Examples

```
>>> s = pd.Series(list('abca'))
```

```
>>> pd.get_dummies(s)
   a  b  c
0  1  0  0
1  0  1  0
2  0  0  1
3  1  0  0
```

```
>>> s1 = ['a', 'b', np.nan]
```

```
>>> pd.get_dummies(s1)
   a  b
0  1  0
1  0  1
2  0  0
```

```
>>> pd.get_dummies(s1, dummy_na=True)
   a  b  NaN
0  1  0    0
1  0  1    0
2  0  0    1
```

```
>>> df = pd.DataFrame({'A': ['a', 'b', 'a'], 'B': ['b', 'a', 'c'],
...                    'C': [1, 2, 3]})
```

```
>>> pd.get_dummies(df, prefix=['col1', 'col2'])
   C  col1_a  col1_b  col2_a  col2_b  col2_c
0  1       1       0       0       1       0
1  2       0       1       1       0       0
2  3       1       0       0       0       1
```

```
>>> pd.get_dummies(pd.Series(list('abcaa')))
   a  b  c
0  1  0  0
1  0  1  0
2  0  0  1
3  1  0  0
4  1  0  0
```

```
>>> pd.get_dummies(pd.Series(list('abcaa')), drop_first=True)
   b  c
0  0  0
1  1  0
2  0  1
3  0  0
4  0  0
```

```
>>> pd.get_dummies(pd.Series(list('abc')), dtype=float)
     a    b    c
0  1.0  0.0  0.0
```

(continues on next page)

(continued from previous page)

```
1  0.0  1.0  0.0
2  0.0  0.0  1.0
```

pandas.factorize

`pandas.factorize`(*values*, *sort: bool = False*, *na_sentinel: int = - 1*, *size_hint: Union[int, NoneType] = None*) → Tuple[numpy.ndarray, Union[numpy.ndarray, pandas.core.dtypes.generic.ABCIndex]]

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

values [sequence] A 1-D sequence. Sequences that aren't pandas objects are coerced to ndarrays before factorization.

sort [bool, default False] Sort *uniques* and shuffle *codes* to maintain the relationship.

na_sentinel [int, default -1] Value to mark "not found".

size_hint [int, optional] Hint to the hashtable sizer.

Returns

codes [ndarray] An integer ndarray that's an indexer into *uniques*. `uniques.take(codes)` 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:

`cut` Discretize continuous-valued array.

`unique` Find the unique value 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()`.

```
>>> codes, uniques = pd.factorize(['b', 'b', 'a', 'c', 'b'])
>>> codes
array([0, 0, 1, 2, 0])
>>> uniques
array(['b', 'a', 'c'], dtype=object)
```

With `sort=True`, the *uniques* will be sorted, and *codes* will be shuffled so that the relationship is the maintained.

```
>>> codes, uniques = pd.factorize(['b', 'b', 'a', 'c', 'b'], sort=True)
>>> codes
array([1, 1, 0, 2, 1])
>>> uniques
array(['a', 'b', 'c'], dtype=object)
```

Missing values are indicated in *codes* with *na_sentinel* (`-1` by default). Note that missing values are never included in *uniques*.

```
>>> codes, uniques = pd.factorize(['b', None, 'a', 'c', 'b'])
>>> codes
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'])
>>> codes, uniques = pd.factorize(cat)
>>> codes
array([0, 0, 1])
>>> uniques
[a, c]
Categories (3, object): [a, b, c]
```

Notice that `'b'` is in `uniques.categories`, despite 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'])
>>> codes, uniques = pd.factorize(cat)
>>> codes
array([0, 0, 1])
>>> uniques
Index(['a', 'c'], dtype='object')
```

pandas.unique

`pandas.`**`unique`**`(`*values*`)`

Hash table-based unique. Uniques are returned in order of appearance. This does NOT sort.

Significantly faster than numpy.unique. Includes NA values.

Parameters

values [1d array-like]

Returns

numpy.ndarray or ExtensionArray The return can be:

- Index : when the input is an Index
- Categorical : when the input is a Categorical dtype
- ndarray : when the input is a Series/ndarray

Return numpy.ndarray or ExtensionArray.

See also:

Index.unique

Series.unique

Examples

```
>>> pd.unique(pd.Series([2, 1, 3, 3]))
array([2, 1, 3])
```

```
>>> pd.unique(pd.Series([2] + [1] * 5))
array([2, 1])
```

```
>>> pd.unique(pd.Series([pd.Timestamp('20160101'),
...                     pd.Timestamp('20160101')]))
array(['2016-01-01T00:00:00.000000000'], dtype='datetime64[ns]')
```

```
>>> pd.unique(pd.Series([pd.Timestamp('20160101', tz='US/Eastern'),
...                      pd.Timestamp('20160101', tz='US/Eastern')]))
array([Timestamp('2016-01-01 00:00:00-0500', tz='US/Eastern')],
      dtype=object)
```

```
>>> pd.unique(pd.Index([pd.Timestamp('20160101', tz='US/Eastern'),
...                     pd.Timestamp('20160101', tz='US/Eastern')]))
DatetimeIndex(['2016-01-01 00:00:00-05:00'],
...           dtype='datetime64[ns, US/Eastern]', freq=None)
```

```
>>> pd.unique(list('baabc'))
array(['b', 'a', 'c'], dtype=object)
```

An unordered Categorical will return categories in the order of appearance.

```
>>> pd.unique(pd.Series(pd.Categorical(list('baabc'))))
[b, a, c]
Categories (3, object): [b, a, c]
```

```
>>> pd.unique(pd.Series(pd.Categorical(list('baabc'),
...                                    categories=list('abc'))))
[b, a, c]
Categories (3, object): [b, a, c]
```

An ordered Categorical preserves the category ordering.

