Essential basic functionality

Here we discuss a lot of the essential functionality common to the pandas data structures. Here's how to create some of the objects used in the examples from the previous section:

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

In [2]: s = pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd', 'e'])

In [3]: df = pd.DataFrame(np.random.randn(8, 3), index=index,
   ...:                   columns=['A', 'B', 'C'])
   ...:
```

Head and tail

To view a small sample of a Series or DataFrame object, use the `head()` and `tail()` methods. The default number of elements to display is five, but you may pass a custom number.

```
In [4]: long_series = pd.Series(np.random.randn(1000))

In [5]: long_series.head()
Out[5]:
0   -1.157892
1   -1.344312
2    0.844885
3    1.075770
4   -0.109050
dtype: float64

In [6]: long_series.tail(3)
Out[6]:
997   -0.289388
998   -1.020544
999    0.589993
dtype: float64
```

Attributes and underlying data

pandas objects have a number of attributes enabling you to access the metadata

- **shape**: gives the axis dimensions of the object, consistent with ndarray
- **Axis labels**
    - **Series**: *index* (only axis)
    - **DataFrame**: *index* (rows) and *columns*

Note, **these attributes can be safely assigned to**!

```
In [7]: df[:2]
Out[7]:
                   A         B         C
2000-01-01 -0.173215  0.119209 -1.044236
2000-01-02 -0.861849 -2.104569 -0.494929

In [8]: df.columns = [x.lower() for x in df.columns]

In [9]: df
Out[9]:
                   a         b         c
2000-01-01 -0.173215  0.119209 -1.044236
2000-01-02 -0.861849 -2.104569 -0.494929
2000-01-03  1.071804  0.721555 -0.706771
2000-01-04 -1.039575  0.271860 -0.424972
2000-01-05  0.567020  0.276232 -1.087401
2000-01-06 -0.673690  0.113648 -1.478427
2000-01-07  0.524988  0.404705  0.577046
2000-01-08 -1.715002 -1.039268 -0.370647
```

Pandas objects (`Index`, `Series`, `DataFrame`) can be thought of as containers for arrays, which hold the actual data and do the actual computation. For many types, the underlying array is a `numpy.ndarray`. However, pandas and 3rd party libraries may *extend* NumPy's type system to add support for custom arrays (see dtypes).

To get the actual data inside a **Index** or **Series**, use the `.array` property

```
In [10]: s.array
Out[10]:
<PandasArray>
[ 0.4691122999071863, -0.2828633443286633, -1.5090585031735124,
 -1.1356323710171934,  1.2121120250208506]
Length: 5, dtype: float64

In [11]: s.index.array
Out[11]:
<PandasArray>
['a', 'b', 'c', 'd', 'e']
Length: 5, dtype: object
```

array will always be an **ExtensionArray**. The exact details of what an **ExtensionArray** is and why pandas uses them is a bit beyond the scope of this introduction. See dtypes for more.

If you know you need a NumPy array, use **to_numpy()** or **numpy.asarray()**.

```
In [12]: s.to_numpy()
Out[12]: array([ 0.4691, -0.2829, -1.5091, -1.1356,  1.2121])

In [13]: np.asarray(s)
Out[13]: array([ 0.4691, -0.2829, -1.5091, -1.1356,  1.2121])
```

When the Series or Index is backed by an **ExtensionArray**, **to_numpy()** may involve copying data and coercing values. See dtypes for more.

to_numpy() gives some control over the `dtype` of the resulting **numpy.ndarray**. For example, consider datetimes with timezones. NumPy doesn't have a dtype to represent timezone-aware datetimes, so there are two possibly useful representations:

1. An object-dtype **numpy.ndarray** with **Timestamp** objects, each with the correct `tz`
2. A `datetime64[ns]` -dtype **numpy.ndarray**, where the values have been converted to UTC and the timezone discarded

Timezones may be preserved with `dtype=object`

```
In [14]: ser = pd.Series(pd.date_range('2000', periods=2, tz="CET"))

In [15]: ser.to_numpy(dtype=object)
Out[15]:
array([Timestamp('2000-01-01 00:00:00+0100', tz='CET', freq='D'),
       Timestamp('2000-01-02 00:00:00+0100', tz='CET', freq='D')],
      dtype=object)
```

Or thrown away with `dtype='datetime64[ns]'`

```
In [16]: ser.to_numpy(dtype="datetime64[ns]")
Out[16]:
array(['1999-12-31T23:00:00.000000000', '2000-01-01T23:00:00.000000000'],
      dtype='datetime64[ns]')
```

Getting the "raw data" inside a **DataFrame** is possibly a bit more complex. When your `DataFrame` only has a single data type for all the columns, **DataFrame.to_numpy()** will return the underlying data:

```
In [17]: df.to_numpy()
Out[17]:
array([[-0.1732,  0.1192, -1.0442],
       [-0.8618, -2.1046, -0.4949],
       [ 1.0718,  0.7216, -0.7068],
       [-1.0396,  0.2719, -0.425 ],
       [ 0.567 ,  0.2762, -1.0874],
       [-0.6737,  0.1136, -1.4784],
       [ 0.525 ,  0.4047,  0.577 ],
       [-1.715 , -1.0393, -0.3706]])
```

If a DataFrame contains homogeneously-typed data, the ndarray can actually be modified in-place, and the changes will be reflected in the data structure. For heterogeneous data (e.g. some of the DataFrame's columns are not all the same dtype), this will not be the case. The values attribute itself, unlike the axis labels, cannot be assigned to.

Note

When working with heterogeneous data, the dtype of the resulting ndarray will be chosen to accommodate all of the data involved. For example, if strings are involved, the result will be of object dtype. If there are only floats and integers, the resulting array will be of float dtype.

In the past, pandas recommended `Series.values` or `DataFrame.values` for extracting the data from a Series or DataFrame. You'll still find references to these in old code bases and online. Going forward, we recommend avoiding `.values` and using `.array` or `.to_numpy()`. `.values` has the following drawbacks:

1. When your Series contains an extension type, it's unclear whether `Series.values` returns a NumPy array or the extension array. `Series.array` will always return an `ExtensionArray`, and will never copy data. `Series.to_numpy()` will always return a NumPy array, potentially at the cost of copying / coercing values.
2. When your DataFrame contains a mixture of data types, `DataFrame.values` may involve copying data and coercing values to a common dtype, a relatively expensive operation. `DataFrame.to_numpy()`, being a method, makes it clearer that the returned NumPy array may not be a view on the same data in the DataFrame.

Accelerated operations

pandas has support for accelerating certain types of binary numerical and boolean operations using the `numexpr` library and the `bottleneck` libraries.

These libraries are especially useful when dealing with large data sets, and provide large speedups. `numexpr` uses smart chunking, caching, and multiple cores. `bottleneck` is a set of specialized cython routines that are especially fast when dealing with arrays that have `nans`.

Here is a sample (using 100 column x 100,000 row `DataFrames`):

Operation	0.11.0 (ms)	Prior Version (ms)	Ratio to Prior
`df1 > df2`	13.32	125.35	0.1063
`df1 * df2`	21.71	36.63	0.5928
`df1 + df2`	22.04	36.50	0.6039

You are highly encouraged to install both libraries. See the section Recommended Dependencies for more installation info.

These are both enabled to be used by default, you can control this by setting the options:

```
pd.set_option('compute.use_bottleneck', False)
pd.set_option('compute.use_numexpr', False)
```

Flexible binary operations

With binary operations between pandas data structures, there are two key points of interest:

- Broadcasting behavior between higher- (e.g. DataFrame) and lower-dimensional (e.g. Series) objects.
- Missing data in computations.

We will demonstrate how to manage these issues independently, though they can be handled simultaneously.

Matching / broadcasting behavior

DataFrame has the methods `add()`, `sub()`, `mul()`, `div()` and related functions `radd()`, `rsub()`, … for carrying out binary operations. For broadcasting behavior, Series input is of primary interest. Using these functions, you can use to either match on the *index* or *columns* via the **axis** keyword:

```
In [18]: 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 [19]: df
Out[19]:
        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 [20]: row = df.iloc[1]

In [21]: column = df['two']

In [22]: df.sub(row, axis='columns')
Out[22]:
        one       two     three
a  1.051928 -0.139606       NaN
b  0.000000  0.000000  0.000000
c  0.352192 -0.433754  1.277825
d       NaN -1.632779 -0.562782

In [23]: df.sub(row, axis=1)
Out[23]:
        one       two     three
a  1.051928 -0.139606       NaN
b  0.000000  0.000000  0.000000
c  0.352192 -0.433754  1.277825
d       NaN -1.632779 -0.562782

In [24]: df.sub(column, axis='index')
Out[24]:
        one  two     three
a -0.377535  0.0       NaN
b -1.569069  0.0 -1.962513
c -0.783123  0.0 -0.250933
d       NaN  0.0 -0.892516

In [25]: df.sub(column, axis=0)
Out[25]:
        one  two     three
a -0.377535  0.0       NaN
b -1.569069  0.0 -1.962513
c -0.783123  0.0 -0.250933
d       NaN  0.0 -0.892516
```

Furthermore you can align a level of a MultiIndexed DataFrame with a Series.

