matplotlib.axes.Axes or np.ndarray of them A NumPy array is returned when *subplots* is True.

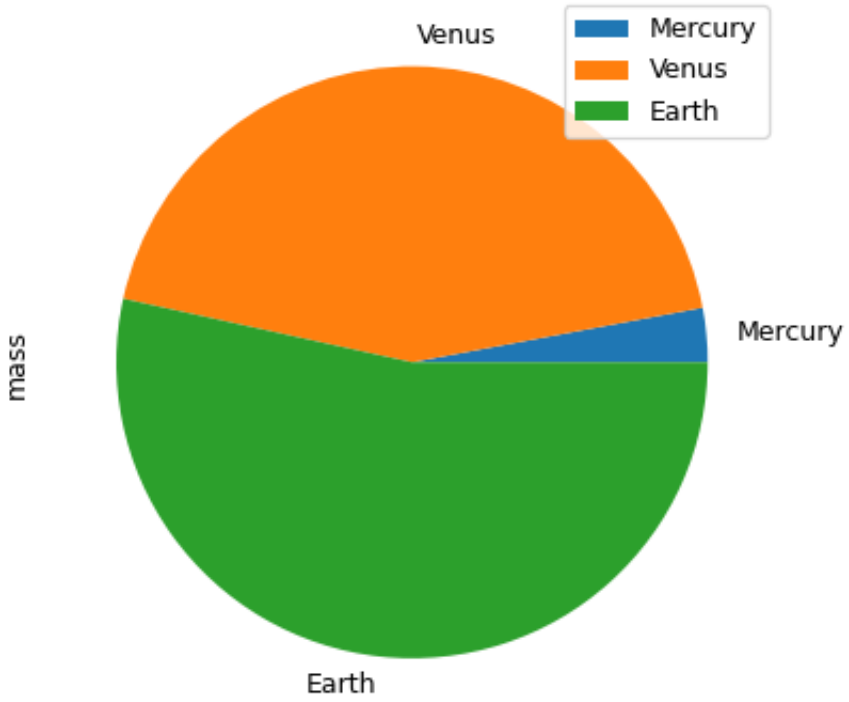
See also:

Series.plot.pie Generate a pie plot for a Series.
DataFrame.plot Make plots of a DataFrame.

Examples

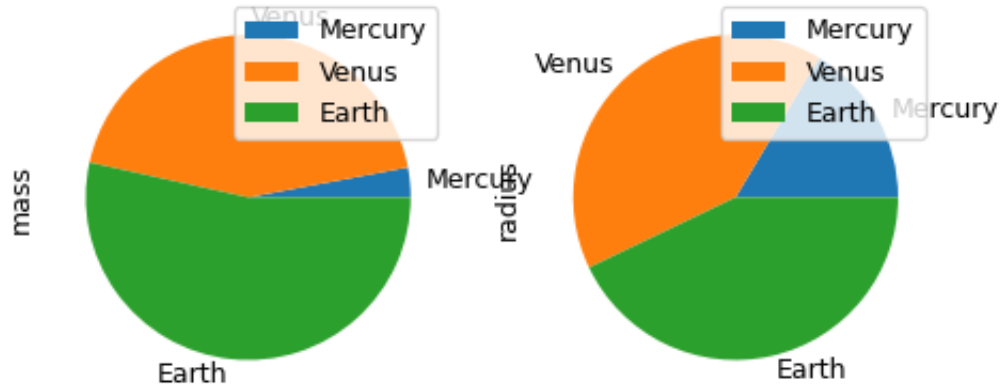
In the example below we have a DataFrame with the information about planet's mass and radius. We pass the the 'mass' column to the pie function to get a pie plot.

```
>>> df = pd.DataFrame({'mass': [0.330, 4.87 , 5.97],
...                    'radius': [2439.7, 6051.8, 6378.1]},
...                   index=['Mercury', 'Venus', 'Earth'])
>>> plot = df.plot.pie(y='mass', figsize=(5, 5))
```




```
>>> plot = df.plot.pie(subplots=True, figsize=(6, 3))
```

Series.hist(self[, by, ax, grid, ...])	Draw histogram of the input series using matplotlib.