```
>>> pd.unique(pd.Series(pd.Categorical(list('baabc'),
...                                    categories=list('abc'),
...                                    ordered=True)))
[b, a, c]
Categories (3, object): [a < b < c]
```

An array of tuples

```
>>> pd.unique([('a', 'b'), ('b', 'a'), ('a', 'c'), ('b', 'a')])
array([('a', 'b'), ('b', 'a'), ('a', 'c')], dtype=object)
```

pandas.wide_to_long

`pandas.wide_to_long`(*df: pandas.core.frame.DataFrame*, *stubnames*, *i*, *j*, *sep: str = ''*, *suffix: str = '\\d+'*) → pandas.core.frame.DataFrame

Wide panel to long format. Less flexible but more user-friendly than melt.

With stubnames ['A', 'B'], this function expects to find one or more group of columns with format A-suffix1, A-suffix2,..., B-suffix1, B-suffix2,... You specify what you want to call this suffix in the resulting long format with *j* (for example *j='year'*)

Each row of these wide variables are assumed to be uniquely identified by *i* (can be a single column name or a list of column names)

All remaining variables in the data frame are left intact.

Parameters

df [DataFrame] The wide-format DataFrame.

stubnames [str or list-like] The stub name(s). The wide format variables are assumed to start with the stub names.

i [str or list-like] Column(s) to use as id variable(s).

j [str] The name of the sub-observation variable. What you wish to name your suffix in the long format.

sep [str, default ""] A character indicating the separation of the variable names in the wide format, to be stripped from the names in the long format. For example, if your column names are A-suffix1, A-suffix2, you can strip the hyphen by specifying *sep='-'*.

suffix [str, default '\d+'] A regular expression capturing the wanted suffixes. '\d+' captures numeric suffixes. Suffixes with no numbers could be specified with the negated character class '\D+'. You can also further disambiguate suffixes, for example, if your wide variables are of the form A-one, B-two,.., and you have an unrelated column A-rating, you can ignore the last one by specifying *suffix='(!?one|two)'*.

Changed in version 0.23.0: When all suffixes are numeric, they are cast to int64/float64.

Returns

DataFrame A DataFrame that contains each stub name as a variable, with new index (i, j).

Notes

All extra variables are left untouched. This simply uses *pandas.melt* under the hood, but is hard-coded to "do the right thing" in a typical case.

Examples

```
>>> np.random.seed(123)
>>> df = pd.DataFrame({"A1970" : {0 : "a", 1 : "b", 2 : "c"},
...                    "A1980" : {0 : "d", 1 : "e", 2 : "f"},
...                    "B1970" : {0 : 2.5, 1 : 1.2, 2 : .7},
...                    "B1980" : {0 : 3.2, 1 : 1.3, 2 : .1},
...                    "X"     : dict(zip(range(3), np.random.randn(3)))
...                   })
>>> df["id"] = df.index
>>> df
```

(continues on next page)

(continued from previous page)

```
  A1970 A1980  B1970  B1980         X  id
0     a     d    2.5    3.2 -1.085631   0
1     b     e    1.2    1.3  0.997345   1
2     c     f    0.7    0.1  0.282978   2
>>> pd.wide_to_long(df, ["A", "B"], i="id", j="year")
...
                X  A    B
id year
0  1970 -1.085631  a  2.5
1  1970  0.997345  b  1.2
2  1970  0.282978  c  0.7
0  1980 -1.085631  d  3.2
1  1980  0.997345  e  1.3
2  1980  0.282978  f  0.1
```

With multiple id columns

```
>>> df = pd.DataFrame({
...     'famid': [1, 1, 1, 2, 2, 2, 3, 3, 3],
...     'birth': [1, 2, 3, 1, 2, 3, 1, 2, 3],
...     'ht1': [2.8, 2.9, 2.2, 2, 1.8, 1.9, 2.2, 2.3, 2.1],
...     'ht2': [3.4, 3.8, 2.9, 3.2, 2.8, 2.4, 3.3, 3.4, 2.9]
... })
>>> df
   famid  birth  ht1  ht2
0      1      1  2.8  3.4
1      1      2  2.9  3.8
2      1      3  2.2  2.9
3      2      1  2.0  3.2
4      2      2  1.8  2.8
5      2      3  1.9  2.4
6      3      1  2.2  3.3
7      3      2  2.3  3.4
8      3      3  2.1  2.9
>>> l = pd.wide_to_long(df, stubnames='ht', i=['famid', 'birth'], j='age')
>>> l
...
                  ht
famid birth age
1     1     1    2.8
            2    3.4
      2     1    2.9
            2    3.8
      3     1    2.2
            2    2.9
2     1     1    2.0
            2    3.2
      2     1    1.8
            2    2.8
      3     1    1.9
            2    2.4
3     1     1    2.2
            2    3.3
      2     1    2.3
            2    3.4
      3     1    2.1
            2    2.9
```

Going from long back to wide just takes some creative use of *unstack*

```
>>> w = l.unstack()
>>> w.columns = w.columns.map('{0[0]}{0[1]}'.format)
>>> w.reset_index()
   famid  birth  ht1  ht2
0      1      1  2.8  3.4
1      1      2  2.9  3.8
2      1      3  2.2  2.9
3      2      1  2.0  3.2
4      2      2  1.8  2.8
5      2      3  1.9  2.4
6      3      1  2.2  3.3
7      3      2  2.3  3.4
8      3      3  2.1  2.9
```

Less wieldy column names are also handled

```
>>> np.random.seed(0)
>>> df = pd.DataFrame({'A(weekly)-2010': np.random.rand(3),
...                    'A(weekly)-2011': np.random.rand(3),
...                    'B(weekly)-2010': np.random.rand(3),
...                    'B(weekly)-2011': np.random.rand(3),
...                    'X' : np.random.randint(3, size=3)})
>>> df['id'] = df.index
>>> df
   A(weekly)-2010  A(weekly)-2011  B(weekly)-2010  B(weekly)-2011  X  id
0        0.548814        0.544883        0.437587        0.383442  0   0
1        0.715189        0.423655        0.891773        0.791725  1   1
2        0.602763        0.645894        0.963663        0.528895  1   2
```

