(continued from previous page)

```
2000-01-06       NaN       NaN
2000-01-07       NaN       NaN
2000-01-08  0.254374 -0.240447
2000-01-09  0.157795  1.791197
2000-01-10  0.030876  1.371900
```

Passing a dict of lists will generate a MultiIndexed DataFrame with these selective transforms.

```
In [192]: tsdf.transform({'A': np.abs, 'B': [lambda x: x + 1, 'sqrt']})
Out[192]: 
                   A         B          
            absolute  <lambda>      sqrt
2000-01-01  0.428759  0.135110       NaN
2000-01-02  0.168731  2.338144  1.156782
2000-01-03  1.621034  1.438107  0.661897
2000-01-04       NaN       NaN       NaN
2000-01-05       NaN       NaN       NaN
2000-01-06       NaN       NaN       NaN
2000-01-07       NaN       NaN       NaN
2000-01-08  0.254374 -0.240447       NaN
2000-01-09  0.157795  1.791197  0.889493
2000-01-10  0.030876  1.371900  0.609836
```

Applying elementwise functions

Since not all functions can be vectorized (accept NumPy arrays and return another array or value), the methods *applymap()* on DataFrame and analogously *map()* on Series accept any Python function taking a single value and returning a single value. For example:

```
In [193]: df4
Out[193]: 
        one       two     three
a  1.394981  1.772517       NaN
b  0.343054  1.912123 -0.050390
c  0.695246  1.478369  1.227435
d       NaN  0.279344 -0.613172

In [194]: def f(x):
   .....:     return len(str(x))
   .....: 

In [195]: df4['one'].map(f)
Out[195]: 
a    18
b    19
c    18
d     3
Name: one, dtype: int64

In [196]: df4.applymap(f)
Out[196]: 
   one  two  three
a   18   17      3
b   19   18     20
```

(continues on next page)

(continued from previous page)

```
c   18   18     16
d    3   19     19
```

`Series.map()` has an additional feature; it can be used to easily "link" or "map" values defined by a secondary series. This is closely related to *merging/joining functionality*:

```
In [197]: s = pd.Series(['six', 'seven', 'six', 'seven', 'six'],
   .....:                index=['a', 'b', 'c', 'd', 'e'])
   .....:

In [198]: t = pd.Series({'six': 6., 'seven': 7.})

In [199]: s
Out[199]:
a      six
b    seven
c      six
d    seven
e      six
dtype: object

In [200]: s.map(t)
Out[200]:
a    6.0
b    7.0
c    6.0
d    7.0
e    6.0
dtype: float64
```

Reindexing and altering labels

`reindex()` is the fundamental data alignment method in pandas. It is used to implement nearly all other features relying on label-alignment functionality. To *reindex* means to conform the data to match a given set of labels along a particular axis. This accomplishes several things:

- Reorders the existing data to match a new set of labels
- Inserts missing value (NA) markers in label locations where no data for that label existed
- If specified, **fill** data for missing labels using logic (highly relevant to working with time series data)

Here is a simple example:

```
In [201]: s = pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd', 'e'])

In [202]: s
Out[202]:
a    1.695148
b    1.328614
c    1.234686
d   -0.385845
e   -1.326508
dtype: float64

In [203]: s.reindex(['e', 'b', 'f', 'd'])
```

(continues on next page)

(continued from previous page)

```
Out[203]:
e   -1.326508
b    1.328614
f         NaN
d   -0.385845
dtype: float64
```

Here, the `f` label was not contained in the Series and hence appears as `NaN` in the result.

With a DataFrame, you can simultaneously reindex the index and columns:

```
In [204]: df
Out[204]:
        one       two     three
a  1.394981  1.772517       NaN
b  0.343054  1.912123 -0.050390
c  0.695246  1.478369  1.227435
d       NaN  0.279344 -0.613172

In [205]: df.reindex(index=['c', 'f', 'b'], columns=['three', 'two', 'one'])
Out[205]:
      three       two       one
c  1.227435  1.478369  0.695246
f       NaN       NaN       NaN
b -0.050390  1.912123  0.343054
```

You may also use `reindex` with an `axis` keyword:

```
In [206]: df.reindex(['c', 'f', 'b'], axis='index')
Out[206]:
        one       two     three
c  0.695246  1.478369  1.227435
f       NaN       NaN       NaN
b  0.343054  1.912123 -0.050390
```

Note that the `Index` objects containing the actual axis labels can be **shared** between objects. So if we have a Series and a DataFrame, the following can be done:

```
In [207]: rs = s.reindex(df.index)

In [208]: rs
Out[208]:
a    1.695148
b    1.328614
c    1.234686
d   -0.385845
dtype: float64

In [209]: rs.index is df.index
Out[209]: True
```

This means that the reindexed Series's index is the same Python object as the DataFrame's index.

New in version 0.21.0.

