pandas.interval_range

pandas.**interval_range**(*start=None*, *end=None*, *periods=None*, *freq=None*, *name=None*, *closed='right'*)

Return a fixed frequency IntervalIndex.

Parameters

start [numeric or datetime-like, default None] Left bound for generating intervals.

end [numeric or datetime-like, default None] Right bound for generating intervals.

periods [int, default None] Number of periods to generate.

freq [numeric, str, or DateOffset, default None] The length of each interval. Must be consistent with the type of start and end, e.g. 2 for numeric, or ‘5H’ for datetime-like. Default is 1 for numeric and ‘D’ for datetime-like.

name [str, default None] Name of the resulting IntervalIndex.

closed [{‘left’, ‘right’, ‘both’, ‘neither’}, default ‘right’] Whether the intervals are closed on the left-side, right-side, both or neither.

Returns

IntervalIndex

See also:

IntervalIndex An Index of intervals that are all closed on the same side.

Notes

Of the four parameters start, end, periods, and freq, exactly three must be specified. If freq is omitted, the resulting IntervalIndex will have periods linearly spaced elements between start and end, inclusively.

To learn more about datetime-like frequency strings, please see this link.

Examples

Numeric start and end is supported.

```
>>> pd.interval_range(start=0, end=5)
IntervalIndex([(0, 1], (1, 2], (2, 3], (3, 4], (4, 5]],
              closed='right', dtype='interval[int64]')
```

Additionally, datetime-like input is also supported.

```
>>> pd.interval_range(start=pd.Timestamp('2017-01-01'),
...                   end=pd.Timestamp('2017-01-04'))
IntervalIndex([(2017-01-01, 2017-01-02], (2017-01-02, 2017-01-03],
               (2017-01-03, 2017-01-04]],
              closed='right', dtype='interval[datetime64[ns]]')
```

The freq parameter specifies the frequency between the left and right. endpoints of the individual intervals within the IntervalIndex. For numeric start and end, the frequency must also be numeric.

```
>>> pd.interval_range(start=0, periods=4, freq=1.5)
IntervalIndex([(0.0, 1.5], (1.5, 3.0], (3.0, 4.5], (4.5, 6.0]],
              closed='right', dtype='interval[float64]')
```

Similarly, for datetime-like `start` and `end`, the frequency must be convertible to a DateOffset.

```
>>> pd.interval_range(start=pd.Timestamp('2017-01-01'),
...                    periods=3, freq='MS')
IntervalIndex([(2017-01-01, 2017-02-01], (2017-02-01, 2017-03-01],
               (2017-03-01, 2017-04-01]],
              closed='right', dtype='interval[datetime64[ns]]')
```

Specify `start`, `end`, and `periods`; the frequency is generated automatically (linearly spaced).

```
>>> pd.interval_range(start=0, end=6, periods=4)
IntervalIndex([(0.0, 1.5], (1.5, 3.0], (3.0, 4.5], (4.5, 6.0]],
          closed='right',
          dtype='interval[float64]')
```

The `closed` parameter specifies which endpoints of the individual intervals within the `IntervalIndex` are closed.

```
>>> pd.interval_range(end=5, periods=4, closed='both')
IntervalIndex([[1, 2], [2, 3], [3, 4], [4, 5]],
              closed='both', dtype='interval[int64]')
```

3.2.6 Top-level evaluation

eval(expr[, parser, truediv, local_dict, . . .])	Evaluate a Python expression as a string using various backends.

pandas.eval

pandas.**eval**(*expr*, *parser='pandas'*, *engine: Union[str, NoneType] = None*, *truediv=<object object at 0x7f5374a06320>*, *local_dict=None*, *global_dict=None*, *resolvers=()*, *level=0*, *target=None*, *inplace=False*)

Evaluate a Python expression as a string using various backends.

The following arithmetic operations are supported: `+`, `-`, `*`, `/`, `**`, `%`, `//` (python engine only) along with the following boolean operations: `|` (or), `&` (and), and `~` (not). Additionally, the `'pandas'` parser allows the use of `and`, `or`, and `not` with the same semantics as the corresponding bitwise operators. `Series` and `DataFrame` objects are supported and behave as they would with plain ol' Python evaluation.