```
In [26]: dfmi = df.copy()

In [27]: dfmi.index = pd.MultiIndex.from_tuples([(1, 'a'), (1, 'b'),
   ....:                                          (1, 'c'), (2, 'a')],
   ....:                                         names=['first', 'second'])
   ....:

In [28]: dfmi.sub(column, axis=0, level='second')
Out[28]:
                   one       two     three
first second
1     a      -0.377535  0.000000       NaN
      b      -1.569069  0.000000 -1.962513
      c      -0.783123  0.000000 -0.250933
2     a            NaN -1.493173 -2.385688
```

Series and Index also support the `divmod()` builtin. This function takes the floor division and modulo operation at the same time returning a two-tuple of the same type as the left hand side. For example:

```
In [29]: s = pd.Series(np.arange(10))

In [30]: s
Out[30]:
0    0
1    1
2    2
3    3
4    4
5    5
6    6
7    7
8    8
9    9
dtype: int64

In [31]: div, rem = divmod(s, 3)

In [32]: div
Out[32]:
0    0
1    0
2    0
3    1
4    1
5    1
6    2
7    2
8    2
9    3
dtype: int64

In [33]: rem
Out[33]:
0    0
1    1
2    2
3    0
4    1
5    2
6    0
7    1
8    2
9    0
dtype: int64

In [34]: idx = pd.Index(np.arange(10))

In [35]: idx
Out[35]: Int64Index([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype='int64')

In [36]: div, rem = divmod(idx, 3)

In [37]: div
Out[37]: Int64Index([0, 0, 0, 1, 1, 1, 2, 2, 2, 3], dtype='int64')

In [38]: rem
Out[38]: Int64Index([0, 1, 2, 0, 1, 2, 0, 1, 2, 0], dtype='int64')
```

We can also do elementwise `divmod()`:

```
In [39]: div, rem = divmod(s, [2, 2, 3, 3, 4, 4, 5, 5, 6, 6])

In [40]: div
Out[40]:
0    0
1    0
2    0
3    1
4    1
5    1
6    1
7    1
8    1
9    1
dtype: int64

In [41]: rem
Out[41]:
0    0
1    1
2    2
3    0
4    0
5    1
6    1
7    2
8    2
9    3
dtype: int64
```

Missing data / operations with fill values

In Series and DataFrame, the arithmetic functions have the option of inputting a *fill_value*, namely a value to substitute when at most one of the values at a location are missing. For example, when adding two DataFrame objects, you may wish to treat NaN as 0 unless both DataFrames are missing that value, in which case the result will be NaN (you can later replace NaN with some other value using `fillna` if you wish).

```
In [42]: df
Out[42]:
        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 [43]: df2
Out[43]:
        one       two     three
a  1.394981  1.772517  1.000000
b  0.343054  1.912123 -0.050390
c  0.695246  1.478369  1.227435
d       NaN  0.279344 -0.613172

In [44]: df + df2
Out[44]:
        one       two     three
a  2.789963  3.545034       NaN
b  0.686107  3.824246 -0.100780
c  1.390491  2.956737  2.454870
d       NaN  0.558688 -1.226343

In [45]: df.add(df2, fill_value=0)
Out[45]:
        one       two     three
a  2.789963  3.545034  1.000000
b  0.686107  3.824246 -0.100780
c  1.390491  2.956737  2.454870
d       NaN  0.558688 -1.226343
```

Flexible comparisons

Series and DataFrame have the binary comparison methods `eq`, `ne`, `lt`, `gt`, `le`, and `ge` whose behavior is analogous to the binary arithmetic operations described above:

```
In [46]: df.gt(df2)
Out[46]:
     one    two  three
a  False  False  False
b  False  False  False
c  False  False  False
d  False  False  False

In [47]: df2.ne(df)
Out[47]:
     one    two  three
a  False  False   True
b  False  False  False
c  False  False  False
d   True  False  False
```

These operations produce a pandas object of the same type as the left-hand-side input that is of dtype `bool`. These `boolean` objects can be used in indexing operations, see the section on [Boolean indexing](#).

Boolean reductions

You can apply the reductions: `empty`, `any()`, `all()`, and `bool()` to provide a way to summarize a boolean result.

```
In [48]: (df > 0).all()
Out[48]:
one      False
two       True
three    False
dtype: bool

In [49]: (df > 0).any()
Out[49]:
one      True
two      True
three    True
dtype: bool
```

You can reduce to a final boolean value.

```
In [50]: (df > 0).any().any()
Out[50]: True
```

You can test if a pandas object is empty, via the `empty` property.

```
In [51]: df.empty
Out[51]: False

In [52]: pd.DataFrame(columns=list('ABC')).empty
Out[52]: True
```

To evaluate single-element pandas objects in a boolean context, use the method `bool()`:

```
In [53]: pd.Series([True]).bool()
Out[53]: True

In [54]: pd.Series([False]).bool()
Out[54]: False

In [55]: pd.DataFrame([[True]]).bool()
Out[55]: True

In [56]: pd.DataFrame([[False]]).bool()
Out[56]: False
```

Warning

You might be tempted to do the following:

```
>>> if df:
...     pass
```

Or

```
>>> df and df2
```

These will both raise errors, as you are trying to compare multiple values.:

```
ValueError: The truth value of an array is ambiguous. Use a.empty, a.any() or a.all().
```

See gotchas for a more detailed discussion.

Comparing if objects are equivalent

Often you may find that there is more than one way to compute the same result. As a simple example, consider `df + df` and `df * 2`. To test that these two computations produce the same result, given the tools shown above, you might imagine using `(df + df == df * 2).all()`. But in fact, this expression is False:

```
In [57]: df + df == df * 2
Out[57]: 
     one   two  three
a   True  True  False
b   True  True   True
c   True  True   True
d  False  True   True

In [58]: (df + df == df * 2).all()
Out[58]: 
one      False
two       True
three    False
dtype: bool
```

Notice that the boolean DataFrame `df + df == df * 2` contains some False values! This is because NaNs do not compare as equals:

```
In [59]: np.nan == np.nan
Out[59]: False
```

So, NDFrames (such as Series and DataFrames) have an `equals()` method for testing equality, with NaNs in corresponding locations treated as equal.

```
In [60]: (df + df).equals(df * 2)
Out[60]: True
```

Note that the Series or DataFrame index needs to be in the same order for equality to be True:

```
In [61]: df1 = pd.DataFrame({'col': ['foo', 0, np.nan]})

In [62]: df2 = pd.DataFrame({'col': [np.nan, 0, 'foo']}, index=[2, 1, 0])

In [63]: df1.equals(df2)
Out[63]: False

In [64]: df1.equals(df2.sort_index())
Out[64]: True
```

Comparing array-like objects

You can conveniently perform element-wise comparisons when comparing a pandas data structure with a scalar value:

```
In [65]: pd.Series(['foo', 'bar', 'baz']) == 'foo'
Out[65]:
0     True
1    False
2    False
dtype: bool

In [66]: pd.Index(['foo', 'bar', 'baz']) == 'foo'
Out[66]: array([ True, False, False])
```

Pandas also handles element-wise comparisons between different array-like objects of the same length:

```
In [67]: pd.Series(['foo', 'bar', 'baz']) == pd.Index(['foo', 'bar', 'qux'])
Out[67]:
0     True
1     True
2    False
dtype: bool

In [68]: pd.Series(['foo', 'bar', 'baz']) == np.array(['foo', 'bar', 'qux'])
Out[68]:
0     True
1     True
2    False
dtype: bool
```

Trying to compare `Index` or `Series` objects of different lengths will raise a ValueError:

```
In [55]: pd.Series(['foo', 'bar', 'baz']) == pd.Series(['foo', 'bar'])
ValueError: Series lengths must match to compare

In [56]: pd.Series(['foo', 'bar', 'baz']) == pd.Series(['foo'])
ValueError: Series lengths must match to compare
```

Note that this is different from the NumPy behavior where a comparison can be broadcast:

```
In [69]: np.array([1, 2, 3]) == np.array([2])
Out[69]: array([False,  True, False])
```

or it can return False if broadcasting can not be done:

```
In [70]: np.array([1, 2, 3]) == np.array([1, 2])
Out[70]: False
```

Combining overlapping data sets

A problem occasionally arising is the combination of two similar data sets where values in one are preferred over the other. An example would be two data series representing a particular economic indicator where one is considered to be of “higher quality”. However, the lower quality series might extend further back in history or have more complete data coverage. As such, we would like to combine two DataFrame objects where missing values in one DataFrame are conditionally filled with like-labeled values from the other DataFrame. The function implementing this operation is `combine_first()`, which we illustrate:

```
In [71]: df1 = pd.DataFrame({'A': [1., np.nan, 3., 5., np.nan],
   ....:                     'B': [np.nan, 2., 3., np.nan, 6.]})
   ....:

In [72]: df2 = pd.DataFrame({'A': [5., 2., 4., np.nan, 3., 7.],
   ....:                     'B': [np.nan, np.nan, 3., 4., 6., 8.]})
   ....:

In [73]: df1
Out[73]:
     A    B
0  1.0  NaN
1  NaN  2.0
2  3.0  3.0
3  5.0  NaN
4  NaN  6.0

In [74]: df2
Out[74]:
     A    B
0  5.0  NaN
1  2.0  NaN
2  4.0  3.0
3  NaN  4.0
4  3.0  6.0
5  7.0  8.0

In [75]: df1.combine_first(df2)
Out[75]:
     A    B
0  1.0  NaN
1  2.0  2.0
2  3.0  3.0
3  5.0  4.0
4  3.0  6.0
5  7.0  8.0
```

General DataFrame combine

The `combine_first()` method above calls the more general `DataFrame.combine()`. This method takes another DataFrame and a combiner function, aligns the input DataFrame and then passes the combiner function pairs of Series (i.e., columns whose names are the same).

So, for instance, to reproduce `combine_first()` as above:

```
In [76]: def combiner(x, y):
   ....:     return np.where(pd.isna(x), y, x)
   ....:
```

Descriptive statistics

There exists a large number of methods for computing descriptive statistics and other related operations on Series, DataFrame. Most of these are aggregations (hence producing a lower-dimensional result) like `sum()`, `mean()`, and `quantile()`, but some of them, like `cumsum()` and `cumprod()`, produce an object of the same size. Generally speaking, these methods take an **axis** argument, just like *ndarray.{sum, std, …}*, but the axis can be specified by name or integer:

- **Series**: no axis argument needed
- **DataFrame**: “index” (axis=0, default), “columns” (axis=1)

For example:

```
In [77]: df
Out[77]:
        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 [78]: df.mean(0)
Out[78]:
one      0.811094
two      1.360588
three    0.187958
dtype: float64

In [79]: df.mean(1)
Out[79]:
a    1.583749
b    0.734929
c    1.133683
d   -0.166914
dtype: float64
```

All such methods have a `skipna` option signaling whether to exclude missing data (`True` by default):

```
In [80]: df.sum(0, skipna=False)
Out[80]:
one           NaN
two      5.442353
three         NaN
dtype: float64

In [81]: df.sum(axis=1, skipna=True)
Out[81]:
a    3.167498
b    2.204786
c    3.401050
d   -0.333828
dtype: float64
```

Combined with the broadcasting / arithmetic behavior, one can describe various statistical procedures, like standardization (rendering data zero mean and standard deviation 1), very concisely:

```
In [82]: ts_stand = (df - df.mean()) / df.std()

In [83]: ts_stand.std()
Out[83]:
one      1.0
two      1.0
three    1.0
dtype: float64

In [84]: xs_stand = df.sub(df.mean(1), axis=0).div(df.std(1), axis=0)

In [85]: xs_stand.std(1)
Out[85]:
a    1.0
b    1.0
c    1.0
d    1.0
dtype: float64
```

Note that methods like `cumsum()` and `cumprod()` preserve the location of `NaN` values. This is somewhat different from `expanding()` and `rolling()`. For more details please see this note.

```
In [86]: df.cumsum()
Out[86]:
        one       two     three
a  1.394981  1.772517       NaN
b  1.738035  3.684640 -0.050390
c  2.433281  5.163008  1.177045
d       NaN  5.442353  0.563873
```

Here is a quick reference summary table of common functions. Each also takes an optional `level` parameter which applies only if the object has a hierarchical index.

Function	Description
`count`	Number of non-NA observations
`sum`	Sum of values

Function	Description
`mean`	Mean of values
`mad`	Mean absolute deviation
`median`	Arithmetic median of values
`min`	Minimum
`max`	Maximum
`mode`	Mode
`abs`	Absolute Value
`prod`	Product of values
`std`	Bessel-corrected sample standard deviation
`var`	Unbiased variance
`sem`	Standard error of the mean
`skew`	Sample skewness (3rd moment)
`kurt`	Sample kurtosis (4th moment)
`quantile`	Sample quantile (value at %)
`cumsum`	Cumulative sum
`cumprod`	Cumulative product
`cummax`	Cumulative maximum
`cummin`	Cumulative minimum

Note that by chance some NumPy methods, like `mean`, `std`, and `sum`, will exclude NAs on Series input by default:

```
In [87]: np.mean(df['one'])
Out[87]: 0.8110935116651192

In [88]: np.mean(df['one'].to_numpy())
Out[88]: nan
```

`Series.nunique()` will return the number of unique non-NA values in a Series:

```
In [89]: series = pd.Series(np.random.randn(500))

In [90]: series[20:500] = np.nan

In [91]: series[10:20] = 5

In [92]: series.nunique()
Out[92]: 11
```

Summarizing data: describe

There is a convenient `describe()` function which computes a variety of summary statistics about a Series or the columns of a DataFrame (excluding NAs of course):

```
In [93]: series = pd.Series(np.random.randn(1000))

In [94]: series[::2] = np.nan

In [95]: series.describe()
Out[95]: 
count    500.000000
mean      -0.021292
std        1.015906
min       -2.683763
25%       -0.699070
50%       -0.069718
75%        0.714483
max        3.160915
dtype: float64

In [96]: frame = pd.DataFrame(np.random.randn(1000, 5),
   ....:                      columns=['a', 'b', 'c', 'd', 'e'])
   ....: 

In [97]: frame.iloc[::2] = np.nan

In [98]: frame.describe()
Out[98]: 
                a           b           c           d           e
count  500.000000  500.000000  500.000000  500.000000  500.000000
mean     0.033387    0.030045   -0.043719   -0.051686    0.005979
std      1.017152    0.978743    1.025270    1.015988    1.006695
min     -3.000951   -2.637901   -3.303099   -3.159200   -3.188821
25%     -0.647623   -0.576449   -0.712369   -0.691338   -0.691115
50%      0.047578   -0.021499   -0.023888   -0.032652   -0.025363
75%      0.729907    0.775880    0.618896    0.670047    0.649748
max      2.740139    2.752332    3.004229    2.728702    3.240991
```

You can select specific percentiles to include in the output:

```
In [99]: series.describe(percentiles=[.05, .25, .75, .95])
Out[99]: 
count    500.000000
mean      -0.021292
std        1.015906
min       -2.683763
5%        -1.645423
25%       -0.699070
50%       -0.069718
75%        0.714483
95%        1.711409
max        3.160915
dtype: float64
```

By default, the median is always included.

For a non-numerical Series object, `describe()` will give a simple summary of the number of unique values and most frequently occurring values:

```
In [100]: s = pd.Series(['a', 'a', 'b', 'b', 'a', 'a', np.nan, 'c', 'd', 'a'])

In [101]: s.describe()
Out[101]: 
count     9
unique    4
top       a
freq      5
dtype: object
```

Note that on a mixed-type DataFrame object, `describe()` will restrict the summary to include only numerical columns or, if none are, only categorical columns:

```
In [102]: frame = pd.DataFrame({'a': ['Yes', 'Yes', 'No', 'No'], 'b': range(4)})

In [103]: frame.describe()
Out[103]: 
              b
count  4.000000
mean   1.500000
std    1.290994
min    0.000000
25%    0.750000
50%    1.500000
75%    2.250000
max    3.000000
```

This behavior can be controlled by providing a list of types as `include`/`exclude` arguments. The special value `all` can also be used:

```
In [104]: frame.describe(include=['object'])
Out[104]:
          a
count     4
unique    2
top     Yes
freq      2

In [105]: frame.describe(include=['number'])
Out[105]:
              b
count  4.000000
mean   1.500000
std    1.290994
min    0.000000
25%    0.750000
50%    1.500000
75%    2.250000
max    3.000000

In [106]: frame.describe(include='all')
Out[106]:
          a         b
count     4  4.000000
unique    2       NaN
top     Yes       NaN
freq      2       NaN
mean    NaN  1.500000
std     NaN  1.290994
min     NaN  0.000000
25%     NaN  0.750000
50%     NaN  1.500000
75%     NaN  2.250000
max     NaN  3.000000
```

That feature relies on select_dtypes. Refer to there for details about accepted inputs.

Index of min/max values

The `idxmin()` and `idxmax()` functions on Series and DataFrame compute the index labels with the minimum and maximum corresponding values:

```
In [107]: s1 = pd.Series(np.random.randn(5))

In [108]: s1
Out[108]:
0    1.118076
1   -0.352051
2   -1.242883
3   -1.277155
4   -0.641184
dtype: float64

In [109]: s1.idxmin(), s1.idxmax()
Out[109]: (3, 0)

In [110]: df1 = pd.DataFrame(np.random.randn(5, 3), columns=['A', 'B', 'C'])

In [111]: df1
Out[111]:
          A         B         C
0 -0.327863 -0.946180 -0.137570
1 -0.186235 -0.257213 -0.486567
2 -0.507027 -0.871259 -0.111110
3  2.000339 -2.430505  0.089759
4 -0.321434 -0.033695  0.096271

In [112]: df1.idxmin(axis=0)
Out[112]:
A    2
B    3
C    1
dtype: int64

In [113]: df1.idxmax(axis=1)
Out[113]:
0    C
1    A
2    C
3    A
4    C
dtype: object
```

When there are multiple rows (or columns) matching the minimum or maximum value, `idxmin()` and `idxmax()` return the first matching index:

```
In [114]: df3 = pd.DataFrame([2, 1, 1, 3, np.nan], columns=['A'], index=list('edcba'))

In [115]: df3
Out[115]:
     A
e  2.0
d  1.0
c  1.0
b  3.0
a  NaN

In [116]: df3['A'].idxmin()
Out[116]: 'd'
```

Note

`idxmin` and `idxmax` are called `argmin` and `argmax` in NumPy.

Value counts (histogramming) / mode

The `value_counts()` Series method and top-level function computes a histogram of a 1D array of values. It can also be used as a function on regular arrays:

```
In [117]: data = np.random.randint(0, 7, size=50)

In [118]: data
Out[118]:
array([6, 6, 2, 3, 5, 3, 2, 5, 4, 5, 4, 3, 4, 5, 0, 2, 0, 4, 2, 0, 3, 2,
       2, 5, 6, 5, 3, 4, 6, 4, 3, 5, 6, 4, 3, 6, 2, 6, 6, 2, 3, 4, 2, 1,
       6, 2, 6, 1, 5, 4])

In [119]: s = pd.Series(data)

In [120]: s.value_counts()
Out[120]:
6    10
2    10
4     9
5     8
3     8
0     3
1     2
dtype: int64

In [121]: pd.value_counts(data)
Out[121]:
6    10
2    10
4     9
5     8
3     8
0     3
1     2
dtype: int64
```

Similarly, you can get the most frequently occurring value(s) (the mode) of the values in a Series or DataFrame:

```
In [122]: s5 = pd.Series([1, 1, 3, 3, 3, 5, 5, 7, 7, 7])

In [123]: s5.mode()
Out[123]:
0    3
1    7
dtype: int64

In [124]: df5 = pd.DataFrame({"A": np.random.randint(0, 7, size=50),
   .....:                     "B": np.random.randint(-10, 15, size=50)})
   .....:

In [125]: df5.mode()
Out[125]:
     A   B
0  1.0  -9
1  NaN  10
2  NaN  13
```

Discretization and quantiling

Continuous values can be discretized using the `cut()` (bins based on values) and `qcut()` (bins based on sample quantiles) functions:

```
In [126]: arr = np.random.randn(20)

In [127]: factor = pd.cut(arr, 4)

In [128]: factor
Out[128]: 
[(-0.251, 0.464], (-0.968, -0.251], (0.464, 1.179], (-0.251, 0.464], (-0.968, -0.251], ...,
(-0.251, 0.464], (-0.968, -0.251], (-0.968, -0.251], (-0.968, -0.251], (-0.968, -0.251]]
Length: 20
Categories (4, interval[float64]): [(-0.968, -0.251] < (-0.251, 0.464] < (0.464, 1.179] <
                                    (1.179, 1.893]]

In [129]: factor = pd.cut(arr, [-5, -1, 0, 1, 5])

In [130]: factor
Out[130]: 
[(0, 1], (-1, 0], (0, 1], (0, 1], (-1, 0], ..., (-1, 0], (-1, 0], (-1, 0], (-1, 0], (-1, 0]]
Length: 20
Categories (4, interval[int64]): [(-5, -1] < (-1, 0] < (0, 1] < (1, 5]]
```

`qcut()` computes sample quantiles. For example, we could slice up some normally distributed data into equal-size quartiles like so:

```
In [131]: arr = np.random.randn(30)

In [132]: factor = pd.qcut(arr, [0, .25, .5, .75, 1])

In [133]: factor
Out[133]: 
[(0.569, 1.184], (-2.278, -0.301], (-2.278, -0.301], (0.569, 1.184], (0.569, 1.184], ...,
(-0.301, 0.569], (1.184, 2.346], (1.184, 2.346], (-0.301, 0.569], (-2.278, -0.301]]
Length: 30
Categories (4, interval[float64]): [(-2.278, -0.301] < (-0.301, 0.569] < (0.569, 1.184] <
                                    (1.184, 2.346]]

In [134]: pd.value_counts(factor)
Out[134]: 
(1.184, 2.346]      8
(-2.278, -0.301]    8
(0.569, 1.184]      7
(-0.301, 0.569]     7
dtype: int64
```

We can also pass infinite values to define the bins:

```
In [135]: arr = np.random.randn(20)

In [136]: factor = pd.cut(arr, [-np.inf, 0, np.inf])

In [137]: factor
Out[137]: 
[(-inf, 0.0], (0.0, inf], (0.0, inf], (-inf, 0.0], (-inf, 0.0], ..., (-inf, 0.0], (-inf, 0.0], (-
inf, 0.0], (0.0, inf], (0.0, inf]]
Length: 20
Categories (2, interval[float64]): [(-inf, 0.0] < (0.0, inf]]
```

Function application

To apply your own or another library's functions to pandas objects, you should be aware of the three methods below. The appropriate method to use depends on whether your function expects to operate on an entire `DataFrame` or `Series`, row- or column-wise, or elementwise.

1. Tablewise Function Application: `pipe()`
2. Row or Column-wise Function Application: `apply()`
3. Aggregation API: `agg()` and `transform()`
4. Applying Elementwise Functions: `applymap()`

Tablewise function application

`DataFrames` and `Series` can be passed into functions. However, if the function needs to be called in a chain, consider using the `pipe()` method.

First some setup:

```
In [138]: def extract_city_name(df):
   .....:     """
   .....:     Chicago, IL -> Chicago for city_name column
   .....:     """
   .....:     df['city_name'] = df['city_and_code'].str.split(",").str.get(0)
   .....:     return df
   .....:

In [139]: def add_country_name(df, country_name=None):
   .....:     """
   .....:     Chicago -> Chicago-US for city_name column
   .....:     """
   .....:     col = 'city_name'
   .....:     df['city_and_country'] = df[col] + country_name
   .....:     return df
   .....:

In [140]: df_p = pd.DataFrame({'city_and_code': ['Chicago, IL']})
```

`extract_city_name` and `add_country_name` are functions taking and returning `DataFrames`.

Now compare the following:

```
In [141]: add_country_name(extract_city_name(df_p), country_name='US')
Out[141]:
  city_and_code city_name city_and_country
0   Chicago, IL   Chicago        ChicagoUS
```

Is equivalent to:

```
In [142]: (df_p.pipe(extract_city_name)
   .....:      .pipe(add_country_name, country_name="US"))
   .....:
Out[142]:
  city_and_code city_name city_and_country
0   Chicago, IL   Chicago        ChicagoUS
```

Pandas encourages the second style, which is known as method chaining. `pipe` makes it easy to use your own or another library's functions in method chains, alongside pandas' methods.

In the example above, the functions `extract_city_name` and `add_country_name` each expected a `DataFrame` as the first positional argument. What if the function you wish to apply takes its data as, say, the second argument? In this case, provide `pipe` with a tuple of `(callable, data_keyword)`. `.pipe` will route the `DataFrame` to the argument specified in the tuple.

For example, we can fit a regression using statsmodels. Their API expects a formula first and a `DataFrame` as the second argument, `data`. We pass in the function, keyword pair `(sm.ols, 'data')` to `pipe`:

```
In [143]: import statsmodels.formula.api as sm

In [144]: bb = pd.read_csv('data/baseball.csv', index_col='id')

In [145]: (bb.query('h > 0')
   .....:    .assign(ln_h=lambda df: np.log(df.h))
   .....:    .pipe((sm.ols, 'data'), 'hr ~ ln_h + year + g + C(lg)')
   .....:    .fit()
   .....:    .summary()
   .....:  )
   .....:
Out[145]:
<class 'statsmodels.iolib.summary.Summary'>
"""
                            OLS Regression Results
==============================================================================
Dep. Variable:                     hr   R-squared:                       0.685
Model:                            OLS   Adj. R-squared:                  0.665
Method:                 Least Squares   F-statistic:                     34.28
Date:                Wed, 18 Mar 2020   Prob (F-statistic):           3.48e-15
Time:                        15:38:44   Log-Likelihood:                -205.92
No. Observations:                  68   AIC:                             421.8
Df Residuals:                      63   BIC:                             432.9
Df Model:                           4
Covariance Type:            nonrobust
===============================================================================
                  coef    std err          t      P>|t|      [0.025      0.975]
-------------------------------------------------------------------------------
Intercept   -8484.7720   4664.146     -1.819      0.074   -1.78e+04     835.780
C(lg)[T.NL]    -2.2736      1.325     -1.716      0.091      -4.922       0.375
ln_h           -1.3542      0.875     -1.547      0.127      -3.103       0.395
year            4.2277      2.324      1.819      0.074      -0.417       8.872
g               0.1841      0.029      6.258      0.000       0.125       0.243
==============================================================================
Omnibus:                       10.875   Durbin-Watson:                   1.999
Prob(Omnibus):                  0.004   Jarque-Bera (JB):               17.298
Skew:                           0.537   Prob(JB):                     0.000175
Kurtosis:                       5.225   Cond. No.                     1.49e+07
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 1.49e+07. This might indicate that there are
strong multicollinearity or other numerical problems.
"""
```

The pipe method is inspired by unix pipes and more recently dplyr and magrittr, which have introduced the popular `(%>%)` (read pipe) operator for R. The implementation of `pipe` here is quite clean and feels right at home in python. We encourage you to view the source code of `pipe()`.

Row or column-wise function application

Arbitrary functions can be applied along the axes of a DataFrame using the `apply()` method, which, like the descriptive statistics methods, takes an optional `axis` argument:

```
In [146]: df.apply(np.mean)
Out[146]: 
one      0.811094
two      1.360588
three    0.187958
dtype: float64

In [147]: df.apply(np.mean, axis=1)
Out[147]: 
a    1.583749
b    0.734929
c    1.133683
d   -0.166914
dtype: float64

In [148]: df.apply(lambda x: x.max() - x.min())
Out[148]: 
one      1.051928
two      1.632779
three    1.840607
dtype: float64

In [149]: df.apply(np.cumsum)
Out[149]: 
        one       two     three
a  1.394981  1.772517       NaN
b  1.738035  3.684640 -0.050390
c  2.433281  5.163008  1.177045
d       NaN  5.442353  0.563873

In [150]: df.apply(np.exp)
Out[150]: 
        one       two     three
a  4.034899  5.885648       NaN
b  1.409244  6.767440  0.950858
c  2.004201  4.385785  3.412466
d       NaN  1.322262  0.541630
```

The `apply()` method will also dispatch on a string method name.

```
In [151]: df.apply('mean')
Out[151]: 
one      0.811094
two      1.360588
three    0.187958
dtype: float64

In [152]: df.apply('mean', axis=1)
Out[152]: 
a    1.583749
b    0.734929
c    1.133683
d   -0.166914
dtype: float64
```

The return type of the function passed to `apply()` affects the type of the final output from `DataFrame.apply` for the default behaviour:

- If the applied function returns a `Series`, the final output is a `DataFrame`. The columns match the index of the `Series` returned by the applied function.
- If the applied function returns any other type, the final output is a `Series`.

This default behaviour can be overridden using the `result_type`, which accepts three options: `reduce`, `broadcast`, and `expand`. These will determine how list-likes return values expand (or not) to a `DataFrame`.

`apply()` combined with some cleverness can be used to answer many questions about a data set. For example, suppose we wanted to extract the date where the maximum value for each column occurred:

```
In [153]: tsdf = pd.DataFrame(np.random.randn(1000, 3), columns=['A', 'B', 'C'],
   .....:                     index=pd.date_range('1/1/2000', periods=1000))
   .....: 

In [154]: tsdf.apply(lambda x: x.idxmax())
Out[154]: 
A   2000-08-06
B   2001-01-18
C   2001-07-18
dtype: datetime64[ns]
```

You may also pass additional arguments and keyword arguments to the `apply()` method. For instance, consider the following function you would like to apply:

```
def subtract_and_divide(x, sub, divide=1):
    return (x - sub) / divide
```

You may then apply this function as follows:

```
df.apply(subtract_and_divide, args=(5,), divide=3)
```

Another useful feature is the ability to pass Series methods to carry out some Series operation on each column or row:

```
In [155]: tsdf
Out[155]:
                   A         B         C
2000-01-01 -0.158131 -0.232466  0.321604
2000-01-02 -1.810340 -3.105758  0.433834
2000-01-03 -1.209847 -1.156793 -0.136794
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 -0.653602  0.178875  1.008298
2000-01-09  1.007996  0.462824  0.254472
2000-01-10  0.307473  0.600337  1.643950

In [156]: tsdf.apply(pd.Series.interpolate)
Out[156]:
                   A         B         C
2000-01-01 -0.158131 -0.232466  0.321604
2000-01-02 -1.810340 -3.105758  0.433834
2000-01-03 -1.209847 -1.156793 -0.136794
2000-01-04 -1.098598 -0.889659  0.092225
2000-01-05 -0.987349 -0.622526  0.321243
2000-01-06 -0.876100 -0.355392  0.550262
2000-01-07 -0.764851 -0.088259  0.779280
2000-01-08 -0.653602  0.178875  1.008298
2000-01-09  1.007996  0.462824  0.254472
2000-01-10  0.307473  0.600337  1.643950
```

Finally, `apply()` takes an argument `raw` which is False by default, which converts each row or column into a Series before applying the function. When set to True, the passed function will instead receive an ndarray object, which has positive performance implications if you do not need the indexing functionality.

Aggregation API

The aggregation API allows one to express possibly multiple aggregation operations in a single concise way. This API is similar across pandas objects, see groupby API, the window functions API, and the resample API. The entry point for aggregation is `DataFrame.aggregate()`, or the alias `DataFrame.agg()`.

We will use a similar starting frame from above:

```
In [157]: tsdf = pd.DataFrame(np.random.randn(10, 3), columns=['A', 'B', 'C'],
   .....:                     index=pd.date_range('1/1/2000', periods=10))
   .....:

In [158]: tsdf.iloc[3:7] = np.nan

In [159]: tsdf
Out[159]:
                   A         B         C
2000-01-01  1.257606  1.004194  0.167574
2000-01-02 -0.749892  0.288112 -0.757304
2000-01-03 -0.207550 -0.298599  0.116018
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  0.814347 -0.257623  0.869226
2000-01-09 -0.250663 -1.206601  0.896839
2000-01-10  2.169758 -1.333363  0.283157
```

Using a single function is equivalent to `apply()`. You can also pass named methods as strings. These will return a `Series` of the aggregated output:

```
In [160]: tsdf.agg(np.sum)
Out[160]:
A    3.033606
B   -1.803879
C    1.575510
dtype: float64

In [161]: tsdf.agg('sum')
Out[161]:
A    3.033606
B   -1.803879
C    1.575510
dtype: float64

# these are equivalent to a ``.sum()`` because we are aggregating
# on a single function
In [162]: tsdf.sum()
Out[162]:
A    3.033606
B   -1.803879
C    1.575510
dtype: float64
```

Single aggregations on a `Series` this will return a scalar value:

```
In [163]: tsdf['A'].agg('sum')
Out[163]: 3.033606102414146
```

Aggregating with multiple functions

You can pass multiple aggregation arguments as a list. The results of each of the passed functions will be a row in the resulting `DataFrame`. These are naturally named from the aggregation function.

```
In [164]: tsdf.agg(['sum'])
Out[164]:
            A         B        C
sum  3.033606 -1.803879  1.57551
```

Multiple functions yield multiple rows:

```
In [165]: tsdf.agg(['sum', 'mean'])
Out[165]:
             A         B         C
sum   3.033606 -1.803879  1.575510
mean  0.505601 -0.300647  0.262585
```

On a `Series`, multiple functions return a `Series`, indexed by the function names:

```
In [166]: tsdf['A'].agg(['sum', 'mean'])
Out[166]:
sum     3.033606
mean    0.505601
Name: A, dtype: float64
```

Passing a `lambda` function will yield a `<lambda>` named row:

```
In [167]: tsdf['A'].agg(['sum', lambda x: x.mean()])
Out[167]:
sum         3.033606
<lambda>    0.505601
Name: A, dtype: float64
```

Passing a named function will yield that name for the row:

```
In [168]: def mymean(x):
   .....:     return x.mean()
   .....:

In [169]: tsdf['A'].agg(['sum', mymean])
Out[169]:
sum       3.033606
mymean    0.505601
Name: A, dtype: float64
```

Aggregating with a dict

Passing a dictionary of column names to a scalar or a list of scalars, to `DataFrame.agg` allows you to customize which functions are applied to which columns. Note that the results are not in any particular order, you can use an `OrderedDict` instead to guarantee ordering.

```
In [170]: tsdf.agg({'A': 'mean', 'B': 'sum'})
Out[170]:
A    0.505601
B   -1.803879
dtype: float64
```

Passing a list-like will generate a `DataFrame` output. You will get a matrix-like output of all of the aggregators. The output will consist of all unique functions. Those that are not noted for a particular column will be `NaN`:

```
In [171]: tsdf.agg({'A': ['mean', 'min'], 'B': 'sum'})
Out[171]:
             A         B
mean  0.505601       NaN
min  -0.749892       NaN
sum        NaN -1.803879
```

Mixed dtypes

When presented with mixed dtypes that cannot aggregate, `.agg` will only take the valid aggregations. This is similar to how groupby `.agg` works.

```
In [172]: mdf = pd.DataFrame({'A': [1, 2, 3],
   .....:                     'B': [1., 2., 3.],
   .....:                     'C': ['foo', 'bar', 'baz'],
   .....:                     'D': pd.date_range('20130101', periods=3)})
   .....:

In [173]: mdf.dtypes
Out[173]:
A             int64
B           float64
C            object
D    datetime64[ns]
dtype: object
```

```
In [174]: mdf.agg(['min', 'sum'])
Out[174]:
     A    B          C          D
min  1  1.0        bar 2013-01-01
sum  6  6.0  foobarbaz        NaT
```

Custom describe

With `.agg()` is it possible to easily create a custom describe function, similar to the built in describe function.