`DataFrame.reindex()` also supports an "axis-style" calling convention, where you specify a single `labels` argument and the `axis` it applies to.

```
In [210]: df.reindex(['c', 'f', 'b'], axis='index')
Out[210]:
        one       two     three
c  0.695246  1.478369  1.227435
f       NaN       NaN       NaN
b  0.343054  1.912123 -0.050390

In [211]: df.reindex(['three', 'two', 'one'], axis='columns')
Out[211]:
      three       two       one
a       NaN  1.772517  1.394981
b -0.050390  1.912123  0.343054
c  1.227435  1.478369  0.695246
d -0.613172  0.279344       NaN
```

See also:

MultiIndex / Advanced Indexing is an even more concise way of doing reindexing.

Note: When writing performance-sensitive code, there is a good reason to spend some time becoming a reindexing ninja: **many operations are faster on pre-aligned data**. Adding two unaligned DataFrames internally triggers a reindexing step. For exploratory analysis you will hardly notice the difference (because `reindex` has been heavily optimized), but when CPU cycles matter sprinkling a few explicit `reindex` calls here and there can have an impact.

Reindexing to align with another object

You may wish to take an object and reindex its axes to be labeled the same as another object. While the syntax for this is straightforward albeit verbose, it is a common enough operation that the `reindex_like()` method is available to make this simpler:

```
In [212]: df2
Out[212]:
        one       two
a  1.394981  1.772517
b  0.343054  1.912123
c  0.695246  1.478369

In [213]: df3
Out[213]:
        one       two
a  0.583888  0.051514
b -0.468040  0.191120
c -0.115848 -0.242634

In [214]: df.reindex_like(df2)
Out[214]:
        one       two
a  1.394981  1.772517
b  0.343054  1.912123
c  0.695246  1.478369
```

Aligning objects with each other with `align`

The *align()* method is the fastest way to simultaneously align two objects. It supports a `join` argument (related to *joining and merging*):

- `join='outer'`: take the union of the indexes (default)
- `join='left'`: use the calling object's index
- `join='right'`: use the passed object's index
- `join='inner'`: intersect the indexes

It returns a tuple with both of the reindexed Series:

```
In [215]: s = pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd', 'e'])

In [216]: s1 = s[:4]

In [217]: s2 = s[1:]

In [218]: s1.align(s2)
Out[218]:
(a   -0.186646
 b   -1.692424
 c   -0.303893
 d   -1.425662
 e         NaN
 dtype: float64,
 a         NaN
 b   -1.692424
 c   -0.303893
 d   -1.425662
 e    1.114285
 dtype: float64)

In [219]: s1.align(s2, join='inner')
Out[219]:
(b   -1.692424
 c   -0.303893
 d   -1.425662
 dtype: float64,
 b   -1.692424
 c   -0.303893
 d   -1.425662
 dtype: float64)

In [220]: s1.align(s2, join='left')
Out[220]:
(a   -0.186646
 b   -1.692424
 c   -0.303893
 d   -1.425662
 dtype: float64,
 a         NaN
 b   -1.692424
 c   -0.303893
 d   -1.425662
 dtype: float64)
```

For DataFrames, the join method will be applied to both the index and the columns by default:

```
In [221]: df.align(df2, join='inner')
Out[221]:
(        one       two
 a  1.394981  1.772517
 b  0.343054  1.912123
 c  0.695246  1.478369,
         one       two
 a  1.394981  1.772517
 b  0.343054  1.912123
 c  0.695246  1.478369)
```

You can also pass an `axis` option to only align on the specified axis:

```
In [222]: df.align(df2, join='inner', axis=0)
Out[222]:
(        one       two     three
 a  1.394981  1.772517       NaN
 b  0.343054  1.912123 -0.050390
 c  0.695246  1.478369  1.227435,
         one       two
 a  1.394981  1.772517
 b  0.343054  1.912123
 c  0.695246  1.478369)
```

If you pass a Series to *`DataFrame.align()`*, you can choose to align both objects either on the DataFrame's index or columns using the `axis` argument:

```
In [223]: df.align(df2.iloc[0], axis=1)
Out[223]:
(        one     three       two
 a  1.394981       NaN  1.772517
 b  0.343054 -0.050390  1.912123
 c  0.695246  1.227435  1.478369
 d       NaN -0.613172  0.279344,
 one      1.394981
 three         NaN
 two      1.772517
 Name: a, dtype: float64)
```

Filling while reindexing

`reindex()` takes an optional parameter `method` which is a filling method chosen from the following table:

Method	Action
pad / ffill	Fill values forward
bfill / backfill	Fill values backward
nearest	Fill from the nearest index value

We illustrate these fill methods on a simple Series:

```
In [224]: rng = pd.date_range('1/3/2000', periods=8)

In [225]: ts = pd.Series(np.random.randn(8), index=rng)
```

(continues on next page)

