (0, 'User 4'), (1, 'User 5')]
```

Overwrite the table with just `df1`.

```
>>> df1.to_sql('users', con=engine, if_exists='replace',
...            index_label='id')
>>> engine.execute("SELECT * FROM users").fetchall()
[(0, 'User 4'), (1, 'User 5')]
```

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('integers', con=engine, index=False,
...           dtype={"A": Integer()})
```

```
>>> engine.execute("SELECT * FROM integers").fetchall()
[(1,), (None,), (2,)]
```

to_stata(*fname*, *convert_dates=None*, *write_index=True*, *encoding='latin-1'*, *byteorder=None*, *time_stamp=None*, *data_label=None*, *variable_labels=None*, *version=114*, *convert_strl=None*)

Export Stata binary dta files.

Parameters

fname [path (string), buffer or path object] string, path object (pathlib.Path or py._path.local.LocalPath) or object implementing a binary write() functions. If using a buffer then the buffer will not be automatically closed after the file data has been written.

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.

encoding [str] Default is latin-1. Unicode is not supported.

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] 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.

New in version 0.19.0.

version [{114, 117}] Version to use in the output dta file. Version 114 can be used read by Stata 10 and later. Version 117 can be read by Stata 13 or later. Version 114 limits

string variables to 244 characters or fewer while 117 allows strings with lengths up to 2,000,000 characters.

New in version 0.23.0.

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.

New in version 0.23.0.

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

New in version 0.19.0.

See also:

pandas.read_stata Import Stata data files

pandas.io.stata.StataWriter low-level writer for Stata data files

pandas.io.stata.StataWriter117 low-level writer for version 117 files

Examples

```
>>> data.to_stata('./data_file.dta')
```

Or with dates

```
>>> data.to_stata('./date_data_file.dta', {2 : 'tw'})
```

Alternatively you can create an instance of the StataWriter class

```
>>> writer = StataWriter('./data_file.dta', data)
>>> writer.write_file()
```

With dates:

```
>>> writer = StataWriter('./date_data_file.dta', data, {2 : 'tw'})
>>> writer.write_file()
```

to_string(*buf=None*, *columns=None*, *col_space=None*, *header=True*, *index=True*, *na_rep='NaN'*, *formatters=None*, *float_format=None*, *sparsify=None*, *index_names=True*, *justify=None*, *line_width=None*, *max_rows=None*, *max_cols=None*, *show_dimensions=False*)

Render a DataFrame to a console-friendly tabular output.

Parameters

buf [StringIO-like, optional] buffer to write to

columns [sequence, optional] the subset of columns to write; default None writes all columns

col_space [int, optional] the minimum width of each column

header [bool, optional] Write out the column names. If a list of strings is given, it is assumed to be aliases for the column names

index [bool, optional] whether to print index (row) labels, default True

na_rep [string, optional] string representation of NAN to use, default 'NaN'

formatters [list or dict of one-parameter functions, optional] formatter functions to apply to columns' elements by position or name, default None. 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, optional] formatter function to apply to columns' elements if they are floats, default None. The result of this function must be a unicode string.

sparsify [bool, optional] Set to False for a DataFrame with a hierarchical index to print every multiindex key at each row, default True

index_names [bool, optional] Prints the names of the indexes, default True

line_width [int, optional] Width to wrap a line in characters, default no wrap

table_id [str, optional] id for the <table> element create by to_html

New in version 0.23.0.

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

Returns

formatted [string (or unicode, depending on data and options)]

to_timestamp (*freq=None*, *how='start'*, *axis=0*, *copy=True*)

Cast to DatetimeIndex of timestamps, at *beginning* of period

Parameters

freq [string, 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 [boolean, default True] If false then underlying input data is not copied

Returns

df [DataFrame with DatetimeIndex]

to_xarray()

Return an xarray object from the pandas object.

Returns

a DataArray for a Series

a Dataset for a DataFrame

a DataArray for higher dims

Notes

See the xarray docs

Examples

```
>>> df = pd.DataFrame({'A' : [1, 1, 2],
                       'B' : ['foo', 'bar', 'foo'],
                       'C' : np.arange(4.,7)})
>>> df
   A    B    C
0  1  foo  4.0
1  1  bar  5.0
2  2  foo  6.0
```

```
>>> df.to_xarray()
<xarray.Dataset>
Dimensions:  (index: 3)
Coordinates:
  * index    (index) int64 0 1 2
Data variables:
    A        (index) int64 1 1 2
    B        (index) object 'foo' 'bar' 'foo'
    C        (index) float64 4.0 5.0 6.0
```

```
>>> df = pd.DataFrame({'A' : [1, 1, 2],
                       'B' : ['foo', 'bar', 'foo'],
                       'C' : np.arange(4.,7)}
                      ).set_index(['B','A'])
>>> df
         C
B   A
foo 1  4.0
```

(continues on next page)

(continued from previous page)

```
bar 1  5.0
foo 2  6.0
```

```
>>> df.to_xarray()
<xarray.Dataset>
Dimensions:  (A: 2, B: 2)
Coordinates:
  * B        (B) object 'bar' 'foo'
  * A        (A) int64 1 2
Data variables:
    C        (B, A) float64 5.0 nan 4.0 6.0
```

```
>>> p = pd.Panel(np.arange(24).reshape(4,3,2),
                 items=list('ABCD'),
                 major_axis=pd.date_range('20130101', periods=3),
                 minor_axis=['first', 'second'])
>>> p
<class 'pandas.core.panel.Panel'>
Dimensions: 4 (items) x 3 (major_axis) x 2 (minor_axis)
Items axis: A to D
Major_axis axis: 2013-01-01 00:00:00 to 2013-01-03 00:00:00
Minor_axis axis: first to second
```

```
>>> p.to_xarray()
<xarray.DataArray (items: 4, major_axis: 3, minor_axis: 2)>
array([[[ 0,  1],
        [ 2,  3],
        [ 4,  5]],
       [[ 6,  7],
        [ 8,  9],
        [10, 11]],
       [[12, 13],
        [14, 15],
        [16, 17]],
       [[18, 19],
        [20, 21],
        [22, 23]]])
Coordinates:
  * items       (items) object 'A' 'B' 'C' 'D'
  * major_axis  (major_axis) datetime64[ns] 2013-01-01 2013-01-02 2013-01-03␣
↪ # noqa
  * minor_axis  (minor_axis) object 'first' 'second'
```

transform(*func*, **args*, ***kwargs*)

Call function producing a like-indexed NDFrame and return a NDFrame with the transformed values

New in version 0.20.0.

Parameters

func [callable, string, dictionary, or list of string/callables] To apply to column

Accepted Combinations are:

- string function name
- function
- list of functions

- dict of column names -> functions (or list of functions)

Returns

transformed [NDFrame]

See also:

`pandas.NDFrame.aggregate`, `pandas.NDFrame.apply`

Examples

```
>>> df = pd.DataFrame(np.random.randn(10, 3), columns=['A', 'B', 'C'],
...                   index=pd.date_range('1/1/2000', periods=10))
df.iloc[3:7] = np.nan
```

```
>>> df.transform(lambda x: (x - x.mean()) / x.std())
                   A         B         C
2000-01-01  0.579457  1.236184  0.123424
2000-01-02  0.370357 -0.605875 -1.231325
2000-01-03  1.455756 -0.277446  0.288967
2000-01-04       NaN       NaN       NaN
2000-01-05       NaN       NaN       NaN
2000-01-06       NaN       NaN       NaN
2000-01-07       NaN       NaN       NaN
2000-01-08 -0.498658  1.274522  1.642524
2000-01-09 -0.540524 -1.012676 -0.828968
2000-01-10 -1.366388 -0.614710  0.005378
```

transpose(**args*, ***kwargs*)

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

copy [bool, default False] If True, the underlying data is copied. Otherwise (default), no copy is made if possible.

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

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
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
```

(continues on next page)

(continued from previous page)

```
dtype: object
>>> df2_transposed.dtypes
0    object
1    object
dtype: object
```

truediv(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

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.

Parameters

other [Series, DataFrame, or constant]

axis [{0, 1, 'index', 'columns'}] For Series input, axis to match Series index on

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

fill_value [None or float value, 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

result [DataFrame]

See also:

`DataFrame.rtruediv`

Notes

Mismatched indices will be unioned together

Examples

None

truncate(*before=None*, *after=None*, *axis=None*, *copy=True*)

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, string, int] Truncate all rows before this index value.

after [date, string, 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.

copy [boolean, 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()
```

(continues on next page)

(continued from previous page)

```
                     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
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
```

tshift (*periods=1*, *freq=None*, *axis=0*)

Shift the time index, using the index's frequency if available.

Parameters

periods [int] Number of periods to move, can be positive or negative

freq [DateOffset, timedelta, or time rule string, default None] Increment to use from the tseries module or time rule (e.g. 'EOM')

axis [int or basestring] Corresponds to the axis that contains the Index

Returns

shifted [NDFrame]

Notes

If freq is not specified then tries to use the freq or inferred_freq attributes of the index. If neither of those attributes exist, a ValueError is thrown

tz_convert (*tz*, *axis=0*, *level=None*, *copy=True*)
Convert tz-aware axis to target time zone.

Parameters

tz [string or pytz.timezone object]

axis [the axis to convert]

level [int, str, default None] If axis ia a MultiIndex, convert a specific level. Otherwise must be None

copy [boolean, default True] Also make a copy of the underlying data

Raises

TypeError If the axis is tz-naive.

tz_localize (*tz*, *axis=0*, *level=None*, *copy=True*, *ambiguous='raise'*)
Localize tz-naive TimeSeries to target time zone.

Parameters

tz [string or pytz.timezone object]

axis [the axis to localize]

level [int, str, default None] If axis ia a MultiIndex, localize a specific level. Otherwise must be None

copy [boolean, default True] Also make a copy of the underlying data

ambiguous ['infer', bool-ndarray, 'NaT', default 'raise']

- '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

Raises

TypeError If the TimeSeries is tz-aware and tz is not None.

unstack (*level=-1*, *fill_value=None*)
Pivot a level of the (necessarily hierarchical) index labels, returning 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). The level involved will automatically get sorted.

Parameters

level [int, string, or list of these, default -1 (last level)] Level(s) of index to unstack, can pass level name

fill_value [replace NaN with this value if the unstack produces] missing values

New in version 0.18.0.

Returns

unstacked [DataFrame or Series]

See also:

`DataFrame.pivot` Pivot a table based on column values.

`DataFrame.stack` Pivot a level of the column labels (inverse operation from *unstack*).

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
```

`update` (*other*, *join='left'*, *overwrite=True*, *filter_func=None*, *raise_conflict=False*)
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) -> boolean 1d-array, optional] Can choose to replace values other than NA. Return True for values that should be updated.

raise_conflict [bool, default False] If True, will raise a ValueError if the DataFrame and *other* both contain non-NA data in the same place.

Raises

ValueError When *raise_conflict* is True and there's overlapping non-NA data.

See also:

`dict.update` Similar method for dictionaries.

`DataFrame.merge` For column(s)-on-columns(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, it's 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])
```

(continues on next page)

(continued from previous page)

```
>>> 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    4.0
1  2  500.0
2  3    6.0
```

values

Return a Numpy representation of the DataFrame.

Only the values in the DataFrame will be returned, the axes labels will be removed.

Returns

numpy.ndarray The values of the DataFrame.

See also:

pandas.DataFrame.index Retrievie the index labels

pandas.DataFrame.columns Retrieving the column names

Notes

The dtype will be a lower-common-denominator dtype (implicit upcasting); that is to say if the dtypes (even of numeric types) are mixed, the one that accommodates all will be chosen. Use this with care if you are not dealing with the blocks.

e.g. If the dtypes are float16 and float32, dtype will be upcast to float32. If dtypes are int32 and uint8, dtype will be upcast to int32. By `numpy.find_common_type()` convention, mixing int64 and uint64 will result in a float64 dtype.

Examples

A DataFrame where all columns are the same type (e.g., int64) results in an array of the same type.

```
>>> df = pd.DataFrame({'age':    [ 3,  29],
...                    'height': [94, 170],
...                    'weight': [31, 115]})
>>> df
   age  height  weight
0    3      94      31
1   29     170     115
```

(continues on next page)

(continued from previous page)

```
>>> df.dtypes
age       int64
height    int64
weight    int64
dtype: object
>>> df.values
array([[  3,  94,  31],
       [ 29, 170, 115]], dtype=int64)
```

A DataFrame with mixed type columns(e.g., str/object, int64, float32) results in an ndarray of the broadest type that accommodates these mixed types (e.g., object).

```
>>> df2 = pd.DataFrame([('parrot',   24.0, 'second'),
...                     ('lion',     80.5, 1),
...                     ('monkey', np.nan, None)],
...                    columns=('name', 'max_speed', 'rank'))
>>> df2.dtypes
name          object
max_speed    float64
rank          object
dtype: object
>>> df2.values
array([['parrot', 24.0, 'second'],
       ['lion', 80.5, 1],
       ['monkey', nan, None]], dtype=object)
```

var(*axis=None*, *skipna=None*, *level=None*, *ddof=1*, *numeric_only=None*, ***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)}]

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

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 [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

Returns

var [Series or DataFrame (if level specified)]

where(*cond*, *other=nan*, *inplace=False*, *axis=None*, *level=None*, *errors='raise'*, *try_cast=False*, *raise_on_error=None*)
Return an object of same shape as self and whose corresponding entries are from self where *cond* is True and otherwise are from *other*.

Parameters

cond [boolean NDFrame, 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 NDFrame and should return boolean NDFrame or array. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1: A callable can be used as cond.

other [scalar, NDFrame, or callable] Entries where *cond* is False are replaced with corresponding value from *other*. If other is callable, it is computed on the NDFrame and should return scalar or NDFrame. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1: A callable can be used as other.

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

axis [alignment axis if needed, default None]

level [alignment level if needed, default None]

errors [str, {'raise', 'ignore'}, default 'raise']

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

Note that currently this parameter won't affect the results and will always coerce to a suitable dtype.

try_cast [boolean, default False] try to cast the result back to the input type (if possible),

raise_on_error [boolean, default True] Whether to raise on invalid data types (e.g. trying to where on strings)

Deprecated since version 0.21.0.

Returns

wh [same type as caller]

See also:

`DataFrame.mask()`

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.

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.

Examples

```
>>> s = pd.Series(range(5))
>>> s.where(s > 0)
0    NaN
1    1.0
2    2.0
```

(continues on next page)

(continued from previous page)

```
3    3.0
4    4.0
```

```
>>> s.mask(s > 0)
0    0.0
1    NaN
2    NaN
3    NaN
4    NaN
```

```
>>> s.where(s > 1, 10)
0    10.0
1    10.0
2    2.0
3    3.0
4    4.0
```

```
>>> df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])
>>> 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
```

xs(*key*, *axis=0*, *level=None*, *drop_level=True*)

Returns a cross-section (row(s) or column(s)) from the Series/DataFrame. Defaults to cross-section on the rows (axis=0).

Parameters

key [object] Some label contained in the index, or partially in a MultiIndex

axis [int, 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 [boolean, default True] If False, returns object with same levels as self.

Returns

xs [Series or DataFrame]

Notes

xs is only for getting, not setting 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

```
>>> df
   A  B  C
a  4  5  2
b  4  0  9
c  9  7  3
>>> df.xs('a')
A    4
B    5
C    2
Name: a
>>> df.xs('C', axis=1)
a    2
b    9
c    3
Name: C
```

```
>>> df
                     A  B  C  D
first second third
bar   one    1       4  1  8  9
      two    1       7  5  5  0
baz   one    1       6  6  8  0
      three  2       5  3  5  3
>>> df.xs(('baz', 'three'))
       A  B  C  D
third
2      5  3  5  3
>>> df.xs('one', level=1)
             A  B  C  D
first third
bar   1      4  1  8  9
baz   1      6  6  8  0
>>> df.xs(('baz', 2), level=[0, 'third'])
        A  B  C  D
second
three   5  3  5  3
```

class oddt.pandas.**ChemPanel**(*data=None*, *items=None*, *major_axis=None*, *minor_axis=None*, *copy=False*, *dtype=None*)

Bases: pandas.core.panel.Panel

Modified *pandas.Panel* to adopt higher dimension data than *ChemDataFrame*. Main purpose is to store molecular fingerprints in one column and keep 2D numpy array underneath.

New in version 0.3.

Attributes

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

axes Return index label(s) of the internal NDFrame

blocks Internal property, property synonym for as_blocks()

dtypes Return the dtypes in the DataFrame.

empty Indicator whether DataFrame is empty.

ftypes Return the ftypes (indication of sparse/dense and dtype) in DataFrame.

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

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

is_copy

items items

ix A primarily label-location based indexer, with integer position fallback.

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

major_axis major_axis

minor_axis minor_axis

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

shape Return a tuple of axis dimensions

size Return an int representing the number of elements in this 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])	Addition of series and other, element-wise (binary operator *add*).
add_prefix(prefix)	Prefix labels with string *prefix*.
add_suffix(suffix)	Suffix labels with string *suffix*.
all([axis, bool_only, skipna, level])	Return whether all elements are True, potentially over an axis.
any([axis, bool_only, skipna, level])	Return whether any element is True over requested axis.
apply(func[, axis])	Applies function along axis (or axes) of the Panel
as_blocks([copy])	Convert the frame to a dict of dtype -> Constructor Types that each has a homogeneous dtype.
asfreq(freq[, method, how, normalize, ...])	Convert TimeSeries to specified frequency.
asof(where[, subset])	The last row without any NaN is taken (or the last row without NaN considering only the subset of columns in the case of a DataFrame)
astype(**kwargs)	Cast a pandas object to a specified dtype `dtype`.
at_time(time[, asof])	Select values at particular time of day (e.g.

Continued on next page

Table 20 – continued from previous page

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])	Synonym for DataFrame.fillna(method='bfill')
bool()	Return the bool of a single element PandasObject.
clip([lower, upper, axis, inplace])	Trim values at input threshold(s).
clip_lower(threshold[, axis, inplace])	Return copy of the input with values below a threshold truncated.
clip_upper(threshold[, axis, inplace])	Return copy of input with values above given value(s) truncated.
compound([axis, skipna, level])	Return the compound percentage of the values for the requested axis
conform(frame[, axis])	Conform input DataFrame to align with chosen axis pair.
consolidate([inplace])	Compute NDFrame with "consolidated" internals (data of each dtype grouped together in a single ndarray).
convert_objects([convert_dates, ...])	Attempt to infer better dtype for object columns.
copy([deep])	Make a copy of this object's indices and data.
count([axis])	Return number of observations over requested axis.
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])	Generates descriptive statistics that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding NaN values.
div(other[, axis])	Floating division of series and other, element-wise (binary operator *truediv*).
divide(other[, axis])	Floating division of series and other, element-wise (binary operator *truediv*).
dropna([axis, how, inplace])	Drop 2D from panel, holding passed axis constant
eq(other[, axis])	Wrapper for comparison method eq
equals(other)	Determines if two NDFrame objects contain the same elements.
ffill([axis, inplace, limit, downcast])	Synonym for DataFrame.fillna(method='ffill')
fillna([value, method, axis, inplace, ...])	Fill NA/NaN values using the specified method
filter([items, like, regex, axis])	Subset rows or columns of dataframe according to labels in the specified index.
first(offset)	Convenience method for subsetting initial periods of time series data based on a date offset.
first_valid_index()	Return index for first non-NA/null value.
floordiv(other[, axis])	Integer division of series and other, element-wise (binary operator *floordiv*).
fromDict(data[, intersect, orient, dtype])	Construct Panel from dict of DataFrame objects
from_dict(data[, intersect, orient, dtype])	Construct Panel from dict of DataFrame objects

Continued on next page

Table 20 – continued from previous page

ge(other[, axis])	Wrapper for comparison method ge
get(key[, default])	Get item from object for given key (DataFrame column, Panel slice, etc.).
get_dtype_counts()	Return counts of unique dtypes in this object.
get_ftype_counts()	Return counts of unique ftypes in this object.
get_value(*args, **kwargs)	Quickly retrieve single value at (item, major, minor) location
get_values()	Return an ndarray after converting sparse values to dense.
groupby(function[, axis])	Group data on given axis, returning GroupBy object
gt(other[, axis])	Wrapper for comparison method gt
infer_objects()	Attempt to infer better dtypes for object columns.
interpolate([method, axis, limit, inplace, ...])	Interpolate values according to different methods.
isna()	Detect missing values.
isnull()	Detect missing values.
iteritems()	Iterate over (label, values) on info axis
join(other[, how, lsuffix, rsuffix])	Join items with other Panel either on major and minor axes column
keys()	Get the 'info axis' (see Indexing for more)
kurt([axis, skipna, level, numeric_only])	Return unbiased kurtosis over requested axis using Fisher's definition of kurtosis (kurtosis of normal == 0.0).
kurtosis([axis, skipna, level, numeric_only])	Return unbiased kurtosis over requested axis using Fisher's definition of kurtosis (kurtosis of normal == 0.0).
last(offset)	Convenience method for subsetting final periods of time series data based on a date offset.
last_valid_index()	Return index for last non-NA/null value.
le(other[, axis])	Wrapper for comparison method le
lt(other[, axis])	Wrapper for comparison method lt
mad([axis, skipna, level])	Return the mean absolute deviation of the values for the requested axis
major_xs(key)	Return slice of panel along major axis
mask(cond[, other, inplace, axis, level, ...])	Return an object of same shape as self and whose corresponding entries are from self where *cond* is False and otherwise are from *other*.
max([axis, skipna, level, numeric_only])	This method returns the maximum of the values in the object.
mean([axis, skipna, level, numeric_only])	Return the mean of the values for the requested axis
median([axis, skipna, level, numeric_only])	Return the median of the values for the requested axis
min([axis, skipna, level, numeric_only])	This method returns the minimum of the values in the object.
minor_xs(key)	Return slice of panel along minor axis
mod(other[, axis])	Modulo of series and other, element-wise (binary operator *mod*).
mul(other[, axis])	Multiplication of series and other, element-wise (binary operator *mul*).
multiply(other[, axis])	Multiplication of series and other, element-wise (binary operator *mul*).
ne(other[, axis])	Wrapper for comparison method ne

Continued on next page

Table 20 – continued from previous page

notna()	Detect existing (non-missing) values.
notnull()	Detect existing (non-missing) values.
pct_change([periods, fill_method, limit, freq])	Percentage change between the current and a prior element.
pipe(func, *args, **kwargs)	Apply func(self, *args, **kwargs)
pop(item)	Return item and drop from frame.
pow(other[, axis])	Exponential power of series and other, element-wise (binary operator *pow*).
prod([axis, skipna, level, numeric_only, ...])	Return the product of the values for the requested axis
product([axis, skipna, level, numeric_only, ...])	Return the product of the values for the requested axis
radd(other[, axis])	Addition of series and other, element-wise (binary operator *radd*).
rank([axis, method, numeric_only, ...])	Compute numerical data ranks (1 through n) along axis.
rdiv(other[, axis])	Floating division of series and other, element-wise (binary operator *rtruediv*).
reindex(*args, **kwargs)	Conform Panel to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index.
reindex_axis(labels[, axis, method, level, ...])	Conform input object to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index.
reindex_like(other[, method, copy, limit, ...])	Return an object with matching indices to myself.
rename([items, major_axis, minor_axis])	Alter axes input function or functions.
rename_axis(mapper[, axis, copy, inplace])	Alter the name of the index or columns.
replace([to_replace, value, inplace, limit, ...])	Replace values given in *to_replace* with *value*.
resample(rule[, how, axis, fill_method, ...])	Convenience method for frequency conversion and resampling of time series.
rfloordiv(other[, axis])	Integer division of series and other, element-wise (binary operator *rfloordiv*).
rmod(other[, axis])	Modulo of series and other, element-wise (binary operator *rmod*).
rmul(other[, axis])	Multiplication of series and other, element-wise (binary operator *rmul*).
round([decimals])	Round each value in Panel to a specified number of decimal places.
rpow(other[, axis])	Exponential power of series and other, element-wise (binary operator *rpow*).
rsub(other[, axis])	Subtraction of series and other, element-wise (binary operator *rsub*).
rtruediv(other[, axis])	Floating division of series and other, element-wise (binary operator *rtruediv*).
sample([n, frac, replace, weights, ...])	Return a random sample of items from an axis of object.
select(crit[, axis])	Return data corresponding to axis labels matching criteria
sem([axis, skipna, level, ddof, numeric_only])	Return unbiased standard error of the mean over requested axis.
set_axis(labels[, axis, inplace])	Assign desired index to given axis.

Continued on next page

Table 20 – continued from previous page

set_value(*args, **kwargs)	Quickly set single value at (item, major, minor) location
shift([periods, freq, axis])	Shift index by desired number of periods with an optional time freq.
skew([axis, skipna, level, numeric_only])	Return unbiased skew over requested axis Normalized by N-1
slice_shift([periods, axis])	Equivalent to *shift* without copying data.
sort_index([axis, level, ascending, ...])	Sort object by labels (along an axis)
sort_values([by, axis, ascending, inplace, ...])	NOT IMPLEMENTED: do not call this method, as sorting values is not supported for Panel objects and will raise an error.
squeeze([axis])	Squeeze length 1 dimensions.
std([axis, skipna, level, ddof, numeric_only])	Return sample standard deviation over requested axis.
sub(other[, axis])	Subtraction of series and other, element-wise (binary operator *sub*).
subtract(other[, axis])	Subtraction of series and other, element-wise (binary operator *sub*).
sum([axis, skipna, level, numeric_only, ...])	Return the sum of the values for 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 on a particular axis
take(indices[, axis, convert, is_copy])	Return the elements in the given *positional* indices along an axis.
to_clipboard([excel, sep])	Copy object to the system clipboard.
to_dense()	Return dense representation of NDFrame (as opposed to sparse)
to_excel(path[, na_rep, engine])	Write each DataFrame in Panel to a separate excel sheet
to_frame([filter_observations])	Transform wide format into long (stacked) format as DataFrame whose columns are the Panel's items and whose index is a MultiIndex formed of the Panel's major and minor axes.
to_hdf(path_or_buf, key, **kwargs)	Write the contained data to an HDF5 file using HDFStore.
to_json([path_or_buf, orient, date_format, ...])	Convert the object to a JSON string.
to_latex([buf, columns, col_space, header, ...])	Render an object to a tabular environment table.
to_msgpack([path_or_buf, encoding])	msgpack (serialize) object to input file path
to_pickle(path[, compression, protocol])	Pickle (serialize) object to file.
to_sparse(*args, **kwargs)	NOT IMPLEMENTED: do not call this method, as sparsifying is not supported for Panel objects and will raise an error.
to_sql(name, con[, schema, if_exists, ...])	Write records stored in a DataFrame to a SQL database.
to_xarray()	Return an xarray object from the pandas object.
transpose(*args, **kwargs)	Permute the dimensions of the Panel
truediv(other[, axis])	Floating division of series 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.

Continued on next page

Table 20 – continued from previous page

tz_localize(tz[, axis, level, copy, ambiguous])	Localize tz-naive TimeSeries to target time zone.
update(other[, join, overwrite, ...])	Modify Panel in place using non-NA values from passed Panel, or object coercible to Panel.
var([axis, skipna, level, ddof, numeric_only])	Return unbiased variance over requested axis.
where(cond[, other, inplace, axis, level, ...])	Return an object of same shape as self and whose corresponding entries are from self where *cond* is True and otherwise are from *other*.
xs(key[, axis])	Return slice of panel along selected axis

agg	
aggregate	
align	
as_matrix	
drop	
head	
tail	
tshift	

abs()

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()
```

(continues on next page)

(continued from previous page)

```
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
```

add(*other*, *axis=0*)

Addition of series and other, element-wise (binary operator *add*). Equivalent to `panel + other`.

Parameters

other [DataFrame or Panel]

axis [{items, major_axis, minor_axis}] Axis to broadcast over

Returns

Panel

See also:

`Panel.radd`

add_prefix(*prefix*)

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.

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
```

add_suffix(*suffix*)

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.

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
```

agg (*func*, **args*, ***kwargs*)

aggregate (*func*, **args*, ***kwargs*)

align (*other*, ***kwargs*)

Align two objects on their axes with the specified join method for each axis Index

Parameters

other [DataFrame or Series]

join [{'outer', 'inner', 'left', 'right'}, default 'outer']

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 [boolean, 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 [str, default None]

limit [int, default None]

fill_axis [int or labels for object, default 0] Filling axis, method and limit

broadcast_axis [int or labels for object, default None] Broadcast values along this axis, if aligning two objects of different dimensions

Returns

(left, right) [(NDFrame, type of other)] Aligned objects

all (*axis=0*, *bool_only=None*, *skipna=True*, *level=None*, ***kwargs*)

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

Returns True if all elements within a series or along a Dataframe axis are non-zero, not-empty or not-False.

Parameters

axis [{0 or 'index', 1 or 'columns', None}, default 0] Indicate which axis or axes should be reduced.

- 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.

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame.

bool_only [boolean, default None] Include only boolean columns. If None, will attempt to use everything, then use only boolean data. Not implemented for Series.

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

Returns

all [DataFrame or Panel (if level specified)]

See also:

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

pandas.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
```

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 column-wise values all return True.

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

Specify `axis='columns'` to check if row-wise values 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
```

any (*axis=0*, *bool_only=None*, *skipna=True*, *level=None*, ***kwargs*)

Return whether any element is True over requested axis.

Unlike `DataFrame.all()`, this performs an *or* operation. If any of the values along the specified axis is True, this will return True.

Parameters

axis [{0 or 'index', 1 or 'columns', None}, default 0] Indicate which axis or axes should be reduced.

- 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.

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame.

bool_only [boolean, default None] Include only boolean columns. If None, will attempt to use everything, then use only boolean data. Not implemented for Series.

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

Returns

any [DataFrame or Panel (if level specified)]

See also:

pandas.DataFrame.all Return whether all elements are True.

Examples

Series

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

```
>>> pd.Series([True, False]).any()
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
       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)
```

apply (*func*, *axis='major'*, ***kwargs*)

Applies function along axis (or axes) of the Panel

Parameters

func [function] Function to apply to each combination of 'other' axes e.g. if axis = 'items', the combination of major_axis/minor_axis will each be passed as a Series; if axis = ('items', 'major'), DataFrames of items & major axis will be passed

axis [{'items', 'minor', 'major'}, or {0, 1, 2}, or a tuple with two] axes

Additional keyword arguments will be passed as keywords to the function

Returns

result [Panel, DataFrame, or Series]

Examples

Returns a Panel with the square root of each element

```
>>> p = pd.Panel(np.random.rand(4,3,2))
>>> p.apply(np.sqrt)
```

Equivalent to p.sum(1), returning a DataFrame

```
>>> p.apply(lambda x: x.sum(), axis=1)
```

Equivalent to previous:

```
>>> p.apply(lambda x: x.sum(), axis='major')
```

Return the shapes of each DataFrame over axis 2 (i.e the shapes of items x major), as a Series

```
>>> p.apply(lambda x: x.shape, axis=(0,1))
```

as_blocks (*copy=True*)

Convert the frame to a dict of dtype -> Constructor Types that each has a homogeneous dtype.

Deprecated since version 0.21.0.

NOTE: the dtypes of the blocks WILL BE PRESERVED HERE (unlike in as_matrix)

Parameters

copy [boolean, default True]

Returns

values [a dict of dtype -> Constructor Types]

as_matrix()

Convert the frame to its Numpy-array representation.

Deprecated since version 0.23.0: Use `DataFrame.values()` instead.

Parameters

columns: list, optional, default:None If None, return all columns, otherwise, returns specified columns.

Returns

values [ndarray] If the caller is heterogeneous and contains booleans or objects, the result will be of dtype=object. See Notes.

See also:

`pandas.DataFrame.values`

Notes

Return is NOT a Numpy-matrix, rather, a Numpy-array.

The dtype will be a lower-common-denominator dtype (implicit upcasting); that is to say if the dtypes (even of numeric types) are mixed, the one that accommodates all will be chosen. Use this with care if you are not dealing with the blocks.

e.g. If the dtypes are float16 and float32, dtype will be upcast to float32. If dtypes are int32 and uint8, dtype will be upcase to int32. By numpy.find_common_type convention, mixing int64 and uint64 will result in a flot64 dtype.

This method is provided for backwards compatibility. Generally, it is recommended to use '.values'.

asfreq(*freq*, *method=None*, *how=None*, *normalize=False*, *fill_value=None*)
Convert TimeSeries to specified frequency.

Optionally provide filling method to pad/backfill missing values.

Returns the original data conformed to a new index with the specified frequency. `resample` is more appropriate if an operation, such as summarization, is necessary to represent the data at the new frequency.

Parameters

freq [DateOffset object, 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).

New in version 0.20.0.

Returns

converted [type of caller]

See also:

reindex

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
```

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

The last row without any NaN is taken (or the last row without NaN considering only the subset of columns in the case of a DataFrame)

New in version 0.19.0: For DataFrame

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

Parameters

where [date or array of dates]

subset [string or list of strings, default None] if not None use these columns for NaN propagation

Returns

where is scalar

- value or NaN if input is Series
- Series if input is DataFrame

where is Index: same shape object as input

See also:

`merge_asof`

Notes

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

astype(***kwargs*)

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

Parameters

dtype [data type, or dict of column name -> data type] Use a numpy.dtype or Python type to cast entire pandas object to the same type. Alternatively, use {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

New in version 0.20.0.

raise_on_error [raise on invalid input] Deprecated since version 0.20.0: Use `errors` instead

kwargs [keyword arguments to pass on to the constructor]

Returns

casted [type of caller]

See also:

pandas.to_datetime Convert argument to datetime.

pandas.to_timedelta Convert argument to timedelta.

pandas.to_numeric Convert argument to a numeric type.

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

Examples

```
>>> 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, int64): [1, 2]
```

Convert to ordered categorical type with custom ordering:

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

Note that using `copy=False` and changing data on a new pandas object may propagate changes:

```
>>> s1 = pd.Series([1,2])
>>> s2 = s1.astype('int64', copy=False)
>>> s2[0] = 10
>>> s1  # note that s1[0] has changed too
0    10
1     2
dtype: int64
```

at

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

Similar to `loc`, in that both provide label-based lookups. Use `at` if you only need to get or set a single value in a DataFrame or Series.

Raises

KeyError When label does not exist in DataFrame

See also:

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

DataFrame.loc Access a group of rows and columns by label(s)

Series.at Access a single value using a label

Examples

```
>>> df = pd.DataFrame([[0, 2, 3], [0, 4, 1], [10, 20, 30]],
...                   index=[4, 5, 6], columns=['A', 'B', 'C'])
>>> df
    A   B   C
4   0   2   3
5   0   4   1
6  10  20  30
```

Get value at specified row/column pair

```
>>> df.at[4, 'B']
2
```

Set value at specified row/column pair

```
>>> df.at[4, 'B'] = 10
>>> df.at[4, 'B']
10
```

Get value within a Series

```
>>> df.loc[5].at['B']
4
```

at_time (*time*, *asof=False*)

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

Parameters

time [datetime.time or string]

Returns

values_at_time [type of caller]

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
```

(continues on next page)

(continued from previous page)

```
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
```

axes
Return index label(s) of the internal NDFrame

between_time (*start_time*, *end_time*, *include_start=True*, *include_end=True*)
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 string]

end_time [datetime.time or string]

include_start [boolean, default True]

include_end [boolean, default True]

Returns

values_between_time [type of caller]

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
```

bfill (*axis=None*, *inplace=False*, *limit=None*, *downcast=None*)
Synonym for `DataFrame.fillna(method='bfill')`

blocks
Internal property, property synonym for as_blocks()

Deprecated since version 0.21.0.

bool ()
Return the bool of a single element PandasObject.

This must be a boolean scalar value, either True or False. Raise a ValueError if the PandasObject does not have exactly 1 element, or that element is not boolean

clip (*lower=None*, *upper=None*, *axis=None*, *inplace=False*, **args*, ***kwargs*)
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.

upper [float or array_like, default None] Maximum threshold value. All values above this threshold will be set to it.

axis [int or string axis name, optional] Align object with lower and upper along the given axis.

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

New in version 0.21.0.

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

Returns

Series or DataFrame Same type as calling object with the values outside the clip boundaries replaced

See also:

clip_lower Clip values below specified threshold(s).

clip_upper Clip values above specified threshold(s).

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
```

clip_lower(*threshold*, *axis=None*, *inplace=False*)

Return copy of the input with values below a threshold truncated.

Parameters

threshold [numeric or array-like] Minimum value allowed. All values below threshold will be set to this value.

- float : every value is compared to *threshold*.
- array-like : The shape of *threshold* should match the object it's compared to. When *self* is a Series, *threshold* should be the length. When *self* is a DataFrame, *threshold* should 2-D and the same shape as *self* for `axis=None`, or 1-D and the same length as the axis being compared.

axis [{0 or 'index', 1 or 'columns'}, default 0] Align *self* with *threshold* along the given axis.

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

New in version 0.21.0.

Returns

clipped [same type as input]

See also:

`Series.clip` Return copy of input with values below and above thresholds truncated.

`Series.clip_upper` Return copy of input with values above threshold truncated.

Examples

Series single threshold clipping:

```
>>> s = pd.Series([5, 6, 7, 8, 9])
>>> s.clip_lower(8)
0    8
1    8
2    8
3    8
4    9
dtype: int64
```

Series clipping element-wise using an array of thresholds. *threshold* should be the same length as the Series.

```
>>> elemwise_thresholds = [4, 8, 7, 2, 5]
>>> s.clip_lower(elemwise_thresholds)
0    5
1    8
2    7
3    8
4    9
dtype: int64
```

DataFrames can be compared to a scalar.

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

```
>>> df.clip_lower(3)
   A  B
0  3  3
1  3  4
2  5  6
```

Or to an array of values. By default, *threshold* should be the same shape as the DataFrame.

```
>>> df.clip_lower(np.array([[3, 4], [2, 2], [6, 2]]))
   A  B
0  3  4
1  3  4
2  6  6
```

Control how *threshold* is broadcast with *axis*. In this case *threshold* should be the same length as the axis specified by *axis*.