3.3.15 Serialization / IO / conversion

`Series.to_pickle`(self, path, compression, ...)	Pickle (serialize) object to file.
`Series.to_csv`(self, path_or_buf, ...)	Write object to a comma-separated values (csv) file.
`Series.to_dict`(self[, into])	Convert Series to {label -> value} dict or dict-like object.
`Series.to_excel`(self, excel_writer[, ...])	Write object to an Excel sheet.
`Series.to_frame`(self[, name])	Convert Series to DataFrame.
`Series.to_xarray`(self)	Return an xarray object from the pandas object.
`Series.to_hdf`(self, path_or_buf, key, mode, ...)	Write the contained data to an HDF5 file using HDFStore.
`Series.to_sql`(self, name, con[, schema, ...])	Write records stored in a DataFrame to a SQL database.
`Series.to_json`(self, path_or_buf, ...)	Convert the object to a JSON string.
`Series.to_string`(self[, buf, na_rep, ...])	Render a string representation of the Series.
`Series.to_clipboard`(self, excel, sep, ...)	Copy object to the system clipboard.
`Series.to_latex`(self[, buf, columns, ...])	Render object to a LaTeX tabular, longtable, or nested table/tabular.
`Series.to_markdown`(self, buf, ...)	Print Series in Markdown-friendly format.

3.4 DataFrame

3.4.1 Constructor

`DataFrame`([data])	Two-dimensional, size-mutable, potentially heterogeneous tabular data.

pandas.DataFrame

class pandas.**DataFrame**(*data=None*, *index: Optional[Collection] = None*, *columns: Optional[Collection] = None*, *dtype: Optional[Union[str, numpy.dtype, ExtensionDtype]] = None*, *copy: bool = False*)

Two-dimensional, size-mutable, potentially heterogeneous tabular data.

Data structure also contains labeled axes (rows and columns). Arithmetic operations align on both row and column labels. Can be thought of as a dict-like container for Series objects. The primary pandas data structure.

Parameters

dtype [dtype, default None] Data type to force. Only a single dtype is allowed. If None, infer.

copy [bool, default False] Copy data from inputs. Only affects DataFrame / 2d ndarray input.

See also:

DataFrame.from_records Constructor from tuples, also record arrays.
DataFrame.from_dict From dicts of Series, arrays, or dicts.
read_csv
read_table
read_clipboard

Examples

Constructing DataFrame from a dictionary.

```
>>> d = {'col1': [1, 2], 'col2': [3, 4]}
>>> df = pd.DataFrame(data=d)
>>> df
   col1  col2
0     1     3
1     2     4
```

Notice that the inferred dtype is int64.

```
>>> df.dtypes
col1    int64
col2    int64
dtype: object
```

To enforce a single dtype:

```
>>> df = pd.DataFrame(data=d, dtype=np.int8)
>>> df.dtypes
col1    int8
col2    int8
dtype: object
```

Constructing DataFrame from numpy ndarray:

```
>>> df2 = pd.DataFrame(np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]),
...                    columns=['a', 'b', 'c'])
>>> df2
   a  b  c
0  1  2  3
1  4  5  6
2  7  8  9
```

Attributes

T	Transpose index and columns.
at	Access a single value for a row/column label pair.
attrs	Dictionary of global attributes on this object.
axes	Return a list representing the axes of the DataFrame.
columns	The column labels of the DataFrame.
dtypes	Return the dtypes in the DataFrame.
empty	Indicator whether DataFrame is empty.
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.
loc	Access a group of rows and columns by label(s) or a boolean array.
ndim	Return an int representing the number of axes / array dimensions.
shape	Return a tuple representing the dimensionality of the DataFrame.
size	Return an int representing the number of elements in this object.
style	Returns a Styler object.
values	Return a Numpy representation of the DataFrame.

pandas.DataFrame.T

property 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

***args** [tuple, optional] Accepted for compatibility with NumPy.

copy [bool, default False] Whether to copy the data after transposing, even for DataFrames with a single dtype.