(continued from previous page)

```
In [226]: ts2 = ts[[0, 3, 6]]

In [227]: ts
Out[227]: 
2000-01-03    0.183051
2000-01-04    0.400528
2000-01-05   -0.015083
2000-01-06    2.395489
2000-01-07    1.414806
2000-01-08    0.118428
2000-01-09    0.733639
2000-01-10   -0.936077
Freq: D, dtype: float64

In [228]: ts2
Out[228]: 
2000-01-03    0.183051
2000-01-06    2.395489
2000-01-09    0.733639
dtype: float64

In [229]: ts2.reindex(ts.index)
Out[229]: 
2000-01-03    0.183051
2000-01-04         NaN
2000-01-05         NaN
2000-01-06    2.395489
2000-01-07         NaN
2000-01-08         NaN
2000-01-09    0.733639
2000-01-10         NaN
Freq: D, dtype: float64

In [230]: ts2.reindex(ts.index, method='ffill')
Out[230]: 
2000-01-03    0.183051
2000-01-04    0.183051
2000-01-05    0.183051
2000-01-06    2.395489
2000-01-07    2.395489
2000-01-08    2.395489
2000-01-09    0.733639
2000-01-10    0.733639
Freq: D, dtype: float64

In [231]: ts2.reindex(ts.index, method='bfill')
Out[231]: 
2000-01-03    0.183051
2000-01-04    2.395489
2000-01-05    2.395489
2000-01-06    2.395489
2000-01-07    0.733639
2000-01-08    0.733639
2000-01-09    0.733639
2000-01-10         NaN
Freq: D, dtype: float64
```

(continues on next page)

(continued from previous page)

```
In [232]: ts2.reindex(ts.index, method='nearest')
Out[232]:
2000-01-03    0.183051
2000-01-04    0.183051
2000-01-05    2.395489
2000-01-06    2.395489
2000-01-07    2.395489
2000-01-08    0.733639
2000-01-09    0.733639
2000-01-10    0.733639
Freq: D, dtype: float64
```

These methods require that the indexes are **ordered** increasing or decreasing.

Note that the same result could have been achieved using *fillna* (except for `method='nearest'`) or *interpolate*:

```
In [233]: ts2.reindex(ts.index).fillna(method='ffill')
Out[233]:
2000-01-03    0.183051
2000-01-04    0.183051
2000-01-05    0.183051
2000-01-06    2.395489
2000-01-07    2.395489
2000-01-08    2.395489
2000-01-09    0.733639
2000-01-10    0.733639
Freq: D, dtype: float64
```

reindex() will raise a ValueError if the index is not monotonically increasing or decreasing. *fillna()* and *interpolate()* will not perform any checks on the order of the index.

Limits on filling while reindexing

The `limit` and `tolerance` arguments provide additional control over filling while reindexing. Limit specifies the maximum count of consecutive matches:

```
In [234]: ts2.reindex(ts.index, method='ffill', limit=1)
Out[234]:
2000-01-03    0.183051
2000-01-04    0.183051
2000-01-05         NaN
2000-01-06    2.395489
2000-01-07    2.395489
2000-01-08         NaN
2000-01-09    0.733639
2000-01-10    0.733639
Freq: D, dtype: float64
```

In contrast, tolerance specifies the maximum distance between the index and indexer values:

```
In [235]: ts2.reindex(ts.index, method='ffill', tolerance='1 day')
Out[235]:
2000-01-03    0.183051
2000-01-04    0.183051
```

(continues on next page)

(continued from previous page)

```
2000-01-05         NaN
2000-01-06    2.395489
2000-01-07    2.395489
2000-01-08         NaN
2000-01-09    0.733639
2000-01-10    0.733639
Freq: D, dtype: float64
```

Notice that when used on a `DatetimeIndex`, `TimedeltaIndex` or `PeriodIndex`, `tolerance` will coerced into a `Timedelta` if possible. This allows you to specify tolerance with appropriate strings.

Dropping labels from an axis

A method closely related to `reindex` is the *drop()* function. It removes a set of labels from an axis:

```
In [236]: df
Out[236]:
        one       two     three
a  1.394981  1.772517       NaN
b  0.343054  1.912123 -0.050390
c  0.695246  1.478369  1.227435
d       NaN  0.279344 -0.613172

In [237]: df.drop(['a', 'd'], axis=0)
Out[237]:
        one       two     three
b  0.343054  1.912123 -0.050390
c  0.695246  1.478369  1.227435

In [238]: df.drop(['one'], axis=1)
Out[238]:
        two     three
a  1.772517       NaN
b  1.912123 -0.050390
c  1.478369  1.227435
d  0.279344 -0.613172
```

Note that the following also works, but is a bit less obvious / clean:

```
In [239]: df.reindex(df.index.difference(['a', 'd']))
Out[239]:
        one       two     three
b  0.343054  1.912123 -0.050390
c  0.695246  1.478369  1.227435
```

Renaming / mapping labels

The *rename()* method allows you to relabel an axis based on some mapping (a dict or Series) or an arbitrary function.

```
In [240]: s
Out[240]:
a   -0.186646
b   -1.692424
c   -0.303893
d   -1.425662
e    1.114285
dtype: float64

In [241]: s.rename(str.upper)
Out[241]:
A   -0.186646
B   -1.692424
C   -0.303893
D   -1.425662
E    1.114285
dtype: float64
```

If you pass a function, it must return a value when called with any of the labels (and must produce a set of unique values). A dict or Series can also be used:

```
In [242]: df.rename(columns={'one': 'foo', 'two': 'bar'},
   .....:           index={'a': 'apple', 'b': 'banana', 'd': 'durian'})
   .....:
Out[242]:
             foo       bar     three
apple   1.394981  1.772517       NaN
banana  0.343054  1.912123 -0.050390
c       0.695246  1.478369  1.227435
durian       NaN  0.279344 -0.613172
```

If the mapping doesn't include a column/index label, it isn't renamed. Note that extra labels in the mapping don't throw an error.