```
In [175]: from functools import partial

In [176]: q_25 = partial(pd.Series.quantile, q=0.25)

In [177]: q_25.__name__ = '25%'

In [178]: q_75 = partial(pd.Series.quantile, q=0.75)

In [179]: q_75.__name__ = '75%'

In [180]: tsdf.agg(['count', 'mean', 'std', 'min', q_25, 'median', q_75, 'max'])
Out[180]:
               A         B         C
count   6.000000  6.000000  6.000000
mean    0.505601 -0.300647  0.262585
std     1.103362  0.887508  0.606860
min    -0.749892 -1.333363 -0.757304
25%    -0.239885 -0.979600  0.128907
median  0.303398 -0.278111  0.225365
75%     1.146791  0.151678  0.722709
max     2.169758  1.004194  0.896839
```

Transform API

The `transform()` method returns an object that is indexed the same (same size) as the original. This API allows you to provide *multiple* operations at the same time rather than one-by-one. Its API is quite similar to the `.agg` API.

We create a frame similar to the one used in the above sections.

```
In [181]: tsdf = pd.DataFrame(np.random.randn(10, 3), columns=['A', 'B', 'C'],
   .....:                     index=pd.date_range('1/1/2000', periods=10))
   .....:

In [182]: tsdf.iloc[3:7] = np.nan

In [183]: tsdf
Out[183]: 
                   A         B         C
2000-01-01 -0.428759 -0.864890 -0.675341
2000-01-02 -0.168731  1.338144 -1.279321
2000-01-03 -1.621034  0.438107  0.903794
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 -1.240447 -0.201052
2000-01-09 -0.157795  0.791197 -1.144209
2000-01-10 -0.030876  0.371900  0.061932
```

Transform the entire frame. `.transform()` allows input functions as: a NumPy function, a string function name or a user defined function.

```
In [184]: tsdf.transform(np.abs)
Out[184]: 
                   A         B         C
2000-01-01  0.428759  0.864890  0.675341
2000-01-02  0.168731  1.338144  1.279321
2000-01-03  1.621034  0.438107  0.903794
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  1.240447  0.201052
2000-01-09  0.157795  0.791197  1.144209
2000-01-10  0.030876  0.371900  0.061932

In [185]: tsdf.transform('abs')
Out[185]: 
                   A         B         C
2000-01-01  0.428759  0.864890  0.675341
2000-01-02  0.168731  1.338144  1.279321
2000-01-03  1.621034  0.438107  0.903794
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  1.240447  0.201052
2000-01-09  0.157795  0.791197  1.144209
2000-01-10  0.030876  0.371900  0.061932

In [186]: tsdf.transform(lambda x: x.abs())
Out[186]: 
                   A         B         C
2000-01-01  0.428759  0.864890  0.675341
2000-01-02  0.168731  1.338144  1.279321
2000-01-03  1.621034  0.438107  0.903794
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  1.240447  0.201052
2000-01-09  0.157795  0.791197  1.144209
2000-01-10  0.030876  0.371900  0.061932
```

Here `transform()` received a single function; this is equivalent to a ufunc application.

```
In [187]: np.abs(tsdf)
Out[187]: 
                   A         B         C
2000-01-01  0.428759  0.864890  0.675341
2000-01-02  0.168731  1.338144  1.279321
2000-01-03  1.621034  0.438107  0.903794
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  1.240447  0.201052
2000-01-09  0.157795  0.791197  1.144209
2000-01-10  0.030876  0.371900  0.061932
```

Passing a single function to `.transform()` with a `Series` will yield a single `Series` in return.

```
In [188]: tsdf['A'].transform(np.abs)
Out[188]:
2000-01-01    0.428759
2000-01-02    0.168731
2000-01-03    1.621034
2000-01-04         NaN
2000-01-05         NaN
2000-01-06         NaN
2000-01-07         NaN
2000-01-08    0.254374
2000-01-09    0.157795
2000-01-10    0.030876
Freq: D, Name: A, dtype: float64
```

Transform with multiple functions

Passing multiple functions will yield a column MultiIndexed DataFrame. The first level will be the original frame column names; the second level will be the names of the transforming functions.

```
In [189]: tsdf.transform([np.abs, lambda x: x + 1])
Out[189]:
                   A                   B                   C
            absolute  <lambda>  absolute  <lambda>  absolute  <lambda>
2000-01-01  0.428759  0.571241  0.864890  0.135110  0.675341  0.324659
2000-01-02  0.168731  0.831269  1.338144  2.338144  1.279321 -0.279321
2000-01-03  1.621034 -0.621034  0.438107  1.438107  0.903794  1.903794
2000-01-04       NaN       NaN       NaN       NaN       NaN       NaN
2000-01-05       NaN       NaN       NaN       NaN       NaN       NaN
2000-01-06       NaN       NaN       NaN       NaN       NaN       NaN
2000-01-07       NaN       NaN       NaN       NaN       NaN       NaN
2000-01-08  0.254374  1.254374  1.240447 -0.240447  0.201052  0.798948
2000-01-09  0.157795  0.842205  0.791197  1.791197  1.144209 -0.144209
2000-01-10  0.030876  0.969124  0.371900  1.371900  0.061932  1.061932
```

Passing multiple functions to a Series will yield a DataFrame. The resulting column names will be the transforming functions.

```
In [190]: tsdf['A'].transform([np.abs, lambda x: x + 1])
Out[190]:
            absolute  <lambda>
2000-01-01  0.428759  0.571241
2000-01-02  0.168731  0.831269
2000-01-03  1.621034 -0.621034
2000-01-04       NaN       NaN
2000-01-05       NaN       NaN
2000-01-06       NaN       NaN
2000-01-07       NaN       NaN
2000-01-08  0.254374  1.254374
2000-01-09  0.157795  0.842205
2000-01-10  0.030876  0.969124
```

Transforming with a dict

Passing a dict of functions will allow selective transforming per column.

```
In [191]: tsdf.transform({'A': np.abs, 'B': lambda x: x + 1})
Out[191]:
                   A         B
2000-01-01  0.428759  0.135110
2000-01-02  0.168731  2.338144
2000-01-03  1.621034  1.438107
2000-01-04       NaN       NaN
2000-01-05       NaN       NaN
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
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'])
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)

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

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
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]:
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
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
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
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
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

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
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
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 Extension data types 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.

```
In [333]: dft = pd.DataFrame({'A': np.random.rand(3),
   .....:                     'B': 1,
   .....:                     'C': 'foo',
   .....:                     'D': pd.Timestamp('20010102'),
   .....:                     'E': pd.Series([1.0] * 3).astype('float32'),
   .....:                     'F': False,
   .....:                     'G': pd.Series([1] * 3, dtype='int8')})
   .....:

In [334]: dft
Out[334]:
          A  B    C          D    E      F  G
0  0.035962  1  foo 2001-01-02  1.0  False  1
1  0.701379  1  foo 2001-01-02  1.0  False  1
2  0.281885  1  foo 2001-01-02  1.0  False  1

In [335]: dft.dtypes
Out[335]:
A           float64
B             int64
C            object
D    datetime64[ns]
E           float32
F              bool
G              int8
dtype: object
```

On a `Series` object, use the **dtype** attribute.

```
In [336]: dft['A'].dtype
Out[336]: dtype('float64')
```

If a pandas object contains data with multiple dtypes *in a single column*, the dtype of the column will be chosen to accommodate all of the data types (`object` is the most general).

```
# these ints are coerced to floats
In [337]: pd.Series([1, 2, 3, 4, 5, 6.])
Out[337]:
0    1.0
1    2.0
2    3.0
3    4.0
4    5.0
5    6.0
dtype: float64

# string data forces an ``object`` dtype
In [338]: pd.Series([1, 2, 3, 6., 'foo'])
Out[338]:
0      1
1      2
2      3
3      6
4    foo
dtype: object
```

The number of columns of each type in a `DataFrame` can be found by calling `DataFrame.dtypes.value_counts()`.

```
In [339]: dft.dtypes.value_counts()
Out[339]:
bool              1
datetime64[ns]    1
object            1
int8              1
int64             1
float32           1
float64           1
dtype: int64
```

Numeric dtypes will propagate and can coexist in DataFrames. If a dtype is passed (either directly via the `dtype` keyword, a passed `ndarray`, or a passed `Series`), then it will be preserved in DataFrame operations. Furthermore, different numeric dtypes will **NOT** be combined. The following example will give you a taste.

```
In [340]: df1 = pd.DataFrame(np.random.randn(8, 1), columns=['A'], dtype='float32')

In [341]: df1
Out[341]:
          A
0  0.224364
1  1.890546
2  0.182879
3  0.787847
4 -0.188449
5  0.667715
6 -0.011736
7 -0.399073

In [342]: df1.dtypes
Out[342]:
A    float32
dtype: object

In [343]: df2 = pd.DataFrame({'A': pd.Series(np.random.randn(8), dtype='float16'),
   .....:                     'B': pd.Series(np.random.randn(8)),
   .....:                     'C': pd.Series(np.array(np.random.randn(8),
   .....:                                             dtype='uint8'))})
   .....:

In [344]: df2
Out[344]:
          A         B    C
0  0.823242  0.256090    0
1  1.607422  1.426469    0
2 -0.333740 -0.416203  255
3 -0.063477  1.139976    0
4 -1.014648 -1.193477    0
5  0.678711  0.096706    0
6 -0.040863 -1.956850    1
7 -0.357422 -0.714337    0

In [345]: df2.dtypes
Out[345]:
A    float16
B    float64
C      uint8
dtype: object
```

defaults

By default integer types are `int64` and float types are `float64`, *regardless* of platform (32-bit or 64-bit). The following will all result in `int64` dtypes.

```
In [346]: pd.DataFrame([1, 2], columns=['a']).dtypes
Out[346]:
a    int64
dtype: object

In [347]: pd.DataFrame({'a': [1, 2]}).dtypes
Out[347]:
a    int64
dtype: object

In [348]: pd.DataFrame({'a': 1}, index=list(range(2))).dtypes
Out[348]:
a    int64
dtype: object
```