```
>>> df.clip_lower(np.array([3, 3, 5]), axis='index')
   A  B
0  3  3
1  3  4
2  5  6
```

```
>>> df.clip_lower(np.array([4, 5]), axis='columns')
   A  B
0  4  5
1  4  5
2  5  6
```

clip_upper(*threshold*, *axis=None*, *inplace=False*)

Return copy of input with values above given value(s) truncated.

Parameters

threshold [float or array_like]

axis [int or string axis name, optional] Align object with threshold along the given axis.

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

New in version 0.21.0.

Returns

clipped [same type as input]

See also:

clip

compound(*axis=None*, *skipna=None*, *level=None*)

Return the compound percentage of the values for the requested axis

Parameters

axis [{items (0), major_axis (1), minor_axis (2)}]

skipna [boolean, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

Returns

compounded [DataFrame or Panel (if level specified)]

conform(*frame*, *axis='items'*)

Conform input DataFrame to align with chosen axis pair.

Parameters

frame [DataFrame]

axis [{'items', 'major', 'minor'}] Axis the input corresponds to. E.g., if axis='major', then the frame's columns would be items, and the index would be values of the minor axis

Returns

DataFrame

consolidate(*inplace=False*)

Compute NDFrame with "consolidated" internals (data of each dtype grouped together in a single ndarray).

Deprecated since version 0.20.0: Consolidate will be an internal implementation only.

convert_objects(*convert_dates=True*, *convert_numeric=False*, *convert_timedeltas=True*, *copy=True*)

Attempt to infer better dtype for object columns.

Deprecated since version 0.21.0.

Parameters

convert_dates [boolean, default True] If True, convert to date where possible. If 'coerce', force conversion, with unconvertible values becoming NaT.

convert_numeric [boolean, default False] If True, attempt to coerce to numbers (including strings), with unconvertible values becoming NaN.

convert_timedeltas [boolean, default True] If True, convert to timedelta where possible. If 'coerce', force conversion, with unconvertible values becoming NaT.

copy [boolean, default True] If True, return a copy even if no copy is necessary (e.g. no conversion was done). Note: This is meant for internal use, and should not be confused with inplace.

Returns

converted [same as input object]

See also:

pandas.to_datetime Convert argument to datetime.

pandas.to_timedelta Convert argument to timedelta.

pandas.to_numeric Return a fixed frequency timedelta index, with day as the default.

copy(*deep=True*)

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

copy [Series, DataFrame or Panel] 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.

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[0] = 3
>>> shallow[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
```

count (*axis='major'*)

Return number of observations over requested axis.

Parameters

axis [{'items', 'major', 'minor'} or {0, 1, 2}]

Returns

count [DataFrame]

cummax (*axis=None*, *skipna=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'.

skipna [boolean, 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

cummax [DataFrame or Panel]

See also:

pandas.core.window.Expanding.max Similar functionality but ignores `NaN` values.

Panel.max Return the maximum over Panel axis.

Panel.cummax Return cumulative maximum over Panel axis.

Panel.cummin Return cumulative minimum over Panel axis.

Panel.cumsum Return cumulative sum over Panel axis.

Panel.cumprod Return cumulative product over Panel 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
```

cummin (*axis=None*, *skipna=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'.

skipna [boolean, 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

cummin [DataFrame or Panel]

See also:

pandas.core.window.Expanding.min Similar functionality but ignores `NaN` values.

Panel.min Return the minimum over Panel axis.

Panel.cummax Return cumulative maximum over Panel axis.

Panel.cummin Return cumulative minimum over Panel axis.

Panel.cumsum Return cumulative sum over Panel axis.

Panel.cumprod Return cumulative product over Panel 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
```

cumprod (*axis=None*, *skipna=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'.

skipna [boolean, 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

cumprod [DataFrame or Panel]

See also:

`pandas.core.window.Expanding.prod` Similar functionality but ignores `NaN` values.

`Panel.prod` Return the product over Panel axis.

`Panel.cummax` Return cumulative maximum over Panel axis.

`Panel.cummin` Return cumulative minimum over Panel axis.

`Panel.cumsum` Return cumulative sum over Panel axis.

`Panel.cumprod` Return cumulative product over Panel 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
```

cumsum(*axis=None*, *skipna=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'.

skipna [boolean, 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

cumsum [DataFrame or Panel]

See also:

pandas.core.window.Expanding.sum Similar functionality but ignores `NaN` values.

Panel.sum Return the sum over Panel axis.

Panel.cummax Return cumulative maximum over Panel axis.

Panel.cummin Return cumulative minimum over Panel axis.

Panel.cumsum Return cumulative sum over Panel axis.

Panel.cumprod Return cumulative product over Panel 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
```

describe(*percentiles=None*, *include=None*, *exclude=None*)

Generates descriptive statistics 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(include=['O'])`). To exclude pandas categorical columns, use `'category'`
- None (default) : The result will exclude nothing.

Returns

summary: Series/DataFrame of summary statistics

See also:

`DataFrame.count`, `DataFrame.max`, `DataFrame.min`, `DataFrame.mean`, `DataFrame.std`, `DataFrame.select_dtypes`

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
```

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
unique                     2
top       2010-01-01 00:00:00
freq                       2
first     2000-01-01 00:00:00
last      2010-01-01 00:00:00
dtype: object
```

Describing a `DataFrame`. By default only numeric fields are returned.

```
>>> df = pd.DataFrame({ 'object': ['a', 'b', 'c'],
...                     'numeric': [1, 2, 3],
...                     'categorical': pd.Categorical(['d','e','f'])
...                   })
>>> 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      c
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=[np.object])
       object
count       3
unique      3
top         c
freq        1
```

Including only categorical columns from a `DataFrame` description.

```
>>> df.describe(include=['category'])
       categorical
count            3
unique           3
top              f
freq             1
```

Excluding numeric columns from a `DataFrame` description.

```
>>> df.describe(exclude=[np.number])
       categorical object
count            3      3
unique           3      3
top              f      c
freq             1      1
```

Excluding object columns from a `DataFrame` description.

```
>>> df.describe(exclude=[np.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
```

div(*other*, *axis=0*)

Floating division of series and other, element-wise (binary operator *truediv*). Equivalent to `panel / other`.

Parameters

other [DataFrame or Panel]

axis [{items, major_axis, minor_axis}] Axis to broadcast over

Returns

Panel

See also:

`Panel.rtruediv`

divide(*other*, *axis=0*)

Floating division of series and other, element-wise (binary operator *truediv*). Equivalent to `panel / other`.

Parameters

other [DataFrame or Panel]

axis [{items, major_axis, minor_axis}] Axis to broadcast over

Returns

Panel

See also:

`Panel.rtruediv`

drop(*labels=None*, *axis=0*, *index=None*, *columns=None*, *level=None*, *inplace=False*, *errors='raise'*)

dropna(*axis=0*, *how='any'*, *inplace=False*)

Drop 2D from panel, holding passed axis constant

Parameters

axis [int, default 0] Axis to hold constant. E.g. axis=1 will drop major_axis entries having a certain amount of NA data

how [{'all', 'any'}, default 'any'] 'any': one or more values are NA in the DataFrame along the axis. For 'all' they all must be.

inplace [bool, default False] If True, do operation inplace and return None.

Returns

dropped [Panel]

dtypes

Return the dtypes in the DataFrame.

This returns a Series with the data type of each column. The result's index is the original DataFrame's columns. Columns with mixed types are stored with the `object` dtype. See the User Guide for more.

Returns

pandas.Series The data type of each column.

See also:

pandas.DataFrame.ftypes dtype and sparsity information.

Examples

```
>>> df = pd.DataFrame({'float': [1.0],
...                    'int': [1],
...                    'datetime': [pd.Timestamp('20180310')],
...                    'string': ['foo']})
>>> df.dtypes
float              float64
int                  int64
datetime    datetime64[ns]
string              object
dtype: object
```

empty
Indicator whether DataFrame is empty.

True if DataFrame is entirely empty (no items), meaning any of the axes are of length 0.

Returns

bool If DataFrame is empty, return True, if not return False.

See also:

`pandas.Series.dropna`, `pandas.DataFrame.dropna`

Notes

If DataFrame contains only NaNs, it is still not considered empty. See the example below.

Examples

An example of an actual empty DataFrame. Notice the index is empty:

```
>>> df_empty = pd.DataFrame({'A' : []})
>>> df_empty
Empty DataFrame
Columns: [A]
Index: []
>>> df_empty.empty
True
```

If we only have NaNs in our DataFrame, it is not considered empty! We will need to drop the NaNs to make the DataFrame empty:

```
>>> df = pd.DataFrame({'A' : [np.nan]})
>>> df
    A
0 NaN
>>> df.empty
False
>>> df.dropna().empty
True
```

eq(*other*, *axis=None*)
Wrapper for comparison method eq

equals(*other*)
Determines if two NDFrame objects contain the same elements. NaNs in the same location are considered equal.

ffill(*axis=None*, *inplace=False*, *limit=None*, *downcast=None*)
Synonym for `DataFrame.fillna(method='ffill')`

fillna(*value=None*, *method=None*, *axis=None*, *inplace=False*, *limit=None*, *downcast=None*, ***kwargs*)
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', '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

axis [{0, 1, 2, 'items', 'major_axis', 'minor_axis'}]

inplace [boolean, 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

filled [Panel]

See also:

interpolate Fill NaN values using interpolation.

reindex, *asfreq*

Examples

```
>>> df = pd.DataFrame([[np.nan, 2, np.nan, 0],
...                    [3, 4, np.nan, 1],
...                    [np.nan, np.nan, np.nan, 5],
...                    [np.nan, 3, np.nan, 4]],
...                   columns=list('ABCD'))
>>> df
     A    B   C  D
0  NaN  2.0 NaN  0
1  3.0  4.0 NaN  1
2  NaN  NaN NaN  5
3  NaN  3.0 NaN  4
```

Replace all NaN elements with 0s.

```
>>> df.fillna(0)
    A   B   C   D
0   0.0 2.0 0.0 0
1   3.0 4.0 0.0 1
2   0.0 0.0 0.0 5
3   0.0 3.0 0.0 4
```

We can also propagate non-null values forward or backward.

```
>>> df.fillna(method='ffill')
    A   B   C   D
0   NaN 2.0 NaN 0
1   3.0 4.0 NaN 1
2   3.0 4.0 NaN 5
3   3.0 3.0 NaN 4
```

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
1   3.0 4.0 2.0 1
2   0.0 1.0 2.0 5
3   0.0 3.0 2.0 4
```

Only replace the first NaN element.

```
>>> df.fillna(value=values, limit=1)
    A   B   C   D
0   0.0 2.0 2.0 0
1   3.0 4.0 NaN 1
2   NaN 1.0 NaN 5
3   NaN 3.0 NaN 4
```

filter (*items=None*, *like=None*, *regex=None*, *axis=None*)

Subset rows or columns of dataframe according to labels in the specified index.

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] List of info axis to restrict to (must not all be present)

like [string] Keep info axis where "arg in col == True"

regex [string (regular expression)] Keep info axis with re.search(regex, col) == True

axis [int or string axis name] The axis to filter on. By default this is the info axis, 'index' for Series, 'columns' for DataFrame

Returns

same type as input object

See also:

`pandas.DataFrame.loc`

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
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
```

first(*offset*)

Convenience method for subsetting initial periods of time series data based on a date offset.

Parameters

offset [string, DateOffset, dateutil.relativedelta]

Returns

subset [type of 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 calender days were returned, not the first 3 days observed in the dataset, and therefore data for 2018-04-13 was not returned.

first_valid_index()
Return index for first non-NA/null value.

Returns

scalar [type of index]

Notes

If all elements are non-NA/null, returns None. Also returns None for empty NDFrame.

floordiv(*other*, *axis=0*)
Integer division of series and other, element-wise (binary operator *floordiv*). Equivalent to `panel // other`.

Parameters

other [DataFrame or Panel]

axis [{items, major_axis, minor_axis}] Axis to broadcast over

Returns

Panel

See also:

`Panel.rfloordiv`

classmethod fromDict(*data*, *intersect=False*, *orient='items'*, *dtype=None*)
Construct Panel from dict of DataFrame objects

Parameters

data [dict] {field : DataFrame}

intersect [boolean] Intersect indexes of input DataFrames

orient [{'items', 'minor'}, default 'items'] The "orientation" of the data. If the keys of the passed dict should be the items of the result panel, pass 'items' (default). Otherwise if the columns of the values of the passed DataFrame objects should be the items (which in the case of mixed-dtype data you should do), instead pass 'minor'

dtype [dtype, default None] Data type to force, otherwise infer

Returns

Panel

classmethod from_dict(*data*, *intersect=False*, *orient='items'*, *dtype=None*)
Construct Panel from dict of DataFrame objects

Parameters

data [dict] {field : DataFrame}

intersect [boolean] Intersect indexes of input DataFrames

orient [{'items', 'minor'}, default 'items'] The "orientation" of the data. If the keys of the passed dict should be the items of the result panel, pass 'items' (default). Otherwise if the columns of the values of the passed DataFrame objects should be the items (which in the case of mixed-dtype data you should do), instead pass 'minor'

dtype [dtype, default None] Data type to force, otherwise infer

Returns

Panel

ftypes

Return the ftypes (indication of sparse/dense and dtype) in DataFrame.

This returns a Series with the data type of each column. The result's index is the original DataFrame's columns. Columns with mixed types are stored with the `object` dtype. See the User Guide for more.

Returns

pandas.Series The data type and indication of sparse/dense of each column.

See also:

pandas.DataFrame.dtypes Series with just dtype information.

pandas.SparseDataFrame Container for sparse tabular data.

Notes

Sparse data should have the same dtypes as its dense representation.

Examples

```
>>> import numpy as np
>>> arr = np.random.RandomState(0).randn(100, 4)
>>> arr[arr < .8] = np.nan
>>> pd.DataFrame(arr).ftypes
0    float64:dense
1    float64:dense
2    float64:dense
3    float64:dense
dtype: object
```

```
>>> pd.SparseDataFrame(arr).ftypes
0    float64:sparse
1    float64:sparse
2    float64:sparse
3    float64:sparse
dtype: object
```

ge(*other*, *axis=None*)

Wrapper for comparison method ge

get(*key*, *default=None*)
Get item from object for given key (DataFrame column, Panel slice, etc.). Returns default value if not found.

Parameters

key [object]

Returns

value [type of items contained in object]

get_dtype_counts()
Return counts of unique dtypes in this object.

Returns

dtype [Series] Series with the count of columns with each dtype.

See also:

dtypes Return the dtypes in this object.

Examples

```
>>> a = [['a', 1, 1.0], ['b', 2, 2.0], ['c', 3, 3.0]]
>>> df = pd.DataFrame(a, columns=['str', 'int', 'float'])
>>> df
  str  int  float
0   a    1    1.0
1   b    2    2.0
2   c    3    3.0
```

```
>>> df.get_dtype_counts()
float64    1
int64      1
object     1
dtype: int64
```

get_ftype_counts()
Return counts of unique ftypes in this object.

Deprecated since version 0.23.0.

This is useful for SparseDataFrame or for DataFrames containing sparse arrays.

Returns

dtype [Series] Series with the count of columns with each type and sparsity (dense/sparse)

See also:

ftypes Return ftypes (indication of sparse/dense and dtype) in this object.

Examples

```
>>> a = [['a', 1, 1.0], ['b', 2, 2.0], ['c', 3, 3.0]]
>>> df = pd.DataFrame(a, columns=['str', 'int', 'float'])
>>> df
  str  int  float
0   a    1    1.0
1   b    2    2.0
2   c    3    3.0
```

```
>>> df.get_ftype_counts()
float64:dense    1
int64:dense      1
object:dense     1
dtype: int64
```

get_value(**args*, ***kwargs*)

Quickly retrieve single value at (item, major, minor) location

Deprecated since version 0.21.0.

Please use .at[] or .iat[] accessors.

Parameters

item [item label (panel item)]

major [major axis label (panel item row)]

minor [minor axis label (panel item column)]

takeable [interpret the passed labels as indexers, default False]

Returns

value [scalar value]

get_values()

Return an ndarray after converting sparse values to dense.

This is the same as `.values` for non-sparse data. For sparse data contained in a *pandas.SparseArray*, the data are first converted to a dense representation.

Returns

numpy.ndarray Numpy representation of DataFrame

See also:

values Numpy representation of DataFrame.

pandas.SparseArray Container for sparse data.

Examples

```
>>> df = pd.DataFrame({'a': [1, 2], 'b': [True, False],
...                    'c': [1.0, 2.0]})
>>> df
   a      b    c
0  1   True  1.0
1  2  False  2.0
```

```
>>> df.get_values()
array([[1, True, 1.0], [2, False, 2.0]], dtype=object)
```

```
>>> df = pd.DataFrame({"a": pd.SparseArray([1, None, None]),
...                    "c": [1.0, 2.0, 3.0]})
>>> df
     a    c
0  1.0  1.0
1  NaN  2.0
2  NaN  3.0
```

```
>>> df.get_values()
array([[ 1.,  1.],
       [nan,  2.],
       [nan,  3.]])
```

groupby (*function*, *axis='major'*)

Group data on given axis, returning GroupBy object

Parameters

function [callable] Mapping function for chosen access

axis [{'major', 'minor', 'items'}, default 'major']

Returns

grouped [PanelGroupBy]

gt (*other*, *axis=None*)

Wrapper for comparison method gt

head (*n=5*)

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.

Parameters

n [int, default 5] Number of rows to select.

Returns

obj_head [type of caller] The first *n* rows of the caller object.

See also:

pandas.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
```

(continues on next page)

(continued from previous page)

```
3       lion
4     monkey
5     parrot
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
```

iat

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

Similar to `iloc`, in that both provide integer-based lookups. Use `iat` if you only need to get or set a single value in a DataFrame or Series.

Raises

IndexError When integer position is out of bounds

See also:

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

DataFrame.loc Access a group of rows and columns by label(s)

DataFrame.iloc Access a group of rows and columns by integer position(s)

Examples

```
>>> df = pd.DataFrame([[0, 2, 3], [0, 4, 1], [10, 20, 30]],
...                   columns=['A', 'B', 'C'])
>>> df
    A   B   C
0   0   2   3
1   0   4   1
2  10  20  30
```

Get value at specified row/column pair

```
>>> df.iat[1, 2]
1
```

Set value at specified row/column pair

```
>>> df.iat[1, 2] = 10
>>> df.iat[1, 2]
10
```

Get value within a series

```
>>> df.loc[0].iat[1]
2
```

iloc

Purely integer-location based indexing for selection by position.

`.iloc[]` is primarily integer position based (from `0` to `length-1` of the axis), but may also be used with a boolean array.

Allowed inputs are:

- An integer, e.g. `5`.
- A list or array of integers, e.g. `[4, 3, 0]`.
- A slice object with ints, e.g. `1:7`.
- A boolean array.
- A `callable` function with one argument (the calling Series, DataFrame or Panel) and that returns valid output for indexing (one of the above)

`.iloc` will raise `IndexError` if a requested indexer is out-of-bounds, except *slice* indexers which allow out-of-bounds indexing (this conforms with python/numpy *slice* semantics).

See more at Selection by Position

infer_objects()

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.

New in version 0.21.0.

Returns

converted [same type as input object]

See also:

pandas.to_datetime Convert argument to datetime.

pandas.to_timedelta Convert argument to timedelta.

pandas.to_numeric Convert argument to numeric typeR

Examples

```
>>> df = pd.DataFrame({"A": ["a", 1, 2, 3]})
>>> df = df.iloc[1:]
>>> df
   A
```

(continues on next page)

(continued from previous page)

```
1  1
2  2
3  3
```

```
>>> df.dtypes
A    object
dtype: object
```

```
>>> df.infer_objects().dtypes
A    int64
dtype: object
```

interpolate(*method='linear'*, *axis=0*, *limit=None*, *inplace=False*, *limit_direction='forward'*, *limit_area=None*, *downcast=None*, ***kwargs*)

Interpolate values according to different methods.

Please note that only `method='linear'` is supported for DataFrames/Series with a MultiIndex.

Parameters

method [{'linear', 'time', 'index', 'values', 'nearest', 'zero',]

'slinear', 'quadratic', 'cubic', 'barycentric', 'krogh', 'polynomial', 'spline', 'piecewise_polynomial', 'from_derivatives', 'pchip', 'akima'}

- 'linear': ignore the index and treat the values as equally spaced. This is the only method supported on MultiIndexes. default
- 'time': interpolation works on daily and higher resolution data to interpolate given length of interval
- 'index', 'values': use the actual numerical values of the index
- 'nearest', 'zero', 'slinear', 'quadratic', 'cubic', 'barycentric', 'polynomial' is passed to `scipy.interpolate.interp1d`. Both 'polynomial' and 'spline' require that you also specify an *order* (int), e.g. df.interpolate(method='polynomial', order=4). These use the actual numerical values of the index.
- 'krogh', 'piecewise_polynomial', 'spline', 'pchip' and 'akima' are all wrappers around the scipy interpolation methods of similar names. These use the actual numerical values of the index. For more information on their behavior, see the scipy documentation and tutorial documentation
- 'from_derivatives' refers to BPoly.from_derivatives which replaces 'piecewise_polynomial' interpolation method in scipy 0.18

New in version 0.18.1: Added support for the 'akima' method Added interpolate method 'from_derivatives' which replaces 'piecewise_polynomial' in scipy 0.18; backwards-compatible with scipy < 0.18

axis [{0, 1}, default 0]

- 0: fill column-by-column
- 1: fill row-by-row

limit [int, default None.] Maximum number of consecutive NaNs to fill. Must be greater than 0.

limit_direction [{'forward', 'backward', 'both'}, default 'forward']

limit_area [{'inside', 'outside'}, default None]

- None: (default) no fill restriction
- 'inside' Only fill NaNs surrounded by valid values (interpolate).
- 'outside' Only fill NaNs outside valid values (extrapolate).

If limit is specified, consecutive NaNs will be filled in this direction.

New in version 0.21.0.

inplace [bool, default False] Update the NDFrame in place if possible.

downcast [optional, 'infer' or None, defaults to None] Downcast dtypes if possible.

kwargs [keyword arguments to pass on to the interpolating function.]

Returns

Series or DataFrame of same shape interpolated at the NaNs

See also:

reindex, *replace*, *fillna*

Examples

Filling in NaNs

```
>>> s = pd.Series([0, 1, np.nan, 3])
>>> s.interpolate()
0    0
1    1
2    2
3    3
dtype: float64
```

is_copy

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

NDFrame Mask of bool values for each element in NDFrame that indicates whether an element is not an NA value.

See also:

NDFrame.isnull alias of isna

NDFrame.notna boolean inverse of isna

NDFrame.dropna omit axes labels with missing values

isna top-level isna

Examples

Show which entries in a DataFrame are NA.

```
>>> df = pd.DataFrame({'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
```

isnull()

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

NDFrame Mask of bool values for each element in NDFrame that indicates whether an element is not an NA value.

See also:

NDFrame.isnull alias of isna

NDFrame.notna boolean inverse of isna

NDFrame.dropna omit axes labels with missing values

isna top-level isna

Examples

Show which entries in a DataFrame are NA.

```
>>> df = pd.DataFrame({'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
```

items

iteritems()
Iterate over (label, values) on info axis

This is index for Series, columns for DataFrame, major_axis for Panel, and so on.

ix

A primarily label-location based indexer, with integer position fallback.

Warning: Starting in 0.20.0, the .ix indexer is deprecated, in favor of the more strict .iloc and .loc indexers.

`.ix[]` supports mixed integer and label based access. It is primarily label based, but will fall back to integer positional access unless the corresponding axis is of integer type.

`.ix` is the most general indexer and will support any of the inputs in `.loc` and `.iloc`. `.ix` also supports floating point label schemes. `.ix` is exceptionally useful when dealing with mixed positional and label based hierarchical indexes.

However, when an axis is integer based, ONLY label based access and not positional access is supported. Thus, in such cases, it's usually better to be explicit and use `.iloc` or `.loc`.

See more at Advanced Indexing.

join(*other*, *how='left'*, *lsuffix=''*, *rsuffix=''*)
Join items with other Panel either on major and minor axes column

Parameters

other [Panel or list of Panels] Index should be similar to one of the columns in this one

how [{'left', 'right', 'outer', 'inner'}] How to handle indexes of the two objects. Default: 'left' for joining on index, None otherwise * left: use calling frame's index * right: use input frame's index * outer: form union of indexes * inner: use intersection of indexes

lsuffix [string] Suffix to use from left frame's overlapping columns

rsuffix [string] Suffix to use from right frame's overlapping columns

Returns

joined [Panel]

keys()
Get the 'info axis' (see Indexing for more)

This is index for Series, columns for DataFrame and major_axis for Panel.

kurt(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)
Return unbiased kurtosis over requested axis using Fisher's definition of kurtosis (kurtosis of normal == 0.0). Normalized by N-1

Parameters

axis [{items (0), major_axis (1), minor_axis (2)}]

skipna [boolean, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

Returns

kurt [DataFrame or Panel (if level specified)]

kurtosis(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)
Return unbiased kurtosis over requested axis using Fisher's definition of kurtosis (kurtosis of normal == 0.0). Normalized by N-1

Parameters

axis [{items (0), major_axis (1), minor_axis (2)}]

skipna [boolean, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

Returns

kurt [DataFrame or Panel (if level specified)]

last(*offset*)

Convenience method for subsetting final periods of time series data based on a date offset.

Parameters

offset [string, DateOffset, dateutil.relativedelta]

Returns

subset [type of 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

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 calender days were returned, not the last 3 observed days in the dataset, and therefore data for 2018-04-11 was not returned.

last_valid_index()

Return index for last non-NA/null value.

Returns

scalar [type of index]

Notes

If all elements are non-NA/null, returns None. Also returns None for empty NDFrame.

le(*other*, *axis=None*)

Wrapper for comparison method le

loc

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

`.loc[]` is primarily label based, but may also be used with a boolean array.

Allowed inputs are:

- A single label, e.g. `5` or `'a'`, (note that `5` is interpreted as a *label* of the index, and **never** as an integer position along the index).
- A list or array of labels, e.g. `['a', 'b', 'c']`.
- A slice object with labels, e.g. `'a':'f'`.

  Warning: Note that contrary to usual python slices, **both** the start and the stop are included

- A boolean array of the same length as the axis being sliced, e.g. `[True, False, True]`.
- A `callable` function with one argument (the calling Series, DataFrame or Panel) and that returns valid output for indexing (one of the above)

See more at Selection by Label

Raises

KeyError: when any items are not found

See also:

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

DataFrame.iloc Access group of rows and columns by integer position(s)

DataFrame.xs Returns a cross-section (row(s) or column(s)) from the Series/DataFrame.

Series.loc Access group of values using labels

Examples

Getting values

```
>>> df = pd.DataFrame([[1, 2], [4, 5], [7, 8]],
...       index=['cobra', 'viper', 'sidewinder'],
...       columns=['max_speed', 'shield'])
>>> df
            max_speed  shield
cobra               1       2
viper               4       5
sidewinder          7       8
```

Single label. Note this returns the row as a Series.

```
>>> df.loc['viper']
max_speed    4
shield       5
Name: viper, dtype: int64
```

List of labels. Note using `[[]]` returns a DataFrame.

```
>>> df.loc[['viper', 'sidewinder']]
            max_speed  shield
viper               4       5
sidewinder          7       8
```

Single label for row and column

```
>>> df.loc['cobra', 'shield']
2
```

Slice with labels for row and single label for column. As mentioned above, note that both the start and stop of the slice are included.

```
>>> df.loc['cobra':'viper', 'max_speed']
cobra    1
viper    4
Name: max_speed, dtype: int64
```

Boolean list with the same length as the row axis

```
>>> df.loc[[False, False, True]]
            max_speed  shield
sidewinder          7       8
```

Conditional that returns a boolean Series

```
>>> df.loc[df['shield'] > 6]
            max_speed  shield
sidewinder          7       8
```

Conditional that returns a boolean Series with column labels specified

```
>>> df.loc[df['shield'] > 6, ['max_speed']]
            max_speed
sidewinder          7
```

Callable that returns a boolean Series

```
>>> df.loc[lambda df: df['shield'] == 8]
            max_speed  shield
sidewinder          7       8
```

Setting values

Set value for all items matching the list of labels

```
>>> df.loc[['viper', 'sidewinder'], ['shield']] = 50
>>> df
            max_speed  shield
cobra               1       2
viper               4      50
sidewinder          7      50
```

Set value for an entire row

```
>>> df.loc['cobra'] = 10
>>> df
```

(continues on next page)

(continued from previous page)

```
            max_speed  shield
cobra              10      10
viper               4      50
sidewinder          7      50
```

Set value for an entire column

```
>>> df.loc[:, 'max_speed'] = 30
>>> df
            max_speed  shield
cobra              30      10
viper              30      50
sidewinder         30      50
```

Set value for rows matching callable condition

```
>>> df.loc[df['shield'] > 35] = 0
>>> df
            max_speed  shield
cobra              30      10
viper               0       0
sidewinder          0       0
```

Getting values on a DataFrame with an index that has integer labels

Another example using integers for the index

```
>>> df = pd.DataFrame([[1, 2], [4, 5], [7, 8]],
...       index=[7, 8, 9], columns=['max_speed', 'shield'])
>>> df
   max_speed  shield
7          1       2
8          4       5
9          7       8
```

Slice with integer labels for rows. As mentioned above, note that both the start and stop of the slice are included.

```
>>> df.loc[7:9]
   max_speed  shield
7          1       2
8          4       5
9          7       8
```

Getting values with a MultiIndex

A number of examples using a DataFrame with a MultiIndex

```
>>> tuples = [
...     ('cobra', 'mark i'), ('cobra', 'mark ii'),
...     ('sidewinder', 'mark i'), ('sidewinder', 'mark ii'),
...     ('viper', 'mark ii'), ('viper', 'mark iii')
... ]
>>> index = pd.MultiIndex.from_tuples(tuples)
>>> values = [[12, 2], [0, 4], [10, 20],
...           [1, 4], [7, 1], [16, 36]]
>>> df = pd.DataFrame(values, columns=['max_speed', 'shield'], index=index)
```

(continues on next page)

(continued from previous page)

```
>>> df
                     max_speed  shield
cobra      mark i            12       2
           mark ii            0       4
sidewinder mark i            10      20
           mark ii            1       4
viper      mark ii            7       1
           mark iii          16      36
```

Single label. Note this returns a DataFrame with a single index.

```
>>> df.loc['cobra']
         max_speed  shield
mark i          12       2
mark ii          0       4
```

Single index tuple. Note this returns a Series.

```
>>> df.loc[('cobra', 'mark ii')]
max_speed    0
shield       4
Name: (cobra, mark ii), dtype: int64
```

Single label for row and column. Similar to passing in a tuple, this returns a Series.

```
>>> df.loc['cobra', 'mark i']
max_speed    12
shield        2
Name: (cobra, mark i), dtype: int64
```

Single tuple. Note using `[[]]` returns a DataFrame.

```
>>> df.loc[[('cobra', 'mark ii')]]
               max_speed  shield
cobra mark ii          0       4
```

Single tuple for the index with a single label for the column

```
>>> df.loc[('cobra', 'mark i'), 'shield']
2
```

Slice from index tuple to single label

```
>>> df.loc[('cobra', 'mark i'):'viper']
                     max_speed  shield
cobra      mark i            12       2
           mark ii            0       4
sidewinder mark i            10      20
           mark ii            1       4
viper      mark ii            7       1
           mark iii          16      36
```

Slice from index tuple to index tuple

```
>>> df.loc[('cobra', 'mark i'):('viper', 'mark ii')]
                     max_speed  shield
```

(continues on next page)

(continued from previous page)

```
cobra      mark i          12       2
           mark ii          0       4
sidewinder mark i          10      20
           mark ii          1       4
viper      mark ii          7       1
```

lt(*other*, *axis=None*)
Wrapper for comparison method lt

mad(*axis=None*, *skipna=None*, *level=None*)
Return the mean absolute deviation of the values for the requested axis

Parameters

axis [{items (0), major_axis (1), minor_axis (2)}]

skipna [boolean, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

Returns

mad [DataFrame or Panel (if level specified)]

major_axis

major_xs(*key*)
Return slice of panel along major axis

Parameters

key [object] Major axis label

Returns

y [DataFrame] index -> minor axis, columns -> items

Notes

major_xs is only for getting, not setting values.

MultiIndex Slicers is a generic way to get/set values on any level or levels and is a superset of major_xs functionality, see MultiIndex Slicers

mask(*cond*, *other=nan*, *inplace=False*, *axis=None*, *level=None*, *errors='raise'*, *try_cast=False*, *raise_on_error=None*)
Return an object of same shape as self and whose corresponding entries are from self where *cond* is False and otherwise are from *other*.

Parameters

cond [boolean NDFrame, 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 NDFrame and should return boolean NDFrame or array. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1: A callable can be used as cond.

other [scalar, NDFrame, or callable] Entries where *cond* is True are replaced with corresponding value from *other*. If other is callable, it is computed on the NDFrame and should return scalar or NDFrame. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1: A callable can be used as other.

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

axis [alignment axis if needed, default None]

level [alignment level if needed, default None]

errors [str, {'raise', 'ignore'}, default 'raise']

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

Note that currently this parameter won't affect the results and will always coerce to a suitable dtype.

try_cast [boolean, default False] try to cast the result back to the input type (if possible),

raise_on_error [boolean, default True] Whether to raise on invalid data types (e.g. trying to where on strings)

Deprecated since version 0.21.0.

Returns

wh [same type as caller]

See also:

`DataFrame.where()`

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.

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.

Examples

```
>>> s = pd.Series(range(5))
>>> s.where(s > 0)
0    NaN
1    1.0
2    2.0
3    3.0
4    4.0
```

```
>>> s.mask(s > 0)
0    0.0
1    NaN
2    NaN
3    NaN
4    NaN
```

```
>>> s.where(s > 1, 10)
0    10.0
1    10.0
2    2.0
3    3.0
4    4.0
```

```
>>> df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])
>>> 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
```

max (*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

This method returns the maximum of the values in the object. If you want the *index* of the maximum, use `idxmax`. This is the equivalent of the `numpy.ndarray` method `argmax`.

Parameters

axis [{items (0), major_axis (1), minor_axis (2)}]

skipna [boolean, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

Returns

max [DataFrame or Panel (if level specified)]

mean(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)
Return the mean of the values for the requested axis

Parameters

axis [{items (0), major_axis (1), minor_axis (2)}]

skipna [boolean, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

Returns

mean [DataFrame or Panel (if level specified)]

median(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)
Return the median of the values for the requested axis

Parameters

axis [{items (0), major_axis (1), minor_axis (2)}]

skipna [boolean, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

Returns

median [DataFrame or Panel (if level specified)]

min(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

This method returns the minimum of the values in the object. If you want the *index* of the minimum, use `idxmin`. This is the equivalent of the `numpy.ndarray` method `argmin`.

Parameters

axis [{items (0), major_axis (1), minor_axis (2)}]

skipna [boolean, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

Returns

min [DataFrame or Panel (if level specified)]

minor_axis

minor_xs(*key*)
Return slice of panel along minor axis

Parameters

key [object] Minor axis label

Returns

y [DataFrame] index -> major axis, columns -> items

Notes

minor_xs is only for getting, not setting values.

MultiIndex Slicers is a generic way to get/set values on any level or levels and is a superset of minor_xs functionality, see MultiIndex Slicers

mod(*other*, *axis=0*)

Modulo of series and other, element-wise (binary operator *mod*). Equivalent to `panel % other`.

Parameters

other [DataFrame or Panel]

axis [{items, major_axis, minor_axis}] Axis to broadcast over

Returns

Panel

See also:

`Panel.rmod`

mul(*other*, *axis=0*)

Multiplication of series and other, element-wise (binary operator *mul*). Equivalent to `panel * other`.

Parameters

other [DataFrame or Panel]

axis [{items, major_axis, minor_axis}] Axis to broadcast over

Returns

Panel

See also:

`Panel.rmul`

multiply(*other*, *axis=0*)

Multiplication of series and other, element-wise (binary operator *mul*). Equivalent to `panel * other`.

Parameters

other [DataFrame or Panel]

axis [{items, major_axis, minor_axis}] Axis to broadcast over

Returns

Panel

See also:

`Panel.rmul`

ndim

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

Return 1 if Series. Otherwise return 2 if DataFrame.

See also:

`ndarray.ndim`

Examples

```
>>> s = pd.Series({'a': 1, 'b': 2, 'c': 3})
>>> s.ndim
1
```

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df.ndim
2
```

ne(*other*, *axis=None*)

Wrapper for comparison method ne

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

NDFrame Mask of bool values for each element in NDFrame that indicates whether an element is not an NA value.

See also:

NDFrame.notnull alias of notna

NDFrame.isna boolean inverse of notna

NDFrame.dropna omit axes labels with missing values

notna top-level notna

Examples

Show which entries in a DataFrame are not NA.

```
>>> df = pd.DataFrame({'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
```

(continues on next page)

(continued from previous page)

```
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
```

notnull()

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

NDFrame Mask of bool values for each element in NDFrame that indicates whether an element is not an NA value.

See also:

NDFrame.notnull alias of notna

NDFrame.isna boolean inverse of notna

NDFrame.dropna omit axes labels with missing values

notna top-level notna

Examples

Show which entries in a DataFrame are not NA.

```
>>> df = pd.DataFrame({'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']})
```

(continues on next page)

(continued from previous page)

```
>>> 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
```

pct_change(*periods=1*, *fill_method='pad'*, *limit=None*, *freq=None*, ***kwargs*)

Percentage change between the current and a prior element.

Computes the percentage change from the immediately previous row by default. This is useful in comparing the percentage of change in a time series of elements.