New in version 0.21.0.

DataFrame.rename() also supports an "axis-style" calling convention, where you specify a single `mapper` and the `axis` to apply that mapping to.

```
In [243]: df.rename({'one': 'foo', 'two': 'bar'}, axis='columns')
Out[243]:
        foo       bar     three
a  1.394981  1.772517       NaN
b  0.343054  1.912123 -0.050390
c  0.695246  1.478369  1.227435
d       NaN  0.279344 -0.613172

In [244]: df.rename({'a': 'apple', 'b': 'banana', 'd': 'durian'}, axis='index')
Out[244]:
             one       two     three
apple   1.394981  1.772517       NaN
banana  0.343054  1.912123 -0.050390
c       0.695246  1.478369  1.227435
durian       NaN  0.279344 -0.613172
```

The *rename()* method also provides an `inplace` named parameter that is by default `False` and copies the underlying data. Pass `inplace=True` to rename the data in place.

Finally, *rename()* also accepts a scalar or list-like for altering the `Series.name` attribute.

```
In [245]: s.rename("scalar-name")
Out[245]:
a   -0.186646
b   -1.692424
c   -0.303893
d   -1.425662
e    1.114285
Name: scalar-name, dtype: float64
```

New in version 0.24.0.

The methods *rename_axis()* and *rename_axis()* allow specific names of a *MultiIndex* to be changed (as opposed to the labels).

```
In [246]: df = pd.DataFrame({'x': [1, 2, 3, 4, 5, 6],
   .....:                    'y': [10, 20, 30, 40, 50, 60]},
   .....:                   index=pd.MultiIndex.from_product([['a', 'b', 'c'], [1,␣
↪2]],
   .....:                   names=['let', 'num']))
   .....:

In [247]: df
Out[247]:
         x   y
let num
a   1    1  10
    2    2  20
b   1    3  30
    2    4  40
c   1    5  50
    2    6  60

In [248]: df.rename_axis(index={'let': 'abc'})
Out[248]:
         x   y
abc num
a   1    1  10
    2    2  20
b   1    3  30
    2    4  40
c   1    5  50
    2    6  60

In [249]: df.rename_axis(index=str.upper)
Out[249]:
         x   y
LET NUM
a   1    1  10
    2    2  20
b   1    3  30
    2    4  40
c   1    5  50
    2    6  60
```

Iteration

The behavior of basic iteration over pandas objects depends on the type. When iterating over a Series, it is regarded as array-like, and basic iteration produces the values. DataFrames follow the dict-like convention of iterating over the "keys" of the objects.

In short, basic iteration (`for i in object`) produces:

- **Series**: values
- **DataFrame**: column labels

Thus, for example, iterating over a DataFrame gives you the column names:

```
In [250]: df = pd.DataFrame({'col1': np.random.randn(3),
   .....:                    'col2': np.random.randn(3)}, index=['a', 'b', 'c'])
   .....:

In [251]: for col in df:
   .....:     print(col)
   .....:
col1
col2
```

Pandas objects also have the dict-like *items()* method to iterate over the (key, value) pairs.

To iterate over the rows of a DataFrame, you can use the following methods:

- *iterrows()*: Iterate over the rows of a DataFrame as (index, Series) pairs. This converts the rows to Series objects, which can change the dtypes and has some performance implications.
- *itertuples()*: Iterate over the rows of a DataFrame as namedtuples of the values. This is a lot faster than *iterrows()*, and is in most cases preferable to use to iterate over the values of a DataFrame.

Warning: Iterating through pandas objects is generally **slow**. In many cases, iterating manually over the rows is not needed and can be avoided with one of the following approaches:

- Look for a *vectorized* solution: many operations can be performed using built-in methods or NumPy functions, (boolean) indexing, …
- When you have a function that cannot work on the full DataFrame/Series at once, it is better to use *apply()* instead of iterating over the values. See the docs on *function application*.
- If you need to do iterative manipulations on the values but performance is important, consider writing the inner loop with cython or numba. See the *enhancing performance* section for some examples of this approach.

Warning: You should **never modify** something you are iterating over. This is not guaranteed to work in all cases. Depending on the data types, the iterator returns a copy and not a view, and writing to it will have no effect!