```
>>> pd.wide_to_long(df, ['A(weekly)', 'B(weekly)'], i='id',
...                 j='year', sep='-')
...
         X  A(weekly)  B(weekly)
id year
0  2010  0   0.548814   0.437587
1  2010  1   0.715189   0.891773
2  2010  1   0.602763   0.963663
0  2011  0   0.544883   0.383442
1  2011  1   0.423655   0.791725
2  2011  1   0.645894   0.528895
```

If we have many columns, we could also use a regex to find our stubnames and pass that list on to wide_to_long

```
>>> stubnames = sorted(
...     set([match[0] for match in df.columns.str.findall(
...         r'[A-B]\(.*\)').values if match != []])
... )
>>> list(stubnames)
['A(weekly)', 'B(weekly)']
```

All of the above examples have integers as suffixes. It is possible to have non-integers as suffixes.

```
>>> df = pd.DataFrame({
...     'famid': [1, 1, 1, 2, 2, 2, 3, 3, 3],
...     'birth': [1, 2, 3, 1, 2, 3, 1, 2, 3],
...     'ht_one': [2.8, 2.9, 2.2, 2, 1.8, 1.9, 2.2, 2.3, 2.1],
```

(continues on next page)

(continued from previous page)

```
...     'ht_two': [3.4, 3.8, 2.9, 3.2, 2.8, 2.4, 3.3, 3.4, 2.9]
... })
>>> df
   famid  birth  ht_one  ht_two
0      1      1     2.8     3.4
1      1      2     2.9     3.8
2      1      3     2.2     2.9
3      2      1     2.0     3.2
4      2      2     1.8     2.8
5      2      3     1.9     2.4
6      3      1     2.2     3.3
7      3      2     2.3     3.4
8      3      3     2.1     2.9
```

```
>>> l = pd.wide_to_long(df, stubnames='ht', i=['famid', 'birth'], j='age',
...                     sep='_', suffix='\w+')
>>> l
...
                  ht
famid birth age
1     1     one  2.8
            two  3.4
      2     one  2.9
            two  3.8
      3     one  2.2
            two  2.9
2     1     one  2.0
            two  3.2
      2     one  1.8
            two  2.8
      3     one  1.9
            two  2.4
3     1     one  2.2
            two  3.3
      2     one  2.3
            two  3.4
      3     one  2.1
            two  2.9
```

3.2.2 Top-level missing data

isna(obj)	Detect missing values for an array-like object.
isnull(obj)	Detect missing values for an array-like object.
notna(obj)	Detect non-missing values for an array-like object.
notnull(obj)	Detect non-missing values for an array-like object.

pandas.isna

`pandas.isna(obj)`

Detect missing values for an array-like object.

This function takes a scalar or array-like object and indicates whether values are missing (`NaN` in numeric arrays, `None` or `NaN` in object arrays, `NaT` in datetimelike).

Parameters

obj [scalar or array-like] Object to check for null or missing values.

Returns

bool or array-like of bool For scalar input, returns a scalar boolean. For array input, returns an array of boolean indicating whether each corresponding element is missing.

See also:

notna Boolean inverse of pandas.isna.

Series.isna Detect missing values in a Series.

DataFrame.isna Detect missing values in a DataFrame.

Index.isna Detect missing values in an Index.

Examples

Scalar arguments (including strings) result in a scalar boolean.

```
>>> pd.isna('dog')
False
```

```
>>> pd.isna(pd.NA)
True
```

```
>>> pd.isna(np.nan)
True
```

ndarrays result in an ndarray of booleans.

```
>>> array = np.array([[1, np.nan, 3], [4, 5, np.nan]])
>>> array
array([[ 1., nan,  3.],
       [ 4.,  5., nan]])
>>> pd.isna(array)
array([[False,  True, False],
       [False, False,  True]])
```

For indexes, an ndarray of booleans is returned.

```
>>> index = pd.DatetimeIndex(["2017-07-05", "2017-07-06", None,
...                           "2017-07-08"])
>>> index
DatetimeIndex(['2017-07-05', '2017-07-06', 'NaT', '2017-07-08'],
              dtype='datetime64[ns]', freq=None)
>>> pd.isna(index)
array([False, False,  True, False])
```

For Series and DataFrame, the same type is returned, containing booleans.

```
>>> df = pd.DataFrame([['ant', 'bee', 'cat'], ['dog', None, 'fly']])
>>> df
     0     1    2
0  ant   bee  cat
1  dog  None  fly
>>> pd.isna(df)
       0      1      2
0  False  False  False
1  False   True  False
```

```
>>> pd.isna(df[1])
0    False
1     True
Name: 1, dtype: bool
```

pandas.isnull

`pandas.isnull(obj)`

Detect missing values for an array-like object.

This function takes a scalar or array-like object and indicates whether values are missing (`NaN` in numeric arrays, `None` or `NaN` in object arrays, `NaT` in datetimelike).

Parameters

obj [scalar or array-like] Object to check for null or missing values.

Returns

bool or array-like of bool For scalar input, returns a scalar boolean. For array input, returns an array of boolean indicating whether each corresponding element is missing.

See also:

notna Boolean inverse of pandas.isna.

Series.isna Detect missing values in a Series.

DataFrame.isna Detect missing values in a DataFrame.

Index.isna Detect missing values in an Index.

Examples

Scalar arguments (including strings) result in a scalar boolean.

```
>>> pd.isna('dog')
False
```

```
>>> pd.isna(pd.NA)
True
```

```
>>> pd.isna(np.nan)
True
```

ndarrays result in an ndarray of booleans.