Parameters

periods [int, default 1] Periods to shift for forming percent change.

fill_method [str, default 'pad'] How to handle NAs before computing percent changes.

limit [int, default None] The number of consecutive NAs to fill before stopping.

freq [DateOffset, timedelta, or offset alias string, 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

chg [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
2     NaN
3    85.0
dtype: float64
```

```
>>> s.pct_change(fill_method='ffill')
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')
      2016      2015      2014
GOOG   NaN -0.151997 -0.086016
APPL   NaN  0.337604  0.012002
```

pipe (*func*, **args*, ***kwargs*)

Apply func(self, *args, **kwargs)

Parameters

func [function] function to apply to the NDFrame. `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 NDFrame.

args [iterable, optional] positional arguments passed into `func`.

kwargs [mapping, optional] a dictionary of keyword arguments passed into `func`.

Returns

object [the return type of `func`.]

See also:

`pandas.DataFrame.apply`, `pandas.DataFrame.applymap`, `pandas.Series.map`

Notes

Use `.pipe` when chaining together functions that expect Series, DataFrames or GroupBy objects. Instead of writing

```
>>> f(g(h(df), arg1=a), arg2=b, arg3=c)
```

You can write

```
>>> (df.pipe(h)
...    .pipe(g, arg1=a)
```

(continues on next page)

(continued from previous page)

```
...     .pipe(f, arg2=b, arg3=c)
... )
```

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 `f` takes its data as `arg2`:

```
>>> (df.pipe(h)
...     .pipe(g, arg1=a)
...     .pipe((f, 'arg2'), arg1=a, arg3=c)
...  )
```

pop(*item*)

Return item and drop from frame. Raise KeyError if not found.

Parameters

item [str] Column label to be popped

Returns

popped [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
```

pow(*other*, *axis=0*)

Exponential power of series and other, element-wise (binary operator *pow*). Equivalent to `panel ** other`.

Parameters

other [DataFrame or Panel]

axis [{items, major_axis, minor_axis}] Axis to broadcast over

Returns

Panel

See also:

`Panel.rpow`

prod(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, *min_count=0*, ***kwargs*)
Return the product of the values for the requested axis

Parameters

axis [{items (0), major_axis (1), minor_axis (2)}]

skipna [boolean, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. 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.

New in version 0.22.0: Added with the default being 0. This means the sum of an all-NA or empty Series is 0, and the product of an all-NA or empty Series is 1.

Returns

prod [DataFrame or Panel (if level specified)]

Examples

By default, the product of an empty or all-NA Series is `1`

```
>>> pd.Series([]).prod()
1.0
```

This can be controlled with the `min_count` parameter

```
>>> pd.Series([]).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
```

product(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, *min_count=0*, ***kwargs*)
Return the product of the values for the requested axis

Parameters

axis [{items (0), major_axis (1), minor_axis (2)}]

skipna [boolean, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. 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.

New in version 0.22.0: Added with the default being 0. This means the sum of an all-NA or empty Series is 0, and the product of an all-NA or empty Series is 1.

Returns

prod [DataFrame or Panel (if level specified)]

Examples

By default, the product of an empty or all-NA Series is `1`

```
>>> pd.Series([]).prod()
1.0
```

This can be controlled with the `min_count` parameter

```
>>> pd.Series([]).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
```

radd(*other*, *axis=0*)

Addition of series and other, element-wise (binary operator *radd*). Equivalent to `other + panel`.

Parameters

other [DataFrame or Panel]

axis [{items, major_axis, minor_axis}] Axis to broadcast over

Returns

Panel

See also:

`Panel.add`

rank(*axis=0*, *method='average'*, *numeric_only=None*, *na_option='keep'*, *ascending=True*, *pct=False*)

Compute numerical data ranks (1 through n) along axis. 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

method [{'average', 'min', 'max', 'first', 'dense'}]

- average: average rank of group
- min: lowest rank in group
- max: highest rank in group
- first: ranks assigned in order they appear in the array
- dense: like 'min', but rank always increases by 1 between groups

numeric_only [boolean, default None] Include only float, int, boolean data. Valid only for DataFrame or Panel objects

na_option [{'keep', 'top', 'bottom'}]

- keep: leave NA values where they are
- top: smallest rank if ascending
- bottom: smallest rank if descending

ascending [boolean, default True] False for ranks by high (1) to low (N)

pct [boolean, default False] Computes percentage rank of data

Returns

ranks [same type as caller]

rdiv(*other*, *axis=0*)

Floating division of series and other, element-wise (binary operator *rtruediv*). Equivalent to `other / panel`.

Parameters

other [DataFrame or Panel]

axis [{items, major_axis, minor_axis}] Axis to broadcast over

Returns

Panel

See also:

`Panel.truediv`

reindex(**args*, ***kwargs*)

Conform Panel to new index with optional filling logic, placing 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

items, major_axis, minor_axis [array-like, optional (should be specified using keywords)] New labels / index to conform to. Preferably an Index object to avoid duplicating data

method [{None, 'backfill'/'bfill', 'pad'/'ffill', 'nearest'}, optional] method to use for filling holes in reindexed DataFrame. Please note: this is only applicable to DataFrames/Series with a monotonically increasing/decreasing index.

- 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 [boolean, 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.

New in version 0.21.0: (list-like tolerance)

Returns

reindexed [Panel]

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
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
```

(continues on next page)

(continued from previous page)

```
            prices
2010-01-01     100
2010-01-02     101
2010-01-03     NaN
2010-01-04     100
2010-01-05      89
2010-01-06      88
```

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
2010-01-02     101
2010-01-03     NaN
2010-01-04     100
2010-01-05      89
2010-01-06      88
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 backpropagate 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
2009-12-30     100
2009-12-31     100
2010-01-01     100
2010-01-02     101
2010-01-03     NaN
2010-01-04     100
2010-01-05      89
2010-01-06      88
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.

reindex_axis (*labels*, *axis=0*, *method=None*, *level=None*, *copy=True*, *limit=None*, *fill_value=nan*)
Conform input object to new index with optional filling logic, placing 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] New labels / index to conform to. Preferably an Index object to avoid

duplicating data

axis [{0, 1, 2, 'items', 'major_axis', 'minor_axis'}]

method [{None, 'backfill'/'bfill', 'pad'/'ffill', 'nearest'}, optional] Method to use for filling holes in reindexed DataFrame:

- 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 [boolean, 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

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.

New in version 0.21.0: (list-like tolerance)

Returns

reindexed [Panel]

See also:

reindex, *reindex_like*

Examples

```
>>> df.reindex_axis(['A', 'B', 'C'], axis=1)
```

reindex_like(*other*, *method=None*, *copy=True*, *limit=None*, *tolerance=None*)

Return an object with matching indices to myself.

Parameters

other [Object]

method [string or None]

copy [boolean, default True]

limit [int, default None] Maximum number of consecutive labels to fill for inexact matches.

tolerance [optional] Maximum distance between labels of the other object and this object for inexact matches. Can be list-like.

New in version 0.21.0: (list-like tolerance)

Returns

reindexed [same as input]

Notes

Like calling s.reindex(index=other.index, columns=other.columns, method=...)

rename (*items=None*, *major_axis=None*, *minor_axis=None*, ***kwargs*)

Alter axes input function or functions. 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. Alternatively, change `Series.name` with a scalar value (Series only).

Parameters

items, major_axis, minor_axis [scalar, list-like, dict-like or function, optional] Scalar or list-like will alter the `Series.name` attribute, and raise on DataFrame or Panel. dict-like or functions are transformations to apply to that axis' values

copy [boolean, default True] Also copy underlying data

inplace [boolean, default False] Whether to return a new Panel. 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.

Returns

renamed [Panel (new object)]

See also:

`pandas.NDFrame.rename_axis`

Examples

```
>>> s = pd.Series([1, 2, 3])
>>> s
0    1
1    2
2    3
dtype: int64
>>> s.rename("my_name") # scalar, changes Series.name
0    1
1    2
2    3
Name: my_name, dtype: int64
>>> s.rename(lambda x: x ** 2)  # function, changes labels
0    1
1    2
4    3
dtype: int64
>>> s.rename({1: 3, 2: 5})  # mapping, changes labels
0    1
3    2
5    3
dtype: int64
```

Since `DataFrame` doesn't have a `.name` attribute, only mapping-type arguments are allowed.

```
>>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
>>> df.rename(2)
Traceback (most recent call last):
...
TypeError: 'int' object is not callable
```

`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.

```
>>> df.rename(index=str, columns={"A": "a", "B": "c"})
   a  c
0  1  4
1  2  5
2  3  6
```

```
>>> df.rename(index=str, columns={"A": "a", "C": "c"})
   a  B
0  1  4
1  2  5
2  3  6
```

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
```

See the user guide for more.

rename_axis(*mapper*, *axis=0*, *copy=True*, *inplace=False*)

Alter the name of the index or columns.

Parameters

mapper [scalar, list-like, optional] Value to set as the axis name attribute.

axis [{0 or 'index', 1 or 'columns'}, default 0] The index or the name of the axis.

copy [boolean, default True] Also copy underlying data.

inplace [boolean, default False] Modifies the object directly, instead of creating a new Series or DataFrame.

Returns

renamed [Series, DataFrame, or None] The same type as the caller or None if *inplace* is True.

See also:

pandas.Series.rename Alter Series index labels or name

pandas.DataFrame.rename Alter DataFrame index labels or name

pandas.Index.rename Set new names on index

Notes

Prior to version 0.21.0, `rename_axis` could also be used to change the axis *labels* by passing a mapping or scalar. This behavior is deprecated and will be removed in a future version. Use `rename` instead.

Examples

Series

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

DataFrame

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

```
>>> df.rename_axis("bar", axis="columns")
bar  A  B
0    1  4
1    2  5
2    3  6
```

replace(*to_replace=None*, *value=None*, *inplace=False*, *limit=None*, *regex=False*, *method='pad'*)
Replace values given in *to_replace* with *value*.

Values of the NDFrame 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 *value* parameter should be *None*.
  - 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 *value* parameter should be `None` 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 [boolean, default False] If True, in place. Note: this will modify any other views on this object (e.g. a column from a DataFrame). Returns the caller if this is True.

limit [int, default None] Maximum size gap to forward or backward fill.

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', *None*}] The method to use when for replacement, when *to_replace* is a scalar, list or tuple and *value* is `None`.

Changed in version 0.23.0: Added to DataFrame.

Returns

NDFrame Object after replacement.

Raises

AssertionError

- If *regex* is not a `bool` and *to_replace* is not `None`.

TypeError

- 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:

`NDFrame.fillna` Fill NA values

`NDFrame.where` Replace values based on boolean condition

`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([0, 1, 2, 3, 4])
>>> s.replace(0, 5)
0    5
1    1
2    2
3    3
4    4
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    0
1    3
2    3
3    3
4    4
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
```

Note that when replacing multiple `bool` or `datetime64` objects, the data types in the *to_replace* parameter must match the data type of the value being replaced:

```
>>> df = pd.DataFrame({'A': [True, False, True],
...                    'B': [False, True, False]})
>>> df.replace({'a string': 'new value', True: False})  # raises
Traceback (most recent call last):
    ...
TypeError: Cannot compare types 'ndarray(dtype=bool)' and 'str'
```

This raises a `TypeError` because one of the `dict` keys is not of the correct type for replacement.

Compare the behavior of `s.replace({'a': None})` and `s.replace('a', None)` to understand the pecularities 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=None` 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. The command `s.replace('a', None)` is actually equivalent to `s.replace(to_replace='a', value=None, method='pad')`:

```
>>> s.replace('a', None)
0    10
1    10
2    10
3     b
4     b
dtype: object
```

resample(*rule*, *how=None*, *axis=0*, *fill_method=None*, *closed=None*, *label=None*, *convention='start'*, *kind=None*, *loffset=None*, *limit=None*, *base=0*, *on=None*, *level=None*)

Convenience method for frequency conversion and resampling of time series. Object must have a datetime-like index (DatetimeIndex, PeriodIndex, or TimedeltaIndex), or pass datetime-like values to the on or level keyword.

Parameters

rule [string] the offset string or object representing target conversion

axis [int, optional, default 0]

closed [{'right', 'left'}] 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'}] 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'}] For PeriodIndex only, controls whether to use the start or end of *rule*

kind: {'timestamp', 'period'}, optional 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.

loffset [timedelta] Adjust the resampled time labels

base [int, default 0] For frequencies that evenly subdivide 1 day, the "origin" of the aggregated intervals. For example, for '5min' frequency, base could range from 0 through 4. Defaults to 0

on [string, optional] For a DataFrame, column to use instead of index for resampling. Column must be datetime-like.

New in version 0.19.0.

level [string or int, optional] For a MultiIndex, level (name or number) to use for resampling. Level must be datetime-like.

New in version 0.19.0.

Returns

Resampler object

See also:

groupby Group by mapping, function, label, or list of labels.

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
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 `pad` method.

```
>>> series.resample('30S').pad()[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
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(array_like):
...     return np.sum(array_like)+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*.

```
>>> 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
```

Resample by month using 'start' *convention*. Values are assigned to the first month of the period.

```
>>> s.resample('M', convention='start').asfreq().head()
2012-01    1.0
2012-02    NaN
2012-03    NaN
2012-04    NaN
2012-05    NaN
Freq: M, dtype: float64
```

Resample by month using 'end' *convention*. Values are assigned to the last month of the period.

```
>>> s.resample('M', convention='end').asfreq()
2012-12    1.0
2013-01    NaN
2013-02    NaN
2013-03    NaN
2013-04    NaN
2013-05    NaN
2013-06    NaN
2013-07    NaN
2013-08    NaN
2013-09    NaN
2013-10    NaN
2013-11    NaN
2013-12    2.0
Freq: M, dtype: float64
```

For DataFrame objects, the keyword `on` can be used to specify the column instead of the index for resampling.

```
>>> df = pd.DataFrame(data=9*[range(4)], columns=['a', 'b', 'c', 'd'])
>>> df['time'] = pd.date_range('1/1/2000', periods=9, freq='T')
>>> df.resample('3T', on='time').sum()
                     a  b  c  d
time
2000-01-01 00:00:00  0  3  6  9
2000-01-01 00:03:00  0  3  6  9
2000-01-01 00:06:00  0  3  6  9
```

For a DataFrame with MultiIndex, the keyword `level` can be used to specify on level the resampling needs to take place.

```
>>> time = pd.date_range('1/1/2000', periods=5, freq='T')
>>> df2 = pd.DataFrame(data=10*[range(4)],
                       columns=['a', 'b', 'c', 'd'],
                       index=pd.MultiIndex.from_product([time, [1, 2]])
                       )
>>> df2.resample('3T', level=0).sum()
```

(continues on next page)

(continued from previous page)

```
                     a  b   c   d
2000-01-01 00:00:00  0  6  12  18
2000-01-01 00:03:00  0  4   8  12
```

rfloordiv(*other*, *axis=0*)

Integer division of series and other, element-wise (binary operator *rfloordiv*). Equivalent to `other // panel`.

Parameters

other [DataFrame or Panel]

axis [{items, major_axis, minor_axis}] Axis to broadcast over

Returns

Panel

See also:

`Panel.floordiv`

rmod(*other*, *axis=0*)

Modulo of series and other, element-wise (binary operator *rmod*). Equivalent to `other % panel`.

Parameters

other [DataFrame or Panel]

axis [{items, major_axis, minor_axis}] Axis to broadcast over

Returns

Panel

See also:

`Panel.mod`

rmul(*other*, *axis=0*)

Multiplication of series and other, element-wise (binary operator *rmul*). Equivalent to `other * panel`.

Parameters

other [DataFrame or Panel]

axis [{items, major_axis, minor_axis}] Axis to broadcast over

Returns

Panel

See also:

`Panel.mul`

round(*decimals=0*, **args*, ***kwargs*)

Round each value in Panel to a specified number of decimal places.

New in version 0.18.0.

Parameters

decimals [int] Number of decimal places to round to (default: 0). If decimals is negative, it specifies the number of positions to the left of the decimal point.

Returns

Panel object

See also:

`numpy.around`

rpow(*other*, *axis=0*)
Exponential power of series and other, element-wise (binary operator *rpow*). Equivalent to `other ** panel`.

Parameters

other [DataFrame or Panel]

axis [{items, major_axis, minor_axis}] Axis to broadcast over

Returns

Panel

See also:

`Panel.pow`

rsub(*other*, *axis=0*)
Subtraction of series and other, element-wise (binary operator *rsub*). Equivalent to `other - panel`.

Parameters

other [DataFrame or Panel]

axis [{items, major_axis, minor_axis}] Axis to broadcast over

Returns

Panel

See also:

`Panel.sub`

rtruediv(*other*, *axis=0*)
Floating division of series and other, element-wise (binary operator *rtruediv*). Equivalent to `other / panel`.

Parameters

other [DataFrame or Panel]

axis [{items, major_axis, minor_axis}] Axis to broadcast over

Returns

Panel

See also:

`Panel.truediv`

sample(*n=None*, *frac=None*, *replace=False*, *weights=None*, *random_state=None*, *axis=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 [boolean, optional] Sample with or without replacement. Default = False.

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. inf and -inf values not allowed.

random_state [int or numpy.random.RandomState, optional] Seed for the random number generator (if int), or numpy RandomState object.

axis [int or string, optional] Axis to sample. Accepts axis number or name. Default is stat axis for given data type (0 for Series and DataFrames, 1 for Panels).

Returns

A new object of same type as caller.

Examples

Generate an example `Series` and `DataFrame`:

```
>>> s = pd.Series(np.random.randn(50))
>>> s.head()
0   -0.038497
1    1.820773
2   -0.972766
3   -1.598270
4   -1.095526
dtype: float64
>>> df = pd.DataFrame(np.random.randn(50, 4), columns=list('ABCD'))
>>> df.head()
          A         B         C         D
0  0.016443 -2.318952 -0.566372 -1.028078
1 -1.051921  0.438836  0.658280 -0.175797
2 -1.243569 -0.364626 -0.215065  0.057736
3  1.768216  0.404512 -0.385604 -1.457834
4  1.072446 -1.137172  0.314194 -0.046661
```

Next extract a random sample from both of these objects...

3 random elements from the `Series`:

```
>>> s.sample(n=3)
27   -0.994689
55   -1.049016
67   -0.224565
dtype: float64
```

And a random 10% of the `DataFrame` with replacement:

```
>>> df.sample(frac=0.1, replace=True)
          A         B         C         D
```

(continues on next page)

(continued from previous page)

```
35  1.981780  0.142106  1.817165 -0.290805
49 -1.336199 -0.448634 -0.789640  0.217116
40  0.823173 -0.078816  1.009536  1.015108
15  1.421154 -0.055301 -1.922594 -0.019696
6  -0.148339  0.832938  1.787600 -1.383767
```

You can use *random state* for reproducibility:

```
>>> df.sample(random_state=1)
A         B         C         D
37 -2.027662  0.103611  0.237496 -0.165867
43 -0.259323 -0.583426  1.516140 -0.479118
12 -1.686325 -0.579510  0.985195 -0.460286
8   1.167946  0.429082  1.215742 -1.636041
9   1.197475 -0.864188  1.554031 -1.505264
```

select(*crit*, *axis=0*)

Return data corresponding to axis labels matching criteria

Deprecated since version 0.21.0: Use df.loc[df.index.map(crit)] to select via labels

Parameters

crit [function] To be called on each index (label). Should return True or False

axis [int]

Returns

selection [type of caller]

sem(*axis=None*, *skipna=None*, *level=None*, *ddof=1*, *numeric_only=None*, ***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 [{items (0), major_axis (1), minor_axis (2)}]

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame

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 [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

Returns

sem [DataFrame or Panel (if level specified)]

set_axis(*labels*, *axis=0*, *inplace=None*)

Assign desired index to given axis.

Indexes for column or row labels can be changed by assigning a list-like or Index.

Changed in version 0.21.0: The signature is now *labels* and *axis*, consistent with the rest of pandas API. Previously, the *axis* and *labels* arguments were respectively the first and second positional arguments.

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, and 1 identifies the columns.

inplace [boolean, default None] Whether to return a new %(klass)s instance.

> **Warning:** `inplace=None` currently falls back to to True, but in a future version, will default to False. Use inplace=True explicitly rather than relying on the default.

Returns

renamed [%(klass)s or None] An object of same type as caller if inplace=False, None otherwise.

See also:

`pandas.DataFrame.rename_axis` Alter the name of the index or columns.

Examples

Series

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

```
>>> s.set_axis(['a', 'b', 'c'], axis=0, inplace=False)
a    1
b    2
c    3
dtype: int64
```

The original object is not modified.

```
>>> s
0    1
1    2
2    3
dtype: int64
```

DataFrame

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

Change the row labels.

```
>>> df.set_axis(['a', 'b', 'c'], axis='index', inplace=False)
   A  B
a  1  4
b  2  5
c  3  6
```

Change the column labels.

```
>>> df.set_axis(['I', 'II'], axis='columns', inplace=False)
   I  II
0  1   4
1  2   5
2  3   6
```

Now, update the labels inplace.

```
>>> df.set_axis(['i', 'ii'], axis='columns', inplace=True)
>>> df
   i  ii
0  1   4
1  2   5
2  3   6
```

set_value(**args*, ***kwargs*)

Quickly set single value at (item, major, minor) location

Deprecated since version 0.21.0.

Please use .at[] or .iat[] accessors.

Parameters

item [item label (panel item)]

major [major axis label (panel item row)]

minor [minor axis label (panel item column)]

value [scalar]

takeable [interpret the passed labels as indexers, default False]

Returns

panel [Panel] If label combo is contained, will be reference to calling Panel, otherwise a new object

shape

Return a tuple of axis dimensions

shift(*periods=1*, *freq=None*, *axis='major'*)

Shift index by desired number of periods with an optional time freq. The shifted data will not include the dropped periods and the shifted axis will be smaller than the original. This is different from the behavior of DataFrame.shift()

Parameters

periods [int] Number of periods to move, can be positive or negative

freq [DateOffset, timedelta, or time rule string, optional]

axis [{'items', 'major', 'minor'} or {0, 1, 2}]

Returns

shifted [Panel]

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

Return the number of rows if Series. Otherwise return the number of rows times number of columns if DataFrame.

See also:

`ndarray.size`

Examples

```
>>> s = pd.Series({'a': 1, 'b': 2, 'c': 3})
>>> s.size
3
```

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df.size
4
```

skew (*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)
Return unbiased skew over requested axis Normalized by N-1

Parameters

axis [{items (0), major_axis (1), minor_axis (2)}]

skipna [boolean, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

Returns

skew [DataFrame or Panel (if level specified)]

slice_shift (*periods=1*, *axis=0*)
Equivalent to *shift* without copying data. The shifted data will not include the dropped periods and the shifted axis will be smaller than the original.

Parameters

periods [int] Number of periods to move, can be positive or negative

Returns

shifted [same type as caller]

Notes

While the *slice_shift* is faster than *shift*, you may pay for it later during alignment.

sort_index (*axis=0*, *level=None*, *ascending=True*, *inplace=False*, *kind='quicksort'*, *na_position='last'*, *sort_remaining=True*)
Sort object by labels (along an axis)

Parameters

axis [axes to direct sorting]

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 [boolean, default True] Sort ascending vs. descending

inplace [bool, default False] if True, perform operation in-place

kind [{'quicksort', 'mergesort', 'heapsort'}, default 'quicksort'] Choice of sorting algorithm. See also ndarray.np.sort for more information. *mergesort* is the only stable algorithm. For DataFrames, this option is only applied when sorting on a single column or label.

na_position [{'first', 'last'}, default 'last'] *first* puts NaNs at the beginning, *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

Returns

sorted_obj [NDFrame]

sort_values(*by=None*, *axis=0*, *ascending=True*, *inplace=False*, *kind='quicksort'*, *na_position='last'*)

NOT IMPLEMENTED: do not call this method, as sorting values is not supported for Panel objects and will raise an error.

squeeze(*axis=None*)

Squeeze length 1 dimensions.

Parameters

axis [None, integer or string axis name, optional] The axis to squeeze if 1-sized.

New in version 0.20.0.

Returns

scalar if 1-sized, else original object

std(*axis=None*, *skipna=None*, *level=None*, *ddof=1*, *numeric_only=None*, ***kwargs*)

Return sample standard deviation over requested axis.

Normalized by N-1 by default. This can be changed using the ddof argument

Parameters

axis [{items (0), major_axis (1), minor_axis (2)}]

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame

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 [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

Returns

std [DataFrame or Panel (if level specified)]

sub(*other*, *axis=0*)

Subtraction of series and other, element-wise (binary operator *sub*). Equivalent to `panel - other`.

Parameters

other [DataFrame or Panel]

axis [{items, major_axis, minor_axis}] Axis to broadcast over

Returns

Panel

See also:

`Panel.rsub`

subtract(*other*, *axis=0*)

Subtraction of series and other, element-wise (binary operator *sub*). Equivalent to `panel - other`.

Parameters

other [DataFrame or Panel]

axis [{items, major_axis, minor_axis}] Axis to broadcast over

Returns

Panel

See also:

`Panel.rsub`

sum(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, *min_count=0*, ***kwargs*)

Return the sum of the values for the requested axis

Parameters

axis [{items (0), major_axis (1), minor_axis (2)}]

skipna [boolean, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. 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.

New in version 0.22.0: Added with the default being 0. This means the sum of an all-NA or empty Series is 0, and the product of an all-NA or empty Series is 1.

Returns

sum [DataFrame or Panel (if level specified)]

Examples

By default, the sum of an empty or all-NA Series is `0`.

```
>>> pd.Series([]).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([]).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
```

swapaxes(*axis1*, *axis2*, *copy=True*)

Interchange axes and swap values axes appropriately

Returns

y [same as input]

swaplevel(*i=-2*, *j=-1*, *axis=0*)

Swap levels i and j in a MultiIndex on a particular axis

Parameters

i, j [int, string (can be mixed)] Level of index to be swapped. Can pass level name as string.

Returns

swapped [type of caller (new object)]

.. versionchanged:: 0.18.1 The indexes `i` and `j` are now optional, and default to the two innermost levels of the index.

tail(*n=5*)

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.

Parameters

n [int, default 5] Number of rows to select.

Returns

type of caller The last *n* rows of the caller object.

See also:

pandas.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
```

take(*indices*, *axis=0*, *convert=None*, *is_copy=True*, ***kwargs*)

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.

convert [bool, default True] Whether to convert negative indices into positive ones. For example, `-1` would map to the `len(axis) - 1`. The conversions are similar to the behavior of indexing a regular Python list.

Deprecated since version 0.21.0: In the future, negative indices will always be converted.

is_copy [bool, default True] Whether to return a copy of the original object or not.

****kwargs** For compatibility with `numpy.take()`. Has no effect on the output.

Returns

taken [type of 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
```

to_clipboard(*excel=True*, *sep=None*, ***kwargs*)

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]

- True, use the provided separator, writing in a csv format for allowing easy pasting into excel.
- 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_table.

Notes

Requirements for your platform.

- Linux : *xclip*, or *xsel* (with *gtk* or *PyQt4* modules)
- Windows : none
- OS X : none

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 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
```

`to_dense`()
Return dense representation of NDFrame (as opposed to sparse)

`to_excel`(*path*, *na_rep=''*, *engine=None*, ***kwargs*)
Write each DataFrame in Panel to a separate excel sheet

Parameters

path [string or ExcelWriter object] File path or existing ExcelWriter

na_rep [string, default ''] Missing data representation

engine [string, default None] write engine to use - you can also set this via the options `io.excel.xlsx.writer`, `io.excel.xls.writer`, and `io.excel.xlsm.writer`.

Other Parameters

float_format [string, default None] Format string for floating point numbers

cols [sequence, optional] Columns to write

header [boolean or list of string, default True] Write out column names. If a list of string is given it is assumed to be aliases for the column names

index [boolean, default True] Write row names (index)

index_label [string or sequence, 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 DataFrame uses MultiIndex.

startrow [upper left cell row to dump data frame]

startcol [upper left cell column to dump data frame]

Notes

Keyword arguments (and na_rep) are passed to the `to_excel` method for each DataFrame written.

`to_frame` (*filter_observations=True*)

Transform wide format into long (stacked) format as DataFrame whose columns are the Panel's items and whose index is a MultiIndex formed of the Panel's major and minor axes.

Parameters

filter_observations [boolean, default True] Drop (major, minor) pairs without a complete set of observations across all the items

Returns

y [DataFrame]

`to_hdf` (*path_or_buf*, *key*, ***kwargs*)

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.

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.

format [{'fixed', 'table'}, 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.

append [bool, default False] For Table formats, append the input data to the existing.

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 io.hdf5-query-data-columns. Applicable only to format='table'.

complevel [{0-9}, optional] Specifies a compression level for data. A value of 0 disables compression.

complib [{'zlib', 'lzo', 'bzip2', 'blosc'}, default 'zlib'] Specifies the compression library to be used. As of v0.20.2 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.

fletcher32 [bool, default False] If applying compression use the fletcher32 checksum.

dropna [bool, default False] If true, ALL nan rows will not be written to store.

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.

See also:

`DataFrame.read_hdf` Read from HDF file.

`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')
```

(continues on next page)

(continued from previous page)

```
0    1
1    2
2    3
3    4
dtype: int64
```

Deleting file with data:

```
>>> import os
>>> os.remove('data.h5')
```

to_json(*path_or_buf=None*, *orient=None*, *date_format=None*, *double_precision=10*, *force_ascii=True*, *date_unit='ms'*, *default_handler=None*, *lines=False*, *compression=None*, *index=True*)

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 [string or file handle, optional] File path or object. If not specified, the result is returned as a string.

orient [string] Indication of expected JSON string format.

- Series
  - default is 'index'
  - allowed values are: {'split','records','index'}
- DataFrame
  - default is 'columns'
  - allowed values are: {'split','records','index','columns','values'}
- 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, and the data component is like `orient='records'`.

    Changed in version 0.20.0.

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.

force_ascii [boolean, default True] Force encoded string to be ASCII.

date_unit [string, 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 [boolean, default False] If 'orient' is 'records' write out line delimited json format. Will throw ValueError if incorrect 'orient' since others are not list like.

New in version 0.19.0.

compression [{None, 'gzip', 'bz2', 'zip', 'xz'}] A string representing the compression to use in the output file, only used when the first argument is a filename.

New in version 0.21.0.

index [boolean, default True] Whether to include the index values in the JSON string. Not including the index (`index=False`) is only supported when orient is 'split' or 'table'.

New in version 0.23.0.

See also:

`pandas.read_json`

Examples

```
>>> df = pd.DataFrame([['a', 'b'], ['c', 'd']],
...                   index=['row 1', 'row 2'],
...                   columns=['col 1', 'col 2'])
>>> df.to_json(orient='split')
'{"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.

```
>>> df.to_json(orient='records')
'[{"col 1":"a","col 2":"b"},{"col 1":"c","col 2":"d"}]'
```

Encoding/decoding a Dataframe using `'index'` formatted JSON:

```
>>> df.to_json(orient='index')
'{"row 1":{"col 1":"a","col 2":"b"},"row 2":{"col 1":"c","col 2":"d"}}'
```

Encoding/decoding a Dataframe using `'columns'` formatted JSON:

```
>>> df.to_json(orient='columns')
'{"col 1":{"row 1":"a","row 2":"c"},"col 2":{"row 1":"b","row 2":"d"}}'
```

Encoding/decoding a Dataframe using `'values'` formatted JSON:

```
>>> df.to_json(orient='values')
'[["a","b"],["c","d"]]'
```

Encoding with Table Schema

```
>>> df.to_json(orient='table')
'{"schema": {"fields": [{"name": "index", "type": "string"},
                        {"name": "col 1", "type": "string"},
                        {"name": "col 2", "type": "string"}],
             "primaryKey": "index",
             "pandas_version": "0.20.0"},
  "data": [{"index": "row 1", "col 1": "a", "col 2": "b"},
           {"index": "row 2", "col 1": "c", "col 2": "d"}]}'
```

to_latex(*buf=None*, *columns=None*, *col_space=None*, *header=True*, *index=True*, *na_rep='NaN'*, *formatters=None*, *float_format=None*, *sparsify=None*, *index_names=True*, *bold_rows=False*, *column_format=None*, *longtable=None*, *escape=None*, *encoding=None*, *decimal='.'*, *multicolumn=None*, *multicolumn_format=None*, *multirow=None*)

Render an object to a tabular environment table. You can splice this into a LaTeX document. Requires \usepackage{booktabs}.

Changed in version 0.20.2: Added to Series

to_latex-specific options:

bold_rows [boolean, default False] Make the row labels bold in the output

column_format [str, default None] The columns format as specified in LaTeX table format e.g 'rcl' for 3 columns

longtable [boolean, default will be read from the pandas config module] Default: False. Use a longtable environment instead of tabular. Requires adding a \usepackage{longtable} to your LaTeX preamble.

escape [boolean, default will be read from the pandas config module] Default: True. When set to False prevents from escaping latex special characters in column names.

encoding [str, default None] A string representing the encoding to use in the output file, defaults to 'ascii' on Python 2 and 'utf-8' on Python 3.

decimal [string, default '.'] Character recognized as decimal separator, e.g. ',' in Europe.

New in version 0.18.0.

multicolumn [boolean, default True] Use multicolumn to enhance MultiIndex columns. The default will be read from the config module.

New in version 0.20.0.

multicolumn_format [str, default 'l'] The alignment for multicolumns, similar to *column_format* The default will be read from the config module.

New in version 0.20.0.

multirow [boolean, default False] 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.

New in version 0.20.0.

to_msgpack(*path_or_buf=None*, *encoding='utf-8'*, ***kwargs*)

msgpack (serialize) object to input file path

THIS IS AN EXPERIMENTAL LIBRARY and the storage format may not be stable until a future release.

Parameters

path [string File path, buffer-like, or None] if None, return generated string

append [boolean whether to append to an existing msgpack] (default is False)

compress [type of compressor (zlib or blosc), default to None (no] compression)

to_pickle(*path*, *compression='infer'*, *protocol=2*)
Pickle (serialize) object to file.

Parameters

path [str] File path where the pickled object will be stored.

compression [{'infer', 'gzip', 'bz2', 'zip', 'xz', None}, default 'infer'] A string representing the compression to use in the output file. By default, infers from the file extension in specified path.

New in version 0.20.0.

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 for this parameter depend on the version of Python. For Python 2.x, possible values are 0, 1, 2. For Python>=3.0, 3 is a valid value. For Python >= 3.4, 4 is a valid value. A negative value for the protocol parameter is equivalent to setting its value to HIGHEST_PROTOCOL.

New in version 0.21.0.

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
```

```
>>> import os
>>> os.remove("./dummy.pkl")
```

to_sparse(**args*, ***kwargs*)

NOT IMPLEMENTED: do not call this method, as sparsifying is not supported for Panel objects and will raise an error.

Convert to SparsePanel

to_sql(*name*, *con*, *schema=None*, *if_exists='fail'*, *index=True*, *index_label=None*, *chunksize=None*, *dtype=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 [string] Name of SQL table.

con [sqlalchemy.engine.Engine or sqlite3.Connection] Using SQLAlchemy makes it possible to use any DB supported by that library. Legacy support is provided for sqlite3.Connection objects.

schema [string, 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 [boolean, default True] Write DataFrame index as a column. Uses *index_label* as the column name in the table.

index_label [string 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] Rows will be written in batches of this size at a time. By default, all rows will be written at once.

dtype [dict, optional] Specifying the datatype for columns. The keys should be the column names and the values should be the SQLAlchemy types or strings for the sqlite3 legacy mode.

Raises

ValueError When the table already exists and *if_exists* is 'fail' (the default).

See also:

pandas.read_sql read a DataFrame from a table

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('users', con=engine)
>>> engine.execute("SELECT * FROM users").fetchall()
[(0, 'User 1'), (1, 'User 2'), (2, 'User 3')]
```

```
>>> df1 = pd.DataFrame({'name' : ['User 4', 'User 5']})
>>> df1.to_sql('users', con=engine, if_exists='append')
>>> engine.execute("SELECT * FROM users").fetchall()
[(0, 'User 1'), (1, 'User 2'), (2, 'User 3'),
 (0, 'User 4'), (1, 'User 5')]
```

Overwrite the table with just `df1`.

```
>>> df1.to_sql('users', con=engine, if_exists='replace',
...            index_label='id')
>>> engine.execute("SELECT * FROM users").fetchall()
[(0, 'User 4'), (1, 'User 5')]
```

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('integers', con=engine, index=False,
...           dtype={"A": Integer()})
```

```
>>> engine.execute("SELECT * FROM integers").fetchall()
[(1,), (None,), (2,)]
```

to_xarray()

Return an xarray object from the pandas object.

Returns

a DataArray for a Series

a Dataset for a DataFrame

a DataArray for higher dims

Notes

See the xarray docs

Examples

```
>>> df = pd.DataFrame({'A' : [1, 1, 2],
                       'B' : ['foo', 'bar', 'foo'],
                       'C' : np.arange(4.,7)})
>>> df
   A    B    C
0  1  foo  4.0
1  1  bar  5.0
2  2  foo  6.0
```

```
>>> df.to_xarray()
<xarray.Dataset>
Dimensions:  (index: 3)
Coordinates:
  * index    (index) int64 0 1 2
Data variables:
    A        (index) int64 1 1 2
    B        (index) object 'foo' 'bar' 'foo'
    C        (index) float64 4.0 5.0 6.0
```

```
>>> df = pd.DataFrame({'A' : [1, 1, 2],
                       'B' : ['foo', 'bar', 'foo'],
                       'C' : np.arange(4.,7)}
                      ).set_index(['B','A'])
>>> df
         C
B   A
foo 1  4.0
bar 1  5.0
foo 2  6.0
```

```
>>> df.to_xarray()
<xarray.Dataset>
Dimensions:  (A: 2, B: 2)
Coordinates:
  * B        (B) object 'bar' 'foo'
  * A        (A) int64 1 2
Data variables:
    C        (B, A) float64 5.0 nan 4.0 6.0
```

```
>>> p = pd.Panel(np.arange(24).reshape(4,3,2),
                 items=list('ABCD'),
                 major_axis=pd.date_range('20130101', periods=3),
                 minor_axis=['first', 'second'])
```

(continues on next page)

(continued from previous page)

```
>>> p
<class 'pandas.core.panel.Panel'>
Dimensions: 4 (items) x 3 (major_axis) x 2 (minor_axis)
Items axis: A to D
Major_axis axis: 2013-01-01 00:00:00 to 2013-01-03 00:00:00
Minor_axis axis: first to second
```

```
>>> p.to_xarray()
<xarray.DataArray (items: 4, major_axis: 3, minor_axis: 2)>
array([[[ 0,  1],
        [ 2,  3],
        [ 4,  5]],
       [[ 6,  7],
        [ 8,  9],
        [10, 11]],
       [[12, 13],
        [14, 15],
        [16, 17]],
       [[18, 19],
        [20, 21],
        [22, 23]]])
Coordinates:
  * items       (items) object 'A' 'B' 'C' 'D'
  * major_axis  (major_axis) datetime64[ns] 2013-01-01 2013-01-02 2013-01-03␣
↪ # noqa
  * minor_axis  (minor_axis) object 'first' 'second'
```

transpose(**args*, ***kwargs*)

Permute the dimensions of the Panel

Parameters

args [three positional arguments: each one of]

{0, 1, 2, 'items', 'major_axis', 'minor_axis'}

copy [boolean, default False] Make a copy of the underlying data. Mixed-dtype data will always result in a copy

Returns

y [same as input]

Examples

```
>>> p.transpose(2, 0, 1)
>>> p.transpose(2, 0, 1, copy=True)
```

truediv(*other*, *axis=0*)

Floating division of series and other, element-wise (binary operator *truediv*). Equivalent to `panel / other`.

Parameters

other [DataFrame or Panel]

axis [{items, major_axis, minor_axis}] Axis to broadcast over

Returns

Panel

See also:

`Panel.rtruediv`

truncate(*before=None*, *after=None*, *axis=None*, *copy=True*)

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, string, int] Truncate all rows before this index value.

after [date, string, 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.

copy [boolean, 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
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
```

(continues on next page)

(continued from previous page)

```
2016-01-10 23:59:58  1
2016-01-10 23:59:59  1
```

tshift (*periods=1*, *freq=None*, *axis='major'*)
Shift the time index, using the index's frequency if available.