For example, in the following case setting the value has no effect:

```
In [252]: df = pd.DataFrame({'a': [1, 2, 3], 'b': ['a', 'b', 'c']})

In [253]: for index, row in df.iterrows():
   .....:     row['a'] = 10
   .....:

In [254]: df
Out[254]:
```

```
   a  b
0  1  a
1  2  b
2  3  c
```

items

Consistent with the dict-like interface, `items()` iterates through key-value pairs:

- **Series**: (index, scalar value) pairs
- **DataFrame**: (column, Series) pairs

For example:

```
In [255]: for label, ser in df.items():
   .....:     print(label)
   .....:     print(ser)
   .....:
a
0    1
1    2
2    3
Name: a, dtype: int64
b
0    a
1    b
2    c
Name: b, dtype: object
```

iterrows

`iterrows()` allows you to iterate through the rows of a DataFrame as Series objects. It returns an iterator yielding each index value along with a Series containing the data in each row:

```
In [256]: for row_index, row in df.iterrows():
   .....:     print(row_index, row, sep='\n')
   .....:
0
a    1
b    a
Name: 0, dtype: object
1
a    2
b    b
Name: 1, dtype: object
2
a    3
b    c
Name: 2, dtype: object
```

Note: Because `iterrows()` returns a Series for each row, it does **not** preserve dtypes across the rows (dtypes are preserved across columns for DataFrames). For example,

```
In [257]: df_orig = pd.DataFrame([[1, 1.5]], columns=['int', 'float'])

In [258]: df_orig.dtypes
Out[258]:
int        int64
float    float64
dtype: object

In [259]: row = next(df_orig.iterrows())[1]

In [260]: row
Out[260]:
int      1.0
float    1.5
Name: 0, dtype: float64
```

All values in `row`, returned as a Series, are now upcasted to floats, also the original integer value in column *x*:

```
In [261]: row['int'].dtype
Out[261]: dtype('float64')

In [262]: df_orig['int'].dtype
Out[262]: dtype('int64')
```

To preserve dtypes while iterating over the rows, it is better to use *itertuples()* which returns namedtuples of the values and which is generally much faster than *iterrows()*.

For instance, a contrived way to transpose the DataFrame would be:

```
In [263]: df2 = pd.DataFrame({'x': [1, 2, 3], 'y': [4, 5, 6]})

In [264]: print(df2)
   x  y
0  1  4
1  2  5
2  3  6

In [265]: print(df2.T)
   0  1  2
x  1  2  3
y  4  5  6

In [266]: df2_t = pd.DataFrame({idx: values for idx, values in df2.iterrows()})

In [267]: print(df2_t)
   0  1  2
x  1  2  3
y  4  5  6
```

itertuples

The `itertuples()` method will return an iterator yielding a namedtuple for each row in the DataFrame. The first element of the tuple will be the row's corresponding index value, while the remaining values are the row values.

For instance:

```
In [268]: for row in df.itertuples():
   .....:     print(row)
   .....:
Pandas(Index=0, a=1, b='a')
Pandas(Index=1, a=2, b='b')
Pandas(Index=2, a=3, b='c')
```

This method does not convert the row to a Series object; it merely returns the values inside a namedtuple. Therefore, `itertuples()` preserves the data type of the values and is generally faster as `iterrows()`.

Note: The column names will be renamed to positional names if they are invalid Python identifiers, repeated, or start with an underscore. With a large number of columns (>255), regular tuples are returned.

.dt accessor

`Series` has an accessor to succinctly return datetime like properties for the *values* of the Series, if it is a datetime/period like Series. This will return a Series, indexed like the existing Series.

```
# datetime
In [269]: s = pd.Series(pd.date_range('20130101 09:10:12', periods=4))

In [270]: s
Out[270]:
0   2013-01-01 09:10:12
1   2013-01-02 09:10:12
2   2013-01-03 09:10:12
3   2013-01-04 09:10:12
dtype: datetime64[ns]

In [271]: s.dt.hour
Out[271]:
0    9
1    9
2    9
3    9
dtype: int64

In [272]: s.dt.second
Out[272]:
0    12
1    12
2    12
3    12
dtype: int64

In [273]: s.dt.day
Out[273]:
```

(continues on next page)

(continued from previous page)

```
0    1
1    2
2    3
3    4
dtype: int64
```

This enables nice expressions like this:

```
In [274]: s[s.dt.day == 2]
Out[274]:
1   2013-01-02 09:10:12
dtype: datetime64[ns]
```

You can easily produces tz aware transformations:

```
In [275]: stz = s.dt.tz_localize('US/Eastern')

In [276]: stz
Out[276]:
0   2013-01-01 09:10:12-05:00
1   2013-01-02 09:10:12-05:00
2   2013-01-03 09:10:12-05:00
3   2013-01-04 09:10:12-05:00
dtype: datetime64[ns, US/Eastern]

In [277]: stz.dt.tz
Out[277]: <DstTzInfo 'US/Eastern' LMT-1 day, 19:04:00 STD>
```