Note that a copy is always required for mixed dtype DataFrames, or for DataFrames with any extension types.

Returns

DataFrame The transposed DataFrame.

See also:

numpy.transpose Permute the dimensions of a given array.

Notes

Transposing a DataFrame with mixed dtypes will result in a homogeneous DataFrame with the *object* dtype. In such a case, a copy of the data is always made.

Examples

Square DataFrame with homogeneous dtype

```
>>> d1 = {'col1': [1, 2], 'col2': [3, 4]}
>>> df1 = pd.DataFrame(data=d1)
>>> df1
   col1  col2
0     1     3
1     2     4
```

```
>>> df1_transposed = df1.T # or df1.transpose()
>>> df1_transposed
      0  1
col1  1  2
col2  3  4
```

When the dtype is homogeneous in the original DataFrame, we get a transposed DataFrame with the same dtype:

```
>>> df1.dtypes
col1    int64
col2    int64
dtype: object
>>> df1_transposed.dtypes
0    int64
1    int64
dtype: object
```

Non-square DataFrame with mixed dtypes

```
>>> d2 = {'name': ['Alice', 'Bob'],
...       'score': [9.5, 8],
...       'employed': [False, True],
...       'kids': [0, 0]}
>>> df2 = pd.DataFrame(data=d2)
>>> df2
    name  score  employed  kids
0  Alice    9.5     False     0
1    Bob    8.0      True     0
```

```
>>> df2_transposed = df2.T # or df2.transpose()
>>> df2_transposed
              0     1
name      Alice   Bob
score       9.5     8
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
```

pandas.DataFrame.at

property DataFrame.**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 If '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
```

pandas.DataFrame.attrs

property DataFrame.**attrs**
Dictionary of global attributes on this object.

> **Warning:** attrs is experimental and may change without warning.

pandas.DataFrame.axes

property DataFrame.**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(['col1', 'col2'],
dtype='object')]
```

pandas.DataFrame.columns

DataFrame.**columns**
The column labels of the DataFrame.

pandas.DataFrame.dtypes

property DataFrame.**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.

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
```

pandas.DataFrame.empty

property DataFrame.**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:

Series.dropna

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
```

pandas.DataFrame.iat

property DataFrame.**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
```

pandas.DataFrame.iloc

property DataFrame.**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 or DataFrame) and that returns valid output for indexing (one of the above). This is useful in method chains, when you don't have a reference to the calling object, but would like to base your selection on some value.

`.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*.

See also:

DataFrame.iat Fast integer location scalar accessor.

DataFrame.loc Purely label-location based indexer for selection by label.

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

Examples

```
>>> mydict = [{'a': 1, 'b': 2, 'c': 3, 'd': 4},
...           {'a': 100, 'b': 200, 'c': 300, 'd': 400},
...           {'a': 1000, 'b': 2000, 'c': 3000, 'd': 4000 }]
>>> df = pd.DataFrame(mydict)
>>> df
      a     b     c     d
0     1     2     3     4
1   100   200   300   400
2  1000  2000  3000  4000
```

Indexing just the rows

With a scalar integer.

```
>>> type(df.iloc[0])
<class 'pandas.core.series.Series'>
>>> df.iloc[0]
a    1
b    2
c    3
d    4
Name: 0, dtype: int64
```

With a list of integers.

```
>>> df.iloc[[0]]
   a  b  c  d
0  1  2  3  4
>>> type(df.iloc[[0]])
<class 'pandas.core.frame.DataFrame'>
```

```
>>> df.iloc[[0, 1]]
     a    b    c    d
0    1    2    3    4
1  100  200  300  400
```

With a *slice* object.