Parameters

periods [int] Number of periods to move, can be positive or negative

freq [DateOffset, timedelta, or time rule string, default None] Increment to use from the tseries module or time rule (e.g. 'EOM')

axis [int or basestring] Corresponds to the axis that contains the Index

Returns

shifted [NDFrame]

Notes

If freq is not specified then tries to use the freq or inferred_freq attributes of the index. If neither of those attributes exist, a ValueError is thrown

tz_convert (*tz*, *axis=0*, *level=None*, *copy=True*)
Convert tz-aware axis to target time zone.

Parameters

tz [string or pytz.timezone object]

axis [the axis to convert]

level [int, str, default None] If axis ia a MultiIndex, convert a specific level. Otherwise must be None

copy [boolean, default True] Also make a copy of the underlying data

Raises

TypeError If the axis is tz-naive.

tz_localize (*tz*, *axis=0*, *level=None*, *copy=True*, *ambiguous='raise'*)
Localize tz-naive TimeSeries to target time zone.

Parameters

tz [string or pytz.timezone object]

axis [the axis to localize]

level [int, str, default None] If axis ia a MultiIndex, localize a specific level. Otherwise must be None

copy [boolean, default True] Also make a copy of the underlying data

ambiguous ['infer', bool-ndarray, 'NaT', default 'raise']

- '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

Raises

TypeError If the TimeSeries is tz-aware and tz is not None.

update (*other*, *join='left'*, *overwrite=True*, *filter_func=None*, *raise_conflict=False*)

Modify Panel in place using non-NA values from passed Panel, or object coercible to Panel. Aligns on items

Parameters

other [Panel, or object coercible to Panel]

join [How to join individual DataFrames] {'left', 'right', 'outer', 'inner'}, default 'left'

overwrite [boolean, default True] If True then overwrite values for common keys in the calling panel

filter_func [callable(1d-array) -> 1d-array<boolean>, default None] Can choose to replace values other than NA. Return True for values that should be updated

raise_conflict [bool] If True, will raise an error if a DataFrame and other both contain data in the same place.

values

Return a Numpy representation of the DataFrame.

Only the values in the DataFrame will be returned, the axes labels will be removed.

Returns

numpy.ndarray The values of the DataFrame.

See also:

pandas.DataFrame.index Retrievie the index labels

pandas.DataFrame.columns Retrieving the column names

Notes

The dtype will be a lower-common-denominator dtype (implicit upcasting); that is to say if the dtypes (even of numeric types) are mixed, the one that accommodates all will be chosen. Use this with care if you are not dealing with the blocks.

e.g. If the dtypes are float16 and float32, dtype will be upcast to float32. If dtypes are int32 and uint8, dtype will be upcast to int32. By `numpy.find_common_type()` convention, mixing int64 and uint64 will result in a float64 dtype.

Examples

A DataFrame where all columns are the same type (e.g., int64) results in an array of the same type.

```
>>> df = pd.DataFrame({'age':    [ 3,  29],
...                    'height': [94, 170],
...                    'weight': [31, 115]})
>>> df
   age  height  weight
0    3      94      31
```

(continues on next page)

(continued from previous page)

```
1   29     170     115
>>> df.dtypes
age       int64
height    int64
weight    int64
dtype: object
>>> df.values
array([[  3,  94,  31],
       [ 29, 170, 115]], dtype=int64)
```

A DataFrame with mixed type columns(e.g., str/object, int64, float32) results in an ndarray of the broadest type that accommodates these mixed types (e.g., object).

```
>>> df2 = pd.DataFrame([('parrot',   24.0, 'second'),
...                     ('lion',     80.5, 1),
...                     ('monkey', np.nan, None)],
...                    columns=('name', 'max_speed', 'rank'))
>>> df2.dtypes
name          object
max_speed    float64
rank          object
dtype: object
>>> df2.values
array([['parrot', 24.0, 'second'],
       ['lion', 80.5, 1],
       ['monkey', nan, None]], dtype=object)
```

var(*axis=None*, *skipna=None*, *level=None*, *ddof=1*, *numeric_only=None*, ***kwargs*)

Return unbiased variance over requested axis.

Normalized by N-1 by default. This can be changed using the ddof argument

Parameters

axis [{items (0), major_axis (1), minor_axis (2)}]

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame

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 [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

Returns

var [DataFrame or Panel (if level specified)]

where(*cond*, *other=nan*, *inplace=False*, *axis=None*, *level=None*, *errors='raise'*, *try_cast=False*, *raise_on_error=None*)

Return an object of same shape as self and whose corresponding entries are from self where *cond* is True and otherwise are from *other*.

Parameters

cond [boolean NDFrame, 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 NDFrame and should return boolean NDFrame or array. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1: A callable can be used as cond.

other [scalar, NDFrame, or callable] Entries where *cond* is False are replaced with corresponding value from *other*. If other is callable, it is computed on the NDFrame and should return scalar or NDFrame. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1: A callable can be used as other.

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

axis [alignment axis if needed, default None]

level [alignment level if needed, default None]

errors [str, {'raise', 'ignore'}, default 'raise']

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

Note that currently this parameter won't affect the results and will always coerce to a suitable dtype.

try_cast [boolean, default False] try to cast the result back to the input type (if possible),

raise_on_error [boolean, default True] Whether to raise on invalid data types (e.g. trying to where on strings)

Deprecated since version 0.21.0.

Returns

wh [same type as caller]

See also:

`DataFrame.mask()`

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.

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.

Examples

```
>>> s = pd.Series(range(5))
>>> s.where(s > 0)
0    NaN
1    1.0
```

(continues on next page)

(continued from previous page)

```
2    2.0
3    3.0
4    4.0
```

```
>>> s.mask(s > 0)
0    0.0
1    NaN
2    NaN
3    NaN
4    NaN
```

```
>>> s.where(s > 1, 10)
0    10.0
1    10.0
2    2.0
3    3.0
4    4.0
```

```
>>> df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])
>>> 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
```

xs (*key*, *axis=1*)

Return slice of panel along selected axis

Parameters

key [object] Label

axis [{'items', 'major', 'minor}, default 1/'major']

Returns

y [ndim(self)-1]

Notes

xs is only for getting, not setting values.

MultiIndex Slicers is a generic way to get/set values on any level or levels and is a superset of xs functionality, see MultiIndex Slicers

class oddt.pandas.**ChemSeries**(*data=None*, *index=None*, *dtype=None*, *name=None*, *copy=False*, *fastpath=False*)

Bases: pandas.core.series.Series

Pandas Series modified to adapt *oddt.toolkit.Molecule* objects and apply molecular methods easily.

New in version 0.3.

Attributes

T return the transpose, which is by definition self

asobject Return object Series which contains boxed values.

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

axes Return a list of the row axis labels

base return the base object if the memory of the underlying data is

blocks Internal property, property synonym for as_blocks()

data return the data pointer of the underlying data

dtype return the dtype object of the underlying data

dtypes return the dtype object of the underlying data

empty

flags return the ndarray.flags for the underlying data

ftype return if the data is sparse|dense

ftypes return if the data is sparse|dense

hasnans return if I have any nans; enables various perf speedups

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

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

imag

index The index (axis labels) of the Series.

is_copy

is_monotonic Return boolean if values in the object are

is_monotonic_decreasing Return boolean if values in the object are

is_monotonic_increasing Return boolean if values in the object are

is_unique Return boolean if values in the object are unique

itemsize return the size of the dtype of the item of the underlying data

ix A primarily label-location based indexer, with integer position fallback.

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

name

nbytes return the number of bytes in the underlying data

ndim return the number of dimensions of the underlying data,

real

shape return a tuple of the shape of the underlying data

size return the number of elements in the underlying data

strides return the strides of the underlying data

values Return Series as ndarray or ndarray-like

Methods

abs()	Return a Series/DataFrame with absolute numeric value of each element.
add(other[, level, fill_value, axis])	Addition of series and other, element-wise (binary operator *add*).
add_prefix(prefix)	Prefix labels with string *prefix*.
add_suffix(suffix)	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 for each axis Index
all([axis, bool_only, skipna, level])	Return whether all elements are True, potentially over an axis.
any([axis, bool_only, skipna, level])	Return whether any element is True over requested axis.
append(to_append[, ignore_index, ...])	Concatenate two or more Series.
apply(func[, convert_dtype, args])	Invoke function on values of Series.
argmax(**kwargs)	
argmin(**kwargs)	
argsort([axis, kind, order])	Overrides ndarray.argsort.
as_blocks([copy])	Convert the frame to a dict of dtype -> Constructor Types that each has a homogeneous dtype.
as_matrix([columns])	Convert the frame to its Numpy-array representation.
asfreq(freq[, method, how, normalize, ...])	Convert TimeSeries to specified frequency.
asof(where[, subset])	The last row without any NaN is taken (or the last row without NaN considering only the subset of columns in the case of a DataFrame)
astype(**kwargs)	Cast a pandas object to a specified dtype `dtype`.
at_time(time[, asof])	Select values at particular time of day (e.g.
autocorr([lag])	Lag-N autocorrelation
between(left, right[, inclusive])	Return boolean Series equivalent to left <= series <= right.
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])	Synonym for `DataFrame.fillna(method='bfill')`

Continued on next page

Table 21 – continued from previous page

bool()	Return the bool of a single element PandasObject.
calcfp(*args, **kwargs)	Helper function to map FP calculation throuugh the series
cat	alias of pandas.core.arrays.categorical.CategoricalAccessor
clip([lower, upper, axis, inplace])	Trim values at input threshold(s).
clip_lower(threshold[, axis, inplace])	Return copy of the input with values below a threshold truncated.
clip_upper(threshold[, axis, inplace])	Return copy of input with values above given value(s) truncated.
combine(other, func[, fill_value])	Perform elementwise binary operation on two Series using given function with optional fill value when an index is missing from one Series or the other
combine_first(other)	Combine Series values, choosing the calling Series's values first.
compound([axis, skipna, level])	Return the compound percentage of the values for the requested axis
compress(condition, *args, **kwargs)	Return selected slices of an array along given axis as a Series
consolidate([inplace])	Compute NDFrame with "consolidated" internals (data of each dtype grouped together in a single ndarray).
convert_objects([convert_dates, ...])	Attempt to infer better dtype for object columns.
copy([deep])	Make a copy of this object's indices and data.
corr(other[, method, min_periods])	Compute correlation with *other* Series, excluding missing values
count([level])	Return number of non-NA/null observations in the Series
cov(other[, min_periods])	Compute covariance with Series, excluding missing 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])	Generates descriptive statistics that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding NaN values.
diff([periods])	First discrete difference of element.
div(other[, level, fill_value, axis])	Floating division of series and other, element-wise (binary operator *truediv*).
divide(other[, level, fill_value, axis])	Floating division of series and other, element-wise (binary operator *truediv*).
divmod(other[, level, fill_value, axis])	Integer division and modulo of series and other, element-wise (binary operator *divmod*).
dot(other)	Matrix multiplication with DataFrame or inner-product with Series objects.
drop([labels, axis, index, columns, level, ...])	Return Series with specified index labels removed.

Continued on next page

Table 21 – continued from previous page

drop_duplicates([keep, inplace])	Return Series with duplicate values removed.
dropna([axis, inplace])	Return a new Series with missing values removed.
duplicated([keep])	Indicate duplicate Series values.
eq(other[, level, fill_value, axis])	Equal to of series and other, element-wise (binary operator *eq*).
equals(other)	Determines if two NDFrame objects contain the same elements.
ewm([com, span, halflife, alpha, ...])	Provides exponential weighted functions
expanding([min_periods, center, axis])	Provides expanding transformations.
factorize([sort, na_sentinel])	Encode the object as an enumerated type or categorical variable.
ffill([axis, inplace, limit, downcast])	Synonym for `DataFrame.fillna(method='ffill')`
fillna([value, method, axis, inplace, ...])	Fill NA/NaN values using the specified method
filter([items, like, regex, axis])	Subset rows or columns of dataframe according to labels in the specified index.
first(offset)	Convenience method for subsetting initial periods of time series data based on a date offset.
first_valid_index()	Return index for first non-NA/null value.
floordiv(other[, level, fill_value, axis])	Integer division of series and other, element-wise (binary operator *floordiv*).
from_array(arr[, index, name, dtype, copy, ...])	Construct Series from array.
from_csv(path[, sep, parse_dates, header, ...])	Read CSV file.
ge(other[, level, fill_value, axis])	Greater than or equal to of series and other, element-wise (binary operator *ge*).
get(key[, default])	Get item from object for given key (DataFrame column, Panel slice, etc.).
get_dtype_counts()	Return counts of unique dtypes in this object.
get_ftype_counts()	Return counts of unique ftypes in this object.
get_value(label[, takeable])	Quickly retrieve single value at passed index label
get_values()	same as values (but handles sparseness conversions); is a view
groupby([by, axis, level, as_index, sort, ...])	Group series using mapper (dict or key function, apply given function to group, return result as series) or by a series of columns.
gt(other[, level, fill_value, axis])	Greater than of series and other, element-wise (binary operator *gt*).
head([n])	Return the first *n* rows.
hist([by, ax, grid, xlabelsize, xrot, ...])	Draw histogram of the input series using matplotlib
idxmax([axis, skipna])	Return the row label of the maximum value.
idxmin([axis, skipna])	Return the row label of the minimum value.
infer_objects()	Attempt to infer better dtypes for object columns.
interpolate([method, axis, limit, inplace, ...])	Interpolate values according to different methods.
isin(values)	Check whether *values* are contained in Series.
isna()	Detect missing values.
isnull()	Detect missing values.
item()	return the first element of the underlying data as a python scalar
items()	Lazily iterate over (index, value) tuples
iteritems()	Lazily iterate over (index, value) tuples
keys()	Alias for index

Continued on next page

Table 21 – continued from previous page

kurt([axis, skipna, level, numeric_only])	Return unbiased kurtosis over requested axis using Fisher's definition of kurtosis (kurtosis of normal == 0.0).
kurtosis([axis, skipna, level, numeric_only])	Return unbiased kurtosis over requested axis using Fisher's definition of kurtosis (kurtosis of normal == 0.0).
last(offset)	Convenience method for subsetting final periods of time series data based on a date offset.
last_valid_index()	Return index for last non-NA/null value.
le(other[, level, fill_value, axis])	Less than or equal to of series and other, element-wise (binary operator *le*).
lt(other[, level, fill_value, axis])	Less than of series and other, element-wise (binary operator *lt*).
mad([axis, skipna, level])	Return the mean absolute deviation of the values for the requested axis
map(arg[, na_action])	Map values of Series using input correspondence (a dict, Series, or function).
mask(cond[, other, inplace, axis, level, ...])	Return an object of same shape as self and whose corresponding entries are from self where *cond* is False and otherwise are from *other*.
max([axis, skipna, level, numeric_only])	This method returns the maximum of the values in the object.
mean([axis, skipna, level, numeric_only])	Return the mean of the values for the requested axis
median([axis, skipna, level, numeric_only])	Return the median of the values for the requested axis
memory_usage([index, deep])	Return the memory usage of the Series.
min([axis, skipna, level, numeric_only])	This method returns the minimum of the values in the object.
mod(other[, level, fill_value, axis])	Modulo of series and other, element-wise (binary operator *mod*).
mode()	Return the mode(s) of the dataset.
mul(other[, level, fill_value, axis])	Multiplication of series and other, element-wise (binary operator *mul*).
multiply(other[, level, fill_value, axis])	Multiplication of series and other, element-wise (binary operator *mul*).
ne(other[, level, fill_value, axis])	Not equal to of series and other, element-wise (binary operator *ne*).
nlargest([n, keep])	Return the largest *n* elements.
nonzero()	Return the *integer* indices of the elements that are non-zero
notna()	Detect existing (non-missing) values.
notnull()	Detect existing (non-missing) values.
nsmallest([n, keep])	Return the smallest *n* elements.
nunique([dropna])	Return number of unique elements in the object.
pct_change([periods, fill_method, limit, freq])	Percentage change between the current and a prior element.
pipe(func, *args, **kwargs)	Apply func(self, *args, **kwargs)
plot	alias of pandas.plotting._core.SeriesPlotMethods
pop(item)	Return item and drop from frame.

Continued on next page

Table 21 – continued from previous page

pow(other[, level, fill_value, axis])	Exponential power of series and other, element-wise (binary operator *pow*).
prod([axis, skipna, level, numeric_only, ...])	Return the product of the values for the requested axis
product([axis, skipna, level, numeric_only, ...])	Return the product of the values for the requested axis
ptp([axis, skipna, level, numeric_only])	Returns the difference between the maximum value and the minimum value in the object.
put(*args, **kwargs)	Applies the *put* method to its *values* attribute if it has one.
quantile([q, interpolation])	Return value at the given quantile, a la numpy.percentile.
radd(other[, level, fill_value, axis])	Addition of series and other, element-wise (binary operator *radd*).
rank([axis, method, numeric_only, ...])	Compute numerical data ranks (1 through n) along axis.
ravel([order])	Return the flattened underlying data as an ndarray
rdiv(other[, level, fill_value, axis])	Floating division of series and other, element-wise (binary operator *rtruediv*).
reindex([index])	Conform Series to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index.
reindex_axis(labels[, axis])	Conform Series to new index with optional filling logic.
reindex_like(other[, method, copy, limit, ...])	Return an object with matching indices to myself.
rename([index])	Alter Series index labels or name
rename_axis(mapper[, axis, copy, inplace])	Alter the name of the index or columns.
reorder_levels(order)	Rearrange index levels using input order.
repeat(**kwargs)	Repeat elements of an Series.
replace([to_replace, value, inplace, limit, ...])	Replace values given in *to_replace* with *value*.
resample(rule[, how, axis, fill_method, ...])	Convenience method for frequency conversion and resampling of time series.
reset_index([level, drop, name, inplace])	Generate a new DataFrame or Series with the index reset.
rfloordiv(other[, level, fill_value, axis])	Integer division of series and other, element-wise (binary operator *rfloordiv*).
rmod(other[, level, fill_value, axis])	Modulo of series and other, element-wise (binary operator *rmod*).
rmul(other[, level, fill_value, axis])	Multiplication of series and other, element-wise (binary operator *rmul*).
rolling(window[, min_periods, center, ...])	Provides rolling window calculations.
round([decimals])	Round each value in a Series to the given number of decimals.
rpow(other[, level, fill_value, axis])	Exponential power of series and other, element-wise (binary operator *rpow*).
rsub(other[, level, fill_value, axis])	Subtraction of series and other, element-wise (binary operator *rsub*).
rtruediv(other[, level, fill_value, axis])	Floating division of series and other, element-wise (binary operator *rtruediv*).
sample([n, frac, replace, weights, ...])	Return a random sample of items from an axis of object.

Continued on next page

Table 21 – continued from previous page

searchsorted(**kwargs)	Find indices where elements should be inserted to maintain order.
select(crit[, axis])	Return data corresponding to axis labels matching criteria
sem([axis, skipna, level, ddof, numeric_only])	Return unbiased standard error of the mean over requested axis.
set_axis(labels[, axis, inplace])	Assign desired index to given axis.
set_value(label, value[, takeable])	Quickly set single value at passed label.
shift([periods, freq, axis])	Shift index by desired number of periods with an optional time freq
skew([axis, skipna, level, numeric_only])	Return unbiased skew over requested axis Normalized by N-1
slice_shift([periods, axis])	Equivalent to *shift* without copying data.
sort_index([axis, level, ascending, ...])	Sort Series by index labels.
sort_values([axis, ascending, inplace, ...])	Sort by the values.
sortlevel([level, ascending, sort_remaining])	Sort Series with MultiIndex by chosen level.
squeeze([axis])	Squeeze length 1 dimensions.
std([axis, skipna, level, ddof, numeric_only])	Return sample standard deviation over requested axis.
str	alias of pandas.core.strings.StringMethods
sub(other[, level, fill_value, axis])	Subtraction of series and other, element-wise (binary operator *sub*).
subtract(other[, level, fill_value, axis])	Subtraction of series and other, element-wise (binary operator *sub*).
sum([axis, skipna, level, numeric_only, ...])	Return the sum of the values for the requested axis
swapaxes(axis1, axis2[, copy])	Interchange axes and swap values axes appropriately
swaplevel([i, j, copy])	Swap levels i and j in a MultiIndex
tail([n])	Return the last *n* rows.
take(indices[, axis, convert, is_copy])	Return the elements in the given *positional* indices along an axis.
to_clipboard([excel, sep])	Copy object to the system clipboard.
to_csv([path, index, sep, na_rep, ...])	Write Series to a comma-separated values (csv) file
to_dense()	Return dense representation of NDFrame (as opposed to sparse)
to_dict([into])	Convert Series to {label -> value} dict or dict-like object.
to_excel(excel_writer[, sheet_name, na_rep, ...])	Write Series to an excel sheet
to_frame([name])	Convert Series to DataFrame
to_hdf(path_or_buf, key, **kwargs)	Write the contained data to an HDF5 file using HDFStore.
to_json([path_or_buf, orient, date_format, ...])	Convert the object to a JSON string.
to_latex([buf, columns, col_space, header, ...])	Render an object to a tabular environment table.
to_msgpack([path_or_buf, encoding])	msgpack (serialize) object to input file path
to_period([freq, copy])	Convert Series from DatetimeIndex to PeriodIndex with desired frequency (inferred from index if not passed)
to_pickle(path[, compression, protocol])	Pickle (serialize) object to file.
to_sparse([kind, fill_value])	Convert Series to SparseSeries

Continued on next page

Table 21 – continued from previous page

to_sql(name, con[, schema, if_exists, ...])	Write records stored in a DataFrame to a SQL database.
to_string([buf, na_rep, float_format, ...])	Render a string representation of the Series
to_timestamp([freq, how, copy])	Cast to datetimeindex of timestamps, at *beginning* of period
to_xarray()	Return an xarray object from the pandas object.
tolist()	Return a list of the values.
transform(func, *args, **kwargs)	Call function producing a like-indexed NDFrame and return a NDFrame with the transformed values
transpose(*args, **kwargs)	return the transpose, which is by definition self
truediv(other[, level, fill_value, axis])	Floating division of series and other, element-wise (binary operator *truediv*).
truncate([before, after, axis, copy])	Truncate a Series or DataFrame before and after some index value.
tshift([periods, freq, axis])	Shift the time index, using the index's frequency if available.
tz_convert(tz[, axis, level, copy])	Convert tz-aware axis to target time zone.
tz_localize(tz[, axis, level, copy, ambiguous])	Localize tz-naive TimeSeries to target time zone.
unique()	Return unique values of Series object.
unstack([level, fill_value])	Unstack, a.k.a.
update(other)	Modify Series in place using non-NA values from passed Series.
valid([inplace])	Return Series without null values.
value_counts([normalize, sort, ascending, ...])	Returns object containing counts of unique values.
var([axis, skipna, level, ddof, numeric_only])	Return unbiased variance over requested axis.
view([dtype])	Create a new view of the Series.
where(cond[, other, inplace, axis, level, ...])	Return an object of same shape as self and whose corresponding entries are from self where *cond* is True and otherwise are from *other*.
xs(key[, axis, level, drop_level])	Returns a cross-section (row(s) or column(s)) from the Series/DataFrame.

dt	
to_mol2	
to_sdf	
to_smiles	

T

return the transpose, which is by definition self

abs()

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
```

add (*other*, *level=None*, *fill_value=None*, *axis=0*)

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

Equivalent to `series + other`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters

other [Series or scalar value]

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

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

Returns

result [Series]

See also:

`Series.radd`

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

add_prefix(*prefix*)

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.

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
```

add_suffix(*suffix*)

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.

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
```

(continues on next page)

(continued from previous page)

```
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
```

agg (*func*, *axis=0*, **args*, ***kwargs*)

Aggregate using one or more operations over the specified axis.

New in version 0.20.0.

Parameters

func [function, string, dictionary, or list of string/functions] Function to use for aggregating the data. If a function, must either work when passed a Series or when passed to Series.apply. For a DataFrame, can pass a dict, if the keys are DataFrame column names.

Accepted combinations are:

- string function name.
- function.
- list of functions.
- dict of column names -> functions (or list of functions).

axis [{0 or 'index'}] Parameter needed for compatibility with DataFrame.

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

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

Returns

aggregated [Series]

See also:

pandas.Series.apply, pandas.Series.transform

Notes

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

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

Examples

```
>>> s = Series(np.random.randn(10))
```

```
>>> s.agg('min')
-1.3018049988556679
```

```
>>> s.agg(['min', 'max'])
min   -1.301805
max    1.127688
dtype: float64
```

aggregate(*func*, *axis=0*, **args*, ***kwargs*)

Aggregate using one or more operations over the specified axis.

New in version 0.20.0.

Parameters

func [function, string, dictionary, or list of string/functions] Function to use for aggregating the data. If a function, must either work when passed a Series or when passed to Series.apply. For a DataFrame, can pass a dict, if the keys are DataFrame column names.

Accepted combinations are:

- string function name.
- function.
- list of functions.
- dict of column names -> functions (or list of functions).

axis [{0 or 'index'}] Parameter needed for compatibility with DataFrame.

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

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

Returns

aggregated [Series]

See also:

pandas.Series.apply, pandas.Series.transform

Notes

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

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

Examples

```
>>> s = Series(np.random.randn(10))
```

```
>>> s.agg('min')
-1.3018049988556679
```

```
>>> s.agg(['min', 'max'])
min   -1.301805
max    1.127688
dtype: float64
```

align(*other*, *join='outer'*, *axis=None*, *level=None*, *copy=True*, *fill_value=None*, *method=None*, *limit=None*, *fill_axis=0*, *broadcast_axis=None*)
Align two objects on their axes with the specified join method for each axis Index

Parameters

other [DataFrame or Series]

join [{'outer', 'inner', 'left', 'right'}, default 'outer']

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 [boolean, 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 [str, default None]

limit [int, default None]

fill_axis [{0 or 'index'}, default 0] Filling axis, method and limit

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

Returns

(left, right) [(Series, type of other)] Aligned objects

all(*axis=0*, *bool_only=None*, *skipna=True*, *level=None*, ***kwargs*)
Return whether all elements are True, potentially over an axis.

Returns True if all elements within a series or along a Dataframe axis are non-zero, not-empty or not-False.

Parameters

axis [{0 or 'index', 1 or 'columns', None}, default 0] Indicate which axis or axes should be reduced.

- 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.

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar.

bool_only [boolean, default None] Include only boolean columns. If None, will attempt to use everything, then use only boolean data. Not implemented for Series.

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

Returns

all [scalar or Series (if level specified)]

See also:

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

`pandas.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
```

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 column-wise values all return True.

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

Specify `axis='columns'` to check if row-wise values 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
```

any(*axis=0*, *bool_only=None*, *skipna=True*, *level=None*, ***kwargs*)

Return whether any element is True over requested axis.

Unlike `DataFrame.all()`, this performs an *or* operation. If any of the values along the specified axis is True, this will return True.

Parameters

axis [{0 or 'index', 1 or 'columns', None}, default 0] Indicate which axis or axes should be reduced.

- 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.

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar.

bool_only [boolean, default None] Include only boolean columns. If None, will attempt to use everything, then use only boolean data. Not implemented for Series.

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

Returns

any [scalar or Series (if level specified)]

See also:

pandas.DataFrame.all Return whether all elements are True.

Examples

Series

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

```
>>> pd.Series([True, False]).any()
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
       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)
```

append(*to_append*, *ignore_index=False*, *verify_integrity=False*)

Concatenate two or more Series.

Parameters

to_append [Series or list/tuple of Series]

ignore_index [boolean, default False] If True, do not use the index labels.

New in version 0.19.0.

verify_integrity [boolean, default False] If True, raise Exception on creating index with duplicates

Returns

appended [Series]

See also:

pandas.concat General function to concatenate DataFrame, Series or Panel objects

Notes

Iteratively appending to a Series can be more computationally intensive than a single concatenate. A better solution is to append values to a list and then concatenate the list with the original Series all at once.

Examples

```
>>> s1 = pd.Series([1, 2, 3])
>>> s2 = pd.Series([4, 5, 6])
>>> s3 = pd.Series([4, 5, 6], index=[3,4,5])
>>> s1.append(s2)
0    1
1    2
2    3
0    4
1    5
2    6
dtype: int64
```

```
>>> s1.append(s3)
0    1
1    2
2    3
3    4
4    5
5    6
dtype: int64
```

With *ignore_index* set to True:

```
>>> s1.append(s2, ignore_index=True)
0    1
1    2
2    3
3    4
4    5
5    6
dtype: int64
```

With *verify_integrity* set to True:

```
>>> s1.append(s2, verify_integrity=True)
Traceback (most recent call last):
...
ValueError: Indexes have overlapping values: [0, 1, 2]
```

apply (*func*, *convert_dtype=True*, *args=()*, ***kwds*)

Invoke function on values of Series. Can be ufunc (a NumPy function that applies to the entire Series) or a Python function that only works on single values

Parameters

func [function]

convert_dtype [boolean, default True] Try to find better dtype for elementwise function results. If False, leave as dtype=object

args [tuple] Positional arguments to pass to function in addition to the value

Additional keyword arguments will be passed as keywords to the function

Returns

y [Series or DataFrame if func returns a Series]

See also:

Series.map For element-wise operations

Series.agg only perform aggregating type operations

Series.transform only perform transformating type operations

Examples

Create a series with typical summer temperatures for each city.

```
>>> import pandas as pd
>>> import numpy as np
>>> series = pd.Series([20, 21, 12], index=['London',
... 'New York','Helsinki'])
>>> series
London      20
New York    21
Helsinki    12
dtype: int64
```

Square the values by defining a function and passing it as an argument to `apply()`.

```
>>> def square(x):
...     return x**2
>>> series.apply(square)
London      400
New York    441
Helsinki    144
dtype: int64
```

Square the values by passing an anonymous function as an argument to `apply()`.

```
>>> series.apply(lambda x: x**2)
London      400
New York    441
Helsinki    144
dtype: int64
```

Define a custom function that needs additional positional arguments and pass these additional arguments using the `args` keyword.

```
>>> def subtract_custom_value(x, custom_value):
...     return x-custom_value
```

```
>>> series.apply(subtract_custom_value, args=(5,))
London      15
New York    16
Helsinki     7
dtype: int64
```

Define a custom function that takes keyword arguments and pass these arguments to `apply`.

```
>>> def add_custom_values(x, **kwargs):
...     for month in kwargs:
...         x+=kwargs[month]
...     return x
```

```
>>> series.apply(add_custom_values, june=30, july=20, august=25)
London      95
New York    96
Helsinki    87
dtype: int64
```

Use a function from the Numpy library.

```
>>> series.apply(np.log)
London      2.995732
New York    3.044522
Helsinki    2.484907
dtype: float64
```

argmax(***kwargs*)

Deprecated since version 0.21.0:will be corrected to return the positional maximum in the future. Use 'series.values.argmax' to get the position of the maximum now.

Return the row label of the maximum value.

If multiple values equal the maximum, the first row label with that value is returned.

'argmax' is deprecated, use 'idxmax' instead. The behavior of 'argmax' Parameters

skipna [boolean, default True] Exclude NA/null values. If the entire Series is NA, the result will be NA.

axis [int, default 0] For compatibility with DataFrame.idxmax. Redundant for application on Series.

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

Returns

idxmax [Index of maximum of values.]

Raises

ValueError If the Series is empty.

See also:

numpy.argmax Return indices of the maximum values along the given axis.

DataFrame.idxmax Return index of first occurrence of maximum over requested axis.

Series.idxmin Return index *label* of the first occurrence of minimum of values.

Notes

This method is the Series version of `ndarray.argmax`. This method returns the label of the maximum, while `ndarray.argmax` returns the position. To get the position, use `series.values.argmax()`.

Examples

```
>>> s = pd.Series(data=[1, None, 4, 3, 4],
...               index=['A', 'B', 'C', 'D', 'E'])
>>> s
A    1.0
B    NaN
C    4.0
D    3.0
E    4.0
dtype: float64
```

```
>>> s.idxmax()
'C'
```

If *skipna* is False and there is an NA value in the data, the function returns `nan`.

```
>>> s.idxmax(skipna=False)
nan
```

argmin(***kwargs*)

Deprecated since version 0.21.0:will be corrected to return the positional minimum in the future. Use 'series.values.argmin' to get the position of the minimum now.

Return the row label of the minimum value.

If multiple values equal the minimum, the first row label with that value is returned.

'argmin' is deprecated, use 'idxmin' instead. The behavior of 'argmin' Parameters

skipna [boolean, default True] Exclude NA/null values. If the entire Series is NA, the result will be NA.

axis [int, default 0] For compatibility with DataFrame.idxmin. Redundant for application on Series.

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

Returns

idxmin [Index of minimum of values.]

Raises

ValueError If the Series is empty.

See also:

numpy.argmin Return indices of the minimum values along the given axis.

DataFrame.idxmin Return index of first occurrence of minimum over requested axis.

Series.idxmax Return index *label* of the first occurrence of maximum of values.

Notes

This method is the Series version of `ndarray.argmin`. This method returns the label of the minimum, while `ndarray.argmin` returns the position. To get the position, use `series.values.argmin()`.

Examples

```
>>> s = pd.Series(data=[1, None, 4, 1],
...               index=['A' ,'B' ,'C' ,'D'])
>>> s
A    1.0
B    NaN
C    4.0
D    1.0
dtype: float64
```

```
>>> s.idxmin()
'A'
```

If *skipna* is False and there is an NA value in the data, the function returns `nan`.

```
>>> s.idxmin(skipna=False)
nan
```

argsort (*axis=0*, *kind='quicksort'*, *order=None*)

Overrides ndarray.argsort. Argsorts the value, omitting NA/null values, and places the result in the same locations as the non-NA values

Parameters

axis [int (can only be zero)]

kind [{'mergesort', 'quicksort', 'heapsort'}, default 'quicksort'] Choice of sorting algorithm. See np.sort for more information. 'mergesort' is the only stable algorithm

order [ignored]

Returns

argsorted [Series, with -1 indicated where nan values are present]

See also:

`numpy.ndarray.argsort`

as_blocks (*copy=True*)

Convert the frame to a dict of dtype -> Constructor Types that each has a homogeneous dtype.

Deprecated since version 0.21.0.

NOTE: the dtypes of the blocks WILL BE PRESERVED HERE (unlike in as_matrix)

Parameters

copy [boolean, default True]

Returns

values [a dict of dtype -> Constructor Types]

as_matrix(*columns=None*)

Convert the frame to its Numpy-array representation.

Deprecated since version 0.23.0: Use `DataFrame.values()` instead.

Parameters

columns: list, optional, default:None If None, return all columns, otherwise, returns specified columns.

Returns

values [ndarray] If the caller is heterogeneous and contains booleans or objects, the result will be of dtype=object. See Notes.

See also:

`pandas.DataFrame.values`

Notes

Return is NOT a Numpy-matrix, rather, a Numpy-array.

The dtype will be a lower-common-denominator dtype (implicit upcasting); that is to say if the dtypes (even of numeric types) are mixed, the one that accommodates all will be chosen. Use this with care if you are not dealing with the blocks.

e.g. If the dtypes are float16 and float32, dtype will be upcast to float32. If dtypes are int32 and uint8, dtype will be upcase to int32. By numpy.find_common_type convention, mixing int64 and uint64 will result in a flot64 dtype.

This method is provided for backwards compatibility. Generally, it is recommended to use '.values'.

asfreq(*freq*, *method=None*, *how=None*, *normalize=False*, *fill_value=None*)

Convert TimeSeries to specified frequency.

Optionally provide filling method to pad/backfill missing values.

Returns the original data conformed to a new index with the specified frequency. `resample` is more appropriate if an operation, such as summarization, is necessary to represent the data at the new frequency.

Parameters

freq [DateOffset object, 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).

New in version 0.20.0.

Returns

converted [type of caller]

See also:

reindex

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
```

asobject
Return object Series which contains boxed values.

Deprecated since version 0.23.0: Use `astype(object)` instead.

this is an internal non-public method

asof(*where*, *subset=None*)
The last row without any NaN is taken (or the last row without NaN considering only the subset of columns in the case of a DataFrame)

New in version 0.19.0: For DataFrame

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

Parameters

where [date or array of dates]

subset [string or list of strings, default None] if not None use these columns for NaN propagation

Returns

where is scalar

- value or NaN if input is Series
- Series if input is DataFrame

where is Index: same shape object as input

See also:

`merge_asof`

Notes

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

astype(***kwargs*)
Cast a pandas object to a specified dtype `dtype`.

Parameters

dtype [data type, or dict of column name -> data type] Use a numpy.dtype or Python type to cast entire pandas object to the same type. Alternatively, use {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

New in version 0.20.0.

raise_on_error [raise on invalid input] Deprecated since version 0.20.0: Use `errors` instead

kwargs [keyword arguments to pass on to the constructor]

Returns

casted [type of caller]

See also:

`pandas.to_datetime` Convert argument to datetime.

`pandas.to_timedelta` Convert argument to timedelta.

`pandas.to_numeric` Convert argument to a numeric type.