You can also chain these types of operations:

```
In [278]: s.dt.tz_localize('UTC').dt.tz_convert('US/Eastern')
Out[278]:
0   2013-01-01 04:10:12-05:00
1   2013-01-02 04:10:12-05:00
2   2013-01-03 04:10:12-05:00
3   2013-01-04 04:10:12-05:00
dtype: datetime64[ns, US/Eastern]
```

You can also format datetime values as strings with *Series.dt.strftime()* which supports the same format as the standard `strftime()`.

```
# DatetimeIndex
In [279]: s = pd.Series(pd.date_range('20130101', periods=4))

In [280]: s
Out[280]:
0   2013-01-01
1   2013-01-02
2   2013-01-03
3   2013-01-04
dtype: datetime64[ns]

In [281]: s.dt.strftime('%Y/%m/%d')
Out[281]:
0    2013/01/01
1    2013/01/02
```

(continues on next page)

(continued from previous page)

```
2    2013/01/03
3    2013/01/04
dtype: object
```

```
# PeriodIndex
In [282]: s = pd.Series(pd.period_range('20130101', periods=4))

In [283]: s
Out[283]:
0    2013-01-01
1    2013-01-02
2    2013-01-03
3    2013-01-04
dtype: period[D]

In [284]: s.dt.strftime('%Y/%m/%d')
Out[284]:
0    2013/01/01
1    2013/01/02
2    2013/01/03
3    2013/01/04
dtype: object
```

The `.dt` accessor works for period and timedelta dtypes.

```
# period
In [285]: s = pd.Series(pd.period_range('20130101', periods=4, freq='D'))

In [286]: s
Out[286]:
0    2013-01-01
1    2013-01-02
2    2013-01-03
3    2013-01-04
dtype: period[D]

In [287]: s.dt.year
Out[287]:
0    2013
1    2013
2    2013
3    2013
dtype: int64

In [288]: s.dt.day
Out[288]:
0    1
1    2
2    3
3    4
dtype: int64
```

```
# timedelta
In [289]: s = pd.Series(pd.timedelta_range('1 day 00:00:05', periods=4, freq='s'))

In [290]: s
```

(continues on next page)

(continued from previous page)

```
Out[290]:
0   1 days 00:00:05
1   1 days 00:00:06
2   1 days 00:00:07
3   1 days 00:00:08
dtype: timedelta64[ns]

In [291]: s.dt.days
Out[291]:
0    1
1    1
2    1
3    1
dtype: int64

In [292]: s.dt.seconds
Out[292]:
0    5
1    6
2    7
3    8
dtype: int64

In [293]: s.dt.components
Out[293]:
   days  hours  minutes  seconds  milliseconds  microseconds  nanoseconds
0     1      0        0        5             0             0            0
1     1      0        0        6             0             0            0
2     1      0        0        7             0             0            0
3     1      0        0        8             0             0            0
```

Note: `Series.dt` will raise a `TypeError` if you access with a non-datetime-like values.

Vectorized string methods

Series is equipped with a set of string processing methods that make it easy to operate on each element of the array. Perhaps most importantly, these methods exclude missing/NA values automatically. These are accessed via the Series's `str` attribute and generally have names matching the equivalent (scalar) built-in string methods. For example:

```
In [294]: s = pd.Series(['A', 'B', 'C', 'Aaba', 'Baca', np.nan, 'CABA', 'dog
↪', 'cat'],
   .....:                 dtype="string")
   .....:

In [295]: s.str.lower()
Out[295]:
0       a
1       b
2       c
3    aaba
4    baca
5    <NA>
6    caba
```

(continues on next page)

(continued from previous page)

```
7      dog
8      cat
dtype: string
```

Powerful pattern-matching methods are provided as well, but note that pattern-matching generally uses regular expressions by default (and in some cases always uses them).

Note: Prior to pandas 1.0, string methods were only available on `object` -dtype `Series`. Pandas 1.0 added the *`StringDtype`* which is dedicated to strings. See *Text Data Types* for more.

Please see *Vectorized String Methods* for a complete description.

Sorting

Pandas supports three kinds of sorting: sorting by index labels, sorting by column values, and sorting by a combination of both.

By index

The *`Series.sort_index()`* and *`DataFrame.sort_index()`* methods are used to sort a pandas object by its index levels.

```
In [296]: df = pd.DataFrame({
   .....:     'one': pd.Series(np.random.randn(3), index=['a', 'b', 'c']),
   .....:     'two': pd.Series(np.random.randn(4), index=['a', 'b', 'c', 'd']),
   .....:     'three': pd.Series(np.random.randn(3), index=['b', 'c', 'd'])})
   .....:

In [297]: unsorted_df = df.reindex(index=['a', 'd', 'c', 'b'],
   .....:                          columns=['three', 'two', 'one'])
   .....:

In [298]: unsorted_df
Out[298]:
      three       two       one
a       NaN -1.152244  0.562973
d -0.252916 -0.109597       NaN
c  1.273388 -0.167123  0.640382
b -0.098217  0.009797 -1.299504

# DataFrame
In [299]: unsorted_df.sort_index()
Out[299]:
      three       two       one
a       NaN -1.152244  0.562973
b -0.098217  0.009797 -1.299504
c  1.273388 -0.167123  0.640382
d -0.252916 -0.109597       NaN

In [300]: unsorted_df.sort_index(ascending=False)
Out[300]:
      three       two       one
```

