(continued from previous page)

```
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
```

(continues on next page)

(continued from previous page)

```
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
```

(continues on next page)

(continued from previous page)

```
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
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]:
```

(continues on next page)

(continued from previous page)

```
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
```

(continues on next page)

(continued from previous page)

```
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     No
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      No       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
```

(continues on next page)

(continued from previous page)

```
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()
```

(continues on next page)

(continued from previous page)

```
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]]
```

(continues on next page)

(continued from previous page)

```
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
   .....:
```

(continues on next page)

(continued from previous page)

```
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
```

(continues on next page)

(continued from previous page)

```
Model:                            OLS   Adj. R-squared:                  0.665
Method:                 Least Squares   F-statistic:                     34.28
Date:                Wed, 17 Jun 2020   Prob (F-statistic):           3.48e-15
Time:                        17:43:40   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())
```

(continues on next page)

(continued from previous page)

```
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())
```

(continues on next page)

(continued from previous page)

```
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
```

(continues on next page)

(continued from previous page)

```
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%'
```

(continues on next page)

(continued from previous page)

```
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
```

(continues on next page)

(continued from previous page)

```
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
```

(continues on next page)

(continued from previous page)

```
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
```

(continues on next page)