```
>>> array = np.array([[1, np.nan, 3], [4, 5, np.nan]])
>>> array
array([[ 1., nan,  3.],
       [ 4.,  5., nan]])
>>> pd.isna(array)
array([[False,  True, False],
       [False, False,  True]])
```

For indexes, an ndarray of booleans is returned.

```
>>> index = pd.DatetimeIndex(["2017-07-05", "2017-07-06", None,
...                           "2017-07-08"])
>>> index
DatetimeIndex(['2017-07-05', '2017-07-06', 'NaT', '2017-07-08'],
              dtype='datetime64[ns]', freq=None)
>>> pd.isna(index)
array([False, False,  True, False])
```

For Series and DataFrame, the same type is returned, containing booleans.

```
>>> df = pd.DataFrame([['ant', 'bee', 'cat'], ['dog', None, 'fly']])
>>> df
     0     1    2
0  ant   bee  cat
1  dog  None  fly
>>> pd.isna(df)
       0      1      2
0  False  False  False
1  False   True  False
```

```
>>> pd.isna(df[1])
0    False
1     True
Name: 1, dtype: bool
```

pandas.notna

pandas.**notna**(*obj*)

Detect non-missing values for an array-like object.

This function takes a scalar or array-like object and indicates whether values are valid (not missing, which is `NaN` in numeric arrays, `None` or `NaN` in object arrays, `NaT` in datetimelike).

Parameters

obj [array-like or object value] Object to check for *not* null or *non*-missing values.

Returns

bool or array-like of bool For scalar input, returns a scalar boolean. For array input, returns an array of boolean indicating whether each corresponding element is valid.

See also:

isna Boolean inverse of pandas.notna.

Series.notna Detect valid values in a Series.

DataFrame.notna Detect valid values in a DataFrame.

Index.notna Detect valid values in an Index.

Examples

Scalar arguments (including strings) result in a scalar boolean.

```
>>> pd.notna('dog')
True
```

```
>>> pd.notna(pd.NA)
False
```

```
>>> pd.notna(np.nan)
False
```

ndarrays result in an ndarray of booleans.

```
>>> array = np.array([[1, np.nan, 3], [4, 5, np.nan]])
>>> array
array([[ 1., nan,  3.],
       [ 4.,  5., nan]])
>>> pd.notna(array)
array([[ True, False,  True],
       [ True,  True, False]])
```

For indexes, an ndarray of booleans is returned.

```
>>> index = pd.DatetimeIndex(["2017-07-05", "2017-07-06", None,
...                           "2017-07-08"])
>>> index
DatetimeIndex(['2017-07-05', '2017-07-06', 'NaT', '2017-07-08'],
              dtype='datetime64[ns]', freq=None)
>>> pd.notna(index)
array([ True,  True, False,  True])
```

For Series and DataFrame, the same type is returned, containing booleans.

```
>>> df = pd.DataFrame([['ant', 'bee', 'cat'], ['dog', None, 'fly']])
>>> df
     0     1    2
0  ant   bee  cat
1  dog  None  fly
>>> pd.notna(df)
      0      1     2
0  True   True  True
1  True  False  True
```

```
>>> pd.notna(df[1])
0     True
1    False
Name: 1, dtype: bool
```

pandas.notnull

`pandas.notnull(obj)`

Detect non-missing values for an array-like object.

This function takes a scalar or array-like object and indicates whether values are valid (not missing, which is `NaN` in numeric arrays, `None` or `NaN` in object arrays, `NaT` in datetimelike).

Parameters

obj [array-like or object value] Object to check for *not* null or *non*-missing values.

Returns

bool or array-like of bool For scalar input, returns a scalar boolean. For array input, returns an array of boolean indicating whether each corresponding element is valid.

See also:

isna Boolean inverse of pandas.notna.

Series.notna Detect valid values in a Series.

DataFrame.notna Detect valid values in a DataFrame.

Index.notna Detect valid values in an Index.

Examples

Scalar arguments (including strings) result in a scalar boolean.

```
>>> pd.notna('dog')
True
```

```
>>> pd.notna(pd.NA)
False
```

```
>>> pd.notna(np.nan)
False
```

ndarrays result in an ndarray of booleans.

```
>>> array = np.array([[1, np.nan, 3], [4, 5, np.nan]])
>>> array
array([[ 1., nan,  3.],
       [ 4.,  5., nan]])
>>> pd.notna(array)
array([[ True, False,  True],
       [ True,  True, False]])
```

For indexes, an ndarray of booleans is returned.

```
>>> index = pd.DatetimeIndex(["2017-07-05", "2017-07-06", None,
...                           "2017-07-08"])
>>> index
DatetimeIndex(['2017-07-05', '2017-07-06', 'NaT', '2017-07-08'],
              dtype='datetime64[ns]', freq=None)
>>> pd.notna(index)
array([ True,  True, False,  True])
```

For Series and DataFrame, the same type is returned, containing booleans.

```
>>> df = pd.DataFrame([['ant', 'bee', 'cat'], ['dog', None, 'fly']])
>>> df
     0     1    2
0  ant   bee  cat
1  dog  None  fly
>>> pd.notna(df)
      0      1     2
0  True   True  True
1  True  False  True
```