(continues on next page)

(continued from previous page)

```
d -0.252916 -0.109597       NaN
c  1.273388 -0.167123  0.640382
b -0.098217  0.009797 -1.299504
a       NaN -1.152244  0.562973

In [301]: unsorted_df.sort_index(axis=1)
Out[301]:
        one     three       two
a  0.562973       NaN -1.152244
d       NaN -0.252916 -0.109597
c  0.640382  1.273388 -0.167123
b -1.299504 -0.098217  0.009797

# Series
In [302]: unsorted_df['three'].sort_index()
Out[302]:
a         NaN
b   -0.098217
c    1.273388
d   -0.252916
Name: three, dtype: float64
```

By values

The `Series.sort_values()` method is used to sort a *Series* by its values. The `DataFrame.sort_values()` method is used to sort a *DataFrame* by its column or row values. The optional `by` parameter to `DataFrame.sort_values()` may used to specify one or more columns to use to determine the sorted order.

```
In [303]: df1 = pd.DataFrame({'one': [2, 1, 1, 1],
   .....:                     'two': [1, 3, 2, 4],
   .....:                     'three': [5, 4, 3, 2]})
   .....:

In [304]: df1.sort_values(by='two')
Out[304]:
   one  two  three
0    2    1      5
2    1    2      3
1    1    3      4
3    1    4      2
```

The `by` parameter can take a list of column names, e.g.:

```
In [305]: df1[['one', 'two', 'three']].sort_values(by=['one', 'two'])
Out[305]:
   one  two  three
2    1    2      3
1    1    3      4
3    1    4      2
0    2    1      5
```

These methods have special treatment of NA values via the `na_position` argument:

```
In [306]: s[2] = np.nan
```

(continues on next page)

(continued from previous page)

```
In [307]: s.sort_values()
Out[307]:
0       A
3    Aaba
1       B
4    Baca
6    CABA
8     cat
7     dog
2    <NA>
5    <NA>
dtype: string

In [308]: s.sort_values(na_position='first')
Out[308]:
2    <NA>
5    <NA>
0       A
3    Aaba
1       B
4    Baca
6    CABA
8     cat
7     dog
dtype: string
```

By indexes and values

New in version 0.23.0.

Strings passed as the by parameter to `DataFrame.sort_values()` may refer to either columns or index level names.

```
# Build MultiIndex
In [309]: idx = pd.MultiIndex.from_tuples([('a', 1), ('a', 2), ('a', 2),
   .....:                                  ('b', 2), ('b', 1), ('b', 1)])
   .....:

In [310]: idx.names = ['first', 'second']

# Build DataFrame
In [311]: df_multi = pd.DataFrame({'A': np.arange(6, 0, -1)},
   .....:                         index=idx)
   .....:

In [312]: df_multi
Out[312]:
              A
first second
a     1       6
      2       5
      2       4
b     2       3
      1       2
      1       1
```

Sort by 'second' (index) and 'A' (column)

```
In [313]: df_multi.sort_values(by=['second', 'A'])
Out[313]:
              A
first second
b     1       1
      1       2
a     1       6
b     2       3
a     2       4
      2       5
```

Note: If a string matches both a column name and an index level name then a warning is issued and the column takes precedence. This will result in an ambiguity error in a future version.

searchsorted

Series has the *searchsorted()* method, which works similarly to `numpy.ndarray.searchsorted()`.

```
In [314]: ser = pd.Series([1, 2, 3])

In [315]: ser.searchsorted([0, 3])
Out[315]: array([0, 2])

In [316]: ser.searchsorted([0, 4])
Out[316]: array([0, 3])

In [317]: ser.searchsorted([1, 3], side='right')
Out[317]: array([1, 3])

In [318]: ser.searchsorted([1, 3], side='left')
Out[318]: array([0, 2])

In [319]: ser = pd.Series([3, 1, 2])

In [320]: ser.searchsorted([0, 3], sorter=np.argsort(ser))
Out[320]: array([0, 2])
```

smallest / largest values

`Series` has the *nsmallest()* and *nlargest()* methods which return the smallest or largest n values. For a large `Series` this can be much faster than sorting the entire Series and calling `head(n)` on the result.

```
In [321]: s = pd.Series(np.random.permutation(10))

In [322]: s
Out[322]:
0    2
1    0
2    3
3    7
```

(continues on next page)

(continued from previous page)

```
4    1
5    5
6    9
7    6
8    8
9    4
dtype: int64

In [323]: s.sort_values()
Out[323]:
1    0
4    1
0    2
2    3
9    4
5    5
7    6
3    7
8    8
6    9
dtype: int64

In [324]: s.nsmallest(3)
Out[324]:
1    0
4    1
0    2
dtype: int64

In [325]: s.nlargest(3)
Out[325]:
6    9
8    8
3    7
dtype: int64
```