Parameters

expr [str] The expression to evaluate. This string cannot contain any Python statements, only Python expressions.

parser [{'pandas', 'python'}, default 'pandas'] The parser to use to construct the syntax tree from the expression. The default of `'pandas'` parses code slightly different than standard Python. Alternatively, you can parse an expression using the `'python'` parser to retain strict Python semantics. See the *enhancing performance* documentation for more details.

engine [{'python', 'numexpr'}, default 'numexpr'] The engine used to evaluate the expression. Supported engines are

- None : tries to use `numexpr`, falls back to `python`
- **`'numexpr'`: This default engine evaluates pandas objects using** numexpr for large speed ups in complex expressions with large frames.
- **`'python'`: Performs operations as if you had `eval`'d in top** level python. This engine is generally not that useful.

More backends may be available in the future.

truediv [bool, optional] Whether to use true division, like in Python >= 3. deprecated:: 1.0.0

local_dict [dict or None, optional] A dictionary of local variables, taken from locals() by default.

global_dict [dict or None, optional] A dictionary of global variables, taken from globals() by default.

resolvers [list of dict-like or None, optional] A list of objects implementing the __`getitem`__ special method that you can use to inject an additional collection of namespaces to use for variable lookup. For example, this is used in the *`query()`* method to inject the `DataFrame.index` and `DataFrame.columns` variables that refer to their respective *`DataFrame`* instance attributes.

level [int, optional] The number of prior stack frames to traverse and add to the current scope. Most users will **not** need to change this parameter.

target [object, optional, default None] This is the target object for assignment. It is used when there is variable assignment in the expression. If so, then *target* must support item assignment with string keys, and if a copy is being returned, it must also support *.copy()*.

inplace [bool, default False] If *target* is provided, and the expression mutates *target*, whether to modify *target* inplace. Otherwise, return a copy of *target* with the mutation.

Returns

ndarray, numeric scalar, DataFrame, Series

Raises

ValueError There are many instances where such an error can be raised:

- *target=None*, but the expression is multiline.
- The expression is multiline, but not all them have item assignment. An example of such an arrangement is this:

  a = b + 1 a + 2

  Here, there are expressions on different lines, making it multiline, but the last line has no variable assigned to the output of *a + 2*.
- *inplace=True*, but the expression is missing item assignment.
- Item assignment is provided, but the *target* does not support string item assignment.
- Item assignment is provided and *inplace=False*, but the *target* does not support the *.copy()* method

See also:

`DataFrame.query`

`DataFrame.eval`

Notes

The `dtype` of any objects involved in an arithmetic `%` operation are recursively cast to `float64`.

See the *enhancing performance* documentation for more details.

3.2.7 Hashing

`util.hash_array`(vals, encoding, hash_key, …)	Given a 1d array, return an array of deterministic integers.
`util.hash_pandas_object`(obj, index, …)	Return a data hash of the Index/Series/DataFrame.

pandas.util.hash_array

`pandas.util.`**`hash_array`**(*vals*, *encoding: str = 'utf8'*, *hash_key: str = '0123456789123456'*, *categorize: bool = True*)

Given a 1d array, return an array of deterministic integers.

Parameters

vals [ndarray, Categorical]

encoding [str, default 'utf8'] Encoding for data & key when strings.

hash_key [str, default _default_hash_key] Hash_key for string key to encode.

categorize [bool, default True] Whether to first categorize object arrays before hashing. This is more efficient when the array contains duplicate values.

Returns

1d uint64 numpy array of hash values, same length as the vals

pandas.util.hash_pandas_object

`pandas.util.`**`hash_pandas_object`**(*obj*, *index: bool = True*, *encoding: str = 'utf8'*, *hash_key: Union[str, NoneType] = '0123456789123456'*, *categorize: bool = True*)

Return a data hash of the Index/Series/DataFrame.

Parameters

index [bool, default True] Include the index in the hash (if Series/DataFrame).

encoding [str, default 'utf8'] Encoding for data & key when strings.

hash_key [str, default _default_hash_key] Hash_key for string key to encode.

categorize [bool, default True] Whether to first categorize object arrays before hashing. This is more efficient when the array contains duplicate values.

Returns

Series of uint64, same length as the object

3.2.8 Testing

`test`([extra_args])	

pandas.test

`pandas.test`(*extra_args=None*)

3.3 Series

3.3.1 Constructor

`Series`([data, index, dtype, name, copy, . . .])	One-dimensional ndarray with axis labels (including time series).

pandas.Series

class `pandas.Series`(*data=None*, *index=None*, *dtype=None*, *name=None*, *copy=False*, *fastpath=False*)

One-dimensional ndarray with axis labels (including time series).