Note that Numpy will choose *platform-dependent* types when creating arrays. The following **WILL** result in `int32` on 32-bit platform.

```
In [349]: frame = pd.DataFrame(np.array([1, 2]))
```

upcasting

Types can potentially be *upcasted* when combined with other types, meaning they are promoted from the current type (e.g. `int` to `float`).

```
In [350]: df3 = df1.reindex_like(df2).fillna(value=0.0) + df2

In [351]: df3
Out[351]:
          A         B      C
0  1.047606  0.256090    0.0
1  3.497968  1.426469    0.0
2 -0.150862 -0.416203  255.0
3  0.724370  1.139976    0.0
4 -1.203098 -1.193477    0.0
5  1.346426  0.096706    0.0
6 -0.052599 -1.956850    1.0
7 -0.756495 -0.714337    0.0

In [352]: df3.dtypes
Out[352]:
A    float32
B    float64
C    float64
dtype: object
```

`DataFrame.to_numpy()` will return the *lower-common-denominator* of the dtypes, meaning the dtype that can accommodate **ALL** of the types in the resulting homogeneous dtyped NumPy array. This can force some *upcasting*.

```
In [353]: df3.to_numpy().dtype
Out[353]: dtype('float64')
```

astype

You can use the `astype()` method to explicitly convert dtypes from one to another. These will by default return a copy, even if the dtype was unchanged (pass `copy=False` to change this behavior). In addition, they will raise an exception if the astype operation is invalid.

Upcasting is always according to the **numpy** rules. If two different dtypes are involved in an operation, then the more *general* one will be used as the result of the operation.

```
In [354]: df3
Out[354]:
          A         B      C
0  1.047606  0.256090    0.0
1  3.497968  1.426469    0.0
2 -0.150862 -0.416203  255.0
3  0.724370  1.139976    0.0
4 -1.203098 -1.193477    0.0
5  1.346426  0.096706    0.0
6 -0.052599 -1.956850    1.0
7 -0.756495 -0.714337    0.0

In [355]: df3.dtypes
Out[355]:
A    float32
B    float64
C    float64
dtype: object

# conversion of dtypes
In [356]: df3.astype('float32').dtypes
Out[356]:
A    float32
B    float32
C    float32
dtype: object
```

Convert a subset of columns to a specified type using `astype()`.

```
In [357]: dft = pd.DataFrame({'a': [1, 2, 3], 'b': [4, 5, 6], 'c': [7, 8, 9]})

In [358]: dft[['a', 'b']] = dft[['a', 'b']].astype(np.uint8)

In [359]: dft
Out[359]:
   a  b  c
0  1  4  7
1  2  5  8
2  3  6  9

In [360]: dft.dtypes
Out[360]:
a    uint8
b    uint8
c    int64
dtype: object
```

Convert certain columns to a specific dtype by passing a dict to `astype()`.

```
In [361]: dft1 = pd.DataFrame({'a': [1, 0, 1], 'b': [4, 5, 6], 'c': [7, 8, 9]})

In [362]: dft1 = dft1.astype({'a': np.bool, 'c': np.float64})

In [363]: dft1
Out[363]: 
       a  b    c
0   True  4  7.0
1  False  5  8.0
2   True  6  9.0

In [364]: dft1.dtypes
Out[364]: 
a       bool
b      int64
c    float64
dtype: object
```

Note

When trying to convert a subset of columns to a specified type using `astype()` and `loc()`, upcasting occurs.

`loc()` tries to fit in what we are assigning to the current dtypes, while `[]` will overwrite them taking the dtype from the right hand side. Therefore the following piece of code produces the unintended result.

```
In [365]: dft = pd.DataFrame({'a': [1, 2, 3], 'b': [4, 5, 6], 'c': [7, 8, 9]})

In [366]: dft.loc[:, ['a', 'b']].astype(np.uint8).dtypes
Out[366]: 
a    uint8
b    uint8
dtype: object

In [367]: dft.loc[:, ['a', 'b']] = dft.loc[:, ['a', 'b']].astype(np.uint8)

In [368]: dft.dtypes
Out[368]: 
a    int64
b    int64
c    int64
dtype: object
```

object conversion

pandas offers various functions to try to force conversion of types from the `object` dtype to other types. In cases where the data is already of the correct type, but stored in an `object` array, the `DataFrame.infer_objects()` and `Series.infer_objects()` methods can be used to soft convert to the correct type.

```
In [369]: import datetime

In [370]: df = pd.DataFrame([[1, 2],
   .....:                    ['a', 'b'],
   .....:                    [datetime.datetime(2016, 3, 2),
   .....:                     datetime.datetime(2016, 3, 2)]])
   .....: 

In [371]: df = df.T

In [372]: df
Out[372]: 
   0  1          2
0  1  a 2016-03-02
1  2  b 2016-03-02

In [373]: df.dtypes
Out[373]: 
0            object
1            object
2    datetime64[ns]
dtype: object
```

Because the data was transposed the original inference stored all columns as object, which `infer_objects` will correct.

Search the docs ...

Installation
Package overview
10 minutes to pandas
Getting started tutorials
Essential basic functionality
Intro to data structures
Comparison with other tools
Tutorials

```
In [374]: df.infer_objects().dtypes
Out[374]:
0              int64
1             object
2     datetime64[ns]
dtype: object
```

The following functions are available for one dimensional object arrays or scalars to perform hard conversion of objects to a specified type:

- `to_numeric()` (conversion to numeric dtypes)

```
In [375]: m = ['1.1', 2, 3]

In [376]: pd.to_numeric(m)
Out[376]: array([1.1, 2. , 3. ])
```

- `to_datetime()` (conversion to datetime objects)

```
In [377]: import datetime

In [378]: m = ['2016-07-09', datetime.datetime(2016, 3, 2)]

In [379]: pd.to_datetime(m)
Out[379]: DatetimeIndex(['2016-07-09', '2016-03-02'], dtype='datetime64[ns]', freq=None)
```

- `to_timedelta()` (conversion to timedelta objects)

```
In [380]: m = ['5us', pd.Timedelta('1day')]

In [381]: pd.to_timedelta(m)
Out[381]: TimedeltaIndex(['0 days 00:00:00.000005', '1 days 00:00:00'],
dtype='timedelta64[ns]', freq=None)
```

To force a conversion, we can pass in an `errors` argument, which specifies how pandas should deal with elements that cannot be converted to desired dtype or object. By default, `errors='raise'`, meaning that any errors encountered will be raised during the conversion process. However, if `errors='coerce'`, these errors will be ignored and pandas will convert problematic elements to `pd.NaT` (for datetime and timedelta) or `np.nan` (for numeric). This might be useful if you are reading in data which is mostly of the desired dtype (e.g. numeric, datetime), but occasionally has non-conforming elements intermixed that you want to represent as missing:

```
In [382]: import datetime

In [383]: m = ['apple', datetime.datetime(2016, 3, 2)]

In [384]: pd.to_datetime(m, errors='coerce')
Out[384]: DatetimeIndex(['NaT', '2016-03-02'], dtype='datetime64[ns]', freq=None)

In [385]: m = ['apple', 2, 3]

In [386]: pd.to_numeric(m, errors='coerce')
Out[386]: array([nan,  2.,  3.])

In [387]: m = ['apple', pd.Timedelta('1day')]

In [388]: pd.to_timedelta(m, errors='coerce')
Out[388]: TimedeltaIndex([NaT, '1 days'], dtype='timedelta64[ns]', freq=None)
```

The `errors` parameter has a third option of `errors='ignore'`, which will simply return the passed in data if it encounters any errors with the conversion to a desired data type:

```
In [389]: import datetime

In [390]: m = ['apple', datetime.datetime(2016, 3, 2)]

In [391]: pd.to_datetime(m, errors='ignore')
Out[391]: Index(['apple', 2016-03-02 00:00:00], dtype='object')

In [392]: m = ['apple', 2, 3]

In [393]: pd.to_numeric(m, errors='ignore')
Out[393]: array(['apple', 2, 3], dtype=object)

In [394]: m = ['apple', pd.Timedelta('1day')]

In [395]: pd.to_timedelta(m, errors='ignore')
Out[395]: array(['apple', Timedelta('1 days 00:00:00')], dtype=object)
```