`DataFrame` also has the `nlargest` and `nsmallest` methods.

```
In [326]: df = pd.DataFrame({'a': [-2, -1, 1, 10, 8, 11, -1],
   .....:                    'b': list('abdceff'),
   .....:                    'c': [1.0, 2.0, 4.0, 3.2, np.nan, 3.0, 4.0]})
   .....:

In [327]: df.nlargest(3, 'a')
Out[327]:
    a  b    c
5  11  f  3.0
3  10  c  3.2
4   8  e  NaN

In [328]: df.nlargest(5, ['a', 'c'])
Out[328]:
    a  b    c
5  11  f  3.0
3  10  c  3.2
4   8  e  NaN
2   1  d  4.0
```

(continues on next page)

(continued from previous page)

```
6  -1  f  4.0

In [329]: df.nsmallest(3, 'a')
Out[329]:
   a  b    c
0 -2  a  1.0
1 -1  b  2.0
6 -1  f  4.0

In [330]: df.nsmallest(5, ['a', 'c'])
Out[330]:
   a  b    c
0 -2  a  1.0
1 -1  b  2.0
6 -1  f  4.0
2  1  d  4.0
4  8  e  NaN
```

Sorting by a MultiIndex column

You must be explicit about sorting when the column is a MultiIndex, and fully specify all levels to `by`.

```
In [331]: df1.columns = pd.MultiIndex.from_tuples([('a', 'one'),
   .....:                                          ('a', 'two'),
   .....:                                          ('b', 'three')])
   .....:

In [332]: df1.sort_values(by=('a', 'two'))
Out[332]:
    a         b
  one two three
0   2   1     5
2   1   2     3
1   1   3     4
3   1   4     2
```

Copying

The `copy()` method on pandas objects copies the underlying data (though not the axis indexes, since they are immutable) and returns a new object. Note that **it is seldom necessary to copy objects**. For example, there are only a handful of ways to alter a DataFrame *in-place*:

- Inserting, deleting, or modifying a column.
- Assigning to the `index` or `columns` attributes.
- For homogeneous data, directly modifying the values via the `values` attribute or advanced indexing.

To be clear, no pandas method has the side effect of modifying your data; almost every method returns a new object, leaving the original object untouched. If the data is modified, it is because you did so explicitly.

dtypes

For the most part, pandas uses NumPy arrays and dtypes for Series or individual columns of a DataFrame. NumPy provides support for `float`, `int`, `bool`, `timedelta64[ns]` and `datetime64[ns]` (note that NumPy does not support timezone-aware datetimes).

Pandas and third-party libraries *extend* NumPy's type system in a few places. This section describes the extensions pandas has made internally. See *Extension types* for how to write your own extension that works with pandas. See ecosystem.extensions for a list of third-party libraries that have implemented an extension.

The following table lists all of pandas extension types. For methods requiring `dtype` arguments, strings can be specified as indicated. See the respective documentation sections for more on each type.

Kind of Data	Data Type	Scalar	Array	String Aliases	Documentation
tz-aware datetime	`DatetimeTZDtype`	`Timestamp`	`arrays.DatetimeArray`	`'datetime64[ns, <tz>]'`	*Time zone handling*
Categorical	`CategoricalDtype`	(none)	`Categorical`	`'category'`	*Categorical data*
period (time spans)	`PeriodDtype`	`Period`	`arrays.PeriodArray`	`'period[<freq>]'`, `'Period[<freq>]'`	*Time span representation*
sparse	`SparseDtype`	(none)	`arrays.SparseArray`	`'Sparse'`, `'Sparse[int]'`, `'Sparse[float]'`	*Sparse data structures*
intervals	`IntervalDtype`	`Interval`	`arrays.IntervalArray`	`'interval'`, `'Interval'`, `'Interval[<numpy_dtype>]'`, `'Interval[datetime64[ns, <tz>]]'`, `'Interval[timedelta64[<freq>]]'`	*IntervalIndex*
nullable integer	`Int64Dtype`, ...	(none)	`arrays.IntegerArray`	`'Int8'`, `'Int16'`, `'Int32'`, `'Int64'`, `'UInt8'`, `'UInt16'`, `'UInt32'`, `'UInt64'`	*Nullable integer data type*
Strings	`StringDtype`	`str`	`arrays.StringArray`	`'string'`	*Working with text data*
Boolean (with NA)	`BooleanDtype`	`bool`	`arrays.BooleanArray`	`'boolean'`	*Boolean data with missing values*

Pandas has two ways to store strings.

1. `object` dtype, which can hold any Python object, including strings.
2. `StringDtype`, which is dedicated to strings.

Generally, we recommend using `StringDtype`. See *Text Data Types* fore more.

Finally, arbitrary objects may be stored using the `object` dtype, but should be avoided to the extent possible (for performance and interoperability with other libraries and methods. See *object conversion*).

A convenient `dtypes` attribute for DataFrame returns a Series with the data type of each column.