Labels need not be unique but must be a hashable type. The object supports both integer- and label-based indexing and provides a host of methods for performing operations involving the index. Statistical methods from ndarray have been overridden to automatically exclude missing data (currently represented as NaN).

Operations between Series (+, -, /, , *) align values based on their associated index values– they need not be the same length. The result index will be the sorted union of the two indexes.

Parameters

data [array-like, Iterable, dict, or scalar value] Contains data stored in Series.

Changed in version 0.23.0: If data is a dict, argument order is maintained for Python 3.6 and later.

index [array-like or Index (1d)] Values must be hashable and have the same length as *data*. Non-unique index values are allowed. Will default to RangeIndex (0, 1, 2, . . . , n) if not provided. If both a dict and index sequence are used, the index will override the keys found in the dict.

dtype [str, numpy.dtype, or ExtensionDtype, optional] Data type for the output Series. If not specified, this will be inferred from *data*. See the *user guide* for more usages.

name [str, optional] The name to give to the Series.

copy [bool, default False] Copy input data.

Attributes

T	Return the transpose, which is by definition self.
array	The ExtensionArray of the data backing this Series or Index.
at	Access a single value for a row/column label pair.
attrs	Dictionary of global attributes on this object.
axes	Return a list of the row axis labels.
dtype	Return the dtype object of the underlying data.
dtypes	Return the dtype object of the underlying data.
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.
index	The index (axis labels) of the Series.
is_monotonic	Return boolean if values in the object are monotonic_increasing.
is_monotonic_decreasing	Return boolean if values in the object are monotonic_decreasing.
is_monotonic_increasing	Return boolean if values in the object are monotonic_increasing.
is_unique	Return boolean if values in the object are unique.
loc	Access a group of rows and columns by label(s) or a boolean array.
nbytes	Return the number of bytes in the underlying data.
ndim	Number of dimensions of the underlying data, by definition 1.
shape	Return a tuple of the shape of the underlying data.
size	Return the number of elements in the underlying data.
values	Return Series as ndarray or ndarray-like depending on the dtype.

pandas.Series.T

property Series.**T**

Return the transpose, which is by definition self.

pandas.Series.array

property Series.**array**

The ExtensionArray of the data backing this Series or Index.

New in version 0.24.0.

Returns

ExtensionArray An ExtensionArray of the values stored within. For extension types, this is the actual array. For NumPy native types, this is a thin (no copy) wrapper around `numpy.ndarray`.

.array differs .values which may require converting the data to a different form.

See also:

Index.to_numpy Similar method that always returns a NumPy array.

Series.to_numpy Similar method that always returns a NumPy array.

Notes

This table lays out the different array types for each extension dtype within pandas.

dtype	array type
category	Categorical
period	PeriodArray
interval	IntervalArray
IntegerNA	IntegerArray
string	StringArray
boolean	BooleanArray
datetime64[ns, tz]	DatetimeArray

For any 3rd-party extension types, the array type will be an ExtensionArray.

For all remaining dtypes .array will be a arrays.NumpyExtensionArray wrapping the actual ndarray stored within. If you absolutely need a NumPy array (possibly with copying / coercing data), then use *Series.to_numpy()* instead.

Examples

For regular NumPy types like int, and float, a PandasArray is returned.

```
>>> pd.Series([1, 2, 3]).array
<PandasArray>
[1, 2, 3]
Length: 3, dtype: int64
```

For extension types, like Categorical, the actual ExtensionArray is returned

```
>>> ser = pd.Series(pd.Categorical(['a', 'b', 'a']))
>>> ser.array
[a, b, a]
Categories (2, object): [a, b]
```

pandas.Series.at

property Series.**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.Series.attrs

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

Warning: attrs is experimental and may change without warning.

pandas.Series.axes

property Series.**axes**
Return a list of the row axis labels.

pandas.Series.dtype

property Series.**dtype**

Return the dtype object of the underlying data.

pandas.Series.dtypes

property Series.**dtypes**

Return the dtype object of the underlying data.

pandas.Series.hasnans

property Series.**hasnans**

Return if I have any nans; enables various perf speedups.

pandas.Series.iat

property Series.**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.Series.iloc

property `Series.`**`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.Series.index

Series.**index**

The index (axis labels) of the Series.

pandas.Series.is_monotonic

property Series.**is_monotonic**

Return boolean if values in the object are monotonic_increasing.

