1 *pandas*

pandas is a high-performance module that provides a comprehensive set of structures for working with data. *pandas* excels at handling structured data, such as data sets containing many variables, working with missing values and merging across multiple data sets.

While extremely useful, *pandas* is not an essential component of the Python scientific stack unlike NumPy, SciPy or matplotlib, and so while *pandas* doesn't make data analysis possible in Python, it makes it much easier. *pandas* also provides high-performance, robust methods for importing from and exporting to a wide range of formats.

1.1 Data Structures

pandas provides a set of data structures which include Series,DataFrames and Panels. Series are 1-dimensional arrays. DataFrames are collections of Series and so are 2-dimensional, and Panels are collections ofDataFrames, and so are 3-dimensional. Note that the Panel type is not covered in this chapter. Series are the primary building block of the data structures in pandas, and in many ways a Series behaves similarly to a NumPy array. A Series is initialized using a list or tupel, or directly from a NumPy array.

```
>>> a = array([0.1, 1.2, 2.3, 3.4, 4.5])
>>> a
>>> from pandas import Series
>>> s = Series([0.1, 1.2, 2.3, 3.4, 4.5])
>>> s
>>> s = Series(a) # NumPy array to Series
```

Series, like arrays, are sliceable. However, unlike a 1-dimensional array, a Series has an additional column – an index – which is a set of values which are associated with the rows of the Series. In this example, pandas has automatically generated an index using the sequence 0, 1, . . . since none was provided. It is also possible to use other values as the index when initializing the Series using a keyword argument.

```
>>> s = Series([0.1, 1.2, 2.3, 3.4, 4.5], index = ['a','b','c','d','e'])
>>> s
```

The index enhances the usefulness of the pandas's data structures (Series and-DataFraame) and allows for dictionary-like access to elements in the index (in addition to both numeric slicing and logical indices).

```
>>> s['a']
0.10000000000000001
>>> s[0]
0.10000000000000001
>>> s[['a','c']]
a 0.1
c 2.3
dtype: float64
>>> s[[0,2]]
a 0.1
c 2.3
dtype: float64
>>> s[:2]
a 0.1
b 1.2
dtype: float64
>>> s[s>2]
c 2.3
d 3.4
e 4.5
dtype: float64
```

In this examples, 'a' and 'c' behave in the same manner as 0 and 2 would in a standard NumPy array. The elements of an index do not have to be unique which another way in which a Series generalizes a NumPy array.

```
>>> s = Series([0.1, 1.2, 2.3, 3.4, 4.5], index = ['a','b','c','a','b'])
```

Using numeric index values other than the default sequence will break scalar selection since there is ambiguity between numerical slicing and index access. For this reason, custom numerical indices should be used with care.

```
>>> s
a 0.1
b 1.2
c 2.3
```

```
a 3.4
b 4.5
dtype: float64
>>> s['a']
a 0.1
a 3.4
dtype: float64
```

Series can also be initialized directly from dictionaries.

```
>>> s = Series({'a': 0.1, 'b': 1.2, 'c': 2.3})
>>> s
```

Series are like NumPy arrays in that they support most numerical operations.

```
>>> s = Series({'a': 0.1, 'b': 1.2, 'c': 2.3})
>>> s * 2.0
>>> s- 1.0
```

However, Series are different from arrays when math operations are performed across two Series. In particular, math operations involving two series operate by aligning indices. The mathematical operation is performed in two steps.
First, the union of all indices is created, and then the mathematical operation is performed on matching indices. Indices that do not match are given the value `NaN` (not a number), and values are computed for all unique pairs of repeated indices.

```
>>> s1 = Series({'a': 0.1, 'b': 1.2, 'c': 2.3})
>>> s2 = Series({'a': 1.0, 'b': 2.0, 'c': 3.0})
>>> s3 = Series({'c': 0.1, 'd': 1.2, 'e': 2.3})
>>> s1 + s2
>>>
>>> s1 * s2
>>>
>>> s1 + s3
>>>
```

Mathematical operations performed on series which have non-unique indices will broadcast the operation to all indices which are common. For example, when one array has 2 elements with the same index, and another has 3, adding the two will produce 6 outputs.

```
>>> s1 = Series([1.0,2,3],index=['a']*3)
>>> s2 = Series([4.0,5],index=['a']*2)
>>> s1 + s2
a 5
a 6
a 6
a 7
a 7
a 8
dtype: float64
```

The underlying NumPy array is accessible through the values property, and the index is accessible the index property, which returns an Index type. The NumPy array underlying the index can be retrieved using values on the Index object returned.

```
>>> s1 = Series([1.0,2,3])
>>> s1.values
>>> s1.index
>>> s1.index.values
>>> s1.index = ['cat','dog','elephant']
>>> s1.index
```

1.2 Notable Methods and Properties

Series provide a large number of methods to manipulate data. These can broadly be categorized into mathematical and non-mathematical functions. The mathematical functions are generally very similar

to those in NumPy due to the underlying structure of a Series, and generally do not warrant a separate discussion. In contrast, the non-mathematical methods are unique to pandas.

`head` and `tail`

`head()` shows the first 5 rows of a series, and `tail()` shows the last 5 rows. An optional argument can be used to return a different number of entries, as in `head(10)`.

`isnull` and `notnull`

- `isnull()` returns a Series with the same indices containing Boolean values indicating True for null values which include NaN and None, among others.
- `notnull()` returns the negation of `isnull()` – that is, True for non-null values, and False otherwise.