In addition to object conversion, `to_numeric()` provides another argument `downcast`, which gives the option of downcasting the newly (or already) numeric data to a smaller dtype, which can conserve memory:

```
In [396]: m = ['1', 2, 3]

In [397]: pd.to_numeric(m, downcast='integer')   # smallest signed int dtype
Out[397]: array([1, 2, 3], dtype=int8)

In [398]: pd.to_numeric(m, downcast='signed')    # same as 'integer'
Out[398]: array([1, 2, 3], dtype=int8)

In [399]: pd.to_numeric(m, downcast='unsigned')  # smallest unsigned int dtype
Out[399]: array([1, 2, 3], dtype=uint8)

In [400]: pd.to_numeric(m, downcast='float')     # smallest float dtype
Out[400]: array([1., 2., 3.], dtype=float32)
```

As these methods apply only to one-dimensional arrays, lists or scalars; they cannot be used directly on multi-dimensional objects such as DataFrames. However, with `apply()`, we can "apply" the function over each column efficiently:

```
In [401]: import datetime

In [402]: df = pd.DataFrame([
   .....:     ['2016-07-09', datetime.datetime(2016, 3, 2)]] * 2, dtype='O')
   .....: 

In [403]: df
Out[403]: 
            0                    1
0  2016-07-09  2016-03-02 00:00:00
1  2016-07-09  2016-03-02 00:00:00

In [404]: df.apply(pd.to_datetime)
Out[404]: 
           0          1
0 2016-07-09 2016-03-02
1 2016-07-09 2016-03-02

In [405]: df = pd.DataFrame([['1.1', 2, 3]] * 2, dtype='O')

In [406]: df
Out[406]: 
     0  1  2
0  1.1  2  3
1  1.1  2  3

In [407]: df.apply(pd.to_numeric)
Out[407]: 
     0  1  2
0  1.1  2  3
1  1.1  2  3

In [408]: df = pd.DataFrame([['5us', pd.Timedelta('1day')]] * 2, dtype='O')

In [409]: df
Out[409]: 
     0                1
0  5us  1 days 00:00:00
1  5us  1 days 00:00:00

In [410]: df.apply(pd.to_timedelta)
Out[410]: 
                0      1
0 00:00:00.000005 1 days
1 00:00:00.000005 1 days
```

gotchas

Performing selection operations on `integer` type data can easily upcast the data to `floating`. The dtype of the input data will be preserved in cases where `nans` are not introduced. See also Support for integer NA.

```
In [411]: dfi = df3.astype('int32')

In [412]: dfi['E'] = 1

In [413]: dfi
Out[413]: 
   A  B    C  E
0  1  0    0  1
1  3  1    0  1
2  0  0  255  1
3  0  1    0  1
4 -1 -1    0  1
5  1  0    0  1
6  0 -1    1  1
7  0  0    0  1

In [414]: dfi.dtypes
Out[414]: 
A    int32
B    int32
C    int32
E    int64
dtype: object

In [415]: casted = dfi[dfi > 0]

In [416]: casted
Out[416]: 
     A    B      C  E
0  1.0  NaN    NaN  1
1  3.0  1.0    NaN  1
2  NaN  NaN  255.0  1
3  NaN  1.0    NaN  1
4  NaN  NaN    NaN  1
5  1.0  NaN    NaN  1
6  NaN  NaN    1.0  1
7  NaN  NaN    NaN  1

In [417]: casted.dtypes
Out[417]: 
A    float64
B    float64
C    float64
E      int64
dtype: object
```

While float dtypes are unchanged.

```
In [418]: dfa = df3.copy()

In [419]: dfa['A'] = dfa['A'].astype('float32')

In [420]: dfa.dtypes
Out[420]: 
A    float32
B    float64
C    float64
dtype: object

In [421]: casted = dfa[df2 > 0]

In [422]: casted
Out[422]: 
          A         B      C
0  1.047606  0.256090    NaN
1  3.497968  1.426469    NaN
2       NaN       NaN  255.0
3       NaN  1.139976    NaN
4       NaN       NaN    NaN
5  1.346426  0.096706    NaN
6       NaN       NaN    1.0
7       NaN       NaN    NaN

In [423]: casted.dtypes
Out[423]: 
A    float32
B    float64
C    float64
dtype: object
```

Selecting columns based on `dtype`

The **`select_dtypes()`** method implements subsetting of columns based on their `dtype`.

First, let's create a **`DataFrame`** with a slew of different dtypes:

```
In [424]: df = pd.DataFrame({'string': list('abc'),
   .....:                    'int64': list(range(1, 4)),
   .....:                    'uint8': np.arange(3, 6).astype('u1'),
   .....:                    'float64': np.arange(4.0, 7.0),
   .....:                    'bool1': [True, False, True],
   .....:                    'bool2': [False, True, False],
   .....:                    'dates': pd.date_range('now', periods=3),
   .....:                    'category': pd.Series(list("ABC")).astype('category')})
   .....:

In [425]: df['tdeltas'] = df.dates.diff()

In [426]: df['uint64'] = np.arange(3, 6).astype('u8')

In [427]: df['other_dates'] = pd.date_range('20130101', periods=3)

In [428]: df['tz_aware_dates'] = pd.date_range('20130101', periods=3, tz='US/Eastern')

In [429]: df
Out[429]:
  string  int64  uint8  float64  bool1  bool2                      dates category tdeltas  uint64
other_dates            tz_aware_dates
0      a      1      3      4.0   True  False 2020-03-18 15:38:47.007134        A     NaT       3
2013-01-01 2013-01-01 00:00:00-05:00
1      b      2      4      5.0  False   True 2020-03-19 15:38:47.007134        B  1 days       4
2013-01-02 2013-01-02 00:00:00-05:00
2      c      3      5      6.0   True  False 2020-03-20 15:38:47.007134        C  1 days       5
2013-01-03 2013-01-03 00:00:00-05:00
```

And the dtypes:

```
In [430]: df.dtypes
Out[430]:
string                                object
int64                                  int64
uint8                                  uint8
float64                              float64
bool1                                   bool
bool2                                   bool
dates                         datetime64[ns]
category                            category
tdeltas                      timedelta64[ns]
uint64                                uint64
other_dates                   datetime64[ns]
tz_aware_dates    datetime64[ns, US/Eastern]
dtype: object
```

[`select_dtypes()`]() has two parameters `include` and `exclude` that allow you to say "give me the columns *with* these dtypes" (`include`) and/or "give the columns *without* these dtypes" (`exclude`).

For example, to select `bool` columns:

```
In [431]: df.select_dtypes(include=[bool])
Out[431]:
   bool1  bool2
0   True  False
1  False   True
2   True  False
```

You can also pass the name of a dtype in the [NumPy dtype hierarchy]():

```
In [432]: df.select_dtypes(include=['bool'])
Out[432]:
   bool1  bool2
0   True  False
1  False   True
2   True  False
```

[`select_dtypes()`]() also works with generic dtypes as well.

For example, to select all numeric and boolean columns while excluding unsigned integers:

```
In [433]: df.select_dtypes(include=['number', 'bool'], exclude=['unsignedinteger'])
Out[433]:
   int64  float64  bool1  bool2 tdeltas
0      1      4.0   True  False     NaT
1      2      5.0  False   True  1 days
2      3      6.0   True  False  1 days
```

To select string columns you must use the `object` dtype:

```
In [434]: df.select_dtypes(include=['object'])
Out[434]: 
  string
0      a
1      b
2      c
```

To see all the child dtypes of a generic `dtype` like `numpy.number` you can define a function that returns a tree of child dtypes:

```
In [435]: def subdtypes(dtype):
   .....:     subs = dtype.__subclasses__()
   .....:     if not subs:
   .....:         return dtype
   .....:     return [dtype, [subdtypes(dt) for dt in subs]]
   .....: 
```

All NumPy dtypes are subclasses of `numpy.generic`:

```
In [436]: subdtypes(np.generic)
Out[436]: 
[numpy.generic,
 [[numpy.number,
   [[numpy.integer,
     [[numpy.signedinteger,
       [numpy.int8,
        numpy.int16,
        numpy.int32,
        numpy.int64,
        numpy.longlong,
        numpy.timedelta64]],
      [numpy.unsignedinteger,
       [numpy.uint8,
        numpy.uint16,
        numpy.uint32,
        numpy.uint64,
        numpy.ulonglong]]]],
    [numpy.inexact,
     [[numpy.floating,
       [numpy.float16, numpy.float32, numpy.float64, numpy.float128]],
      [numpy.complexfloating,
       [numpy.complex64, numpy.complex128, numpy.complex256]]]]]],
  [numpy.flexible,
   [[numpy.character, [numpy.bytes_, numpy.str_]],
    [numpy.void, [numpy.record]]]],
  numpy.bool_,
  numpy.datetime64,
  numpy.object_]]
```

Note

Pandas also defines the types `category`, and `datetime64[ns, tz]`, which are not integrated into the normal NumPy hierarchy and won't show up with the above function.

[<< How to manipulate textual data?] [Intro to data structures >>]