Returns

bool

pandas.Series.is_monotonic_decreasing

property Series.**is_monotonic_decreasing**

Return boolean if values in the object are monotonic_decreasing.

Returns

bool

pandas.Series.is_monotonic_increasing

property `Series.`**`is_monotonic_increasing`**

Return boolean if values in the object are monotonic_increasing.

Returns

bool

pandas.Series.is_unique

property `Series.`**`is_unique`**

Return boolean if values in the object are unique.

Returns

bool

pandas.Series.loc

property `Series.`**`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
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
```

pandas.Series.nbytes

property Series.**nbytes**
Return the number of bytes in the underlying data.

pandas.Series.ndim

property Series.**ndim**
Number of dimensions of the underlying data, by definition 1.

pandas.Series.shape

property Series.**shape**
Return a tuple of the shape of the underlying data.

pandas.Series.size

property Series.**size**
Return the number of elements in the underlying data.

pandas.Series.values

property Series.**values**

Return Series as ndarray or ndarray-like depending on the dtype.

> **Warning:** We recommend using *Series.array* or *Series.to_numpy()*, depending on whether you need a reference to the underlying data or a NumPy array.

Returns

numpy.ndarray or ndarray-like

See also:

Series.array Reference to the underlying data.

Series.to_numpy A NumPy array representing the underlying data.

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]')
```

empty	
name	

Methods

abs(self)	Return a Series/DataFrame with absolute numeric value of each element.
add(self, other[, level, fill_value, axis])	Return Addition of series 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*.

continues on next page

Table 29 – continued from previous page

`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, to_append[, ignore_index, . . .])	Concatenate two or more Series.
`apply`(self, func[, convert_dtype, args])	Invoke function on values of Series.
`argmax`(self[, axis, skipna])	Return an ndarray of the maximum argument indexer.
`argmin`(self[, axis, skipna])	Return a ndarray of the minimum argument indexer.
`argsort`(self[, axis, kind, order])	Override ndarray.argsort.
`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*.
`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.
`autocorr`(self[, lag])	Compute the lag-N autocorrelation.
`between`(self, left, right[, inclusive])	Return boolean Series equivalent to left <= series <= right.
`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.
`cat`	alias of `pandas.core.arrays.categorical.CategoricalAccessor`
`clip`(self[, lower, upper, axis])	Trim values at input threshold(s).
`combine`(self, other, func[, fill_value])	Combine the Series with a Series or scalar according to *func*.
`combine_first`(self, other)	Combine Series values, choosing the calling Series's values first.
`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, other[, method, min_periods])	Compute correlation with *other* Series, excluding missing values.
`count`(self[, level])	Return number of non-NA/null observations in the Series.
`cov`(self, other[, min_periods])	Compute covariance with Series, excluding missing 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.

continues on next page

Table 29 – continued from previous page

`cumsum`(self[, axis, skipna])	Return cumulative sum over a DataFrame or Series axis.
`describe`(self[, percentiles, include, exclude])	Generate descriptive statistics.
`diff`(self[, periods])	First discrete difference of element.
`div`(self, other[, level, fill_value, axis])	Return Floating division of series and other, element-wise (binary operator *truediv*).
`divide`(self, other[, level, fill_value, axis])	Return Floating division of series and other, element-wise (binary operator *truediv*).
`divmod`(self, other[, level, fill_value, axis])	Return Integer division and modulo of series and other, element-wise (binary operator *divmod*).
`dot`(self, other)	Compute the dot product between the Series and the columns of other.
`drop`(self[, labels, axis, index, columns, . . .])	Return Series with specified index labels removed.
`drop_duplicates`(self[, keep, inplace])	Return Series with duplicate values removed.
`droplevel`(self, level[, axis])	Return DataFrame with requested index / column level(s) removed.
`dropna`(self[, axis, inplace, how])	Return a new Series with missing values removed.
`dt`	alias of `pandas.core.indexes.accessors.CombinedDatetimelikeProperties`
`duplicated`(self[, keep])	Indicate duplicate Series values.
`eq`(self, other[, level, fill_value, axis])	Return Equal to of series and other, element-wise (binary operator *eq*).
`equals`(self, other)	Test whether two objects contain the same elements.
`ewm`(self[, com, span, halflife, alpha, . . .])	Provide exponential weighted functions.
`expanding`(self[, min_periods, center, axis])	Provide expanding transformations.
`explode`(self)	Transform each element of a list-like to a row, replicating the index values.
`factorize`(self[, sort, na_sentinel])	Encode the object as an enumerated type or categorical variable.
`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.
`floordiv`(self, other[, level, fill_value, axis])	Return Integer division of series and other, element-wise (binary operator *floordiv*).
`ge`(self, other[, level, fill_value, axis])	Return Greater than or equal to of series 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 Series using a mapper or by a Series of columns.
`gt`(self, other[, level, fill_value, axis])	Return Greater than of series and other, element-wise (binary operator *gt*).
`head`(self, n)	Return the first *n* rows.
`hist`(self[, by, ax, grid, xlabelsize, xrot, . . .])	Draw histogram of the input series using matplotlib.
`idxmax`(self[, axis, skipna])	Return the row label of the maximum value.