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

Examples

```
>>> 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, int64): [1, 2]
```

Convert to ordered categorical type with custom ordering:

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

Note that using `copy=False` and changing data on a new pandas object may propagate changes:

```
>>> s1 = pd.Series([1,2])
>>> s2 = s1.astype('int64', copy=False)
>>> s2[0] = 10
>>> s1  # note that s1[0] has changed too
0    10
1     2
dtype: int64
```

at

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

Similar to `loc`, in that both provide label-based lookups. Use `at` if you only need to get or set a single value in a DataFrame or Series.

Raises

KeyError When label does not exist in DataFrame

See also:

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

DataFrame.loc Access a group of rows and columns by label(s)

Series.at Access a single value using a label

Examples

```
>>> df = pd.DataFrame([[0, 2, 3], [0, 4, 1], [10, 20, 30]],
...                   index=[4, 5, 6], columns=['A', 'B', 'C'])
>>> df
    A   B   C
4   0   2   3
5   0   4   1
6  10  20  30
```

Get value at specified row/column pair

```
>>> df.at[4, 'B']
2
```

Set value at specified row/column pair

```
>>> df.at[4, 'B'] = 10
>>> df.at[4, 'B']
10
```

Get value within a Series

```
>>> df.loc[5].at['B']
4
```

at_time (*time*, *asof=False*)

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

Parameters

time [datetime.time or string]

Returns

values_at_time [type of caller]

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
```

`autocorr`(*lag=1*)
Lag-N autocorrelation

Parameters

lag [int, default 1] Number of lags to apply before performing autocorrelation.

Returns

autocorr [float]

`axes`
Return a list of the row axis labels

`base`
return the base object if the memory of the underlying data is shared

`between`(*left*, *right*, *inclusive=True*)
Return boolean Series equivalent to left <= series <= right.

This function returns a boolean vector containing *True* wherever the corresponding Series element is between the boundary values *left* and *right*. NA values are treated as *False*.

Parameters

left [scalar] Left boundary.

right [scalar] Right boundary.

inclusive [bool, default True] Include boundaries.

Returns

Series Each element will be a boolean.

See also:

pandas.Series.gt Greater than of series and other

pandas.Series.lt Less than of series and other

Notes

This function is equivalent to `(left <= ser) & (ser <= right)`

Examples

```
>>> s = pd.Series([2, 0, 4, 8, np.nan])
```

Boundary values are included by default:

```
>>> s.between(1, 4)
0     True
1    False
2     True
3    False
4    False
dtype: bool
```

With *inclusive* set to `False` boundary values are excluded:

```
>>> s.between(1, 4, inclusive=False)
0     True
1    False
2    False
3    False
4    False
dtype: bool
```

left and *right* can be any scalar value:

```
>>> s = pd.Series(['Alice', 'Bob', 'Carol', 'Eve'])
>>> s.between('Anna', 'Daniel')
0    False
1     True
2     True
3    False
dtype: bool
```

between_time (*start_time*, *end_time*, *include_start=True*, *include_end=True*)

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 string]

end_time [datetime.time or string]

include_start [boolean, default True]

include_end [boolean, default True]

Returns

values_between_time [type of caller]

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
```

`bfill`(*axis=None*, *inplace=False*, *limit=None*, *downcast=None*)
Synonym for `DataFrame.fillna(method='bfill')`

`blocks`
Internal property, property synonym for as_blocks()

Deprecated since version 0.21.0.

bool()
Return the bool of a single element PandasObject.

This must be a boolean scalar value, either True or False. Raise a ValueError if the PandasObject does not have exactly 1 element, or that element is not boolean

calcfp(**args*, ***kwargs*)
Helper function to map FP calculation throuugh the series

cat
alias of `pandas.core.arrays.categorical.CategoricalAccessor`

clip(*lower=None*, *upper=None*, *axis=None*, *inplace=False*, **args*, ***kwargs*)
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.

upper [float or array_like, default None] Maximum threshold value. All values above this threshold will be set to it.

axis [int or string axis name, optional] Align object with lower and upper along the given axis.

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

New in version 0.21.0.

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

Returns

Series or DataFrame Same type as calling object with the values outside the clip boundaries replaced

See also:

clip_lower Clip values below specified threshold(s).

clip_upper Clip values above specified threshold(s).

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
```

clip_lower(*threshold*, *axis=None*, *inplace=False*)

Return copy of the input with values below a threshold truncated.

Parameters

threshold [numeric or array-like] Minimum value allowed. All values below threshold will be set to this value.

- float : every value is compared to *threshold*.
- array-like : The shape of *threshold* should match the object it's compared to. When *self* is a Series, *threshold* should be the length. When *self* is a DataFrame, *threshold* should 2-D and the same shape as *self* for `axis=None`, or 1-D and the same length as the axis being compared.

axis [{0 or 'index', 1 or 'columns'}, default 0] Align *self* with *threshold* along the given axis.

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

New in version 0.21.0.

Returns

clipped [same type as input]

See also:

Series.clip Return copy of input with values below and above thresholds truncated.

Series.clip_upper Return copy of input with values above threshold truncated.

Examples

Series single threshold clipping:

```
>>> s = pd.Series([5, 6, 7, 8, 9])
>>> s.clip_lower(8)
0    8
1    8
2    8
3    8
4    9
dtype: int64
```

Series clipping element-wise using an array of thresholds. *threshold* should be the same length as the Series.

```
>>> elemwise_thresholds = [4, 8, 7, 2, 5]
>>> s.clip_lower(elemwise_thresholds)
0    5
1    8
2    7
3    8
4    9
dtype: int64
```

DataFrames can be compared to a scalar.

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

```
>>> df.clip_lower(3)
   A  B
0  3  3
1  3  4
2  5  6
```

Or to an array of values. By default, *threshold* should be the same shape as the DataFrame.

```
>>> df.clip_lower(np.array([[3, 4], [2, 2], [6, 2]]))
   A  B
0  3  4
1  3  4
2  6  6
```

Control how *threshold* is broadcast with *axis*. In this case *threshold* should be the same length as the axis specified by *axis*.

```
>>> df.clip_lower(np.array([3, 3, 5]), axis='index')
   A  B
0  3  3
1  3  4
2  5  6
```

```
>>> df.clip_lower(np.array([4, 5]), axis='columns')
   A  B
0  4  5
1  4  5
2  5  6
```

clip_upper (*threshold*, *axis=None*, *inplace=False*)
Return copy of input with values above given value(s) truncated.

Parameters

threshold [float or array_like]

axis [int or string axis name, optional] Align object with threshold along the given axis.

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

New in version 0.21.0.

Returns

clipped [same type as input]

See also:

clip

combine (*other*, *func*, *fill_value=nan*)
Perform elementwise binary operation on two Series using given function with optional fill value when an index is missing from one Series or the other

Parameters

other [Series or scalar value]

func [function] Function that takes two scalars as inputs and return a scalar

fill_value [scalar value]

Returns

result [Series]

See also:

Series.combine_first Combine Series values, choosing the calling Series's values first

Examples

```
>>> s1 = Series([1, 2])
>>> s2 = Series([0, 3])
>>> s1.combine(s2, lambda x1, x2: x1 if x1 < x2 else x2)
0    0
1    2
dtype: int64
```

combine_first (*other*)
Combine Series values, choosing the calling Series's values first. Result index will be the union of the two indexes

Parameters

other [Series]

Returns

combined [Series]

See also:

Series.combine Perform elementwise operation on two Series using a given function

Examples

```
>>> s1 = pd.Series([1, np.nan])
>>> s2 = pd.Series([3, 4])
>>> s1.combine_first(s2)
0    1.0
1    4.0
dtype: float64
```

compound(*axis=None*, *skipna=None*, *level=None*)

Return the compound percentage of the values for the requested axis

Parameters

axis [{index (0)}]

skipna [boolean, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

Returns

compounded [scalar or Series (if level specified)]

compress(*condition*, **args*, ***kwargs*)

Return selected slices of an array along given axis as a Series

See also:

numpy.ndarray.compress

consolidate(*inplace=False*)

Compute NDFrame with "consolidated" internals (data of each dtype grouped together in a single ndarray).

Deprecated since version 0.20.0: Consolidate will be an internal implementation only.

convert_objects(*convert_dates=True*, *convert_numeric=False*, *convert_timedeltas=True*, *copy=True*)

Attempt to infer better dtype for object columns.

Deprecated since version 0.21.0.

Parameters

convert_dates [boolean, default True] If True, convert to date where possible. If 'coerce', force conversion, with unconvertible values becoming NaT.

convert_numeric [boolean, default False] If True, attempt to coerce to numbers (including strings), with unconvertible values becoming NaN.

convert_timedeltas [boolean, default True] If True, convert to timedelta where possible. If 'coerce', force conversion, with unconvertible values becoming NaT.

copy [boolean, default True] If True, return a copy even if no copy is necessary (e.g. no conversion was done). Note: This is meant for internal use, and should not be confused with inplace.

Returns

converted [same as input object]

See also:

pandas.to_datetime Convert argument to datetime.

pandas.to_timedelta Convert argument to timedelta.

pandas.to_numeric Return a fixed frequency timedelta index, with day as the default.

copy (*deep=True*)

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

copy [Series, DataFrame or Panel] 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.

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[0] = 3
>>> shallow[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
```

corr(*other*, *method='pearson'*, *min_periods=None*)
Compute correlation with *other* Series, excluding missing values

Parameters

other [Series]

method [{'pearson', 'kendall', 'spearman'}]

- pearson : standard correlation coefficient
- kendall : Kendall Tau correlation coefficient
- spearman : Spearman rank correlation

min_periods [int, optional] Minimum number of observations needed to have a valid result

Returns

correlation [float]

count(*level=None*)
Return number of non-NA/null observations in the Series

Parameters

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a smaller Series

Returns

nobs [int or Series (if level specified)]

cov(*other*, *min_periods=None*)
Compute covariance with Series, excluding missing values

Parameters

other [Series]

min_periods [int, optional] Minimum number of observations needed to have a valid result

Returns

covariance [float]

Normalized by N-1 (unbiased estimator).

cummax(*axis=None*, *skipna=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'.

skipna [boolean, 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

cummax [scalar or Series]

See also:

pandas.core.window.Expanding.max Similar functionality but ignores `NaN` values.

Series.max Return the maximum over Series axis.

Series.cummax Return cumulative maximum over Series axis.

Series.cummin Return cumulative minimum over Series axis.

Series.cumsum Return cumulative sum over Series axis.

Series.cumprod Return cumulative product over Series 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
```

(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 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
```

cummin (*axis=None*, *skipna=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'.

skipna [boolean, 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

cummin [scalar or Series]

See also:

pandas.core.window.Expanding.min Similar functionality but ignores `NaN` values.

Series.min Return the minimum over Series axis.

Series.cummax Return cumulative maximum over Series axis.

Series.cummin Return cumulative minimum over Series axis.

Series.cumsum Return cumulative sum over Series axis.

Series.cumprod Return cumulative product over Series 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
```

cumprod(*axis=None*, *skipna=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'.

skipna [boolean, 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

cumprod [scalar or Series]

See also:

pandas.core.window.Expanding.prod Similar functionality but ignores `NaN` values.

Series.prod Return the product over Series axis.

Series.cummax Return cumulative maximum over Series axis.

Series.cummin Return cumulative minimum over Series axis.

Series.cumsum Return cumulative sum over Series axis.

Series.cumprod Return cumulative product over Series 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
```

cumsum(*axis=None*, *skipna=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'.

skipna [boolean, 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

cumsum [scalar or Series]

See also:

pandas.core.window.Expanding.sum Similar functionality but ignores `NaN` values.

Series.sum Return the sum over Series axis.

Series.cummax Return cumulative maximum over Series axis.

Series.cummin Return cumulative minimum over Series axis.

Series.cumsum Return cumulative sum over Series axis.

Series.cumprod Return cumulative product over Series 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
```

data
return the data pointer of the underlying data

describe (*percentiles=None*, *include=None*, *exclude=None*)
Generates descriptive statistics 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(include=['O'])`). To exclude pandas categorical columns, use `'category'`
- None (default) : The result will exclude nothing.

Returns

summary: Series/DataFrame of summary statistics

See also:

`DataFrame.count`, `DataFrame.max`, `DataFrame.min`, `DataFrame.mean`, `DataFrame.std`, `DataFrame.select_dtypes`

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
```

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
unique                      2
top       2010-01-01 00:00:00
```

(continues on next page)

(continued from previous page)

```
freq                          2
first     2000-01-01 00:00:00
last      2010-01-01 00:00:00
dtype: object
```

Describing a `DataFrame`. By default only numeric fields are returned.

```
>>> df = pd.DataFrame({ 'object': ['a', 'b', 'c'],
...                     'numeric': [1, 2, 3],
...                     'categorical': pd.Categorical(['d','e','f'])
...                   })
>>> 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      c
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
```

(continues on next page)

(continued from previous page)

```
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=[np.object])
       object
count       3
unique      3
top         c
freq        1
```

Including only categorical columns from a `DataFrame` description.

```
>>> df.describe(include=['category'])
       categorical
count            3
unique           3
top              f
freq             1
```

Excluding numeric columns from a `DataFrame` description.

```
>>> df.describe(exclude=[np.number])
       categorical object
count            3      3
unique           3      3
top              f      c
freq             1      1
```

Excluding object columns from a `DataFrame` description.

```
>>> df.describe(exclude=[np.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
```

diff (*periods=1*)

First discrete difference of element.

Calculates the difference of a Series element compared with another element in the Series (default is element in previous row).

Parameters

periods [int, default 1] Periods to shift for calculating difference, accepts negative values.

Returns

diffed [Series]

See also:

`Series.pct_change` Percent change over given number of periods.

`Series.shift` Shift index by desired number of periods with an optional time freq.

`DataFrame.diff` First discrete difference of object

Examples

Difference with previous row

```
>>> s = pd.Series([1, 1, 2, 3, 5, 8])
>>> s.diff()
0    NaN
1    0.0
2    1.0
3    1.0
4    2.0
5    3.0
dtype: float64
```

Difference with 3rd previous row

```
>>> s.diff(periods=3)
0    NaN
1    NaN
2    NaN
3    2.0
4    4.0
5    6.0
dtype: float64
```

Difference with following row

```
>>> s.diff(periods=-1)
0    0.0
1   -1.0
2   -1.0
3   -2.0
4   -3.0
5    NaN
dtype: float64
```

`div`(*other*, *level=None*, *fill_value=None*, *axis=0*)

Floating division of series and other, element-wise (binary operator *truediv*).

Equivalent to `series / other`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters

other [Series or scalar value]

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

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

Returns

result [Series]

See also:

`Series.rtruediv`

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

divide (*other*, *level=None*, *fill_value=None*, *axis=0*)

Floating division of series and other, element-wise (binary operator *truediv*).

Equivalent to `series / other`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters

other [Series or scalar value]

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

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

Returns

result [Series]

See also:

`Series.rtruediv`

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

divmod(*other*, *level=None*, *fill_value=None*, *axis=0*)

Integer division and modulo of series and other, element-wise (binary operator *divmod*).

Equivalent to `series divmod other`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters

other [Series or scalar value]

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

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

Returns

result [Series]

See also:

`Series.None`

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

dot (*other*)

Matrix multiplication with DataFrame or inner-product with Series objects. Can also be called using *self @ other* in Python >= 3.5.

Parameters

other [Series or DataFrame]

Returns

dot_product [scalar or Series]

drop (*labels=None*, *axis=0*, *index=None*, *columns=None*, *level=None*, *inplace=False*, *errors='raise'*)

Return Series with specified index labels removed.

Remove elements of a Series based on specifying the index labels. When using a multi-index, labels on different levels can be removed by specifying the level.

Parameters

labels [single label or list-like] Index labels to drop.

axis [0, default 0] Redundant for application on Series.

index, columns [None] Redundant for application on Series, but index can be used instead of labels.

New in version 0.21.0.

level [int or level name, optional] For MultiIndex, level for which the labels will be removed.

inplace [bool, default False] If True, do operation inplace and return None.

errors [{'ignore', 'raise'}, default 'raise'] If 'ignore', suppress error and only existing labels are dropped.

Returns

dropped [pandas.Series]

Raises

KeyError If none of the labels are found in the index.

See also:

`Series.reindex` Return only specified index labels of Series.

`Series.dropna` Return series without null values.

`Series.drop_duplicates` Return Series with duplicate values removed.

`DataFrame.drop` Drop specified labels from rows or columns.

Examples

```
>>> s = pd.Series(data=np.arange(3), index=['A','B','C'])
>>> s
A  0
B  1
C  2
dtype: int64
```

Drop labels B en C

```
>>> s.drop(labels=['B','C'])
A  0
dtype: int64
```

Drop 2nd level label in MultiIndex Series

```
>>> midx = pd.MultiIndex(levels=[['lama', 'cow', 'falcon'],
...                              ['speed', 'weight', 'length']],
...                      labels=[[0, 0, 0, 1, 1, 1, 2, 2, 2],
...                              [0, 1, 2, 0, 1, 2, 0, 1, 2]])
>>> s = pd.Series([45, 200, 1.2, 30, 250, 1.5, 320, 1, 0.3],
...               index=midx)
>>> s
lama    speed      45.0
        weight    200.0
        length      1.2
cow     speed      30.0
        weight    250.0
        length      1.5
falcon  speed     320.0
        weight      1.0
        length      0.3
dtype: float64
```

```
>>> s.drop(labels='weight', level=1)
lama    speed      45.0
        length      1.2
cow     speed      30.0
        length      1.5
falcon  speed     320.0
```

(continues on next page)

(continued from previous page)

```
        length      0.3
dtype: float64
```

drop_duplicates(*keep='first'*, *inplace=False*)

Return Series with duplicate values removed.

Parameters

keep [{'first', 'last', `False`}, default 'first']

- 'first' : Drop duplicates except for the first occurrence.
- 'last' : Drop duplicates except for the last occurrence.
- `False` : Drop all duplicates.

inplace [boolean, default `False`] If `True`, performs operation inplace and returns None.

Returns

deduplicated [Series]

See also:

Index.drop_duplicates equivalent method on Index

DataFrame.drop_duplicates equivalent method on DataFrame

Series.duplicated related method on Series, indicating duplicate Series values.

Examples

Generate an Series with duplicated entries.

```
>>> s = pd.Series(['lama', 'cow', 'lama', 'beetle', 'lama', 'hippo'],
...               name='animal')
>>> s
0      lama
1       cow
2      lama
3    beetle
4      lama
5     hippo
Name: animal, dtype: object
```

With the 'keep' parameter, the selection behaviour of duplicated values can be changed. The value 'first' keeps the first occurrence for each set of duplicated entries. The default value of keep is 'first'.

```
>>> s.drop_duplicates()
0      lama
1       cow
3    beetle
5     hippo
Name: animal, dtype: object
```

The value 'last' for parameter 'keep' keeps the last occurrence for each set of duplicated entries.

```
>>> s.drop_duplicates(keep='last')
1       cow
3    beetle
4      lama
5     hippo
Name: animal, dtype: object
```

The value `False` for parameter 'keep' discards all sets of duplicated entries. Setting the value of 'inplace' to `True` performs the operation inplace and returns `None`.

```
>>> s.drop_duplicates(keep=False, inplace=True)
>>> s
1       cow
3    beetle
5     hippo
Name: animal, dtype: object
```

dropna (*axis=0*, *inplace=False*, ***kwargs*)

Return a new Series with missing values removed.

See the User Guide for more on which values are considered missing, and how to work with missing data.

Parameters

axis [{0 or 'index'}, default 0] There is only one axis to drop values from.

inplace [bool, default False] If True, do operation inplace and return None.

****kwargs** Not in use.

Returns

Series Series with NA entries dropped from it.

See also:

Series.isna Indicate missing values.

Series.notna Indicate existing (non-missing) values.

Series.fillna Replace missing values.

DataFrame.dropna Drop rows or columns which contain NA values.

Index.dropna Drop missing indices.

Examples

```
>>> ser = pd.Series([1., 2., np.nan])
>>> ser
0    1.0
1    2.0
2    NaN
dtype: float64
```

Drop NA values from a Series.

```
>>> ser.dropna()
0    1.0
1    2.0
dtype: float64
```

Keep the Series with valid entries in the same variable.

```
>>> ser.dropna(inplace=True)
>>> ser
0    1.0
1    2.0
dtype: float64
```

Empty strings are not considered NA values. `None` is considered an NA value.

```
>>> ser = pd.Series([np.NaN, 2, pd.NaT, '', None, 'I stay'])
>>> ser
0       NaN
1         2
2       NaT
3
4      None
5    I stay
dtype: object
>>> ser.dropna()
1         2
3
5    I stay
dtype: object
```

dt
alias of `pandas.core.indexes.accessors.CombinedDatetimelikeProperties`

dtype
return the dtype object of the underlying data

dtypes
return the dtype object of the underlying data

duplicated(*keep='first'*)
Indicate duplicate Series values.

Duplicated values are indicated as `True` values in the resulting Series. Either all duplicates, all except the first or all except the last occurrence of duplicates can be indicated.

Parameters

keep [{'first', 'last', False}, default 'first']

- '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

pandas.core.series.Series

See also:

pandas.Index.duplicated Equivalent method on pandas.Index

pandas.DataFrame.duplicated Equivalent method on pandas.DataFrame

pandas.Series.drop_duplicates Remove duplicate values from Series

Examples

By default, for each set of duplicated values, the first occurrence is set on False and all others on True:

```
>>> animals = pd.Series(['lama', 'cow', 'lama', 'beetle', 'lama'])
>>> animals.duplicated()
0    False
1    False
2     True
3    False
4     True
dtype: bool
```

which is equivalent to

```
>>> animals.duplicated(keep='first')
0    False
1    False
2     True
3    False
4     True
dtype: bool
```

By using 'last', the last occurrence of each set of duplicated values is set on False and all others on True:

```
>>> animals.duplicated(keep='last')
0     True
1    False
2     True
3    False
4    False
dtype: bool
```

By setting keep on `False`, all duplicates are True:

```
>>> animals.duplicated(keep=False)
0     True
1    False
2     True
3    False
4     True
dtype: bool
```

empty

eq(*other*, *level=None*, *fill_value=None*, *axis=0*)

Equal to of series and other, element-wise (binary operator *eq*).

Equivalent to `series == other`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters

other [Series or scalar value]

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

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

Returns

result [Series]

See also:

`Series.None`

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

equals(*other*)
Determines if two NDFrame objects contain the same elements. NaNs in the same location are considered equal.

ewm(*com=None*, *span=None*, *halflife=None*, *alpha=None*, *min_periods=0*, *adjust=True*, *ignore_na=False*, *axis=0*)
Provides exponential weighted functions

New in version 0.18.0.

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, optional] Specify decay in terms of half-life, $\alpha = 1 - exp(log(0.5)/halflife)$, for $halflife > 0$

alpha [float, optional] Specify smoothing factor α directly, $0 < \alpha \leq 1$

New in version 0.18.0.

min_periods [int, default 0] Minimum number of observations in window required to have a value (otherwise result is NA).

adjust [boolean, default True] Divide by decaying adjustment factor in beginning periods to account for imbalance in relative weightings (viewing EWMA as a moving average)

ignore_na [boolean, default False] Ignore missing values when calculating weights; specify True to reproduce pre-0.15.0 behavior

Returns

a Window sub-classed for the particular operation

See also:

rolling Provides rolling window calculations

expanding Provides expanding transformations.

Notes

Exactly one of center of mass, span, half-life, and alpha must be provided. Allowed values and relationship between the parameters are specified in the parameter descriptions above; see the link at the end of this section for a detailed explanation.

When adjust is True (default), weighted averages are calculated using weights (1-alpha)**(n-1), (1-alpha)**(n-2), . . . , 1-alpha, 1.

When adjust is False, weighted averages are calculated recursively as: weighted_average[0] = arg[0]; weighted_average[i] = (1-alpha)*weighted_average[i-1] + alpha*arg[i].

When ignore_na is False (default), weights are based on absolute positions. For example, the weights of x and y used in calculating the final weighted average of [x, None, y] are (1-alpha)**2 and 1 (if adjust is True), and (1-alpha)**2 and alpha (if adjust is False).

When ignore_na is True (reproducing pre-0.15.0 behavior), weights are based on relative positions. For example, the weights of x and y used in calculating the final weighted average of [x, None, y] are 1-alpha and 1 (if adjust is True), and 1-alpha and alpha (if adjust is False).

More details can be found at http://pandas.pydata.org/pandas-docs/stable/computation.html#exponentially-weighted-windows

Examples

```
>>> df = DataFrame({'B': [0, 1, 2, np.nan, 4]})
     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
```

expanding(*min_periods=1*, *center=False*, *axis=0*)

Provides expanding transformations.

New in version 0.18.0.

Parameters

min_periods [int, default 1] Minimum number of observations in window required to have a value (otherwise result is NA).

center [boolean, default False] Set the labels at the center of the window.

axis [int or string, default 0]

Returns

a Window sub-classed for the particular operation

See also:

rolling Provides rolling window calculations

ewm Provides exponential weighted functions

Notes

By default, the result is set to the right edge of the window. This can be changed to the center of the window by setting `center=True`.

Examples

```
>>> df = DataFrame({'B': [0, 1, 2, np.nan, 4]})
     B
0  0.0
1  1.0
2  2.0
3  NaN
4  4.0
```

```
>>> df.expanding(2).sum()
     B
0  NaN
1  1.0
2  3.0
3  3.0
4  7.0
```

factorize(*sort=False*, *na_sentinel=-1*)

Encode the object as an enumerated type or categorical variable.

This method is useful for obtaining a numeric representation of an array when all that matters is identifying distinct values. *factorize* is available as both a top-level function `pandas.factorize()`, and as a method `Series.factorize()` and `Index.factorize()`.

Parameters

sort [boolean, default False] Sort *uniques* and shuffle *labels* to maintain the relationship.

na_sentinel [int, default -1] Value to mark "not found".

Returns

labels [ndarray] An integer ndarray that's an indexer into *uniques*. `uniques.take(labels)` will have the same values as *values*.

uniques [ndarray, Index, or Categorical] The unique valid values. When *values* is Categorical, *uniques* is a Categorical. When *values* is some other pandas object, an *Index* is returned. Otherwise, a 1-D ndarray is returned.

Note: Even if there's a missing value in *values*, *uniques* will *not* contain an entry for it.

See also:

`pandas.cut` Discretize continuous-valued array.

`pandas.unique` Find the unique valuse in an array.

Examples

These examples all show factorize as a top-level method like `pd.factorize(values)`. The results are identical for methods like `Series.factorize()`.

```
>>> labels, uniques = pd.factorize(['b', 'b', 'a', 'c', 'b'])
>>> labels
array([0, 0, 1, 2, 0])
>>> uniques
array(['b', 'a', 'c'], dtype=object)
```

With `sort=True`, the *uniques* will be sorted, and *labels* will be shuffled so that the relationship is the maintained.

```
>>> labels, uniques = pd.factorize(['b', 'b', 'a', 'c', 'b'], sort=True)
>>> labels
array([1, 1, 0, 2, 1])
>>> uniques
array(['a', 'b', 'c'], dtype=object)
```

Missing values are indicated in *labels* with *na_sentinel* (`-1` by default). Note that missing values are never included in *uniques*.

```
>>> labels, uniques = pd.factorize(['b', None, 'a', 'c', 'b'])
>>> labels
array([ 0, -1,  1,  2,  0])
>>> uniques
array(['b', 'a', 'c'], dtype=object)
```

Thus far, we've only factorized lists (which are internally coerced to NumPy arrays). When factorizing pandas objects, the type of *uniques* will differ. For Categoricals, a *Categorical* is returned.

```
>>> cat = pd.Categorical(['a', 'a', 'c'], categories=['a', 'b', 'c'])
>>> labels, uniques = pd.factorize(cat)
>>> labels
array([0, 0, 1])
>>> uniques
[a, c]
Categories (3, object): [a, b, c]
```

Notice that `'b'` is in `uniques.categories`, desipite not being present in `cat.values`.

For all other pandas objects, an Index of the appropriate type is returned.

```
>>> cat = pd.Series(['a', 'a', 'c'])
>>> labels, uniques = pd.factorize(cat)
>>> labels
array([0, 0, 1])
>>> uniques
Index(['a', 'c'], dtype='object')
```

ffill(*axis=None*, *inplace=False*, *limit=None*, *downcast=None*)
Synonym for `DataFrame.fillna(method='ffill')`

fillna(*value=None*, *method=None*, *axis=None*, *inplace=False*, *limit=None*, *downcast=None*, ***kwargs*)
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', '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

axis [{0 or 'index'}]

inplace [boolean, 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

filled [Series]

See also:

interpolate Fill NaN values using interpolation.

reindex, *asfreq*

Examples

```
>>> df = pd.DataFrame([[np.nan, 2, np.nan, 0],
...                    [3, 4, np.nan, 1],
...                    [np.nan, np.nan, np.nan, 5],
...                    [np.nan, 3, np.nan, 4]],
...                    columns=list('ABCD'))
>>> df
     A    B   C  D
0  NaN  2.0 NaN  0
1  3.0  4.0 NaN  1
2  NaN  NaN NaN  5
3  NaN  3.0 NaN  4
```

Replace all NaN elements with 0s.

```
>>> df.fillna(0)
    A   B   C   D
0   0.0 2.0 0.0 0
1   3.0 4.0 0.0 1
2   0.0 0.0 0.0 5
3   0.0 3.0 0.0 4
```

We can also propagate non-null values forward or backward.

```
>>> df.fillna(method='ffill')
    A   B   C   D
0   NaN 2.0 NaN 0
1   3.0 4.0 NaN 1
2   3.0 4.0 NaN 5
3   3.0 3.0 NaN 4
```

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
1   3.0 4.0 2.0 1
2   0.0 1.0 2.0 5
3   0.0 3.0 2.0 4
```

Only replace the first NaN element.

```
>>> df.fillna(value=values, limit=1)
    A   B   C   D
0   0.0 2.0 2.0 0
1   3.0 4.0 NaN 1
2   NaN 1.0 NaN 5
3   NaN 3.0 NaN 4
```

filter(*items=None*, *like=None*, *regex=None*, *axis=None*)

Subset rows or columns of dataframe according to labels in the specified index.

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] List of info axis to restrict to (must not all be present)

like [string] Keep info axis where "arg in col == True"

regex [string (regular expression)] Keep info axis with re.search(regex, col) == True

axis [int or string axis name] The axis to filter on. By default this is the info axis, 'index' for Series, 'columns' for DataFrame

Returns

same type as input object

See also:

`pandas.DataFrame.loc`

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
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
```

first(*offset*)

Convenience method for subsetting initial periods of time series data based on a date offset.

Parameters

offset [string, DateOffset, dateutil.relativedelta]

Returns

subset [type of 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 calender days were returned, not the first 3 days observed in the dataset, and therefore data for 2018-04-13 was not returned.

first_valid_index()
Return index for first non-NA/null value.

Returns

scalar [type of index]

Notes

If all elements are non-NA/null, returns None. Also returns None for empty NDFrame.

flags
return the ndarray.flags for the underlying data

floordiv(*other*, *level=None*, *fill_value=None*, *axis=0*)
Integer division of series and other, element-wise (binary operator *floordiv*).

Equivalent to `series // other`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters

other [Series or scalar value]

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

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

Returns

result [Series]

See also:

`Series.rfloordiv`

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

classmethod from_array(*arr*, *index=None*, *name=None*, *dtype=None*, *copy=False*, *fastpath=False*)

Construct Series from array.

Deprecated since version 0.23.0: Use pd.Series(..) constructor instead.

classmethod from_csv(*path*, *sep=','*, *parse_dates=True*, *header=None*, *index_col=0*, *encoding=None*, *infer_datetime_format=False*)

Read CSV file.

Deprecated since version 0.21.0: Use `pandas.read_csv()` instead.

It is preferable to use the more powerful `pandas.read_csv()` for most general purposes, but `from_csv` makes for an easy roundtrip to and from a file (the exact counterpart of `to_csv`), especially with a time Series.

This method only differs from `pandas.read_csv()` in some defaults:

- *index_col* is `0` instead of `None` (take first column as index by default)
- *header* is `None` instead of `0` (the first row is not used as the column names)

- *parse_dates* is `True` instead of `False` (try parsing the index as datetime by default)

With `pandas.read_csv()`, the option `squeeze=True` can be used to return a Series like `from_csv`.

Parameters

path [string file path or file handle / StringIO]

sep [string, default ‘,’] Field delimiter

parse_dates [boolean, default True] Parse dates. Different default from read_table

header [int, default None] Row to use as header (skip prior rows)

index_col [int or sequence, default 0] Column to use for index. If a sequence is given, a MultiIndex is used. Different default from read_table

encoding [string, optional] a string representing the encoding to use if the contents are non-ascii, for python versions prior to 3

infer_datetime_format: boolean, default False If True and *parse_dates* is True for a column, try to infer the datetime format based on the first datetime string. If the format can be inferred, there often will be a large parsing speed-up.

Returns

y [Series]

See also:

`pandas.read_csv`

ftype
return if the data is sparseldense

ftypes
return if the data is sparseldense

ge(*other*, *level=None*, *fill_value=None*, *axis=0*)
Greater than or equal to of series and other, element-wise (binary operator *ge*).

Equivalent to `series >= other`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters

other [Series or scalar value]

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

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

Returns

result [Series]

See also:

`Series.None`

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

get(*key*, *default=None*)

Get item from object for given key (DataFrame column, Panel slice, etc.). Returns default value if not found.

Parameters

key [object]

Returns

value [type of items contained in object]

get_dtype_counts()

Return counts of unique dtypes in this object.

Returns

dtype [Series] Series with the count of columns with each dtype.

See also:

dtypes Return the dtypes in this object.

Examples

```
>>> a = [['a', 1, 1.0], ['b', 2, 2.0], ['c', 3, 3.0]]
>>> df = pd.DataFrame(a, columns=['str', 'int', 'float'])
>>> df
  str  int  float
0   a    1    1.0
1   b    2    2.0
2   c    3    3.0
```

```
>>> df.get_dtype_counts()
float64    1
int64      1
object     1
dtype: int64
```

get_ftype_counts()
Return counts of unique ftypes in this object.

Deprecated since version 0.23.0.

This is useful for SparseDataFrame or for DataFrames containing sparse arrays.

Returns

dtype [Series] Series with the count of columns with each type and sparsity (dense/sparse)

See also:

ftypes Return ftypes (indication of sparse/dense and dtype) in this object.

Examples

```
>>> a = [['a', 1, 1.0], ['b', 2, 2.0], ['c', 3, 3.0]]
>>> df = pd.DataFrame(a, columns=['str', 'int', 'float'])
>>> df
  str  int  float
0   a    1    1.0
1   b    2    2.0
2   c    3    3.0
```

```
>>> df.get_ftype_counts()
float64:dense    1
int64:dense      1
object:dense     1
dtype: int64
```

get_value(*label*, *takeable=False*)
Quickly retrieve single value at passed index label

Deprecated since version 0.21.0: Please use .at[] or .iat[] accessors.

Parameters

label [object]

takeable [interpret the index as indexers, default False]

Returns

value [scalar value]

get_values()
same as values (but handles sparseness conversions); is a view

groupby(*by=None*, *axis=0*, *level=None*, *as_index=True*, *sort=True*, *group_keys=True*, *squeeze=False*, *observed=False*, ***kwargs*)
Group series using mapper (dict or key function, apply given function to group, return result as series) or by a series of columns.

Parameters

by [mapping, function, label, or list of labels] 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 an ndarray is passed, the values are used as-is 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 a (single) key.

axis [int, default 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

as_index [boolean, default True] For aggregated output, return object with group labels as the index. Only relevant for DataFrame input. as_index=False is effectively "SQL-style" grouped output

sort [boolean, 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.

group_keys [boolean, default True] When calling apply, add group keys to index to identify pieces

squeeze [boolean, default False] reduce the dimensionality of the return type if possible, otherwise return a consistent type

observed [boolean, 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.

New in version 0.23.0.

Returns

GroupBy object

See also:

resample Convenience method for frequency conversion and resampling of time series.

Notes

See the user guide for more.

Examples

DataFrame results

```
>>> data.groupby(func, axis=0).mean()
>>> data.groupby(['col1', 'col2'])['col3'].mean()
```

DataFrame with hierarchical index

```
>>> data.groupby(['col1', 'col2']).mean()
```

gt(*other*, *level=None*, *fill_value=None*, *axis=0*)

Greater than of series and other, element-wise (binary operator *gt*).

Equivalent to `series > other`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters

other [Series or scalar value]

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

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

Returns

result [Series]

See also:

`Series.None`

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

hasnans

return if I have any nans; enables various perf speedups

head(*n=5*)

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.

Parameters

n [int, default 5] Number of rows to select.

Returns

obj_head [type of caller] The first *n* rows of the caller object.

See also:

pandas.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
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
```

hist(*by=None*, *ax=None*, *grid=True*, *xlabelsize=None*, *xrot=None*, *ylabelsize=None*, *yrot=None*, *figsize=None*, *bins=10*, ***kwds*)
Draw histogram of the input series using matplotlib

Parameters

by [object, optional] If passed, then used to form histograms for separate groups

ax [matplotlib axis object] If not passed, uses gca()

grid [boolean, 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

ylabelsize [int, default None] If specified changes the y-axis label size

yrot [float, default None] rotation of y axis labels

figsize [tuple, default None] figure size in inches by default

bins [integer 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.

bins: integer, default 10 Number of histogram bins to be used

'kwds'** [keywords] To be passed to the actual plotting function

See also:

matplotlib.axes.Axes.hist Plot a histogram using matplotlib.

iat

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

Similar to `iloc`, in that both provide integer-based lookups. Use `iat` if you only need to get or set a single value in a DataFrame or Series.

Raises

IndexError When integer position is out of bounds

See also:

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

DataFrame.loc Access a group of rows and columns by label(s)

DataFrame.iloc Access a group of rows and columns by integer position(s)

Examples

```
>>> df = pd.DataFrame([[0, 2, 3], [0, 4, 1], [10, 20, 30]],
...                   columns=['A', 'B', 'C'])
>>> df
    A   B   C
0   0   2   3
1   0   4   1
2  10  20  30
```

Get value at specified row/column pair

```
>>> df.iat[1, 2]
1
```

Set value at specified row/column pair

```
>>> df.iat[1, 2] = 10
>>> df.iat[1, 2]
10
```

Get value within a series

```
>>> df.loc[0].iat[1]
2
```

idxmax(*axis=0*, *skipna=True*, **args*, ***kwargs*)

Return the row label of the maximum value.