```
>>> df.iloc[:3]
      a     b     c     d
0     1     2     3     4
1   100   200   300   400
2  1000  2000  3000  4000
```

With a boolean mask the same length as the index.

```
>>> df.iloc[[True, False, True]]
      a     b     c     d
0     1     2     3     4
2  1000  2000  3000  4000
```

With a callable, useful in method chains. The *x* passed to the `lambda` is the DataFrame being sliced. This selects the rows whose index label even.

```
>>> df.iloc[lambda x: x.index % 2 == 0]
      a     b     c     d
0     1     2     3     4
2  1000  2000  3000  4000
```

Indexing both axes

You can mix the indexer types for the index and columns. Use `:` to select the entire axis.

With scalar integers.

```
>>> df.iloc[0, 1]
2
```

With lists of integers.

```
>>> df.iloc[[0, 2], [1, 3]]
      b     d
0     2     4
2  2000  4000
```

With *slice* objects.

```
>>> df.iloc[1:3, 0:3]
      a     b     c
1   100   200   300
2  1000  2000  3000
```

With a boolean array whose length matches the columns.

```
>>> df.iloc[:, [True, False, True, False]]
      a     c
0     1     3
1   100   300
2  1000  3000
```

With a callable function that expects the Series or DataFrame.

```
>>> df.iloc[:, lambda df: [0, 2]]
      a     c
0     1     3
1   100   300
2  1000  3000
```

pandas.DataFrame.index

DataFrame.**index**
The index (row labels) of the DataFrame.

pandas.DataFrame.loc

property DataFrame.**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 or DataFrame) and that returns valid output for indexing (one of the above)

See more at *Selection by Label*

Raises

KeyError If 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
```

(continues on next page)

(continued from previous page)

```
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'),
```

(continues on next page)

(continued from previous page)

```
...     ('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
```

(continues on next page)

(continued from previous page)

```
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
```

pandas.DataFrame.ndim

property DataFrame.**ndim**

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

Return 1 if Series. Otherwise return 2 if DataFrame.

See also:

ndarray.ndim Number of array dimensions.

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
```

pandas.DataFrame.shape

property DataFrame.**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)
```

pandas.DataFrame.size

property DataFrame.**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 Number of elements in the array.

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
```

pandas.DataFrame.style

property DataFrame.**style**

Returns a Styler object.

Contains methods for building a styled HTML representation of the DataFrame. a styled HTML representation fo the DataFrame.

See also:

io.formats.style.Styler

pandas.DataFrame.values

property DataFrame.**values**

Return a Numpy representation of the DataFrame.

> **Warning:** We recommend using *DataFrame.to_numpy()* instead.

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:

DataFrame.to_numpy Recommended alternative to this method.

DataFrame.index Retrieve the index labels.

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
>>> 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)
```