continues on next page

Table 29 – continued from previous page

idxmin(self[, axis, skipna])	Return the row label of the minimum value.
infer_objects(self)	Attempt to infer better dtypes for object columns.
interpolate(self[, method, axis, limit, . . .])	Interpolate values according to different methods.
isin(self, values)	Check whether *values* are contained in Series.
isna(self)	Detect missing values.
isnull(self)	Detect missing values.
item(self)	Return the first element of the underlying data as a python scalar.
items(self)	Lazily iterate over (index, value) tuples.
iteritems(self)	Lazily iterate over (index, value) tuples.
keys(self)	Return alias for index.
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[, level, fill_value, axis])	Return Less than or equal to of series and other, element-wise (binary operator *le*).
lt(self, other[, level, fill_value, axis])	Return Less than of series and other, element-wise (binary operator *lt*).
mad(self[, axis, skipna, level])	Return the mean absolute deviation of the values for the requested axis.
map(self, arg[, na_action])	Map values of Series according to input correspondence.
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.
memory_usage(self[, index, deep])	Return the memory usage of the Series.
min(self[, axis, skipna, level, numeric_only])	Return the minimum of the values for the requested axis.
mod(self, other[, level, fill_value, axis])	Return Modulo of series and other, element-wise (binary operator *mod*).
mode(self[, dropna])	Return the mode(s) of the dataset.
mul(self, other[, level, fill_value, axis])	Return Multiplication of series and other, element-wise (binary operator *mul*).
multiply(self, other[, level, fill_value, axis])	Return Multiplication of series and other, element-wise (binary operator *mul*).
ne(self, other[, level, fill_value, axis])	Return Not equal to of series and other, element-wise (binary operator *ne*).
nlargest(self[, n, keep])	Return the largest *n* elements.
notna(self)	Detect existing (non-missing) values.
notnull(self)	Detect existing (non-missing) values.
nsmallest(self[, n, keep])	Return the smallest *n* elements.
nunique(self[, dropna])	Return number of unique elements in the object.
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).

continues on next page

Table 29 – continued from previous page

`plot`	alias of `pandas.plotting._core.PlotAccessor`
`pop`(self, item)	Return item and drop from frame.
`pow`(self, other[, level, fill_value, axis])	Return Exponential power of series 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, interpolation])	Return value at the given quantile.
`radd`(self, other[, level, fill_value, axis])	Return Addition of series and other, element-wise (binary operator *radd*).
`rank`(self[, axis])	Compute numerical data ranks (1 through n) along axis.
`ravel`(self[, order])	Return the flattened underlying data as an ndarray.
`rdiv`(self, other[, level, fill_value, axis])	Return Floating division of series and other, element-wise (binary operator *rtruediv*).
`rdivmod`(self, other[, level, fill_value, axis])	Return Integer division and modulo of series and other, element-wise (binary operator *rdivmod*).
`reindex`(self[, index])	Conform Series to new index with optional filling logic.
`reindex_like`(self, other, method, . . . [, . . .])	Return an object with matching indices as other object.
`rename`(self[, index, axis, copy, inplace, . . .])	Alter Series index labels or name.
`rename_axis`(self[, mapper, index, columns, . . .])	Set the name of the axis for the index or columns.
`reorder_levels`(self, order)	Rearrange index levels using input order.
`repeat`(self, repeats[, axis])	Repeat elements of a Series.
`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, drop, name, inplace])	Generate a new DataFrame or Series with the index reset.
`rfloordiv`(self, other[, level, fill_value, axis])	Return Integer division of series and other, element-wise (binary operator *rfloordiv*).
`rmod`(self, other[, level, fill_value, axis])	Return Modulo of series and other, element-wise (binary operator *rmod*).
`rmul`(self, other[, level, fill_value, axis])	Return Multiplication of series and other, element-wise (binary operator *rmul*).
`rolling`(self, window[, min_periods, center, . . .])	Provide rolling window calculations.
`round`(self[, decimals])	Round each value in a Series to the given number of decimals.
`rpow`(self, other[, level, fill_value, axis])	Return Exponential power of series and other, element-wise (binary operator *rpow*).
`rsub`(self, other[, level, fill_value, axis])	Return Subtraction of series and other, element-wise (binary operator *rsub*).
`rtruediv`(self, other[, level, fill_value, axis])	Return Floating division of series and other, element-wise (binary operator *rtruediv*).
`sample`(self[, n, frac, replace, weights, . . .])	Return a random sample of items from an axis of object.
`searchsorted`(self, value[, side, sorter])	Find indices where elements should be inserted to maintain order.