If multiple values equal the maximum, the first row label with that value is returned.

Parameters

skipna [boolean, default True] Exclude NA/null values. If the entire Series is NA, the result will be NA.

axis [int, default 0] For compatibility with DataFrame.idxmax. Redundant for application on Series.

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

Returns

idxmax [Index of maximum of values.]

Raises

ValueError If the Series is empty.

See also:

numpy.argmax Return indices of the maximum values along the given axis.

DataFrame.idxmax Return index of first occurrence of maximum over requested axis.

Series.idxmin Return index *label* of the first occurrence of minimum of values.

Notes

This method is the Series version of `ndarray.argmax`. This method returns the label of the maximum, while `ndarray.argmax` returns the position. To get the position, use `series.values.argmax()`.

Examples

```
>>> s = pd.Series(data=[1, None, 4, 3, 4],
...               index=['A', 'B', 'C', 'D', 'E'])
>>> s
A    1.0
B    NaN
C    4.0
D    3.0
E    4.0
dtype: float64
```

```
>>> s.idxmax()
'C'
```

If *skipna* is False and there is an NA value in the data, the function returns `nan`.

```
>>> s.idxmax(skipna=False)
nan
```

idxmin(*axis=None*, *skipna=True*, **args*, ***kwargs*)

Return the row label of the minimum value.

If multiple values equal the minimum, the first row label with that value is returned.

Parameters

skipna [boolean, default True] Exclude NA/null values. If the entire Series is NA, the result will be NA.

axis [int, default 0] For compatibility with DataFrame.idxmin. Redundant for application on Series.

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

Returns

idxmin [Index of minimum of values.]

Raises

ValueError If the Series is empty.

See also:

numpy.argmin Return indices of the minimum values along the given axis.

DataFrame.idxmin Return index of first occurrence of minimum over requested axis.

Series.idxmax Return index *label* of the first occurrence of maximum of values.

Notes

This method is the Series version of `ndarray.argmin`. This method returns the label of the minimum, while `ndarray.argmin` returns the position. To get the position, use `series.values.argmin()`.

Examples

```
>>> s = pd.Series(data=[1, None, 4, 1],
...               index=['A' ,'B' ,'C' ,'D'])
>>> s
A    1.0
B    NaN
C    4.0
D    1.0
dtype: float64
```

```
>>> s.idxmin()
'A'
```

If *skipna* is False and there is an NA value in the data, the function returns `nan`.

```
>>> s.idxmin(skipna=False)
nan
```

iloc

Purely integer-location based indexing for selection by position.

`.iloc[]` is primarily integer position based (from `0` to `length-1` of the axis), but may also be used with a boolean array.

Allowed inputs are:

- An integer, e.g. `5`.
- A list or array of integers, e.g. `[4, 3, 0]`.
- A slice object with ints, e.g. `1:7`.
- A boolean array.
- A `callable` function with one argument (the calling Series, DataFrame or Panel) and that returns valid output for indexing (one of the above)

`.iloc` will raise `IndexError` if a requested indexer is out-of-bounds, except *slice* indexers which allow out-of-bounds indexing (this conforms with python/numpy *slice* semantics).

See more at Selection by Position

imag

index

The index (axis labels) of the Series.

infer_objects()

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.

New in version 0.21.0.

Returns

converted [same type as input object]

See also:

pandas.to_datetime Convert argument to datetime.

pandas.to_timedelta Convert argument to timedelta.

pandas.to_numeric Convert argument to numeric typeR

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
```

interpolate(*method='linear'*, *axis=0*, *limit=None*, *inplace=False*, *limit_direction='forward'*, *limit_area=None*, *downcast=None*, ***kwargs*)

Interpolate values according to different methods.

Please note that only `method='linear'` is supported for DataFrames/Series with a MultiIndex.

Parameters

method [{'linear', 'time', 'index', 'values', 'nearest', 'zero',]

'slinear', 'quadratic', 'cubic', 'barycentric', 'krogh', 'polynomial', 'spline', 'piecewise_polynomial', 'from_derivatives', 'pchip', 'akima'}

- 'linear': ignore the index and treat the values as equally spaced. This is the only method supported on MultiIndexes. default
- 'time': interpolation works on daily and higher resolution data to interpolate given length of interval
- 'index', 'values': use the actual numerical values of the index
- 'nearest', 'zero', 'slinear', 'quadratic', 'cubic', 'barycentric', 'polynomial' is passed to `scipy.interpolate.interp1d`. Both 'polynomial' and 'spline' require that you also specify an *order* (int), e.g. df.interpolate(method='polynomial', order=4). These use the actual numerical values of the index.
- 'krogh', 'piecewise_polynomial', 'spline', 'pchip' and 'akima' are all wrappers around the scipy interpolation methods of similar names. These use the actual numerical values of the index. For more information on their behavior, see the scipy documentation and tutorial documentation
- 'from_derivatives' refers to BPoly.from_derivatives which replaces 'piecewise_polynomial' interpolation method in scipy 0.18

New in version 0.18.1: Added support for the 'akima' method Added interpolate method 'from_derivatives' which replaces 'piecewise_polynomial' in scipy 0.18; backwards-compatible with scipy < 0.18

axis [{0, 1}, default 0]

- 0: fill column-by-column
- 1: fill row-by-row

limit [int, default None.] Maximum number of consecutive NaNs to fill. Must be greater than 0.

limit_direction [{'forward', 'backward', 'both'}, default 'forward']

limit_area [{'inside', 'outside'}, default None]

- None: (default) no fill restriction
- 'inside' Only fill NaNs surrounded by valid values (interpolate).
- 'outside' Only fill NaNs outside valid values (extrapolate).

If limit is specified, consecutive NaNs will be filled in this direction.

New in version 0.21.0.

inplace [bool, default False] Update the NDFrame in place if possible.

downcast [optional, 'infer' or None, defaults to None] Downcast dtypes if possible.

kwargs [keyword arguments to pass on to the interpolating function.]

Returns

Series or DataFrame of same shape interpolated at the NaNs

See also:

reindex, *replace*, *fillna*

Examples

Filling in NaNs

```
>>> s = pd.Series([0, 1, np.nan, 3])
>>> s.interpolate()
0    0
1    1
2    2
3    3
dtype: float64
```

is_copy

is_monotonic

Return boolean if values in the object are monotonic_increasing

New in version 0.19.0.

Returns

is_monotonic [boolean]

is_monotonic_decreasing

Return boolean if values in the object are monotonic_decreasing

New in version 0.19.0.

Returns

is_monotonic_decreasing [boolean]

is_monotonic_increasing

Return boolean if values in the object are monotonic_increasing

New in version 0.19.0.

Returns

is_monotonic [boolean]

is_unique

Return boolean if values in the object are unique

Returns

is_unique [boolean]

isin(*values*)

Check whether *values* are contained in Series.

Return a boolean Series showing whether each element in the Series matches an element in the passed sequence of *values* exactly.

Parameters

values [set or list-like] The sequence of values to test. Passing in a single string will raise a `TypeError`. Instead, turn a single string into a list of one element.

New in version 0.18.1: Support for values as a set.

Returns

isin [Series (bool dtype)]

Raises

TypeError

- If *values* is a string

See also:

pandas.DataFrame.isin equivalent method on DataFrame

Examples

```
>>> s = pd.Series(['lama', 'cow', 'lama', 'beetle', 'lama',
...                'hippo'], name='animal')
>>> s.isin(['cow', 'lama'])
0     True
1     True
2     True
3    False
4     True
5    False
Name: animal, dtype: bool
```

Passing a single string as `s.isin('lama')` will raise an error. Use a list of one element instead:

```
>>> s.isin(['lama'])
0     True
1    False
2     True
3    False
4     True
5    False
Name: animal, dtype: bool
```

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

Series Mask of bool values for each element in Series that indicates whether an element is not an NA value.

See also:

Series.isnull alias of isna

Series.notna boolean inverse of isna

Series.dropna omit axes labels with missing values

isna top-level isna

Examples

Show which entries in a DataFrame are NA.

```
>>> df = pd.DataFrame({'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
```

isnull()

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

Series Mask of bool values for each element in Series that indicates whether an element is not an NA value.

See also:

Series.isnull alias of isna

Series.notna boolean inverse of isna

Series.dropna omit axes labels with missing values

isna top-level isna

Examples

Show which entries in a DataFrame are NA.

```
>>> df = pd.DataFrame({'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
```

item()
return the first element of the underlying data as a python scalar

items()
Lazily iterate over (index, value) tuples

itemsize
return the size of the dtype of the item of the underlying data

iteritems()
Lazily iterate over (index, value) tuples

ix
A primarily label-location based indexer, with integer position fallback.

Warning: Starting in 0.20.0, the .ix indexer is deprecated, in favor of the more strict .iloc and .loc indexers.

`.ix[]` supports mixed integer and label based access. It is primarily label based, but will fall back to integer positional access unless the corresponding axis is of integer type.

`.ix` is the most general indexer and will support any of the inputs in `.loc` and `.iloc`. `.ix` also supports floating point label schemes. `.ix` is exceptionally useful when dealing with mixed positional and label based hierarchical indexes.

However, when an axis is integer based, ONLY label based access and not positional access is supported. Thus, in such cases, it's usually better to be explicit and use `.iloc` or `.loc`.

See more at Advanced Indexing.

keys()
Alias for index

kurt(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)
Return unbiased kurtosis over requested axis using Fisher's definition of kurtosis (kurtosis of normal == 0.0). Normalized by N-1

Parameters

axis [{index (0)}]

skipna [boolean, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

Returns

kurt [scalar or Series (if level specified)]

kurtosis(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)
Return unbiased kurtosis over requested axis using Fisher's definition of kurtosis (kurtosis of normal == 0.0). Normalized by N-1

Parameters

axis [{index (0)}]

skipna [boolean, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

Returns

kurt [scalar or Series (if level specified)]

last(*offset*)

Convenience method for subsetting final periods of time series data based on a date offset.

Parameters

offset [string, DateOffset, dateutil.relativedelta]

Returns

subset [type of 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

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 calender days were returned, not the last 3 observed days in the dataset, and therefore data for 2018-04-11 was not returned.

last_valid_index()

Return index for last non-NA/null value.

Returns

scalar [type of index]

Notes

If all elements are non-NA/null, returns None. Also returns None for empty NDFrame.

le(*other*, *level=None*, *fill_value=None*, *axis=0*)

Less than or equal to of series and other, element-wise (binary operator *le*).

Equivalent to `series <= other`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters

other [Series or scalar value]

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

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

Returns

result [Series]

See also:

`Series.None`

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

`loc`

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

`.loc[]` is primarily label based, but may also be used with a boolean array.

Allowed inputs are:

- A single label, e.g. `5` or `'a'`, (note that `5` is interpreted as a *label* of the index, and **never** as an integer position along the index).
- A list or array of labels, e.g. `['a', 'b', 'c']`.
- A slice object with labels, e.g. `'a':'f'`.

> **Warning:** Note that contrary to usual python slices, **both** the start and the stop are included

- A boolean array of the same length as the axis being sliced, e.g. `[True, False, True]`.
- A `callable` function with one argument (the calling Series, DataFrame or Panel) and that returns valid output for indexing (one of the above)

See more at Selection by Label

Raises

KeyError: when any items are not found

See also:

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

`DataFrame.iloc` Access group of rows and columns by integer position(s)

`DataFrame.xs` Returns a cross-section (row(s) or column(s)) from the Series/DataFrame.

`Series.loc` Access group of values using labels

Examples

Getting values

```
>>> df = pd.DataFrame([[1, 2], [4, 5], [7, 8]],
...       index=['cobra', 'viper', 'sidewinder'],
...       columns=['max_speed', 'shield'])
>>> df
            max_speed  shield
cobra               1       2
viper               4       5
sidewinder          7       8
```

Single label. Note this returns the row as a Series.

```
>>> df.loc['viper']
max_speed    4
shield       5
Name: viper, dtype: int64
```

List of labels. Note using `[[]]` returns a DataFrame.

```
>>> df.loc[['viper', 'sidewinder']]
            max_speed  shield
viper               4       5
sidewinder          7       8
```

Single label for row and column

```
>>> df.loc['cobra', 'shield']
2
```

Slice with labels for row and single label for column. As mentioned above, note that both the start and stop of the slice are included.

```
>>> df.loc['cobra':'viper', 'max_speed']
cobra    1
viper    4
Name: max_speed, dtype: int64
```

Boolean list with the same length as the row axis

```
>>> df.loc[[False, False, True]]
            max_speed  shield
sidewinder          7       8
```

Conditional that returns a boolean Series

```
>>> df.loc[df['shield'] > 6]
            max_speed  shield
sidewinder          7       8
```

Conditional that returns a boolean Series with column labels specified

```
>>> df.loc[df['shield'] > 6, ['max_speed']]
            max_speed
sidewinder          7
```

Callable that returns a boolean Series

```
>>> df.loc[lambda df: df['shield'] == 8]
            max_speed  shield
sidewinder          7       8
```

Setting values

Set value for all items matching the list of labels

```
>>> df.loc[['viper', 'sidewinder'], ['shield']] = 50
>>> df
            max_speed  shield
cobra               1       2
viper               4      50
sidewinder          7      50
```

Set value for an entire row

```
>>> df.loc['cobra'] = 10
>>> df
            max_speed  shield
cobra              10      10
viper               4      50
sidewinder          7      50
```

Set value for an entire column

```
>>> df.loc[:, 'max_speed'] = 30
>>> df
            max_speed  shield
cobra              30      10
viper              30      50
sidewinder         30      50
```

Set value for rows matching callable condition

```
>>> df.loc[df['shield'] > 35] = 0
>>> df
            max_speed  shield
cobra              30      10
viper               0       0
sidewinder          0       0
```

Getting values on a DataFrame with an index that has integer labels

Another example using integers for the index

```
>>> df = pd.DataFrame([[1, 2], [4, 5], [7, 8]],
...      index=[7, 8, 9], columns=['max_speed', 'shield'])
>>> df
   max_speed  shield
7          1       2
8          4       5
9          7       8
```

Slice with integer labels for rows. As mentioned above, note that both the start and stop of the slice are included.

```
>>> df.loc[7:9]
   max_speed  shield
7          1       2
8          4       5
9          7       8
```

Getting values with a MultiIndex

A number of examples using a DataFrame with a MultiIndex

```
>>> tuples = [
...     ('cobra', 'mark i'), ('cobra', 'mark ii'),
...     ('sidewinder', 'mark i'), ('sidewinder', 'mark ii'),
...     ('viper', 'mark ii'), ('viper', 'mark iii')
... ]
>>> index = pd.MultiIndex.from_tuples(tuples)
>>> values = [[12, 2], [0, 4], [10, 20],
...           [1, 4], [7, 1], [16, 36]]
>>> df = pd.DataFrame(values, columns=['max_speed', 'shield'], index=index)
>>> df
                     max_speed  shield
cobra      mark i           12       2
           mark ii           0       4
sidewinder mark i           10      20
           mark ii           1       4
viper      mark ii           7       1
           mark iii         16      36
```

Single label. Note this returns a DataFrame with a single index.

```
>>> df.loc['cobra']
         max_speed  shield
mark i          12       2
mark ii          0       4
```

Single index tuple. Note this returns a Series.

```
>>> df.loc[('cobra', 'mark ii')]
max_speed    0
shield       4
Name: (cobra, mark ii), dtype: int64
```

Single label for row and column. Similar to passing in a tuple, this returns a Series.

```
>>> df.loc['cobra', 'mark i']
max_speed    12
shield        2
Name: (cobra, mark i), dtype: int64
```

Single tuple. Note using `[[]]` returns a DataFrame.

```
>>> df.loc[[('cobra', 'mark ii')]]
               max_speed  shield
cobra mark ii          0       4
```

Single tuple for the index with a single label for the column

```
>>> df.loc[('cobra', 'mark i'), 'shield']
2
```

Slice from index tuple to single label

```
>>> df.loc[('cobra', 'mark i'):'viper']
                     max_speed  shield
cobra      mark i           12       2
           mark ii           0       4
sidewinder mark i           10      20
           mark ii           1       4
viper      mark ii           7       1
           mark iii         16      36
```

Slice from index tuple to index tuple

```
>>> df.loc[('cobra', 'mark i'):('viper', 'mark ii')]
                    max_speed  shield
cobra      mark i          12       2
           mark ii          0       4
sidewinder mark i          10      20
           mark ii          1       4
viper      mark ii          7       1
```

lt (*other*, *level=None*, *fill_value=None*, *axis=0*)

Less than of series and other, element-wise (binary operator *lt*).

Equivalent to `series < other`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters

other [Series or scalar value]

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

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

Returns

result [Series]

See also:

`Series.None`

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

mad(*axis=None*, *skipna=None*, *level=None*)

Return the mean absolute deviation of the values for the requested axis

Parameters

axis [{index (0)}]

skipna [boolean, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

Returns

mad [scalar or Series (if level specified)]

map(*arg*, *na_action=None*)

Map values of Series using input correspondence (a dict, Series, or function).

Parameters

arg [function, dict, or Series] Mapping correspondence.

na_action [{None, 'ignore'}] If 'ignore', propagate NA values, without passing them to the mapping correspondence.

Returns

y [Series] Same index as caller.

See also:

`Series.apply` For applying more complex functions on a Series.

`DataFrame.apply` Apply a function row-/column-wise.

`DataFrame.applymap` Apply a function elementwise on a whole DataFrame.

Notes

When *arg* is a dictionary, values in Series that are not in the dictionary (as keys) are converted to `NaN`. However, if the dictionary is a `dict` subclass that defines `__missing__` (i.e. provides a method for default values), then this default is used rather than `NaN`:

```
>>> from collections import Counter
>>> counter = Counter()
>>> counter['bar'] += 1
>>> y.map(counter)
1    0
2    1
3    0
dtype: int64
```

Examples

Map inputs to outputs (both of type *Series*):

```
>>> x = pd.Series([1,2,3], index=['one', 'two', 'three'])
>>> x
one      1
two      2
three    3
dtype: int64
```

```
>>> y = pd.Series(['foo', 'bar', 'baz'], index=[1,2,3])
>>> y
1    foo
2    bar
3    baz
```

```
>>> x.map(y)
one   foo
two   bar
three baz
```

If *arg* is a dictionary, return a new Series with values converted according to the dictionary's mapping:

```
>>> z = {1: 'A', 2: 'B', 3: 'C'}
```

```
>>> x.map(z)
one   A
two   B
three C
```

Use na_action to control whether NA values are affected by the mapping function.

```
>>> s = pd.Series([1, 2, 3, np.nan])
```

```
>>> s2 = s.map('this is a string {}'.format, na_action=None)
0    this is a string 1.0
1    this is a string 2.0
2    this is a string 3.0
3    this is a string nan
dtype: object
```

```
>>> s3 = s.map('this is a string {}'.format, na_action='ignore')
0    this is a string 1.0
1    this is a string 2.0
2    this is a string 3.0
3                    NaN
dtype: object
```

mask(*cond*, *other=nan*, *inplace=False*, *axis=None*, *level=None*, *errors='raise'*, *try_cast=False*, *raise_on_error=None*)

Return an object of same shape as self and whose corresponding entries are from self where *cond* is False and otherwise are from *other*.

Parameters

cond [boolean NDFrame, 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 NDFrame and should return boolean NDFrame or array. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1: A callable can be used as cond.

other [scalar, NDFrame, or callable] Entries where *cond* is True are replaced with corresponding value from *other*. If other is callable, it is computed on the NDFrame and should return scalar or NDFrame. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1: A callable can be used as other.

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

axis [alignment axis if needed, default None]

level [alignment level if needed, default None]

errors [str, {'raise', 'ignore'}, default 'raise']

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

Note that currently this parameter won't affect the results and will always coerce to a suitable dtype.

try_cast [boolean, default False] try to cast the result back to the input type (if possible),

raise_on_error [boolean, default True] Whether to raise on invalid data types (e.g. trying to where on strings)

Deprecated since version 0.21.0.

Returns

wh [same type as caller]

See also:

`DataFrame.where()`

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.

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.

Examples

```
>>> s = pd.Series(range(5))
>>> s.where(s > 0)
0    NaN
1    1.0
2    2.0
3    3.0
4    4.0
```

```
>>> s.mask(s > 0)
0    0.0
1    NaN
2    NaN
3    NaN
4    NaN
```

```
>>> s.where(s > 1, 10)
0    10.0
1    10.0
2    2.0
3    3.0
4    4.0
```

```
>>> df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])
>>> 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
```

max(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

This method returns the maximum of the values in the object. If you want the *index* of the maximum, use `idxmax`. This is the equivalent of the `numpy.ndarray` method `argmax`.

Parameters

axis [{index (0)}]

skipna [boolean, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

Returns

max [scalar or Series (if level specified)]

mean(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

Return the mean of the values for the requested axis

Parameters

axis [{index (0)}]

skipna [boolean, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

Returns

mean [scalar or Series (if level specified)]

median(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

Return the median of the values for the requested axis

Parameters

axis [{index (0)}]

skipna [boolean, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

Returns

median [scalar or Series (if level specified)]

memory_usage(*index=True*, *deep=False*)

Return the memory usage of the Series.

The memory usage can optionally include the contribution of the index and of elements of *object* dtype.

Parameters

index [bool, default True] Specifies whether to include the memory usage of the Series index.

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 value.

Returns

int Bytes of memory consumed.

See also:

numpy.ndarray.nbytes Total bytes consumed by the elements of the array.

DataFrame.memory_usage Bytes consumed by a DataFrame.

Examples

```
>>> s = pd.Series(range(3))
>>> s.memory_usage()
104
```

Not including the index gives the size of the rest of the data, which is necessarily smaller:

```
>>> s.memory_usage(index=False)
24
```

The memory footprint of *object* values is ignored by default:

```
>>> s = pd.Series(["a", "b"])
>>> s.values
array(['a', 'b'], dtype=object)
>>> s.memory_usage()
96
>>> s.memory_usage(deep=True)
212
```

min(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

This method returns the minimum of the values in the object. If you want the *index* of the minimum, use `idxmin`. This is the equivalent of the `numpy.ndarray` method `argmin`.

Parameters

axis [{index (0)}]

skipna [boolean, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

Returns

min [scalar or Series (if level specified)]

mod (*other*, *level=None*, *fill_value=None*, *axis=0*)

Modulo of series and other, element-wise (binary operator *mod*).

Equivalent to `series % other`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters

other [Series or scalar value]

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

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

Returns

result [Series]

See also:

`Series.rmod`

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
```

(continues on next page)

(continued from previous page)

```
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

mode()

Return the mode(s) of the dataset.

Always returns Series even if only one value is returned.

Returns

modes [Series (sorted)]

mul(*other*, *level=None*, *fill_value=None*, *axis=0*)

Multiplication of series and other, element-wise (binary operator *mul*).

Equivalent to `series * other`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters

other [Series or scalar value]

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

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

Returns

result [Series]

See also:

`Series.rmul`

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
```

(continues on next page)

(continued from previous page)

```
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

multiply(*other*, *level=None*, *fill_value=None*, *axis=0*)

Multiplication of series and other, element-wise (binary operator *mul*).

Equivalent to `series * other`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters

other [Series or scalar value]

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

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

Returns

result [Series]

See also:

`Series.rmul`

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
```

(continues on next page)

(continued from previous page)

```
e    NaN
dtype: float64
```

name

nbytes
return the number of bytes in the underlying data

ndim
return the number of dimensions of the underlying data, by definition 1

ne(*other*, *level=None*, *fill_value=None*, *axis=0*)
Not equal to of series and other, element-wise (binary operator *ne*).

Equivalent to `series != other`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters

other [Series or scalar value]

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

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

Returns

result [Series]

See also:

`Series.None`

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
```

(continues on next page)

(continued from previous page)

```
e    NaN
dtype: float64
```

nlargest (*n=5*, *keep='first'*)

Return the largest *n* elements.

Parameters

n [int] Return this many descending sorted values

keep [{'first', 'last'}, default 'first'] Where there are duplicate values: - `first` : take the first occurrence. - `last` : take the last occurrence.

Returns

top_n [Series] The n largest values in the Series, in sorted order

See also:

`Series.nsmallest`

Notes

Faster than `.sort_values(ascending=False).head(n)` for small *n* relative to the size of the `Series` object.

Examples

```
>>> import pandas as pd
>>> import numpy as np
>>> s = pd.Series(np.random.randn(10**6))
>>> s.nlargest(10)  # only sorts up to the N requested
219921    4.644710
82124     4.608745
421689    4.564644
425277    4.447014
718691    4.414137
43154     4.403520
283187    4.313922
595519    4.273635
503969    4.250236
121637    4.240952
dtype: float64
```

nonzero ()

Return the *integer* indices of the elements that are non-zero

This method is equivalent to calling *numpy.nonzero* on the series data. For compatibility with NumPy, the return value is the same (a tuple with an array of indices for each dimension), but it will always be a one-item tuple because series only have one dimension.

See also:

`numpy.nonzero`

Examples

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

```
>>> s = pd.Series([0, 3, 0, 4], index=['a', 'b', 'c', 'd'])
# same return although index of s is different
>>> s.nonzero()
(array([1, 3]),)
>>> s.iloc[s.nonzero()[0]]
b    3
d    4
dtype: int64
```

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

Series Mask of bool values for each element in Series that indicates whether an element is not an NA value.

See also:

Series.notnull alias of notna

Series.isna boolean inverse of notna

Series.dropna omit axes labels with missing values

notna top-level notna

Examples

Show which entries in a DataFrame are not NA.

```
>>> df = pd.DataFrame({'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
```

notnull()

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

Series Mask of bool values for each element in Series that indicates whether an element is not an NA value.

See also:

Series.notnull alias of notna

Series.isna boolean inverse of notna

Series.dropna omit axes labels with missing values

notna top-level notna

Examples

Show which entries in a DataFrame are not NA.

```
>>> df = pd.DataFrame({'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
```

nsmallest (*n=5*, *keep='first'*)

Return the smallest *n* elements.

Parameters

n [int] Return this many ascending sorted values

keep [{'first', 'last'}, default 'first'] Where there are duplicate values: - `first` : take the first occurrence. - `last` : take the last occurrence.

Returns

bottom_n [Series] The n smallest values in the Series, in sorted order

See also:

`Series.nlargest`

Notes

Faster than `.sort_values().head(n)` for small *n* relative to the size of the `Series` object.

Examples

```
>>> import pandas as pd
>>> import numpy as np
>>> s = pd.Series(np.random.randn(10**6))
>>> s.nsmallest(10)  # only sorts up to the N requested
288532   -4.954580
732345   -4.835960
64803    -4.812550
446457   -4.609998
501225   -4.483945
669476   -4.472935
973615   -4.401699
```

(continues on next page)

(continued from previous page)

```
621279   -4.355126
773916   -4.347355
359919   -4.331927
dtype: float64
```

nunique (*dropna=True*)

Return number of unique elements in the object.

Excludes NA values by default.

Parameters

dropna [boolean, default True] Don't include NaN in the count.

Returns

nunique [int]

pct_change (*periods=1*, *fill_method='pad'*, *limit=None*, *freq=None*, ***kwargs*)

Percentage change between the current and a prior element.

Computes the percentage change from the immediately previous row by default. This is useful in comparing the percentage of change in a time series of elements.

Parameters

periods [int, default 1] Periods to shift for forming percent change.

fill_method [str, default 'pad'] How to handle NAs before computing percent changes.

limit [int, default None] The number of consecutive NAs to fill before stopping.

freq [DateOffset, timedelta, or offset alias string, 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

chg [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
2     NaN
3    85.0
dtype: float64
```

```
>>> s.pct_change(fill_method='ffill')
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],
```

(continues on next page)

(continued from previous page)

```
...      '2014': [1371819, 41403351]},
...      index=['GOOG', 'APPL'])
>>> df
          2016      2015      2014
GOOG   1769950   1500923   1371819
APPL  30586265  40912316  41403351
```

```
>>> df.pct_change(axis='columns')
      2016      2015      2014
GOOG   NaN -0.151997 -0.086016
APPL   NaN  0.337604  0.012002
```

pipe (*func*, **args*, ***kwargs*)

Apply func(self, *args, **kwargs)

Parameters

func [function] function to apply to the NDFrame. `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 NDFrame.

args [iterable, optional] positional arguments passed into `func`.

kwargs [mapping, optional] a dictionary of keyword arguments passed into `func`.

Returns

object [the return type of `func`.]

See also:

`pandas.DataFrame.apply`, `pandas.DataFrame.applymap`, `pandas.Series.map`

Notes

Use `.pipe` when chaining together functions that expect Series, DataFrames or GroupBy objects. Instead of writing

```
>>> f(g(h(df), arg1=a), arg2=b, arg3=c)
```

You can write

```
>>> (df.pipe(h)
...    .pipe(g, arg1=a)
...    .pipe(f, arg2=b, arg3=c)
... )
```

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 `f` takes its data as `arg2`:

```
>>> (df.pipe(h)
...    .pipe(g, arg1=a)
...    .pipe((f, 'arg2'), arg1=a, arg3=c)
...  )
```

plot

alias of `pandas.plotting._core.SeriesPlotMethods`

pop(*item*)

Return item and drop from frame. Raise KeyError if not found.

Parameters

item [str] Column label to be popped

Returns

popped [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
```

pow(*other*, *level=None*, *fill_value=None*, *axis=0*)

Exponential power of series and other, element-wise (binary operator *pow*).

Equivalent to `series ** other`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters

other [Series or scalar value]

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

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

Returns

result [Series]

See also:

`Series.rpow`

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

prod(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, *min_count=0*, ***kwargs*)
Return the product of the values for the requested axis

Parameters

axis [{index (0)}]

skipna [boolean, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. 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.

New in version 0.22.0: Added with the default being 0. This means the sum of an all-NA or empty Series is 0, and the product of an all-NA or empty Series is 1.

Returns

prod [scalar or Series (if level specified)]

Examples

By default, the product of an empty or all-NA Series is `1`

```
>>> pd.Series([]).prod()
1.0
```

This can be controlled with the `min_count` parameter

```
>>> pd.Series([]).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
```

product (*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, *min_count=0*, ***kwargs*)

Return the product of the values for the requested axis

Parameters

axis [{index (0)}]

skipna [boolean, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. 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.

New in version 0.22.0: Added with the default being 0. This means the sum of an all-NA or empty Series is 0, and the product of an all-NA or empty Series is 1.

Returns

prod [scalar or Series (if level specified)]

Examples

By default, the product of an empty or all-NA Series is `1`

```
>>> pd.Series([]).prod()
1.0
```

This can be controlled with the `min_count` parameter

```
>>> pd.Series([]).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
```

ptp(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

Returns the difference between the maximum value and the minimum value in the object. This is the equivalent of the `numpy.ndarray` method `ptp`.

Parameters

axis [{index (0)}]

skipna [boolean, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

Returns

ptp [scalar or Series (if level specified)]

put(**args*, ***kwargs*)

Applies the *put* method to its *values* attribute if it has one.

See also:

`numpy.ndarray.put`

quantile(*q=0.5*, *interpolation='linear'*)

Return value at the given quantile, a la numpy.percentile.

Parameters

q [float or array-like, default 0.5 (50% quantile)] 0 <= q <= 1, the quantile(s) to compute

interpolation [{'linear', 'lower', 'higher', 'midpoint', 'nearest'}] New in version 0.18.0.

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.

Returns

quantile [float or Series] if `q` is an array, a Series will be returned where the index is `q` and the values are the quantiles.

See also:

`pandas.core.window.Rolling.quantile`

Examples

```
>>> s = Series([1, 2, 3, 4])
>>> s.quantile(.5)
2.5
>>> s.quantile([.25, .5, .75])
0.25    1.75
0.50    2.50
0.75    3.25
dtype: float64
```

radd(*other*, *level=None*, *fill_value=None*, *axis=0*)

Addition of series and other, element-wise (binary operator *radd*).

Equivalent to `other + series`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters

other [Series or scalar value]

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

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

Returns

result [Series]

See also:

`Series.add`

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
```

(continues on next page)

(continued from previous page)

```
e    NaN
dtype: float64
```

rank(*axis=0*, *method='average'*, *numeric_only=None*, *na_option='keep'*, *ascending=True*, *pct=False*)

Compute numerical data ranks (1 through n) along axis. 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

method [{'average', 'min', 'max', 'first', 'dense'}]

- average: average rank of group
- min: lowest rank in group
- max: highest rank in group
- first: ranks assigned in order they appear in the array
- dense: like 'min', but rank always increases by 1 between groups

numeric_only [boolean, default None] Include only float, int, boolean data. Valid only for DataFrame or Panel objects

na_option [{'keep', 'top', 'bottom'}]

- keep: leave NA values where they are
- top: smallest rank if ascending
- bottom: smallest rank if descending

ascending [boolean, default True] False for ranks by high (1) to low (N)

pct [boolean, default False] Computes percentage rank of data

Returns

ranks [same type as caller]

ravel(*order='C'*)

Return the flattened underlying data as an ndarray

See also:

`numpy.ndarray.ravel`

rdiv(*other*, *level=None*, *fill_value=None*, *axis=0*)

Floating division of series and other, element-wise (binary operator *rtruediv*).

Equivalent to `other / series`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters

other [Series or scalar value]

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

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

Returns

result [Series]

See also:

`Series.truediv`

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

real

reindex(*index=None*, ***kwargs*)

Conform Series to new index with optional filling logic, placing 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

index [array-like, optional (should be specified using keywords)] New labels / index to conform to. Preferably an Index object to avoid duplicating data

method [{None, 'backfill'/'bfill', 'pad'/'ffill', 'nearest'}, optional] method to use for filling holes in reindexed DataFrame. Please note: this is only applicable to DataFrames/Series with a monotonically increasing/decreasing index.

- 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 [boolean, 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.

New in version 0.21.0: (list-like tolerance)

Returns

reindexed [Series]

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
```

(continues on next page)

(continued from previous page)

```
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
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
2010-01-02     101
2010-01-03     NaN
2010-01-04     100
```

(continues on next page)

(continued from previous page)

```
2010-01-05      89
2010-01-06      88
```

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
2010-01-02     101
2010-01-03     NaN
2010-01-04     100
2010-01-05      89
2010-01-06      88
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 backpropagate 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
2009-12-30     100
2009-12-31     100
2010-01-01     100
2010-01-02     101
2010-01-03     NaN
2010-01-04     100
2010-01-05      89
2010-01-06      88
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.

reindex_axis (*labels*, *axis=0*, ***kwargs*)
Conform Series to new index with optional filling logic.

Deprecated since version 0.21.0: Use `Series.reindex` instead.

reindex_like (*other*, *method=None*, *copy=True*, *limit=None*, *tolerance=None*)
Return an object with matching indices to myself.

Parameters

other [Object]

method [string or None]

copy [boolean, default True]

limit [int, default None] Maximum number of consecutive labels to fill for inexact matches.

tolerance [optional] Maximum distance between labels of the other object and this object for inexact matches. Can be list-like.

New in version 0.21.0: (list-like tolerance)

Returns

reindexed [same as input]

Notes

Like calling s.reindex(index=other.index, columns=other.columns, method=...)

rename(*index=None*, ***kwargs*)
Alter Series index labels or name

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.

Alternatively, change `Series.name` with a scalar value.

See the user guide for more.

Parameters

index [scalar, hashable sequence, dict-like or function, optional] dict-like or functions are transformations to apply to the index. Scalar or hashable sequence-like will alter the `Series.name` attribute.

copy [boolean, default True] Also copy underlying data

inplace [boolean, default False] Whether to return a new Series. 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.

Returns

renamed [Series (new object)]

See also:

`pandas.Series.rename_axis`

Examples

```
>>> s = pd.Series([1, 2, 3])
>>> s
0    1
1    2
2    3
dtype: int64
>>> s.rename("my_name") # scalar, changes Series.name
0    1
1    2
2    3
```

(continues on next page)

(continued from previous page)

```
Name: my_name, dtype: int64
>>> s.rename(lambda x: x ** 2)  # function, changes labels
0    1
1    2
4    3
dtype: int64
>>> s.rename({1: 3, 2: 5})  # mapping, changes labels
0    1
3    2
5    3
dtype: int64
```

rename_axis(*mapper*, *axis=0*, *copy=True*, *inplace=False*)

Alter the name of the index or columns.

Parameters

mapper [scalar, list-like, optional] Value to set as the axis name attribute.

axis [{0 or 'index', 1 or 'columns'}, default 0] The index or the name of the axis.

copy [boolean, default True] Also copy underlying data.

inplace [boolean, default False] Modifies the object directly, instead of creating a new Series or DataFrame.

Returns

renamed [Series, DataFrame, or None] The same type as the caller or None if *inplace* is True.

See also:

pandas.Series.rename Alter Series index labels or name

pandas.DataFrame.rename Alter DataFrame index labels or name

pandas.Index.rename Set new names on index

Notes

Prior to version 0.21.0, `rename_axis` could also be used to change the axis *labels* by passing a mapping or scalar. This behavior is deprecated and will be removed in a future version. Use `rename` instead.

Examples

Series

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

DataFrame

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

```
>>> df.rename_axis("bar", axis="columns")
bar  A  B
0    1  4
1    2  5
2    3  6
```

reorder_levels(*order*)

Rearrange index levels using input order. May not drop or duplicate levels

Parameters

order [list of int representing new level order.] (reference level by number or key)

axis [where to reorder levels]

Returns

type of caller (new object)

repeat(***kwargs*)

Repeat elements of an Series. Refer to *numpy.ndarray.repeat* for more information about the *repeats* argument.

See also:

`numpy.ndarray.repeat`

replace(*to_replace=None*, *value=None*, *inplace=False*, *limit=None*, *regex=False*, *method='pad'*)

Replace values given in *to_replace* with *value*.

Values of the Series 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 *value* parameter should be *None*.
  - 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 *value* parameter should be `None` 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 [boolean, default False] If True, in place. Note: this will modify any other views on this object (e.g. a column from a DataFrame). Returns the caller if this is True.

limit [int, default None] Maximum size gap to forward or backward fill.

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', *None*}] The method to use when for replacement, when *to_replace* is a scalar, list or tuple and *value* is `None`.

Changed in version 0.23.0: Added to DataFrame.

Returns

Series Object after replacement.

Raises

AssertionError

- If *regex* is not a `bool` and *to_replace* is not `None`.

TypeError

- 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

`Series.where` Replace values based on boolean condition

`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([0, 1, 2, 3, 4])
>>> s.replace(0, 5)
0    5
1    1
2    2
3    3
4    4
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    0
1    3
2    3
3    3
4    4
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
```

(continues on next page)

(continued from previous page)

```
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
```

Note that when replacing multiple `bool` or `datetime64` objects, the data types in the *to_replace* parameter must match the data type of the value being replaced:

```
>>> df = pd.DataFrame({'A': [True, False, True],
...                    'B': [False, True, False]})
>>> df.replace({'a string': 'new value', True: False})  # raises
Traceback (most recent call last):
    ...
TypeError: Cannot compare types 'ndarray(dtype=bool)' and 'str'
```

This raises a `TypeError` because one of the `dict` keys is not of the correct type for replacement.