Methods

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

continues on next page

Table 59 – continued from previous page

combine_first(self, other)	Update null elements with value in the same location in *other*.
convert_dtypes(self, infer_objects, ...)	Convert columns to best possible dtypes using dtypes supporting `pd.NA`.
copy(self, deep)	Make a copy of this object's indices and data.
corr(self[, method, min_periods])	Compute pairwise correlation of columns, excluding NA/null values.
corrwith(self, other[, axis, drop, method])	Compute pairwise correlation.
count(self[, axis, level, numeric_only])	Count non-NA cells for each column or row.
cov(self[, min_periods])	Compute pairwise covariance of columns, excluding NA/null values.
cummax(self[, axis, skipna])	Return cumulative maximum over a DataFrame or Series axis.
cummin(self[, axis, skipna])	Return cumulative minimum over a DataFrame or Series axis.
cumprod(self[, axis, skipna])	Return cumulative product over a DataFrame or Series axis.
cumsum(self[, axis, skipna])	Return cumulative sum over a DataFrame or Series axis.
describe(self[, percentiles, include, exclude])	Generate descriptive statistics.
diff(self[, periods, axis])	First discrete difference of element.
div(self, other[, axis, level, fill_value])	Get Floating division of dataframe and other, element-wise (binary operator *truediv*).
divide(self, other[, axis, level, fill_value])	Get Floating division of dataframe and other, element-wise (binary operator *truediv*).
dot(self, other)	Compute the matrix multiplication between the DataFrame and other.
drop(self[, labels, axis, index, columns, ...])	Drop specified labels from rows or columns.
drop_duplicates(self, subset, ...)	Return DataFrame with duplicate rows removed.
droplevel(self, level[, axis])	Return DataFrame with requested index / column level(s) removed.
dropna(self[, axis, how, thresh, subset, ...])	Remove missing values.
duplicated(self, subset, Sequence[Hashable], ...)	Return boolean Series denoting duplicate rows.
eq(self, other[, axis, level])	Get Equal to of dataframe and other, element-wise (binary operator *eq*).
equals(self, other)	Test whether two objects contain the same elements.
eval(self, expr[, inplace])	Evaluate a string describing operations on DataFrame columns.
ewm(self[, com, span, halflife, alpha, ...])	Provide exponential weighted functions.
expanding(self[, min_periods, center, axis])	Provide expanding transformations.
explode(self, column, Tuple])	Transform each element of a list-like to a row, replicating index values.
ffill(self[, axis, limit, downcast])	Synonym for *DataFrame.fillna()* with `method='ffill'`.
fillna(self[, value, method, axis, inplace, ...])	Fill NA/NaN values using the specified method.
filter(self[, items, axis])	Subset the dataframe rows or columns according to the specified index labels.
first(self, offset)	Method to subset initial periods of time series data based on a date offset.
first_valid_index(self)	Return index for first non-NA/null value.

continues on next page

Table 59 – continued from previous page

floordiv(self, other[, axis, level, fill_value])	Get Integer division of dataframe and other, element-wise (binary operator *floordiv*).
from_dict(data[, orient, dtype, columns])	Construct DataFrame from dict of array-like or dicts.
from_records(data[, index, exclude, ...])	Convert structured or record ndarray to DataFrame.
ge(self, other[, axis, level])	Get Greater than or equal to of dataframe and other, element-wise (binary operator *ge*).
get(self, key[, default])	Get item from object for given key (ex: DataFrame column).
groupby(self[, by, axis, level])	Group DataFrame using a mapper or by a Series of columns.
gt(self, other[, axis, level])	Get Greater than of dataframe and other, element-wise (binary operator *gt*).
head(self, n)	Return the first *n* rows.
hist(data[, column, by, grid, xlabelsize, ...])	Make a histogram of the DataFrame's.
idxmax(self[, axis, skipna])	Return index of first occurrence of maximum over requested axis.
idxmin(self[, axis, skipna])	Return index of first occurrence of minimum over requested axis.
infer_objects(self)	Attempt to infer better dtypes for object columns.
info(self[, verbose, buf, max_cols, ...])	Print a concise summary of a DataFrame.
insert(self, loc, column, value[, ...])	Insert column into DataFrame at specified location.
interpolate(self[, method, axis, limit, ...])	Interpolate values according to different methods.
isin(self, values)	Whether each element in the DataFrame is contained in values.
isna(self)	Detect missing values.
isnull(self)	Detect missing values.
items(self)	Iterate over (column name, Series) pairs.
iteritems(self)	Iterate over (column name, Series) pairs.
iterrows(self)	Iterate over DataFrame rows as (index, Series) pairs.
itertuples(self[, index, name])	Iterate over DataFrame rows as namedtuples.
join(self, other[, on, how, lsuffix, ...])	Join columns of another DataFrame.
keys(self)	Get the 'info axis' (see Indexing for more).
kurt(self[, axis, skipna, level, numeric_only])	Return unbiased kurtosis over requested axis.
kurtosis(self[, axis, skipna, level, ...])	Return unbiased kurtosis over requested axis.
last(self, offset)	Method to subset final periods of time series data based on a date offset.
last_valid_index(self)	Return index for last non-NA/null value.
le(self, other[, axis, level])	Get Less than or equal to of dataframe and other, element-wise (binary operator *le*).
lookup(self, row_labels, col_labels)	Label-based "fancy indexing" function for DataFrame.
lt(self, other[, axis, level])	Get Less than of dataframe and other, element-wise (binary operator *lt*).
mad(self[, axis, skipna, level])	Return the mean absolute deviation of the values for the requested axis.
mask(self, cond[, other, inplace, axis, ...])	Replace values where the condition is True.
max(self[, axis, skipna, level, numeric_only])	Return the maximum of the values for the requested axis.
mean(self[, axis, skipna, level, numeric_only])	Return the mean of the values for the requested axis.
median(self[, axis, skipna, level, numeric_only])	Return the median of the values for the requested axis.