```
>>> pd.notna(df[1])
0     True
1    False
Name: 1, dtype: bool
```

3.2.3 Top-level conversions

to_numeric(arg[, errors, downcast])	Convert argument to a numeric type.

pandas.to_numeric

pandas.**to_numeric**(*arg*, *errors='raise'*, *downcast=None*)

Convert argument to a numeric type.

The default return dtype is *float64* or *int64* depending on the data supplied. Use the *downcast* parameter to obtain other dtypes.

Please note that precision loss may occur if really large numbers are passed in. Due to the internal limitations of *ndarray*, if numbers smaller than *-9223372036854775808* (np.iinfo(np.int64).min) or larger than *18446744073709551615* (np.iinfo(np.uint64).max) are passed in, it is very likely they will be converted to float so that they can stored in an *ndarray*. These warnings apply similarly to *Series* since it internally leverages *ndarray*.

Parameters

arg [scalar, list, tuple, 1-d array, or Series]

errors [{'ignore', 'raise', 'coerce'}, default 'raise']

- If 'raise', then invalid parsing will raise an exception.
- If 'coerce', then invalid parsing will be set as NaN.
- If 'ignore', then invalid parsing will return the input.

downcast [{'integer', 'signed', 'unsigned', 'float'}, default None] If not None, and if the data has been successfully cast to a numerical dtype (or if the data was numeric to begin with), downcast that resulting data to the smallest numerical dtype possible according to the following rules:

- 'integer' or 'signed': smallest signed int dtype (min.: np.int8)
- 'unsigned': smallest unsigned int dtype (min.: np.uint8)
- 'float': smallest float dtype (min.: np.float32)

As this behaviour is separate from the core conversion to numeric values, any errors raised during the downcasting will be surfaced regardless of the value of the 'errors' input.

In addition, downcasting will only occur if the size of the resulting data's dtype is strictly larger than the dtype it is to be cast to, so if none of the dtypes checked satisfy that specification, no downcasting will be performed on the data.

Returns

ret [numeric if parsing succeeded.] Return type depends on input. Series if Series, otherwise ndarray.

See also:

DataFrame.astype Cast argument to a specified dtype.

to_datetime Convert argument to datetime.

to_timedelta Convert argument to timedelta.

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

convert_dtypes Convert dtypes.

Examples

Take separate series and convert to numeric, coercing when told to

```
>>> s = pd.Series(['1.0', '2', -3])
>>> pd.to_numeric(s)
0    1.0
1    2.0
2   -3.0
dtype: float64
>>> pd.to_numeric(s, downcast='float')
0    1.0
1    2.0
2   -3.0
dtype: float32
>>> pd.to_numeric(s, downcast='signed')
0    1
1    2
2   -3
dtype: int8
>>> s = pd.Series(['apple', '1.0', '2', -3])
>>> pd.to_numeric(s, errors='ignore')
0    apple
1      1.0
2        2
3       -3
dtype: object
>>> pd.to_numeric(s, errors='coerce')
0    NaN
1    1.0
2    2.0
3   -3.0
dtype: float64
```

3.2.4 Top-level dealing with datetimelike

`to_datetime`(arg[, errors, dayfirst, . . .])	Convert argument to datetime.
`to_timedelta`(arg[, unit, errors])	Convert argument to timedelta.
`date_range`([start, end, periods, freq, tz, . . .])	Return a fixed frequency DatetimeIndex.
`bdate_range`([start, end, periods, freq, tz, . . .])	Return a fixed frequency DatetimeIndex, with business day as the default frequency.
`period_range`([start, end, periods, freq, name])	Return a fixed frequency PeriodIndex.
`timedelta_range`([start, end, periods, freq, . . .])	Return a fixed frequency TimedeltaIndex, with day as the default frequency.
`infer_freq`(index, warn)	Infer the most likely frequency given the input index.

pandas.to_datetime

`pandas.`**`to_datetime`**(*arg*, *errors='raise'*, *dayfirst=False*, *yearfirst=False*, *utc=None*, *format=None*, *exact=True*, *unit=None*, *infer_datetime_format=False*, *origin='unix'*, *cache=True*)

Convert argument to datetime.

Parameters

arg [int, float, str, datetime, list, tuple, 1-d array, Series DataFrame/dict-like] The object to convert to a datetime.

errors [{'ignore', 'raise', 'coerce'}, default 'raise']

- If 'raise', then invalid parsing will raise an exception.
- If 'coerce', then invalid parsing will be set as NaT.
- If 'ignore', then invalid parsing will return the input.

dayfirst [bool, default False] Specify a date parse order if *arg* is str or its list-likes. If True, parses dates with the day first, eg 10/11/12 is parsed as 2012-11-10. Warning: dayfirst=True is not strict, but will prefer to parse with day first (this is a known bug, based on dateutil behavior).

yearfirst [bool, default False] Specify a date parse order if *arg* is str or its list-likes.

- If True parses dates with the year first, eg 10/11/12 is parsed as 2010-11-12.
- If both dayfirst and yearfirst are True, yearfirst is preceded (same as dateutil).

Warning: yearfirst=True is not strict, but will prefer to parse with year first (this is a known bug, based on dateutil behavior).

utc [bool, default None] Return UTC DatetimeIndex if True (converting any tz-aware datetime.datetime objects as well).

format [str, default None] The strftime to parse time, eg "%d/%m/%Y", note that "%f" will parse all the way up to nanoseconds. See strftime documentation for more information on choices: https://docs.python.org/3/library/datetime.html#strftime-and-strptime-behavior.

exact [bool, True by default] Behaves as: - If True, require an exact format match. - If False, allow the format to match anywhere in the target string.

unit [str, default 'ns'] The unit of the arg (D,s,ms,us,ns) denote the unit, which is an integer or float number. This will be based off the origin. Example, with unit='ms' and origin='unix' (the default), this would calculate the number of milliseconds to the unix epoch start.

infer_datetime_format [bool, default False] If True and no *format* is given, attempt to infer the format of the datetime strings, and if it can be inferred, switch to a faster method of parsing them. In some cases this can increase the parsing speed by ~5-10x.

origin [scalar, default 'unix'] Define the reference date. The numeric values would be parsed as number of units (defined by *unit*) since this reference date.

- If 'unix' (or POSIX) time; origin is set to 1970-01-01.
- If 'julian', unit must be 'D', and origin is set to beginning of Julian Calendar. Julian day number 0 is assigned to the day starting at noon on January 1, 4713 BC.
- If Timestamp convertible, origin is set to Timestamp identified by origin.

cache [bool, default True] If True, use a cache of unique, converted dates to apply the datetime conversion. May produce significant speed-up when parsing duplicate date strings, especially ones with timezone offsets. The cache is only used when there are at least 50 values. The presence of out-of-bounds values will render the cache unusable and may slow down parsing.

New in version 0.23.0.

Changed in version 0.25.0: - changed default value from False to True.

Returns

datetime If parsing succeeded. Return type depends on input:

- list-like: DatetimeIndex
- Series: Series of datetime64 dtype
- scalar: Timestamp

In case when it is not possible to return designated types (e.g. when any element of input is before Timestamp.min or after Timestamp.max) return will have datetime.datetime type (or corresponding array/Series).

See also:

`DataFrame.astype` Cast argument to a specified dtype.

`to_timedelta` Convert argument to timedelta.

`convert_dtypes` Convert dtypes.

Examples

Assembling a datetime from multiple columns of a DataFrame. The keys can be common abbreviations like ['year', 'month', 'day', 'minute', 'second', 'ms', 'us', 'ns']) or plurals of the same