Compare the behavior of `s.replace({'a': None})` and `s.replace('a', None)` to understand the pecularities 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
```

(continues on next page)

(continued from previous page)

```
4    None
dtype: object
```

When `value=None` 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. The command `s.replace('a', None)` is actually equivalent to `s.replace(to_replace='a', value=None, method='pad')`:

```
>>> s.replace('a', None)
0    10
1    10
2    10
3     b
4     b
dtype: object
```

resample(*rule*, *how=None*, *axis=0*, *fill_method=None*, *closed=None*, *label=None*, *convention='start'*, *kind=None*, *loffset=None*, *limit=None*, *base=0*, *on=None*, *level=None*)

Convenience method for frequency conversion and resampling of time series. Object must have a datetime-like index (DatetimeIndex, PeriodIndex, or TimedeltaIndex), or pass datetime-like values to the on or level keyword.

Parameters

rule [string] the offset string or object representing target conversion

axis [int, optional, default 0]

closed [{'right', 'left'}] 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'}] 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'}] For PeriodIndex only, controls whether to use the start or end of *rule*

kind: {'timestamp', 'period'}, optional 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.

loffset [timedelta] Adjust the resampled time labels

base [int, default 0] For frequencies that evenly subdivide 1 day, the "origin" of the aggregated intervals. For example, for '5min' frequency, base could range from 0 through 4. Defaults to 0

on [string, optional] For a DataFrame, column to use instead of index for resampling. Column must be datetime-like.

New in version 0.19.0.

level [string or int, optional] For a MultiIndex, level (name or number) to use for resampling. Level must be datetime-like.

New in version 0.19.0.

Returns

Resampler object

See also:

groupby Group by mapping, function, label, or list of labels.

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
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
```

(continues on next page)

(continued from previous page)

```
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 `pad` method.

```
>>> series.resample('30S').pad()[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
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(array_like):
...     return np.sum(array_like)+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*.

```
>>> 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
```

Resample by month using 'start' *convention*. Values are assigned to the first month of the period.

```
>>> s.resample('M', convention='start').asfreq().head()
2012-01    1.0
2012-02    NaN
2012-03    NaN
2012-04    NaN
2012-05    NaN
Freq: M, dtype: float64
```

Resample by month using 'end' *convention*. Values are assigned to the last month of the period.

```
>>> s.resample('M', convention='end').asfreq()
2012-12    1.0
2013-01    NaN
2013-02    NaN
2013-03    NaN
2013-04    NaN
2013-05    NaN
2013-06    NaN
2013-07    NaN
2013-08    NaN
2013-09    NaN
2013-10    NaN
2013-11    NaN
2013-12    2.0
Freq: M, dtype: float64
```

For DataFrame objects, the keyword `on` can be used to specify the column instead of the index for resampling.

```
>>> df = pd.DataFrame(data=9*[range(4)], columns=['a', 'b', 'c', 'd'])
>>> df['time'] = pd.date_range('1/1/2000', periods=9, freq='T')
>>> df.resample('3T', on='time').sum()
                     a  b  c  d
time
2000-01-01 00:00:00  0  3  6  9
2000-01-01 00:03:00  0  3  6  9
2000-01-01 00:06:00  0  3  6  9
```

For a DataFrame with MultiIndex, the keyword `level` can be used to specify on level the resampling needs to take place.

```
>>> time = pd.date_range('1/1/2000', periods=5, freq='T')
>>> df2 = pd.DataFrame(data=10*[range(4)],
                       columns=['a', 'b', 'c', 'd'],
                       index=pd.MultiIndex.from_product([time, [1, 2]])
                       )
>>> df2.resample('3T', level=0).sum()
                     a  b   c   d
2000-01-01 00:00:00  0  6  12  18
2000-01-01 00:03:00  0  4   8  12
```

reset_index(*level=None*, *drop=False*, *name=None*, *inplace=False*)

Generate a new DataFrame or Series with the index reset.

This is useful when the index needs to be treated as a column, or when the index is meaningless and needs to be reset to the default before another operation.

Parameters

level [int, str, tuple, or list, default optional] For a Series with a MultiIndex, only remove the specified levels from the index. Removes all levels by default.

drop [bool, default False] Just reset the index, without inserting it as a column in the new DataFrame.

name [object, optional] The name to use for the column containing the original Series values. Uses `self.name` by default. This argument is ignored when *drop* is True.

inplace [bool, default False] Modify the Series in place (do not create a new object).

Returns

Series or DataFrame When *drop* is False (the default), a DataFrame is returned. The newly created columns will come first in the DataFrame, followed by the original Series values. When *drop* is True, a *Series* is returned. In either case, if `inplace=True`, no value is returned.

See also:

`DataFrame.reset_index` Analogous function for DataFrame.

Examples

```
>>> s = pd.Series([1, 2, 3, 4], name='foo',
...               index=pd.Index(['a', 'b', 'c', 'd'], name='idx'))
```

Generate a DataFrame with default index.

```
>>> s.reset_index()
  idx  foo
0   a    1
1   b    2
2   c    3
3   d    4
```

To specify the name of the new column use *name*.

```
>>> s.reset_index(name='values')
  idx  values
0   a       1
1   b       2
2   c       3
3   d       4
```

To generate a new Series with the default set *drop* to True.

```
>>> s.reset_index(drop=True)
0    1
1    2
2    3
3    4
Name: foo, dtype: int64
```

To update the Series in place, without generating a new one set *inplace* to True. Note that it also requires `drop=True`.

```
>>> s.reset_index(inplace=True, drop=True)
>>> s
0    1
1    2
2    3
3    4
Name: foo, dtype: int64
```

The *level* parameter is interesting for Series with a multi-level index.

```
>>> arrays = [np.array(['bar', 'bar', 'baz', 'baz']),
...           np.array(['one', 'two', 'one', 'two'])]
>>> s2 = pd.Series(
...     range(4), name='foo',
...     index=pd.MultiIndex.from_arrays(arrays,
...                                     names=['a', 'b']))
```

To remove a specific level from the Index, use *level*.

```
>>> s2.reset_index(level='a')
       a  foo
b
one  bar    0
two  bar    1
one  baz    2
two  baz    3
```

If *level* is not set, all levels are removed from the Index.

```
>>> s2.reset_index()
     a    b  foo
0  bar  one    0
1  bar  two    1
2  baz  one    2
3  baz  two    3
```

rfloordiv(*other*, *level=None*, *fill_value=None*, *axis=0*)

Integer division of series and other, element-wise (binary operator *rfloordiv*).

Equivalent to `other // series`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters

other [Series or scalar value]

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

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

Returns

result [Series]

See also:

`Series.floordiv`

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

rmod(*other*, *level=None*, *fill_value=None*, *axis=0*)

Modulo of series and other, element-wise (binary operator *rmod*).

Equivalent to `other % series`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters

other [Series or scalar value]

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

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

Returns

result [Series]

See also:

`Series.mod`

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
```

(continues on next page)

(continued from previous page)

```
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

rmul (*other*, *level=None*, *fill_value=None*, *axis=0*)

Multiplication of series and other, element-wise (binary operator *rmul*).

Equivalent to `other * series`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters

other [Series or scalar value]

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

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

Returns

result [Series]

See also:

`Series.mul`

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
```

(continues on next page)

(continued from previous page)

```
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

rolling(*window*, *min_periods=None*, *center=False*, *win_type=None*, *on=None*, *axis=0*, *closed=None*)

Provides rolling window calculations.

New in version 0.18.0.

Parameters

window [int, or offset] Size of the moving window. This is the number of observations used for calculating the statistic. Each window will be a fixed size.

If its an offset then this will be 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. This is new in 0.19.0

min_periods [int, default None] Minimum number of observations in window required to have a value (otherwise result is NA). For a window that is specified by an offset, this will default to 1.

center [boolean, default False] Set the labels at the center of the window.

win_type [string, default None] Provide a window type. If `None`, all points are evenly weighted. See the notes below for further information.

on [string, optional] For a DataFrame, column on which to calculate the rolling window, rather than the index

closed [string, default None] Make the interval closed on the 'right', 'left', 'both' or 'neither' endpoints. For offset-based windows, it defaults to 'right'. For fixed windows, defaults to 'both'. Remaining cases not implemented for fixed windows.

New in version 0.20.0.

axis [int or string, default 0]

Returns

a Window or Rolling sub-classed for the particular operation

See also:

expanding Provides expanding transformations.

ewm Provides exponential weighted functions

Notes

By default, the result is set to the right edge of the window. This can be changed to the center of the window by setting `center=True`.

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

The recognized win_types are:

- `boxcar`
- `triang`
- `blackman`
- `hamming`
- `bartlett`
- `parzen`
- `bohman`
- `blackmanharris`
- `nuttall`
- `barthann`
- `kaiser` (needs beta)
- `gaussian` (needs std)
- `general_gaussian` (needs power, width)
- `slepian` (needs width).

If `win_type=None` all points are evenly weighted. To learn more about different window types see scipy.signal window functions.

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
```

Rolling sum with a window length of 2, using the 'triang' window type.

```
>>> df.rolling(2, win_type='triang').sum()
     B
0  NaN
1  1.0
2  2.5
3  NaN
4  NaN
```

Rolling sum with a window length of 2, min_periods defaults to the window length.

```
>>> df.rolling(2).sum()
     B
0  NaN
1  1.0
2  3.0
```

(continues on next page)

(continued from previous page)

```
3  NaN
4  NaN
```

Same as above, but explicitly set the min_periods

```
>>> df.rolling(2, min_periods=1).sum()
     B
0  0.0
1  1.0
2  3.0
3  2.0
4  4.0
```

A ragged (meaning not-a-regular frequency), time-indexed DataFrame

```
>>> df = 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
                     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
```

Contrasting to an integer rolling window, this will roll a variable length window corresponding to the time period. The default for min_periods is 1.

```
>>> df.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
```

round(*decimals=0*, **args*, ***kwargs*)

Round each value in a Series to the given number of decimals.

Parameters

decimals [int] Number of decimal places to round to (default: 0). If decimals is negative, it specifies the number of positions to the left of the decimal point.

Returns

Series object

See also:

`numpy.around`, `DataFrame.round`

rpow(*other*, *level=None*, *fill_value=None*, *axis=0*)

Exponential power of series and other, element-wise (binary operator *rpow*).

Equivalent to `other ** series`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters

other [Series or scalar value]

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

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

Returns

result [Series]

See also:

`Series.pow`

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

rsub (*other*, *level=None*, *fill_value=None*, *axis=0*)
Subtraction of series and other, element-wise (binary operator *rsub*).

Equivalent to `other - series`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters

other [Series or scalar value]

fill_value [None or float value, default None (NaN)] Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before

computation. If data in both corresponding Series locations is missing the result will be missing

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

Returns

result [Series]

See also:

`Series.sub`

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

rtruediv(*other*, *level=None*, *fill_value=None*, *axis=0*)

Floating division of series and other, element-wise (binary operator *rtruediv*).

Equivalent to `other / series`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters

other [Series or scalar value]

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

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

Returns

result [Series]

See also:

`Series.truediv`

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

sample (*n=None*, *frac=None*, *replace=False*, *weights=None*, *random_state=None*, *axis=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 [boolean, optional] Sample with or without replacement. Default = False.

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. inf and -inf values not allowed.

random_state [int or numpy.random.RandomState, optional] Seed for the random number generator (if int), or numpy RandomState object.

axis [int or string, optional] Axis to sample. Accepts axis number or name. Default is stat axis for given data type (0 for Series and DataFrames, 1 for Panels).

Returns

A new object of same type as caller.

Examples

Generate an example `Series` and `DataFrame`:

```
>>> s = pd.Series(np.random.randn(50))
>>> s.head()
0   -0.038497
1    1.820773
2   -0.972766
3   -1.598270
4   -1.095526
dtype: float64
>>> df = pd.DataFrame(np.random.randn(50, 4), columns=list('ABCD'))
>>> df.head()
          A         B         C         D
0  0.016443 -2.318952 -0.566372 -1.028078
1 -1.051921  0.438836  0.658280 -0.175797
2 -1.243569 -0.364626 -0.215065  0.057736
3  1.768216  0.404512 -0.385604 -1.457834
4  1.072446 -1.137172  0.314194 -0.046661
```

Next extract a random sample from both of these objects...

3 random elements from the `Series`:

```
>>> s.sample(n=3)
27   -0.994689
55   -1.049016
67   -0.224565
dtype: float64
```

And a random 10% of the `DataFrame` with replacement:

```
>>> df.sample(frac=0.1, replace=True)
           A         B         C         D
35  1.981780  0.142106  1.817165 -0.290805
49 -1.336199 -0.448634 -0.789640  0.217116
40  0.823173 -0.078816  1.009536  1.015108
15  1.421154 -0.055301 -1.922594 -0.019696
6  -0.148339  0.832938  1.787600 -1.383767
```

You can use *random state* for reproducibility:

```
>>> df.sample(random_state=1)
A         B         C         D
37 -2.027662  0.103611  0.237496 -0.165867
43 -0.259323 -0.583426  1.516140 -0.479118
12 -1.686325 -0.579510  0.985195 -0.460286
8   1.167946  0.429082  1.215742 -1.636041
9   1.197475 -0.864188  1.554031 -1.505264
```

searchsorted(***kwargs*)

Find indices where elements should be inserted to maintain order.

Find the indices into a sorted Series *self* such that, if the corresponding elements in *value* were inserted before the indices, the order of *self* would be preserved.

Parameters

value [array_like] Values to insert into *self*.

side [{'left', 'right'}, optional] If 'left', the index of the first suitable location found is given. If 'right', return the last such index. If there is no suitable index, return either 0 or N (where N is the length of *self*).

sorter [1-D array_like, optional] Optional array of integer indices that sort *self* into ascending order. They are typically the result of `np.argsort`.

Returns

indices [array of ints] Array of insertion points with the same shape as *value*.

See also:

`numpy.searchsorted`

Notes

Binary search is used to find the required insertion points.

Examples

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

```
>>> x.searchsorted(4)
array([3])
```

```
>>> x.searchsorted([0, 4])
array([0, 3])
```

```
>>> x.searchsorted([1, 3], side='left')
array([0, 2])
```

```
>>> x.searchsorted([1, 3], side='right')
array([1, 3])
```

```
>>> x = pd.Categorical(['apple', 'bread', 'bread',
                        'cheese', 'milk'], ordered=True)
[apple, bread, bread, cheese, milk]
Categories (4, object): [apple < bread < cheese < milk]
```

```
>>> x.searchsorted('bread')
array([1])      # Note: an array, not a scalar
```

```
>>> x.searchsorted(['bread'], side='right')
array([3])
```

select(*crit*, *axis=0*)
Return data corresponding to axis labels matching criteria

Deprecated since version 0.21.0: Use df.loc[df.index.map(crit)] to select via labels

Parameters

crit [function] To be called on each index (label). Should return True or False

axis [int]

Returns

selection [type of caller]

sem(*axis=None*, *skipna=None*, *level=None*, *ddof=1*, *numeric_only=None*, ***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)}]

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar

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 [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

Returns

sem [scalar or Series (if level specified)]

set_axis(*labels*, *axis=0*, *inplace=None*)
Assign desired index to given axis.

Indexes for column or row labels can be changed by assigning a list-like or Index.

Changed in version 0.21.0: The signature is now *labels* and *axis*, consistent with the rest of pandas API. Previously, the *axis* and *labels* arguments were respectively the first and second positional arguments.

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, and 1 identifies the columns.

inplace [boolean, default None] Whether to return a new %(klass)s instance.

> **Warning:** `inplace=None` currently falls back to to True, but in a future version, will default to False. Use inplace=True explicitly rather than relying on the default.

Returns

renamed [%(klass)s or None] An object of same type as caller if inplace=False, None otherwise.

See also:

pandas.DataFrame.rename_axis Alter the name of the index or columns.

Examples

Series

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

```
>>> s.set_axis(['a', 'b', 'c'], axis=0, inplace=False)
a    1
b    2
c    3
dtype: int64
```

The original object is not modified.

```
>>> s
0    1
1    2
2    3
dtype: int64
```

DataFrame

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

Change the row labels.

```
>>> df.set_axis(['a', 'b', 'c'], axis='index', inplace=False)
   A  B
a  1  4
b  2  5
c  3  6
```

Change the column labels.

```
>>> df.set_axis(['I', 'II'], axis='columns', inplace=False)
   I  II
0  1   4
1  2   5
2  3   6
```

Now, update the labels inplace.

```
>>> df.set_axis(['i', 'ii'], axis='columns', inplace=True)
>>> df
```

(continues on next page)

(continued from previous page)

```
   i  ii
0  1   4
1  2   5
2  3   6
```

set_value(*label*, *value*, *takeable=False*)
Quickly set single value at passed label. If label is not contained, a new object is created with the label placed at the end of the result index.

Deprecated since version 0.21.0: Please use .at[] or .iat[] accessors.

Parameters

label [object] Partial indexing with MultiIndex not allowed

value [object] Scalar value

takeable [interpret the index as indexers, default False]

Returns

series [Series] If label is contained, will be reference to calling Series, otherwise a new object

shape
return a tuple of the shape of the underlying data

shift(*periods=1*, *freq=None*, *axis=0*)
Shift index by desired number of periods with an optional time freq

Parameters

periods [int] Number of periods to move, can be positive or negative

freq [DateOffset, timedelta, or time rule string, optional] Increment to use from the tseries module or time rule (e.g. 'EOM'). See Notes.

axis [{0 or 'index'}]

Returns

shifted [Series]

Notes

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.

size
return the number of elements in the underlying data

skew(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)
Return unbiased skew over requested axis Normalized by N-1

Parameters

axis [{index (0)}]

skipna [boolean, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

Returns

skew [scalar or Series (if level specified)]

slice_shift (*periods=1*, *axis=0*)

Equivalent to *shift* without copying data. The shifted data will not include the dropped periods and the shifted axis will be smaller than the original.

Parameters

periods [int] Number of periods to move, can be positive or negative

Returns

shifted [same type as caller]

Notes

While the *slice_shift* is faster than *shift*, you may pay for it later during alignment.

sort_index (*axis=0*, *level=None*, *ascending=True*, *inplace=False*, *kind='quicksort'*, *na_position='last'*, *sort_remaining=True*)

Sort Series by index labels.

Returns a new Series sorted by label if *inplace* argument is `False`, otherwise updates the original series and returns None.

Parameters

axis [int, default 0] Axis to direct sorting. This can only be 0 for Series.

level [int, optional] If not None, sort on values in specified index level(s).

ascending [bool, default true] Sort ascending vs. descending.

inplace [bool, default False] If True, perform operation in-place.

kind [{'quicksort', 'mergesort', 'heapsort'}, default 'quicksort'] Choice of sorting algorithm. See also `numpy.sort()` for more information. 'mergesort' is the only stable algorithm. For DataFrames, this option is only applied when sorting on a single column or label.

na_position [{'first', 'last'}, default 'last'] If 'first' puts NaNs at the beginning, '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.

Returns

pandas.Series The original Series sorted by the labels

See also:

DataFrame.sort_index Sort DataFrame by the index

DataFrame.sort_values Sort DataFrame by the value

Series.sort_values Sort Series by the value

Examples

```
>>> s = pd.Series(['a', 'b', 'c', 'd'], index=[3, 2, 1, 4])
>>> s.sort_index()
1    c
2    b
3    a
4    d
dtype: object
```

Sort Descending

```
>>> s.sort_index(ascending=False)
4    d
3    a
2    b
1    c
dtype: object
```

Sort Inplace

```
>>> s.sort_index(inplace=True)
>>> s
1    c
2    b
3    a
4    d
dtype: object
```

By default NaNs are put at the end, but use *na_position* to place them at the beginning

```
>>> s = pd.Series(['a', 'b', 'c', 'd'], index=[3, 2, 1, np.nan])
>>> s.sort_index(na_position='first')
NaN     d
 1.0    c
 2.0    b
 3.0    a
dtype: object
```

Specify index level to sort

```
>>> arrays = [np.array(['qux', 'qux', 'foo', 'foo',
...                     'baz', 'baz', 'bar', 'bar']),
...           np.array(['two', 'one', 'two', 'one',
...                     'two', 'one', 'two', 'one'])]
>>> s = pd.Series([1, 2, 3, 4, 5, 6, 7, 8], index=arrays)
>>> s.sort_index(level=1)
bar  one    8
baz  one    6
foo  one    4
qux  one    2
bar  two    7
baz  two    5
foo  two    3
qux  two    1
dtype: int64
```

Does not sort by remaining levels when sorting by levels

```
>>> s.sort_index(level=1, sort_remaining=False)
qux  one    2
foo  one    4
baz  one    6
bar  one    8
qux  two    1
foo  two    3
baz  two    5
bar  two    7
dtype: int64
```

sort_values(*axis=0*, *ascending=True*, *inplace=False*, *kind='quicksort'*, *na_position='last'*)

Sort by the values.

Sort a Series in ascending or descending order by some criterion.

Parameters

axis [{0 or 'index'}, default 0] Axis to direct sorting. The value 'index' is accepted for compatibility with DataFrame.sort_values.

ascending [bool, default True] If True, sort values in ascending order, otherwise descending.

inplace [bool, default False] If True, perform operation in-place.

kind [{'quicksort', 'mergesort' or 'heapsort'}, default 'quicksort'] Choice of sorting algorithm. See also `numpy.sort()` for more information. 'mergesort' is the only stable algorithm.

na_position [{'first' or 'last'}, default 'last'] Argument 'first' puts NaNs at the beginning, 'last' puts NaNs at the end.

Returns

Series Series ordered by values.

See also:

Series.sort_index Sort by the Series indices.

DataFrame.sort_values Sort DataFrame by the values along either axis.

DataFrame.sort_index Sort DataFrame by indices.

Examples

```
>>> s = pd.Series([np.nan, 1, 3, 10, 5])
>>> s
0     NaN
1     1.0
2     3.0
3     10.0
4     5.0
dtype: float64
```

Sort values ascending order (default behaviour)

```
>>> s.sort_values(ascending=True)
1     1.0
2     3.0
4     5.0
3    10.0
0     NaN
dtype: float64
```

Sort values descending order

```
>>> s.sort_values(ascending=False)
3    10.0
4     5.0
2     3.0
1     1.0
0     NaN
dtype: float64
```

Sort values inplace

```
>>> s.sort_values(ascending=False, inplace=True)
>>> s
3    10.0
4     5.0
2     3.0
1     1.0
0     NaN
dtype: float64
```

Sort values putting NAs first

```
>>> s.sort_values(na_position='first')
0     NaN
1     1.0
2     3.0
4     5.0
3    10.0
dtype: float64
```

Sort a series of strings

```
>>> s = pd.Series(['z', 'b', 'd', 'a', 'c'])
>>> s
0    z
1    b
2    d
3    a
4    c
dtype: object
```

```
>>> s.sort_values()
3    a
1    b
4    c
2    d
0    z
dtype: object
```

sortlevel(*level=0*, *ascending=True*, *sort_remaining=True*)
Sort Series with MultiIndex by chosen level. Data will be lexicographically sorted by the chosen level followed by the other levels (in order),

Deprecated since version 0.20.0: Use `Series.sort_index()`

Parameters

level [int or level name, default None]

ascending [bool, default True]

Returns

sorted [Series]

See also:

`Series.sort_index`

squeeze(*axis=None*)
Squeeze length 1 dimensions.

Parameters

axis [None, integer or string axis name, optional] The axis to squeeze if 1-sized.

New in version 0.20.0.

Returns

scalar if 1-sized, else original object

std(*axis=None*, *skipna=None*, *level=None*, *ddof=1*, *numeric_only=None*, ***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)}]

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar

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 [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

Returns

std [scalar or Series (if level specified)]

str
alias of `pandas.core.strings.StringMethods`

strides
return the strides of the underlying data

sub(*other*, *level=None*, *fill_value=None*, *axis=0*)

Subtraction of series and other, element-wise (binary operator *sub*).

Equivalent to `series - other`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters

other [Series or scalar value]

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

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

Returns

result [Series]

See also:

`Series.rsub`

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

subtract(*other*, *level=None*, *fill_value=None*, *axis=0*)

Subtraction of series and other, element-wise (binary operator *sub*).

Equivalent to `series - other`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters

other [Series or scalar value]

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

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

Returns

result [Series]

See also:

`Series.rsub`

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

sum(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, *min_count=0*, ***kwargs*)
Return the sum of the values for the requested axis

Parameters

axis [{index (0)}]

skipna [boolean, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. 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.

New in version 0.22.0: Added with the default being 0. This means the sum of an all-NA or empty Series is 0, and the product of an all-NA or empty Series is 1.

Returns

sum [scalar or Series (if level specified)]

Examples

By default, the sum of an empty or all-NA Series is `0`.

```
>>> pd.Series([]).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([]).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
```

swapaxes (*axis1*, *axis2*, *copy=True*)

Interchange axes and swap values axes appropriately

Returns

y [same as input]

swaplevel (*i=-2*, *j=-1*, *copy=True*)

Swap levels i and j in a MultiIndex

Parameters

i, j [int, string (can be mixed)] Level of index to be swapped. Can pass level name as string.

Returns

swapped [Series]

.. versionchanged:: 0.18.1 The indexes `i` and `j` are now optional, and default to the two innermost levels of the index.

tail (*n=5*)

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.

Parameters

n [int, default 5] Number of rows to select.

Returns

type of caller The last *n* rows of the caller object.

See also:

`pandas.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
```

`take`(*indices*, *axis=0*, *convert=None*, *is_copy=True*, ***kwargs*)

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.

convert [bool, default True] Whether to convert negative indices into positive ones. For example, `-1` would map to the `len(axis) - 1`. The conversions are similar to the behavior of indexing a regular Python list.

Deprecated since version 0.21.0: In the future, negative indices will always be converted.

is_copy [bool, default True] Whether to return a copy of the original object or not.

****kwargs** For compatibility with `numpy.take()`. Has no effect on the output.

Returns

taken [type of 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
```

to_clipboard(*excel=True*, *sep=None*, ***kwargs*)
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]

- True, use the provided separator, writing in a csv format for allowing easy pasting into excel.
- 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_table.

Notes

Requirements for your platform.

- Linux : *xclip*, or *xsel* (with *gtk* or *PyQt4* modules)
- Windows : none
- OS X : none

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 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
```

to_csv(*path=None*, *index=True*, *sep=', '*, *na_rep=''*, *float_format=None*, *header=False*, *index_label=None*, *mode='w'*, *encoding=None*, *compression=None*, *date_format=None*, *decimal='.'*)
Write Series to a comma-separated values (csv) file

Parameters

path [string or file handle, default None] File path or object, if None is provided the result is returned as a string.

na_rep [string, default ‘’] Missing data representation

float_format [string, default None] Format string for floating point numbers

header [boolean, default False] Write out series name

index [boolean, default True] Write row names (index)

index_label [string or sequence, 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 DataFrame uses MultiIndex.

mode [Python write mode, default ‘w’]

sep [character, default “,”] Field delimiter for the output file.

encoding [string, optional] a string representing the encoding to use if the contents are non-ascii, for python versions prior to 3

compression [string, optional] A string representing the compression to use in the output file. Allowed values are ‘gzip’, ‘bz2’, ‘zip’, ‘xz’. This input is only used when the first argument is a filename.

date_format: string, default None Format string for datetime objects.

decimal: string, default ‘.’ Character recognized as decimal separator. E.g. use ‘,’ for European data

to_dense()
Return dense representation of NDFrame (as opposed to sparse)

to_dict(*into=<type ’dict’>*)
Convert Series to {label -> value} dict or dict-like object.

Parameters

into [class, default dict] The collections.Mapping subclass to use as the return object. 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.

New in version 0.21.0.

Returns

value_dict [collections.Mapping]

Examples

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s.to_dict()
{0: 1, 1: 2, 2: 3, 3: 4}
>>> from collections import OrderedDict, defaultdict
>>> s.to_dict(OrderedDict)
OrderedDict([(0, 1), (1, 2), (2, 3), (3, 4)])
>>> dd = defaultdict(list)
>>> s.to_dict(dd)
defaultdict(<type 'list'>, {0: 1, 1: 2, 2: 3, 3: 4})
```

to_excel(*excel_writer*, *sheet_name='Sheet1'*, *na_rep=''*, *float_format=None*, *columns=None*, *header=True*, *index=True*, *index_label=None*, *startrow=0*, *startcol=0*, *engine=None*, *merge_cells=True*, *encoding=None*, *inf_rep='inf'*, *verbose=True*)

Write Series to an excel sheet

New in version 0.20.0.

Parameters

excel_writer [string or ExcelWriter object] File path or existing ExcelWriter

sheet_name [string, default 'Sheet1'] Name of sheet which will contain DataFrame

na_rep [string, default ''] Missing data representation

float_format [string, default None] Format string for floating point numbers

columns [sequence, optional] Columns to write

header [boolean or list of string, 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 [boolean, default True] Write row names (index)

index_label [string or sequence, 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 DataFrame uses MultiIndex.

startrow : upper left cell row to dump data frame

startcol : upper left cell column to dump data frame

engine [string, default None] write engine to use - you can also set this via the options `io.excel.xlsx.writer`, `io.excel.xls.writer`, and `io.excel.xlsm.writer`.

merge_cells [boolean, default True] Write MultiIndex and Hierarchical Rows as merged cells.

encoding: string, default None encoding of the resulting excel file. Only necessary for xlwt, other writers support unicode natively.

inf_rep [string, default 'inf'] Representation for infinity (there is no native representation for infinity in Excel)

freeze_panes [tuple of integer (length 2), default None] Specifies the one-based bottommost row and rightmost column that is to be frozen

New in version 0.20.0.

Notes

If passing an existing ExcelWriter object, then the sheet will be added to the existing workbook. This can be used to save different DataFrames to one workbook:

```
>>> writer = pd.ExcelWriter('output.xlsx')
>>> df1.to_excel(writer,'Sheet1')
>>> df2.to_excel(writer,'Sheet2')
>>> writer.save()
```

For compatibility with to_csv, to_excel serializes lists and dicts to strings before writing.

to_frame(*name=None*)
Convert Series to DataFrame

Parameters

name [object, default None] The passed name should substitute for the series name (if it has one).

Returns

data_frame [DataFrame]

to_hdf(*path_or_buf*, *key*, ***kwargs*)
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.

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.

format [{'fixed', 'table'}, 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.

append [bool, default False] For Table formats, append the input data to the existing.

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 io.hdf5-query-data-columns. Applicable only to format='table'.

complevel [{0-9}, optional] Specifies a compression level for data. A value of 0 disables compression.

complib [{'zlib', 'lzo', 'bzip2', 'blosc'}, default 'zlib'] Specifies the compression library to be used. As of v0.20.2 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.

fletcher32 [bool, default False] If applying compression use the fletcher32 checksum.

dropna [bool, default False] If true, ALL nan rows will not be written to store.

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.

See also:

`DataFrame.read_hdf` Read from HDF file.

`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
```

Deleting file with data:

```
>>> import os
>>> os.remove('data.h5')
```

`to_json`(*path_or_buf=None*, *orient=None*, *date_format=None*, *double_precision=10*, *force_ascii=True*, *date_unit='ms'*, *default_handler=None*, *lines=False*, *compression=None*, *index=True*)

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 [string or file handle, optional] File path or object. If not specified, the result is returned as a string.

orient [string] Indication of expected JSON string format.

- Series
  - default is 'index'
  - allowed values are: {'split','records','index'}
- DataFrame
  - default is 'columns'
  - allowed values are: {'split','records','index','columns','values'}
- 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, and the data component is like `orient='records'`.

    Changed in version 0.20.0.

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.

force_ascii [boolean, default True] Force encoded string to be ASCII.

date_unit [string, 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 [boolean, default False] If 'orient' is 'records' write out line delimited json format. Will throw ValueError if incorrect 'orient' since others are not list like.

New in version 0.19.0.

compression [{None, 'gzip', 'bz2', 'zip', 'xz'}] A string representing the compression to use in the output file, only used when the first argument is a filename.

New in version 0.21.0.

index [boolean, default True] Whether to include the index values in the JSON string. Not including the index (`index=False`) is only supported when orient is 'split' or 'table'.

New in version 0.23.0.

See also:

`pandas.read_json`

Examples

```
>>> df = pd.DataFrame([['a', 'b'], ['c', 'd']],
...                   index=['row 1', 'row 2'],
...                   columns=['col 1', 'col 2'])
>>> df.to_json(orient='split')
'{"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.

```
>>> df.to_json(orient='records')
'[{"col 1":"a","col 2":"b"},{"col 1":"c","col 2":"d"}]'
```

Encoding/decoding a Dataframe using `'index'` formatted JSON:

```
>>> df.to_json(orient='index')
'{"row 1":{"col 1":"a","col 2":"b"},"row 2":{"col 1":"c","col 2":"d"}}'
```

Encoding/decoding a Dataframe using `'columns'` formatted JSON:

```
>>> df.to_json(orient='columns')
'{"col 1":{"row 1":"a","row 2":"c"},"col 2":{"row 1":"b","row 2":"d"}}'
```

Encoding/decoding a Dataframe using `'values'` formatted JSON:

```
>>> df.to_json(orient='values')
'[["a","b"],["c","d"]]'
```

Encoding with Table Schema

```
>>> df.to_json(orient='table')
'{"schema": {"fields": [{"name": "index", "type": "string"},
                        {"name": "col 1", "type": "string"},
                        {"name": "col 2", "type": "string"}],
             "primaryKey": "index",
             "pandas_version": "0.20.0"},
  "data": [{"index": "row 1", "col 1": "a", "col 2": "b"},
           {"index": "row 2", "col 1": "c", "col 2": "d"}]}'
```

to_latex(*buf=None*, *columns=None*, *col_space=None*, *header=True*, *index=True*, *na_rep='NaN'*, *formatters=None*, *float_format=None*, *sparsify=None*, *index_names=True*, *bold_rows=False*, *column_format=None*, *longtable=None*, *escape=None*, *encoding=None*, *decimal='.'*, *multicolumn=None*, *multicolumn_format=None*, *multirow=None*)

Render an object to a tabular environment table. You can splice this into a LaTeX document. Requires \usepackage{booktabs}.

Changed in version 0.20.2: Added to Series

to_latex-specific options:

bold_rows [boolean, default False] Make the row labels bold in the output

column_format [str, default None] The columns format as specified in LaTeX table format e.g 'rcl' for 3 columns

longtable [boolean, default will be read from the pandas config module] Default: False. Use a longtable environment instead of tabular. Requires adding a \usepackage{longtable} to your LaTeX preamble.

escape [boolean, default will be read from the pandas config module] Default: True. When set to False prevents from escaping latex special characters in column names.

encoding [str, default None] A string representing the encoding to use in the output file, defaults to 'ascii' on Python 2 and 'utf-8' on Python 3.

decimal [string, default '.'] Character recognized as decimal separator, e.g. ',' in Europe.

New in version 0.18.0.

multicolumn [boolean, default True] Use multicolumn to enhance MultiIndex columns. The default will be read from the config module.

New in version 0.20.0.

multicolumn_format [str, default 'l'] The alignment for multicolumns, similar to *column_format* The default will be read from the config module.

New in version 0.20.0.

multirow [boolean, default False] 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.

New in version 0.20.0.

to_mol2(*filepath_or_buffer=None*)

to_msgpack(*path_or_buf=None*, *encoding='utf-8'*, ***kwargs*)
msgpack (serialize) object to input file path

THIS IS AN EXPERIMENTAL LIBRARY and the storage format may not be stable until a future release.

Parameters

path [string File path, buffer-like, or None] if None, return generated string

append [boolean whether to append to an existing msgpack] (default is False)

compress [type of compressor (zlib or blosc), default to None (no] compression)

to_period(*freq=None*, *copy=True*)
Convert Series from DatetimeIndex to PeriodIndex with desired frequency (inferred from index if not passed)

Parameters

freq [string, default]

Returns

ts [Series with PeriodIndex]

to_pickle(*path*, *compression='infer'*, *protocol=2*)
Pickle (serialize) object to file.

Parameters

path [str] File path where the pickled object will be stored.

compression [{'infer', 'gzip', 'bz2', 'zip', 'xz', None}, default 'infer'] A string representing the compression to use in the output file. By default, infers from the file extension in specified path.

New in version 0.20.0.

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 for this parameter depend on the version of Python. For Python 2.x, possible values are 0, 1, 2. For Python>=3.0, 3 is a valid value. For Python >= 3.4, 4 is a valid value. A negative value for the protocol parameter is equivalent to setting its value to HIGHEST_PROTOCOL.

New in version 0.21.0.

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
```

```
>>> import os
>>> os.remove("./dummy.pkl")
```

to_sdf(*filepath_or_buffer=None*)

to_smiles(*filepath_or_buffer=None*)

to_sparse(*kind='block'*, *fill_value=None*)
Convert Series to SparseSeries

Parameters

kind [{'block', 'integer'}]

fill_value [float, defaults to NaN (missing)]

Returns

sp [SparseSeries]

to_sql(*name*, *con*, *schema=None*, *if_exists='fail'*, *index=True*, *index_label=None*, *chunksize=None*, *dtype=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 [string] Name of SQL table.

con [sqlalchemy.engine.Engine or sqlite3.Connection] Using SQLAlchemy makes it possible to use any DB supported by that library. Legacy support is provided for sqlite3.Connection objects.

schema [string, 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 [boolean, default True] Write DataFrame index as a column. Uses *index_label* as the column name in the table.

index_label [string 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] Rows will be written in batches of this size at a time. By default, all rows will be written at once.

dtype [dict, optional] Specifying the datatype for columns. The keys should be the column names and the values should be the SQLAlchemy types or strings for the sqlite3 legacy mode.