continues on next page

Table 59 – continued from previous page

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

continues on next page

Table 59 – continued from previous page

reindex(self[, labels, index, columns, ...])	Conform DataFrame to new index with optional filling logic.
reindex_like(self, other, method, ...[, ...])	Return an object with matching indices as other object.
rename(self[, mapper, index, columns, axis, ...])	Alter axes labels.
rename_axis(self[, mapper, index, columns, ...])	Set the name of the axis for the index or columns.
reorder_levels(self, order[, axis])	Rearrange index levels using input order.
replace(self[, to_replace, value, inplace, ...])	Replace values given in *to_replace* with *value*.
resample(self, rule[, axis, loffset, on, level])	Resample time-series data.
reset_index(self, level, Sequence[Hashable], ...)	Reset the index, or a level of it.
rfloordiv(self, other[, axis, level, fill_value])	Get Integer division of dataframe and other, element-wise (binary operator *rfloordiv*).
rmod(self, other[, axis, level, fill_value])	Get Modulo of dataframe and other, element-wise (binary operator *rmod*).
rmul(self, other[, axis, level, fill_value])	Get Multiplication of dataframe and other, element-wise (binary operator *rmul*).
rolling(self, window[, min_periods, center, ...])	Provide rolling window calculations.
round(self[, decimals])	Round a DataFrame to a variable number of decimal places.
rpow(self, other[, axis, level, fill_value])	Get Exponential power of dataframe and other, element-wise (binary operator *rpow*).
rsub(self, other[, axis, level, fill_value])	Get Subtraction of dataframe and other, element-wise (binary operator *rsub*).
rtruediv(self, other[, axis, level, fill_value])	Get Floating division of dataframe and other, element-wise (binary operator *rtruediv*).
sample(self[, n, frac, replace, weights, ...])	Return a random sample of items from an axis of object.
select_dtypes(self[, include, exclude])	Return a subset of the DataFrame's columns based on the column dtypes.
sem(self[, axis, skipna, level, ddof, ...])	Return unbiased standard error of the mean over requested axis.
set_axis(self, labels[, axis, inplace])	Assign desired index to given axis.
set_index(self, keys[, drop, append, ...])	Set the DataFrame index using existing columns.
shift(self[, periods, freq, axis, fill_value])	Shift index by desired number of periods with an optional time *freq*.
skew(self[, axis, skipna, level, numeric_only])	Return unbiased skew over requested axis.
slice_shift(self, periods[, axis])	Equivalent to *shift* without copying data.
sort_index(self[, axis, level, ascending, ...])	Sort object by labels (along an axis).
sort_values(self, by[, axis, ascending, ...])	Sort by the values along either axis.
sparse	alias of pandas.core.arrays.sparse.accessor.SparseFrameAccessor
squeeze(self[, axis])	Squeeze 1 dimensional axis objects into scalars.
stack(self[, level, dropna])	Stack the prescribed level(s) from columns to index.
std(self[, axis, skipna, level, ddof, ...])	Return sample standard deviation over requested axis.
sub(self, other[, axis, level, fill_value])	Get Subtraction of dataframe and other, element-wise (binary operator *sub*).
subtract(self, other[, axis, level, fill_value])	Get Subtraction of dataframe and other, element-wise (binary operator *sub*).
sum(self[, axis, skipna, level, ...])	Return the sum of the values for the requested axis.