```
>>> df = pd.DataFrame({'year': [2015, 2016],
...                    'month': [2, 3],
...                    'day': [4, 5]})
>>> pd.to_datetime(df)
0   2015-02-04
1   2016-03-05
dtype: datetime64[ns]
```

If a date does not meet the timestamp limitations, passing errors='ignore' will return the original input instead of raising any exception.

Passing errors='coerce' will force an out-of-bounds date to NaT, in addition to forcing non-dates (or non-parseable dates) to NaT.

```
>>> pd.to_datetime('13000101', format='%Y%m%d', errors='ignore')
datetime.datetime(1300, 1, 1, 0, 0)
>>> pd.to_datetime('13000101', format='%Y%m%d', errors='coerce')
NaT
```

Passing infer_datetime_format=True can often-times speedup a parsing if its not an ISO8601 format exactly, but in a regular format.

```
>>> s = pd.Series(['3/11/2000', '3/12/2000', '3/13/2000'] * 1000)
>>> s.head()
0    3/11/2000
1    3/12/2000
2    3/13/2000
3    3/11/2000
4    3/12/2000
dtype: object
```

```
>>> %timeit pd.to_datetime(s, infer_datetime_format=True)
100 loops, best of 3: 10.4 ms per loop
```

```
>>> %timeit pd.to_datetime(s, infer_datetime_format=False)
1 loop, best of 3: 471 ms per loop
```

Using a unix epoch time

```
>>> pd.to_datetime(1490195805, unit='s')
Timestamp('2017-03-22 15:16:45')
>>> pd.to_datetime(1490195805433502912, unit='ns')
Timestamp('2017-03-22 15:16:45.433502912')
```

Warning: For float arg, precision rounding might happen. To prevent unexpected behavior use a fixed-width exact type.

Using a non-unix epoch origin

```
>>> pd.to_datetime([1, 2, 3], unit='D',
...                origin=pd.Timestamp('1960-01-01'))
DatetimeIndex(['1960-01-02', '1960-01-03', '1960-01-04'], dtype='datetime64[ns]',␣
↪freq=None)
```

pandas.to_timedelta

`pandas.to_timedelta(arg, unit='ns', errors='raise')`

Convert argument to timedelta.

Timedeltas are absolute differences in times, expressed in difference units (e.g. days, hours, minutes, seconds). This method converts an argument from a recognized timedelta format / value into a Timedelta type.

Parameters

arg [str, timedelta, list-like or Series] The data to be converted to timedelta.

unit [str, default 'ns'] Denotes the unit of the arg. Possible values: ('Y', 'M', 'W', 'D', 'days', 'day', 'hours', hour', 'hr', 'h', 'm', 'minute', 'min', 'minutes', 'T', 'S', 'seconds', 'sec', 'second', 'ms', 'milliseconds', 'millisecond', 'milli', 'millis', 'L', 'us', 'microseconds', 'microsecond', 'micro', 'micros', 'U', 'ns', 'nanoseconds', 'nano', 'nanos', 'nanosecond', 'N').

errors [{'ignore', 'raise', 'coerce'}, default 'raise']

- If 'raise', then invalid parsing will raise an exception.
- If 'coerce', then invalid parsing will be set as NaT.
- If 'ignore', then invalid parsing will return the input.

Returns

timedelta64 or numpy.array of timedelta64 Output type returned if parsing succeeded.

See also:

`DataFrame.astype` Cast argument to a specified dtype.

`to_datetime` Convert argument to datetime.

`convert_dtypes` Convert dtypes.

Examples

Parsing a single string to a Timedelta:

```
>>> pd.to_timedelta('1 days 06:05:01.00003')
Timedelta('1 days 06:05:01.000030')
>>> pd.to_timedelta('15.5us')
Timedelta('0 days 00:00:00.000015')
```

Parsing a list or array of strings:

```
>>> pd.to_timedelta(['1 days 06:05:01.00003', '15.5us', 'nan'])
TimedeltaIndex(['1 days 06:05:01.000030', '0 days 00:00:00.000015', NaT],
               dtype='timedelta64[ns]', freq=None)
```

Converting numbers by specifying the *unit* keyword argument:

```
>>> pd.to_timedelta(np.arange(5), unit='s')
TimedeltaIndex(['00:00:00', '00:00:01', '00:00:02',
                '00:00:03', '00:00:04'],
               dtype='timedelta64[ns]', freq=None)
>>> pd.to_timedelta(np.arange(5), unit='d')
TimedeltaIndex(['0 days', '1 days', '2 days', '3 days', '4 days'],
               dtype='timedelta64[ns]', freq=None)
```

pandas.date_range

`pandas.date_range`(*start=None*, *end=None*, *periods=None*, *freq=None*, *tz=None*, *normalize=False*, *name=None*, *closed=None*, ***kwargs*) → pandas.core.indexes.datetimes.DatetimeIndex

Return a fixed frequency DatetimeIndex.

Parameters

start [str or datetime-like, optional] Left bound for generating dates.

end [str or datetime-like, optional] Right bound for generating dates.

periods [int, optional] Number of periods to generate.

freq [str or DateOffset, default 'D'] Frequency strings can have multiples, e.g. '5H'. See *here* for a list of frequency aliases.

tz [str or tzinfo, optional] Time zone name for returning localized DatetimeIndex, for example 'Asia/Hong_Kong'. By default, the resulting DatetimeIndex is timezone-naive.

normalize [bool, default False] Normalize start/end dates to midnight before generating date range.

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

closed [{None, 'left', 'right'}, optional] Make the interval closed with respect to the given frequency to the 'left', 'right', or both sides (None, the default).

****kwargs** For compatibility. Has no effect on the result.

Returns

rng [DatetimeIndex]

See also:

`DatetimeIndex` An immutable container for datetimes.

`timedelta_range` Return a fixed frequency TimedeltaIndex.

`period_range` Return a fixed frequency PeriodIndex.

`interval_range` Return a fixed frequency IntervalIndex.

Notes

Of the four parameters `start`, `end`, `periods`, and `freq`, exactly three must be specified. If `freq` is omitted, the resulting `DatetimeIndex` will have `periods` linearly spaced elements between `start` and `end` (closed on both sides).

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

Examples

Specifying the values

The next four examples generate the same *DatetimeIndex*, but vary the combination of *start*, *end* and *periods*.

Specify *start* and *end*, with the default daily frequency.

```
>>> pd.date_range(start='1/1/2018', end='1/08/2018')
DatetimeIndex(['2018-01-01', '2018-01-02', '2018-01-03', '2018-01-04',
               '2018-01-05', '2018-01-06', '2018-01-07', '2018-01-08'],
              dtype='datetime64[ns]', freq='D')
```

Specify *start* and *periods*, the number of periods (days).

```
>>> pd.date_range(start='1/1/2018', periods=8)
DatetimeIndex(['2018-01-01', '2018-01-02', '2018-01-03', '2018-01-04',
               '2018-01-05', '2018-01-06', '2018-01-07', '2018-01-08'],
              dtype='datetime64[ns]', freq='D')
```

Specify *end* and *periods*, the number of periods (days).

```
>>> pd.date_range(end='1/1/2018', periods=8)
DatetimeIndex(['2017-12-25', '2017-12-26', '2017-12-27', '2017-12-28',
               '2017-12-29', '2017-12-30', '2017-12-31', '2018-01-01'],
              dtype='datetime64[ns]', freq='D')
```

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

```
>>> pd.date_range(start='2018-04-24', end='2018-04-27', periods=3)
DatetimeIndex(['2018-04-24 00:00:00', '2018-04-25 12:00:00',
               '2018-04-27 00:00:00'],
              dtype='datetime64[ns]', freq=None)
```

Other Parameters

Changed the *freq* (frequency) to `'M'` (month end frequency).

```
>>> pd.date_range(start='1/1/2018', periods=5, freq='M')
DatetimeIndex(['2018-01-31', '2018-02-28', '2018-03-31', '2018-04-30',
               '2018-05-31'],
              dtype='datetime64[ns]', freq='M')
```

Multiples are allowed

```
>>> pd.date_range(start='1/1/2018', periods=5, freq='3M')
DatetimeIndex(['2018-01-31', '2018-04-30', '2018-07-31', '2018-10-31',
               '2019-01-31'],
              dtype='datetime64[ns]', freq='3M')
```

freq can also be specified as an Offset object.

```
>>> pd.date_range(start='1/1/2018', periods=5, freq=pd.offsets.MonthEnd(3))
DatetimeIndex(['2018-01-31', '2018-04-30', '2018-07-31', '2018-10-31',
               '2019-01-31'],
              dtype='datetime64[ns]', freq='3M')
```

Specify *tz* to set the timezone.

```
>>> pd.date_range(start='1/1/2018', periods=5, tz='Asia/Tokyo')
DatetimeIndex(['2018-01-01 00:00:00+09:00', '2018-01-02 00:00:00+09:00',
               '2018-01-03 00:00:00+09:00', '2018-01-04 00:00:00+09:00',
               '2018-01-05 00:00:00+09:00'],
              dtype='datetime64[ns, Asia/Tokyo]', freq='D')
```

closed controls whether to include *start* and *end* that are on the boundary. The default includes boundary points on either end.

```
>>> pd.date_range(start='2017-01-01', end='2017-01-04', closed=None)
DatetimeIndex(['2017-01-01', '2017-01-02', '2017-01-03', '2017-01-04'],
              dtype='datetime64[ns]', freq='D')
```

Use `closed='left'` to exclude *end* if it falls on the boundary.

```
>>> pd.date_range(start='2017-01-01', end='2017-01-04', closed='left')
DatetimeIndex(['2017-01-01', '2017-01-02', '2017-01-03'],
              dtype='datetime64[ns]', freq='D')
```

Use `closed='right'` to exclude *start* if it falls on the boundary.