continues on next page

Table 29 – continued from previous page

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.
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 Series by index labels.
sort_values(self[, axis, ascending, …])	Sort by the values.
sparse	alias of pandas.core.arrays.sparse.accessor.SparseAccessor
squeeze(self[, axis])	Squeeze 1 dimensional axis objects into scalars.
std(self[, axis, skipna, level, ddof, …])	Return sample standard deviation over requested axis.
str	alias of pandas.core.strings.StringMethods
sub(self, other[, level, fill_value, axis])	Return Subtraction of series and other, element-wise (binary operator *sub*).
subtract(self, other[, level, fill_value, axis])	Return Subtraction of series and other, element-wise (binary operator *sub*).
sum(self[, axis, skipna, level, …])	Return the sum of the values for the requested axis.
swapaxes(self, axis1, axis2[, copy])	Interchange axes and swap values axes appropriately.
swaplevel(self[, i, j, copy])	Swap levels i and j in a MultiIndex.
tail(self, n)	Return the last *n* rows.
take(self, indices[, axis, is_copy])	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[, into])	Convert Series to {label -> value} dict or dict-like object.
to_excel(self, excel_writer[, sheet_name, …])	Write object to an Excel sheet.
to_frame(self[, name])	Convert Series to DataFrame.
to_hdf(self, path_or_buf, key, mode, …[, …])	Write the contained data to an HDF5 file using HDFStore.
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_list(self)	Return a list of the values.
to_markdown(self, buf, NoneType] = None, …)	Print Series in Markdown-friendly format.
to_numpy(self[, dtype, copy, na_value])	A NumPy ndarray representing the values in this Series or Index.
to_period(self[, freq, copy])	Convert Series from DatetimeIndex to PeriodIndex with desired frequency (inferred from index if not passed).
to_pickle(self, path, compression, …)	Pickle (serialize) object to file.
to_sql(self, name, con[, schema, …])	Write records stored in a DataFrame to a SQL database.
to_string(self[, buf, na_rep, float_format, …])	Render a string representation of the Series.
to_timestamp(self[, freq, how, copy])	Cast to DatetimeIndex of Timestamps, at *beginning* of period.
to_xarray(self)	Return an xarray object from the pandas object.

continues on next page

Table 29 – continued from previous page

`tolist`(self)	Return a list of the values.
`transform`(self, func[, axis])	Call `func` on self producing a Series with transformed values.
`transpose`(self, *args, **kwargs)	Return the transpose, which is by definition self.
`truediv`(self, other[, level, fill_value, axis])	Return Floating division of series 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.
`unique`(self)	Return unique values of Series object.
`unstack`(self[, level, fill_value])	Unstack, a.k.a.
`update`(self, other)	Modify Series in place using non-NA values from passed Series.
`value_counts`(self[, normalize, sort, . . .])	Return a Series containing counts of unique values.
`var`(self[, axis, skipna, level, ddof, . . .])	Return unbiased variance over requested axis.
`view`(self[, dtype])	Create a new view of the Series.
`where`(self, cond[, other, inplace, axis, . . .])	Replace values where the condition is False.
`xs`(self, key[, axis, level])	Return cross-section from the Series/DataFrame.

pandas.Series.abs

`Series.`**`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]
... })
>>> 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
```