continues on next page

Table 59 – continued from previous page

swapaxes(self, axis1, axis2[, copy])	Interchange axes and swap values axes appropriately.
swaplevel(self[, i, j, axis])	Swap levels i and j in a MultiIndex on a particular axis.
tail(self, n)	Return the last *n* rows.
take(self, indices[, axis])	Return the elements in the given *positional* indices along an axis.
to_clipboard(self, excel, sep, ...)	Copy object to the system clipboard.
to_csv(self, path_or_buf, pathlib.Path, ...)	Write object to a comma-separated values (csv) file.
to_dict(self[, orient, into])	Convert the DataFrame to a dictionary.
to_excel(self, excel_writer[, sheet_name, ...])	Write object to an Excel sheet.
to_feather(self, path)	Write out the binary feather-format for DataFrames.
to_gbq(self, destination_table[, ...])	Write a DataFrame to a Google BigQuery table.
to_hdf(self, path_or_buf, key, mode, ...[, ...])	Write the contained data to an HDF5 file using HDFStore.
to_html(self[, buf, columns, col_space, ...])	Render a DataFrame as an HTML table.
to_json(self, path_or_buf, pathlib.Path, ...)	Convert the object to a JSON string.
to_latex(self[, buf, columns, col_space, ...])	Render object to a LaTeX tabular, longtable, or nested table/tabular.
to_markdown(self, buf, NoneType] = None, ...)	Print DataFrame in Markdown-friendly format.
to_numpy(self[, dtype, copy])	Convert the DataFrame to a NumPy array.
to_parquet(self, path[, engine, ...])	Write a DataFrame to the binary parquet format.
to_period(self[, freq, axis, copy])	Convert DataFrame from DatetimeIndex to PeriodIndex.
to_pickle(self, path, compression, ...)	Pickle (serialize) object to file.
to_records(self[, index, column_dtypes, ...])	Convert DataFrame to a NumPy record array.
to_sql(self, name, con[, schema, ...])	Write records stored in a DataFrame to a SQL database.
to_stata(self, path[, convert_dates, ...])	Export DataFrame object to Stata dta format.
to_string(self, buf, pathlib.Path, IO[str], ...)	Render a DataFrame to a console-friendly tabular output.
to_timestamp(self[, freq, how, axis, copy])	Cast to DatetimeIndex of timestamps, at *beginning* of period.
to_xarray(self)	Return an xarray object from the pandas object.
transform(self, func[, axis])	Call `func` on self producing a DataFrame with transformed values.
transpose(self, *args, copy)	Transpose index and columns.
truediv(self, other[, axis, level, fill_value])	Get Floating division of dataframe and other, element-wise (binary operator *truediv*).
truncate(self[, before, after, axis])	Truncate a Series or DataFrame before and after some index value.
tshift(self, periods[, freq, axis])	Shift the time index, using the index's frequency if available.
tz_convert(self, tz[, axis, level])	Convert tz-aware axis to target time zone.
tz_localize(self, tz[, axis, level, ambiguous])	Localize tz-naive index of a Series or DataFrame to target time zone.
unstack(self[, level, fill_value])	Pivot a level of the (necessarily hierarchical) index labels.
update(self, other[, join, overwrite, ...])	Modify in place using non-NA values from another DataFrame.
var(self[, axis, skipna, level, ddof, ...])	Return unbiased variance over requested axis.
where(self, cond[, other, inplace, axis, ...])	Replace values where the condition is False.

continues on next page

Table 59 – continued from previous page

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

pandas.DataFrame.abs

DataFrame.**abs**(*self: ~ FrameOrSeries*) → ~FrameOrSeries

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]
... })
```

(continues on next page)