```
>>> pd.date_range(start='2017-01-01', end='2017-01-04', closed='right')
DatetimeIndex(['2017-01-02', '2017-01-03', '2017-01-04'],
              dtype='datetime64[ns]', freq='D')
```

pandas.bdate_range

`pandas.`**`bdate_range`**(*start=None*, *end=None*, *periods=None*, *freq='B'*, *tz=None*, *normalize=True*, *name=None*, *weekmask=None*, *holidays=None*, *closed=None*, ***kwargs*) → pandas.core.indexes.datetimes.DatetimeIndex

Return a fixed frequency DatetimeIndex, with business day as the default frequency.

Parameters

start [str or datetime-like, default None] Left bound for generating dates.

end [str or datetime-like, default None] Right bound for generating dates.

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

freq [str or DateOffset, default 'B' (business daily)] Frequency strings can have multiples, e.g. '5H'.

tz [str or None] Time zone name for returning localized DatetimeIndex, for example Asia/Beijing.

normalize [bool, default False] Normalize start/end dates to midnight before generating date range.

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

weekmask [str or None, default None] Weekmask of valid business days, passed to `numpy.busdaycalendar`, only used when custom frequency strings are passed. The default value None is equivalent to 'Mon Tue Wed Thu Fri'.

New in version 0.21.0.

holidays [list-like or None, default None] Dates to exclude from the set of valid business days, passed to `numpy.busdaycalendar`, only used when custom frequency strings are passed.

New in version 0.21.0.

closed [str, default None] Make the interval closed with respect to the given frequency to the 'left', 'right', or both sides (None).

****kwargs** For compatibility. Has no effect on the result.

Returns

DatetimeIndex

Notes

Of the four parameters: `start`, `end`, `periods`, and `freq`, exactly three must be specified. Specifying `freq` is a requirement for `bdate_range`. Use `date_range` if specifying `freq` is not desired.

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

Examples

Note how the two weekend days are skipped in the result.

```
>>> pd.bdate_range(start='1/1/2018', end='1/08/2018')
DatetimeIndex(['2018-01-01', '2018-01-02', '2018-01-03', '2018-01-04',
           '2018-01-05', '2018-01-08'],
          dtype='datetime64[ns]', freq='B')
```

pandas.period_range

`pandas.`**`period_range`**(*start=None*, *end=None*, *periods=None*, *freq=None*, *name=None*) → pandas.core.indexes.period.PeriodIndex

Return a fixed frequency PeriodIndex.

The day (calendar) is the default frequency.

Parameters

start [str or period-like, default None] Left bound for generating periods.

end [str or period-like, default None] Right bound for generating periods.

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

freq [str or DateOffset, optional] Frequency alias. By default the freq is taken from *start* or *end* if those are Period objects. Otherwise, the default is `"D"` for daily frequency.

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

Returns

PeriodIndex

Notes

Of the three parameters: `start`, `end`, and `periods`, exactly two must be specified.

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

Examples

```
>>> pd.period_range(start='2017-01-01', end='2018-01-01', freq='M')
PeriodIndex(['2017-01', '2017-02', '2017-03', '2017-04', '2017-05',
             '2017-06', '2017-06', '2017-07', '2017-08', '2017-09',
             '2017-10', '2017-11', '2017-12', '2018-01'],
            dtype='period[M]', freq='M')
```

If `start` or `end` are `Period` objects, they will be used as anchor endpoints for a `PeriodIndex` with frequency matching that of the `period_range` constructor.

```
>>> pd.period_range(start=pd.Period('2017Q1', freq='Q'),
...                 end=pd.Period('2017Q2', freq='Q'), freq='M')
PeriodIndex(['2017-03', '2017-04', '2017-05', '2017-06'],
            dtype='period[M]', freq='M')
```

pandas.timedelta_range

`pandas.`**`timedelta_range`**(*start=None*, *end=None*, *periods=None*, *freq=None*, *name=None*, *closed=None*) → pandas.core.indexes.timedeltas.TimedeltaIndex

Return a fixed frequency TimedeltaIndex, with day as the default frequency.

Parameters

start [str or timedelta-like, default None] Left bound for generating timedeltas.

end [str or timedelta-like, default None] Right bound for generating timedeltas.

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

freq [str or DateOffset, default 'D'] Frequency strings can have multiples, e.g. '5H'.

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

closed [str, default None] Make the interval closed with respect to the given frequency to the 'left', 'right', or both sides (None).

Returns

rng [TimedeltaIndex]

Notes

Of the four parameters `start`, `end`, `periods`, and `freq`, exactly three must be specified. If `freq` is omitted, the resulting `TimedeltaIndex` will have `periods` linearly spaced elements between `start` and `end` (closed on both sides).

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

Examples

```
>>> pd.timedelta_range(start='1 day', periods=4)
TimedeltaIndex(['1 days', '2 days', '3 days', '4 days'],
               dtype='timedelta64[ns]', freq='D')
```

The `closed` parameter specifies which endpoint is included. The default behavior is to include both endpoints.

```
>>> pd.timedelta_range(start='1 day', periods=4, closed='right')
TimedeltaIndex(['2 days', '3 days', '4 days'],
               dtype='timedelta64[ns]', freq='D')
```

The `freq` parameter specifies the frequency of the TimedeltaIndex. Only fixed frequencies can be passed, non-fixed frequencies such as 'M' (month end) will raise.

```
>>> pd.timedelta_range(start='1 day', end='2 days', freq='6H')
TimedeltaIndex(['1 days 00:00:00', '1 days 06:00:00', '1 days 12:00:00',
                '1 days 18:00:00', '2 days 00:00:00'],
               dtype='timedelta64[ns]', freq='6H')
```

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

```
>>> pd.timedelta_range(start='1 day', end='5 days', periods=4)
TimedeltaIndex(['1 days 00:00:00', '2 days 08:00:00', '3 days 16:00:00',
            '5 days 00:00:00'],
           dtype='timedelta64[ns]', freq=None)
```

pandas.infer_freq

`pandas.`**`infer_freq`**(*index*, *warn: bool = True*) → Union[str, NoneType]

Infer the most likely frequency given the input index. If the frequency is uncertain, a warning will be printed.

Parameters

index [DatetimeIndex or TimedeltaIndex] If passed a Series will use the values of the series (NOT THE INDEX).

warn [bool, default True]

Returns

str or None None if no discernible frequency TypeError if the index is not datetime-like ValueError if there are less than three values.

3.2.5 Top-level dealing with intervals

interval_range([start, end, periods, freq, …])	Return a fixed frequency IntervalIndex.