Raises

ValueError When the table already exists and *if_exists* is 'fail' (the default).

See also:

pandas.read_sql read a DataFrame from a table

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('users', con=engine)
>>> engine.execute("SELECT * FROM users").fetchall()
[(0, 'User 1'), (1, 'User 2'), (2, 'User 3')]
```

```
>>> df1 = pd.DataFrame({'name' : ['User 4', 'User 5']})
>>> df1.to_sql('users', con=engine, if_exists='append')
>>> engine.execute("SELECT * FROM users").fetchall()
[(0, 'User 1'), (1, 'User 2'), (2, 'User 3'),
 (0, 'User 4'), (1, 'User 5')]
```

Overwrite the table with just `df1`.

```
>>> df1.to_sql('users', con=engine, if_exists='replace',
...            index_label='id')
>>> engine.execute("SELECT * FROM users").fetchall()
[(0, 'User 4'), (1, 'User 5')]
```

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('integers', con=engine, index=False,
...           dtype={"A": Integer()})
```

```
>>> engine.execute("SELECT * FROM integers").fetchall()
[(1,), (None,), (2,)]
```

to_string(*buf=None*, *na_rep='NaN'*, *float_format=None*, *header=True*, *index=True*, *length=False*, *dtype=False*, *name=False*, *max_rows=None*)

Render a string representation of the Series

Parameters

buf [StringIO-like, optional] buffer to write to

na_rep [string, optional] string representation of NAN to use, default 'NaN'

float_format [one-parameter function, optional] formatter function to apply to columns' elements if they are floats default None

header: boolean, default True Add the Series header (index name)

index [bool, optional] Add index (row) labels, default True

length [boolean, default False] Add the Series length

dtype [boolean, default False] Add the Series dtype

name [boolean, default False] Add the Series name if not None

max_rows [int, optional] Maximum number of rows to show before truncating. If None, show all.

Returns

formatted [string (if not buffer passed)]

to_timestamp (*freq=None*, *how='start'*, *copy=True*)
Cast to datetimeindex of timestamps, at *beginning* of period

Parameters

freq [string, default frequency of PeriodIndex] Desired frequency

how [{'s', 'e', 'start', 'end'}] Convention for converting period to timestamp; start of period vs. end

Returns

ts [Series with DatetimeIndex]

to_xarray()
Return an xarray object from the pandas object.

Returns

a DataArray for a Series

a Dataset for a DataFrame

a DataArray for higher dims

Notes

See the xarray docs

Examples

```
>>> df = pd.DataFrame({'A' : [1, 1, 2],
                       'B' : ['foo', 'bar', 'foo'],
                       'C' : np.arange(4.,7)})
>>> df
   A    B    C
0  1  foo  4.0
1  1  bar  5.0
2  2  foo  6.0
```

```
>>> df.to_xarray()
<xarray.Dataset>
Dimensions:  (index: 3)
Coordinates:
  * index    (index) int64 0 1 2
```

(continues on next page)

(continued from previous page)

```
Data variables:
    A        (index) int64 1 1 2
    B        (index) object 'foo' 'bar' 'foo'
    C        (index) float64 4.0 5.0 6.0
```

```
>>> df = pd.DataFrame({'A' : [1, 1, 2],
                       'B' : ['foo', 'bar', 'foo'],
                       'C' : np.arange(4.,7)}
                     ).set_index(['B','A'])
>>> df
         C
B   A
foo 1  4.0
bar 1  5.0
foo 2  6.0
```

```
>>> df.to_xarray()
<xarray.Dataset>
Dimensions:  (A: 2, B: 2)
Coordinates:
  * B        (B) object 'bar' 'foo'
  * A        (A) int64 1 2
Data variables:
    C        (B, A) float64 5.0 nan 4.0 6.0
```

```
>>> p = pd.Panel(np.arange(24).reshape(4,3,2),
                 items=list('ABCD'),
                 major_axis=pd.date_range('20130101', periods=3),
                 minor_axis=['first', 'second'])
>>> p
<class 'pandas.core.panel.Panel'>
Dimensions: 4 (items) x 3 (major_axis) x 2 (minor_axis)
Items axis: A to D
Major_axis axis: 2013-01-01 00:00:00 to 2013-01-03 00:00:00
Minor_axis axis: first to second
```

```
>>> p.to_xarray()
<xarray.DataArray (items: 4, major_axis: 3, minor_axis: 2)>
array([[[ 0,  1],
        [ 2,  3],
        [ 4,  5]],
       [[ 6,  7],
        [ 8,  9],
        [10, 11]],
       [[12, 13],
        [14, 15],
        [16, 17]],
       [[18, 19],
        [20, 21],
        [22, 23]]])
Coordinates:
  * items       (items) object 'A' 'B' 'C' 'D'
  * major_axis  (major_axis) datetime64[ns] 2013-01-01 2013-01-02 2013-01-03␣
↪ # noqa
  * minor_axis  (minor_axis) object 'first' 'second'
```

tolist()
Return a list of the values.

These are each a scalar type, which is a Python scalar (for str, int, float) or a pandas scalar (for Timestamp/Timedelta/Interval/Period)

See also:

`numpy.ndarray.tolist`

transform(*func*, **args*, ***kwargs*)
Call function producing a like-indexed NDFrame and return a NDFrame with the transformed values

New in version 0.20.0.

Parameters

func [callable, string, dictionary, or list of string/callables] To apply to column

Accepted Combinations are:

- string function name
- function
- list of functions
- dict of column names -> functions (or list of functions)

Returns

transformed [NDFrame]

See also:

`pandas.NDFrame.aggregate`, `pandas.NDFrame.apply`

Examples

```
>>> df = pd.DataFrame(np.random.randn(10, 3), columns=['A', 'B', 'C'],
...                   index=pd.date_range('1/1/2000', periods=10))
df.iloc[3:7] = np.nan
```

```
>>> df.transform(lambda x: (x - x.mean()) / x.std())
                   A         B         C
2000-01-01  0.579457  1.236184  0.123424
2000-01-02  0.370357 -0.605875 -1.231325
2000-01-03  1.455756 -0.277446  0.288967
2000-01-04       NaN       NaN       NaN
2000-01-05       NaN       NaN       NaN
2000-01-06       NaN       NaN       NaN
2000-01-07       NaN       NaN       NaN
2000-01-08 -0.498658  1.274522  1.642524
2000-01-09 -0.540524 -1.012676 -0.828968
2000-01-10 -1.366388 -0.614710  0.005378
```

transpose(**args*, ***kwargs*)
return the transpose, which is by definition self

truediv(*other*, *level=None*, *fill_value=None*, *axis=0*)
Floating division of series and other, element-wise (binary operator *truediv*).

Equivalent to `series / other`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters

other [Series or scalar value]

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

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

Returns

result [Series]

See also:

`Series.rtruediv`

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

truncate(*before=None*, *after=None*, *axis=None*, *copy=True*)

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, string, int] Truncate all rows before this index value.

after [date, string, 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.

copy [boolean, 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
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
```

tshift (*periods=1*, *freq=None*, *axis=0*)

Shift the time index, using the index's frequency if available.

Parameters

periods [int] Number of periods to move, can be positive or negative

freq [DateOffset, timedelta, or time rule string, default None] Increment to use from the tseries module or time rule (e.g. 'EOM')

axis [int or basestring] Corresponds to the axis that contains the Index

Returns

shifted [NDFrame]

Notes

If freq is not specified then tries to use the freq or inferred_freq attributes of the index. If neither of those attributes exist, a ValueError is thrown

tz_convert(*tz*, *axis=0*, *level=None*, *copy=True*)
Convert tz-aware axis to target time zone.

Parameters

tz [string or pytz.timezone object]

axis [the axis to convert]

level [int, str, default None] If axis ia a MultiIndex, convert a specific level. Otherwise must be None

copy [boolean, default True] Also make a copy of the underlying data

Raises

TypeError If the axis is tz-naive.

tz_localize(*tz*, *axis=0*, *level=None*, *copy=True*, *ambiguous='raise'*)
Localize tz-naive TimeSeries to target time zone.

Parameters

tz [string or pytz.timezone object]

axis [the axis to localize]

level [int, str, default None] If axis ia a MultiIndex, localize a specific level. Otherwise must be None

copy [boolean, default True] Also make a copy of the underlying data

ambiguous ['infer', bool-ndarray, 'NaT', default 'raise']

- '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

Raises

TypeError If the TimeSeries is tz-aware and tz is not None.

unique()
Return unique values of Series object.

Uniques are returned in order of appearance. Hash table-based unique, therefore does NOT sort.

Returns

ndarray or Categorical The unique values returned as a NumPy array. In case of categorical data type, returned as a Categorical.

See also:

pandas.unique top-level unique method for any 1-d array-like object.

Index.unique return Index with unique values from an Index object.

Examples

```
>>> pd.Series([2, 1, 3, 3], name='A').unique()
array([2, 1, 3])
```

```
>>> pd.Series([pd.Timestamp('2016-01-01') for _ in range(3)]).unique()
array(['2016-01-01T00:00:00.000000000'], dtype='datetime64[ns]')
```

```
>>> pd.Series([pd.Timestamp('2016-01-01', tz='US/Eastern')
...             for _ in range(3)]).unique()
array([Timestamp('2016-01-01 00:00:00-0500', tz='US/Eastern')],
      dtype=object)
```

An unordered Categorical will return categories in the order of appearance.

```
>>> pd.Series(pd.Categorical(list('baabc'))).unique()
[b, a, c]
Categories (3, object): [b, a, c]
```

An ordered Categorical preserves the category ordering.

```
>>> pd.Series(pd.Categorical(list('baabc'), categories=list('abc'),
...                          ordered=True)).unique()
[b, a, c]
Categories (3, object): [a < b < c]
```

unstack (*level=-1*, *fill_value=None*)

Unstack, a.k.a. pivot, Series with MultiIndex to produce DataFrame. The level involved will automatically get sorted.

Parameters

level [int, string, or list of these, default last level] Level(s) to unstack, can pass level name

fill_value [replace NaN with this value if the unstack produces] missing values

New in version 0.18.0.

Returns

unstacked [DataFrame]

Examples

```
>>> s = pd.Series([1, 2, 3, 4],
...     index=pd.MultiIndex.from_product([['one', 'two'], ['a', 'b']]))
>>> s
one  a    1
     b    2
two  a    3
     b    4
dtype: int64
```

```
>>> s.unstack(level=-1)
     a  b
one  1  2
two  3  4
```

```
>>> s.unstack(level=0)
   one  two
a    1    3
b    2    4
```

update(*other*)

Modify Series in place using non-NA values from passed Series. Aligns on index

Parameters

other [Series]

Examples

```
>>> s = pd.Series([1, 2, 3])
>>> s.update(pd.Series([4, 5, 6]))
>>> s
0    4
1    5
2    6
dtype: int64
```

```
>>> s = pd.Series(['a', 'b', 'c'])
>>> s.update(pd.Series(['d', 'e'], index=[0, 2]))
>>> s
0    d
1    b
2    e
dtype: object
```

```
>>> s = pd.Series([1, 2, 3])
>>> s.update(pd.Series([4, 5, 6, 7, 8]))
>>> s
0    4
1    5
2    6
dtype: int64
```

If `other` contains NaNs the corresponding values are not updated in the original Series.

```
>>> s = pd.Series([1, 2, 3])
>>> s.update(pd.Series([4, np.nan, 6]))
>>> s
0    4
1    2
2    6
dtype: int64
```

valid(*inplace=False*, ***kwargs*)

Return Series without null values.

Deprecated since version 0.23.0: Use `Series.dropna()` instead.

value_counts(*normalize=False*, *sort=True*, *ascending=False*, *bins=None*, *dropna=True*)
Returns object containing counts of unique values.

The resulting object will be in descending order so that the first element is the most frequently-occurring element. Excludes NA values by default.

Parameters

normalize [boolean, default False] If True then the object returned will contain the relative frequencies of the unique values.

sort [boolean, default True] Sort by values

ascending [boolean, default False] Sort in ascending order

bins [integer, optional] Rather than count values, group them into half-open bins, a convenience for pd.cut, only works with numeric data

dropna [boolean, default True] Don't include counts of NaN.

Returns

counts [Series]

values
Return Series as ndarray or ndarray-like depending on the dtype

Returns

arr [numpy.ndarray or ndarray-like]

Examples

```
>>> pd.Series([1, 2, 3]).values
array([1, 2, 3])
```

```
>>> pd.Series(list('aabc')).values
array(['a', 'a', 'b', 'c'], dtype=object)
```

```
>>> pd.Series(list('aabc')).astype('category').values
[a, a, b, c]
Categories (3, object): [a, b, c]
```

Timezone aware datetime data is converted to UTC:

```
>>> pd.Series(pd.date_range('20130101', periods=3,
...                         tz='US/Eastern')).values
array(['2013-01-01T05:00:00.000000000',
       '2013-01-02T05:00:00.000000000',
       '2013-01-03T05:00:00.000000000'], dtype='datetime64[ns]')
```

var(*axis=None*, *skipna=None*, *level=None*, *ddof=1*, *numeric_only=None*, ***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)}]

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar

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 [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

Returns

var [scalar or Series (if level specified)]

view(*dtype=None*)

Create a new view of the Series.

This function will return a new Series with a view of the same underlying values in memory, optionally reinterpreted with a new data type. The new data type must preserve the same size in bytes as to not cause index misalignment.

Parameters

dtype [data type] Data type object or one of their string representations.

Returns

Series A new Series object as a view of the same data in memory.

See also:

numpy.ndarray.view Equivalent numpy function to create a new view of the same data in memory.

Notes

Series are instantiated with `dtype=float64` by default. While `numpy.ndarray.view()` will return a view with the same data type as the original array, `Series.view()` (without specified dtype) will try using `float64` and may fail if the original data type size in bytes is not the same.

Examples

```
>>> s = pd.Series([-2, -1, 0, 1, 2], dtype='int8')
>>> s
0   -2
1   -1
2    0
3    1
4    2
dtype: int8
```

The 8 bit signed integer representation of *-1* is *0b11111111*, but the same bytes represent 255 if read as an 8 bit unsigned integer:

```
>>> us = s.view('uint8')
>>> us
0    254
1    255
2      0
3      1
4      2
dtype: uint8
```

The views share the same underlying values:

```
>>> us[0] = 128
>>> s
0   -128
1     -1
2      0
3      1
4      2
dtype: int8
```

where(*cond*, *other=nan*, *inplace=False*, *axis=None*, *level=None*, *errors='raise'*, *try_cast=False*, *raise_on_error=None*)

Return an object of same shape as self and whose corresponding entries are from self where *cond* is True and otherwise are from *other*.

Parameters

cond [boolean NDFrame, 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 NDFrame and should return boolean NDFrame or array. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1: A callable can be used as cond.

other [scalar, NDFrame, or callable] Entries where *cond* is False are replaced with corresponding value from *other*. If other is callable, it is computed on the NDFrame and should return scalar or NDFrame. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1: A callable can be used as other.

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

axis [alignment axis if needed, default None]

level [alignment level if needed, default None]

errors [str, {'raise', 'ignore'}, default 'raise']

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

Note that currently this parameter won't affect the results and will always coerce to a suitable dtype.

try_cast [boolean, default False] try to cast the result back to the input type (if possible),

raise_on_error [boolean, default True] Whether to raise on invalid data types (e.g. trying to where on strings)

Deprecated since version 0.21.0.

Returns

wh [same type as caller]

See also:

`DataFrame.mask()`

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.

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.

Examples

```
>>> s = pd.Series(range(5))
>>> s.where(s > 0)
0    NaN
1    1.0
2    2.0
3    3.0
4    4.0
```

```
>>> s.mask(s > 0)
0    0.0
1    NaN
2    NaN
3    NaN
4    NaN
```

```
>>> s.where(s > 1, 10)
0    10.0
1    10.0
2    2.0
3    3.0
4    4.0
```

```
>>> df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])
>>> 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
```

(continues on next page)

(continued from previous page)

```
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
```

xs (*key*, *axis=0*, *level=None*, *drop_level=True*)

Returns a cross-section (row(s) or column(s)) from the Series/DataFrame. Defaults to cross-section on the rows (axis=0).

Parameters

key [object] Some label contained in the index, or partially in a MultiIndex

axis [int, 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 [boolean, default True] If False, returns object with same levels as self.

Returns

xs [Series or DataFrame]

Notes

xs is only for getting, not setting 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

```
>>> df
   A  B  C
a  4  5  2
b  4  0  9
c  9  7  3
>>> df.xs('a')
A    4
B    5
C    2
Name: a
>>> df.xs('C', axis=1)
a    2
b    9
c    3
Name: C
```

```
>>> df
                     A  B  C  D
first second third
bar   one    1       4  1  8  9
      two    1       7  5  5  0
baz   one    1       6  6  8  0
      three  2       5  3  5  3
>>> df.xs(('baz', 'three'))
       A  B  C  D
third
2      5  3  5  3
>>> df.xs('one', level=1)
             A  B  C  D
first third
bar   1      4  1  8  9
baz   1      6  6  8  0
>>> df.xs(('baz', 2), level=[0, 'third'])
        A  B  C  D
second
three   5  3  5  3
```

oddt.pandas.**read_csv**(**args*, ***kwargs*)

TODO: Support Chunks

oddt.pandas.**read_mol2**(*filepath_or_buffer=None*, *usecols=None*, *molecule_column='mol'*, *molecule_name_column='mol_name'*, *smiles_column=None*, *skip_bad_mols=False*, *chunksize=None*, ***kwargs*)

Read Mol2 multi molecular file to ChemDataFrame. UCSF Dock 6 comments style is supported, i.e. *#### var_name: value* before molecular block.

New in version 0.3.

Parameters

filepath_or_buffer [string or None] File path

usecols [list or None, optional (default=None)] A list of columns to read from file. If None then all available fields are read.

molecule_column [string or None, optional (default='mol')] Name of molecule column. If None the molecules will be skipped and the reading will be speed up significantly.

molecule_name_column [string or None, optional (default='mol_name')] Column name which will contain molecules' title/name. Column is skipped when set to None.

smiles_column [string or None, optional (default=None)] Column name containg molecules' SMILES, by default it is disabled.

skip_bad_mols [bool, optional (default=False)] Switch to skip empty (bad) molecules. Useful for RDKit, which Returns None if molecule can not sanitize.

chunksize [int or None, optional (default=None)] Size of chunk to return. If set to None whole set is returned.

Returns

result : A *ChemDataFrame* containg all molecules if *chunksize* is None or genrerator of *ChemDataFrame* with *chunksize* molecules.

oddt.pandas.**read_sdf**(*filepath_or_buffer=None*, *usecols=None*, *molecule_column='mol'*, *molecule_name_column='mol_name'*, *smiles_column=None*, *skip_bad_mols=False*, *chunksize=None*, ***kwargs*)

Read SDF/MDL multi molecular file to ChemDataFrame

New in version 0.3.

Parameters

filepath_or_buffer [string or None] File path

usecols [list or None, optional (default=None)] A list of columns to read from file. If None then all available fields are read.

molecule_column [string or None, optional (default='mol')] Name of molecule column. If None the molecules will be skipped and the reading will be speed up significantly.

molecule_name_column [string or None, optional (default='mol_name')] Column name which will contain molecules' title/name. Column is skipped when set to None.

smiles_column [string or None, optional (default=None)] Column name containg molecules' SMILES, by default it is disabled.

skip_bad_mols [bool, optional (default=False)] Switch to skip empty (bad) molecules. Useful for RDKit, which Returns None if molecule can not sanitize.

chunksize [int or None, optional (default=None)] Size of chunk to return. If set to None whole set is returned.

Returns

result : A *ChemDataFrame* containg all molecules if *chunksize* is None or genrerator of *ChemDataFrame* with *chunksize* molecules.

5.1.8 oddt.shape module

oddt.shape.**common_usr**(*molecule*, *ctd=None*, *cst=None*, *fct=None*, *ftf=None*, *atoms_type=None*)

Function used in USR and USRCAT function

Parameters

molecule [oddt.toolkit.Molecule] Molecule to compute USR shape descriptor

ctd [numpy array or None (default = None)] Coordinates of the molecular centroid If 'None', the point is calculated

cst [numpy array or None (default = None)] Coordinates of the closest atom to the molecular centroid If 'None', the point is calculated

fct [numpy array or None (default = None)] Coordinates of the farthest atom to the molecular centroid If 'None', the point is calculated

ftf [numpy array or None (default = None)] Coordinates of the farthest atom to the farthest atom to the molecular centroid If 'None', the point is calculated

atoms_type [str or None (default None)] Type of atoms to be selected from atom_dict If 'None', all atoms are used to calculate shape descriptor

Returns

shape_descriptor [numpy array, shape = (12)] Array describing shape of molecule

oddt.shape.**electroshape**(*mol*)

Computes shape descriptor based on Armstrong, M. S. et al. ElectroShape: fast molecular similarity calculations incorporating shape, chirality and electrostatics. J Comput Aided Mol Des 24, 789-801 (2010). http://dx.doi.org/doi:10.1007/s10822-010-9374-0

Aside from spatial coordinates, atoms' charges are also used as the fourth dimension to describe shape of the molecule.

Parameters

mol [oddt.toolkit.Molecule] Molecule to compute Electroshape descriptor

Returns

shape_descriptor [numpy array, shape = (15)] Array describing shape of molecule

oddt.shape.**usr**(*molecule*)

Computes USR shape descriptor based on Ballester PJ, Richards WG (2007). Ultrafast shape recognition to search compound databases for similar molecular shapes. Journal of computational chemistry, 28(10):1711-23. http://dx.doi.org/10.1002/jcc.20681

Parameters

molecule [oddt.toolkit.Molecule] Molecule to compute USR shape descriptor

Returns

shape_descriptor [numpy array, shape = (12)] Array describing shape of molecule

oddt.shape.**usr_cat**(*molecule*)

Computes USRCAT shape descriptor based on Adrian M Schreyer, Tom Blundell (2012). USRCAT: real-time ultrafast shape recognition with pharmacophoric constraints. Journal of Cheminformatics, 2012 4:27. http://dx.doi.org/10.1186/1758-2946-4-27

Parameters

molecule [oddt.toolkit.Molecule] Molecule to compute USRCAT shape descriptor

Returns

shape_descriptor [numpy array, shape = (60)] Array describing shape of molecule

oddt.shape.**usr_similarity**(*mol1_shape*, *mol2_shape*, *ow=1.0*, *hw=1.0*, *rw=1.0*, *aw=1.0*, *dw=1.0*)

Computes similarity between molecules

Parameters

mol1_shape [numpy array] USR shape descriptor

mol2_shape [numpy array] USR shape descriptor

ow [float (default = 1.)] Scaling factor for all atoms Only used for USRCAT, ignored for other types

hw [float (default = 1.)] Scaling factor for hydrophobic atoms Only used for USRCAT, ignored for other types

rw [float (default = 1.)] Scaling factor for aromatic atoms Only used for USRCAT, ignored for other types

aw [float (default = 1.)] Scaling factor for acceptors Only used for USRCAT, ignored for other types

dw [float (default = 1.)] Scaling factor for donors Only used for USRCAT, ignored for other types

Returns

similarity [float from 0 to 1] Similarity between shapes of molecules, 1 indicates identical molecules

5.1.9 oddt.spatial module

Spatial functions included in ODDT Mainly used by other modules, but can be accessed directly.

oddt.spatial.**angle**(*p1*, *p2*, *p3*)
Returns an angle from a series of 3 points (point #2 is centroid). Angle is returned in degrees.

Parameters

p1,p2,p3 [numpy arrays, shape = [n_points, n_dimensions]] Triplets of points in n-dimensional space, aligned in rows.

Returns

angles [numpy array, shape = [n_points]] Series of angles in degrees

oddt.spatial.**angle_2v**(*v1*, *v2*)
Returns an angle between two vecors.Angle is returned in degrees.

Parameters

v1,v2 [numpy arrays, shape = [n_vectors, n_dimensions]] Pairs of vectors in n-dimensional space, aligned in rows.

Returns

angles [numpy array, shape = [n_vectors]] Series of angles in degrees

oddt.spatial.**dihedral**(*p1*, *p2*, *p3*, *p4*)
Returns an dihedral angle from a series of 4 points. Dihedral is returned in degrees. Function distingishes clockwise and antyclockwise dihedrals.

Parameters

p1, p2, p3, p4 [numpy arrays, shape = [n_points, n_dimensions]] Quadruplets of points in n-dimensional space, aligned in rows.

Returns

angles [numpy array, shape = [n_points]] Series of angles in degrees

oddt.spatial.**distance**(*x*, *y*)
Computes distance between each pair of points from x and y.

Parameters

x [numpy arrays, shape = [n_x, 3]] Array of poinds in 3D

y [numpy arrays, shape = [n_y, 3]] Array of poinds in 3D

Returns

dist_matrix [numpy arrays, shape = [n_x, n_y]] Distance matrix

oddt.spatial.**rmsd**(*ref*, *mol*, *ignore_h=True*, *method=None*, *normalize=False*)
Computes root mean square deviation (RMSD) between two molecules (including or excluding Hydrogens). No symmetry checks are performed.

Parameters

ref [oddt.toolkit.Molecule object] Reference molecule for the RMSD calculation

mol [oddt.toolkit.Molecule object] Query molecule for RMSD calculation

ignore_h [bool (default=False)] Flag indicating to ignore Hydrogen atoms while performing RMSD calculation. This toggle works only with 'hungarian' method and without sorting (method=None).

method [str (default=None)] The method to be used for atom asignment between ref and mol. None means that direct matching is applied, which is the default behavior. Available methods:

- canonize - match heavy atoms using canonical ordering (it forces

ignoring H's) - hungarian - minimize RMSD using Hungarian algorithm - min_symmetry - makes multiple molecule-molecule matches and finds minimal RMSD (the slowest). Hydrogens are ignored.

normalize [bool (default=False)] Normalize RMSD by square root of rot. bonds

Returns

rmsd [float] RMSD between two molecules

oddt.spatial.**rotate**(*coords*, *alpha*, *beta*, *gamma*)

Rotate coords by cerain angle in X, Y, Z. Angles are specified in radians.

Parameters

coords [numpy arrays, shape = [n_points, 3]] Coordinates in 3-dimensional space.

alpha, beta, gamma: float Angles to rotate the coordinates along X, Y and Z axis. Angles are specified in radians.

Returns

new_coords [numpy arrays, shape = [n_points, 3]] Rorated coordinates in 3-dimensional space.

5.1.10 oddt.surface module

This module generates and does computation with molecular surfaces.

oddt.surface.**find_surface_residues**(*molecule*, *max_dist=None*, *scaling=1.0*)

Finds residues close to the molecular surface using generate_surface_marching_cubes. Ignores hydrogens and waters present in the molecule.

Parameters

molecule [oddt.toolkit.Molecule] Molecule to find surface residues in.

max_dist [array_like, numeric or None (default = None)] Maximum distance from the surface where residues would still be considered close. If None, compares distances to radii of respective atoms.

scaling [float (default = 1.0)] Expands the grid in which computation is done by generate_surface_marching_cubes by a factor of scaling. Results in a more accurate representation of the surface, and therefore more accurate computation of distances but increases computation time.

Returns

atom_dict [numpy array] An atom_dict containing only the surface residues from the original molecule.

oddt.surface.**generate_surface_marching_cubes**(*molecule*, *remove_hoh=False*, *scaling=1.0*, *probe_radius=1.4*)

Generates a molecular surface mesh using the marching_cubes method from scikit-image. Ignores hydrogens present in the molecule.

Parameters

molecule [oddt.toolkit.Molecule object] Molecule for which the surface will be generated

remove_hoh [bool (default = False)] If True, remove waters from the molecule before generating the surface. Requires molecule.protein to be set to True.

scaling [float (default = 1.0)] Expands the grid in which computation is done by a factor of scaling. Results in a more accurate representation of the surface, but increases computation time.

probe_radius [float (default = 1.4)] Radius of a ball used to patch up holes inside the molecule resulting from some molecular distances being larger (usually in protein). Basically reduces the surface to one accesible by other molecules of radius smaller than probe_radius.

Returns

verts [numpy array] Spatial coordinates for mesh vertices.

faces [numpy array] Faces are defined by referencing vertices from verts.

5.1.11 oddt.utils module

Common utilities for ODDT

oddt.utils.**check_molecule**(*mol*, *force_protein=False*, *force_coords=False*, *non_zero_atoms=False*)

Universal validator of molecule objects. Usage of positional arguments is allowed only for molecule object, otherwise it is prohibitted (i.e. the order of arguments **will** change). Desired properties of molecule are validated based on specified arguments. By default only the object type is checked. In case of discrepancy to the specification a *ValueError* is raised with appropriate message.

New in version 0.6.

Parameters

mol: oddt.toolkit.Molecule object Object to verify

force_protein: bool (default=False) Force the molecule to be marked as protein (mol.protein).

force_coords: bool (default=False) Force the molecule to have non-zero coordinates.

non_zero_atoms: bool (default=False) Check if molecule has at least one atom.

oddt.utils.**chunker**(*iterable*, *chunksize=100*)

Generate chunks from a generator object. If iterable is passed which is not a generator it will be converted to one with *iter()*.

New in version 0.6.

oddt.utils.**compose_iter**(*iterable*, *funcs*)

Chain functions and apply them to iterable, by exhausting the iterable. Functions are executed in the order from funcs.

New in version 0.6.

oddt.utils.**is_molecule**(*obj*)

Check whether an object is an *oddt.toolkits.{rdk,ob}.Molecule* instance.

New in version 0.6.

oddt.utils.**is_openbabel_molecule**(*obj*)

Check whether an object is an *oddt.toolkits.ob.Molecule* instance.

New in version 0.6.

oddt.utils.**is_rdkit_molecule**(*obj*)
Check whether an object is an *oddt.toolkits.rdk.Molecule* instance.

New in version 0.6.

oddt.utils.**method_caller**(*obj*, *methodname*, **args*, ***kwargs*)
Helper function to workaround Python 2 pickle limitations to parallelize methods and generator objects

5.1.12 oddt.virtualscreening module

ODDT pipeline framework for virtual screening

class oddt.virtualscreening.**virtualscreening**(*n_cpu=-1*, *verbose=False*, *chunksize=100*)
Virtual Screening pipeline stack

Parameters

n_cpu: int (default=-1) The number of parallel procesors to use

verbose: bool (default=False) Verbosity flag for some methods

Methods

apply_filter(expression[, soft_fail])	Filtering method, can use raw expressions (strings to be evaled in if statement, can use oddt.toolkit.Molecule methods, eg.
dock(engine, protein, *args, **kwargs)	Docking procedure.
fetch()	A method to exhaust the pipeline.
load_ligands(fmt, ligands_file, **kwargs)	Loads file with ligands.
score(function[, protein])	Scoring procedure compatible with any scoring function implemented in ODDT and other pickled SFs which are subclasses of *oddt.scoring.scorer*.
similarity(method, query[, cutoff, protein])	Similarity filter.
write(fmt, filename[, csv_filename])	Outputs molecules to a file
write_csv(csv_filename[, fields, keep_pipe])	Outputs molecules to a csv file

apply_filter(*expression*, *soft_fail=0*)
Filtering method, can use raw expressions (strings to be evaled in if statement, can use oddt.toolkit.Molecule methods, eg. *mol.molwt < 500*) Currently supported presets:

- Lipinski Rule of 5 ('ro5' or 'l5')
- Fragment Rule of 3 ('ro3')
- PAINS filter ('pains')

Parameters

expression: string or list of strings Expresion(s) to be used while filtering.

soft_fail: int (default=0) The number of faulures molecule can have to pass filter, aka. soft-fails.

dock(*engine*, *protein*, **args*, ***kwargs*)
Docking procedure.

Parameters

engine: string Which docking engine to use.

Notes

Additional parameters are passed directly to the engine. Following docking engines are supported:

1. Audodock Vina (`engine="autodock_vina"`), see *oddt.docking.autodock_vina*.

fetch()
A method to exhaust the pipeline. Itself it is lazy (a generator)

load_ligands(*fmt*, *ligands_file*, ***kwargs*)
Loads file with ligands.

Parameters

file_type: string Type of molecular file

ligands_file: string Path to a file, which is loaded to pipeline

score(*function*, *protein=None*, **args*, ***kwargs*)
Scoring procedure compatible with any scoring function implemented in ODDT and other pickled SFs which are subclasses of *oddt.scoring.scorer*.

Parameters

function: string Which scoring function to use.

protein: oddt.toolkit.Molecule Default protein to use as reference

Notes

Additional parameters are passed directly to the scoring function.

similarity(*method*, *query*, *cutoff=0.9*, *protein=None*)

Similarity filter. Supported structural methods:

- ift: interaction fingerprints
- sift: simple interaction fingerprints
- usr: Ultrafast Shape recognition
- usr_cat: Ultrafast Shape recognition, Credo Atom Types
- electroshape: Electroshape, an USR method including partial charges

Parameters

method: string Similarity method used to compare molecules. Avaiale methods: * *ifp* - interaction fingerprint (requires a receptor) * *sifp* - simple interaction fingerprint (requires a receptor) * *usr* - Ultrafast Shape Reckognition * *usr_cat* - USR, with CREDO atom types * *electroshape* - Electroshape, USR with moments representing partial charge

query: oddt.toolkit.Molecule or list of oddt.toolkit.Molecule Query molecules to compare the pipeline to.

cutoff: float Similarity cutoff for filtering molecules. Any similarity lower than it will be filtered out.

protein: oddt.toolkit.Molecule (default = None) Protein for underling method. By default it's empty, but sturctural fingerprints need one.

write (*fmt*, *filename*, *csv_filename=None*, ***kwargs*)
Outputs molecules to a file

Parameters

file_type: string Type of molecular file

ligands_file: string Path to a output file

csv_filename: string Optional path to a CSV file

write_csv (*csv_filename*, *fields=None*, *keep_pipe=False*, ***kwargs*)
Outputs molecules to a csv file

Parameters

csv_filename: string Optional path to a CSV file

fields: list (default None) List of fields to save in CSV file

keep_pipe: bool (default=False) If set to True, the ligand pipe is sustained.

5.1.13 Module contents

Open Drug Discovery Toolkit

Universal and easy to use resource for various drug discovery tasks, ie docking, virutal screening, rescoring.

Attributes

toolkit [module,] Toolkits backend module, currenlty OpenBabel [ob] and RDKit [rdk]. This setting is toolkit-wide, and sets given toolkit as default

CHAPTER 6

References

To be announced.

CHAPTER 7

Documentation Indices and tables

- genindex
- modindex
- search

Bibliography

[1] Durrant JD, McCammon JA. NNScore 2.0: a neural-network receptor-ligand scoring function. J Chem Inf Model. 2011;51: 2897-2903. doi:10.1021/ci2003889

[2] Durrant JD, McCammon JA. BINANA: a novel algorithm for ligand-binding characterization. J Mol Graph Model. 2011;29: 888-893. doi:10.1016/j.jmgm.2011.01.004

[1] Ballester PJ, Mitchell JBO. A machine learning approach to predicting protein-ligand binding affinity with applications to molecular docking. Bioinformatics. 2010;26: 1169-1175. doi:10.1093/bioinformatics/btq112

[2] Ballester PJ, Schreyer A, Blundell TL. Does a more precise chemical description of protein-ligand complexes lead to more accurate prediction of binding affinity? J Chem Inf Model. 2014;54: 944-955. doi:10.1021/ci500091r

[3] Li H, Leung K-S, Wong M-H, Ballester PJ. Improving AutoDock Vina Using Random Forest: The Growing Accuracy of Binding Affinity Prediction by the Effective Exploitation of Larger Data Sets. Mol Inform. WILEY-VCH Verlag; 2015;34: 115-126. doi:10.1002/minf.201400132

[1] Ballester PJ, Mitchell JBO. A machine learning approach to predicting protein-ligand binding affinity with applications to molecular docking. Bioinformatics. 2010;26: 1169-1175. doi:10.1093/bioinformatics/btq112

[2] Ballester PJ, Schreyer A, Blundell TL. Does a more precise chemical description of protein-ligand complexes lead to more accurate prediction of binding affinity? J Chem Inf Model. 2014;54: 944-955. doi:10.1021/ci500091r

[3] Li H, Leung K-S, Wong M-H, Ballester PJ. Improving AutoDock Vina Using Random Forest: The Growing Accuracy of Binding Affinity Prediction by the Effective Exploitation of Larger Data Sets. Mol Inform. WILEY-VCH Verlag; 2015;34: 115-126. doi:10.1002/minf.201400132

[1] Durrant JD, McCammon JA. NNScore 2.0: a neural-network receptor-ligand scoring function. J Chem Inf Model. 2011;51: 2897-2903. doi:10.1021/ci2003889

[2] Durrant JD, McCammon JA. BINANA: a novel algorithm for ligand-binding characterization. J Mol Graph Model. 2011;29: 888-893. doi:10.1016/j.jmgm.2011.01.004

[1] Wikipedia entry for the Receiver operating characteristic

[1] Sheridan, R. P.; Singh, S. B.; Fluder, E. M.; Kearsley, S. K. Protocols for bridging the peptide to nonpeptide gap in topological similarity searches. J. Chem. Inf. Comput. Sci. 2001, 41, 1395-1406. DOI: 10.1021/ci0100144

[1] Truchon J-F, Bayly CI. Evaluating virtual screening methods: good and bad metrics for the "early recognition" problem. J Chem Inf Model. 2007;47: 488-508. DOI: 10.1021/ci600426e

[1] https://docs.python.org/3/library/pickle.html

[1] http://docs.sqlalchemy.org

[2] https://www.python.org/dev/peps/pep-0249/

[1] https://docs.python.org/3/library/pickle.html

[1] http://docs.sqlalchemy.org

[2] https://www.python.org/dev/peps/pep-0249/

[1] https://docs.python.org/3/library/pickle.html

[1] http://docs.sqlalchemy.org

[2] https://www.python.org/dev/peps/pep-0249/

Python Module Index

o

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X