`ix`

`ix` is the indexing function and `s.ix[0:2]` is the same as `s[0:2]`. `ix` is quite useful for DataFrames.

`describe`

`describe()` returns a simple set of summary statistics about a Series. The values returned is a series where the index contains name of the statistics computed.

```
>>> s1 = Series(arange(10.0,20.0))
>>> s1.describe()
>>> summ = s1.describe()
>>> summ['mean']
```

`unique` and `nunique`

`unique()` returns the unique elements of a series and `nunique()` returns the number of unique values in a Series.

`drop and dropna`

`drop(labels)` drop elements with the selected labels from a Series.

```
drop(labels) dro~+p elements with the selected labels forma Series.
>>> s1 = Series(arange(1.0,6),index=['a','a','b','c','d'])
>>> s1
>>> s1.drop('a')
```

`dropna()` is similar to `drop()` except that it only drops null values – NaN or similar.

```
>>> s1 = Series(arange(1.0,4.0),index=['a','b','c'])
>>> s2 = Series(arange(1.0,4.0),index=['c','d','e'])
>>> s3 = s1 + s2
>>> s3
>>> s3.dropna()
```

Both return Series and so it is necessary to assign the values to have a series with the selected elements dropped.

`fillna`

`fillna(value)` fills all null values in a series with a specific value.

```
>>> s1 = Series(arange(1.0,4.0),index=['a','b','c'])
>>> s2 = Series(arange(1.0,4.0),index=['c','d','e'])
>>> s3 = s1 + s2
>>> s3.fillna(1.0)
a 1
b 1
c 4
d 1
e 1
dtype: float64
```

`append`

`append(series)` appends one series to another, and is similar to `list.append`.

`replace`

replace(list,values) replaces a set of values in a Series with a new value. replace is similar to fillna except that replace also replaces non-null values.

`update`

`update(series)` replaces values in a series with those in another series, matching on the index, and is similar to a SQL update operation.

```
>>> s1 = Series(arange(1.0,4.0),index=['a','b','c'])
>>> s1

dtype: float64
>>> s2 = Series(1.0
* arange(1.0,4.0),index=['c','d','e'])
>>> s1.update(s2)
>>> s1
```

1.3 DataFrame

While the Series class is the building block of data structures in pandas, the DataFrame is the work-horse. DataFrames collect multiple series in the same way that a spreadsheet collects multiple columns of data. In a simple sense, a DataFrame is like a 2-dimensional NumPy array – and when all data is numeric and of the same type (e.g. float64), it is virtually indistinguishable. However, a DataFrame is composed of Series and each Series has its own data type, and so not allDataFrames are representable as homogeneous NumPy arrays. A number of methods are available to initialize a DataFrame. The simplest is from a homogeneous NumPy array.

```
>>> from pandas import DataFrame
>>> a = array([[1.0,2],[3,4]])
>>> df = DataFrame(a)
>>> df
0 1
0 1 2
177
1 3 4
```

Like a Series, a DataFrame contains the input data as well as row labels. However, since a DataFrame is a collection of columns, it also contains column labels (located along the top edge). When none are provided, the numeric sequence 0, 1, . . . is used. Column names are entered using a keyword argument or later by assigning to columns.

```
>>> df = DataFrame(array([[1,2],[3,4]]),columns=['a','b'])
>>> df
a b
0 1 2
1 3 4
>>> df = DataFrame(array([[1,2],[3,4]]))
>>> df.columns = ['dogs','cats']
>>> df
dogs cats
0 1 2
1 3 4
```

Index values are similarly assigned using either the keyword argument index or by setting the index property.

```
>>> df = DataFrame(array([[1,2],[3,4]]), columns=['dogs','cats'], index=['Alice','Bob']
>>> df
dogs cats
Alice 1 2
Bob 3 4
```

DataFrames can also be created from NumPy arrays with structured data.

```
>>> import datetime
>>> t = dtype([('datetime', 'O8'), ('value', 'f4')])
>>> x = zeros(1,dtype=t)
>>> x[0][0] = datetime.datetime(2013,01,01)
>>> x[0][1] = 99.99
>>> x
array([(datetime.datetime(2013, 1, 1, 0, 0), 99.98999786376953)],
dtype=[('datetime', 'O'), ('value', '<f4')])
>>> df = DataFrame(x)
```

```
>>> df
datetime value
0 20130101
99.989998
```

In the previous part, the DataFrame has automatically pulled the column names and column types from the NumPy structured data. The final method to create a DataFrame uses a dictionary containing Series, where the keys contain the column names. The DataFrame will automatically align the data using the common indices.

```
>>> s1 = Series(arange(0.0,5))
>>> s2 = Series(arange(1.0,3))
>>> DataFrame({'one': s1, 'two': s2})
178
one two
0 0 1
1 1 2
2 2 3
3 3 4
4 4 5
>>> s3 = Series(arange(0.0,3))
>>> DataFrame({'one': s1, 'two': s2, 'three': s3})
one three two
0 0 0 1
1 1 1 2
2 2 2 3
3 3 NaN 4
4 4 NaN 5
```

In the final example, the third series (s3) has fewer values and the DataFrame automatically fills missing values as NaN. Note that is possible to create DataFrames from Series which do not have unique index values, although in these cases the index values of the two series must match exactly – that is, have the same index values in the same order.

2 Manipulating DataFrames

The use of DataFrames will be demonstrated using a data set containing a mix of data types using statelevel GDP data from the US. The data set contains both the GDP level

between 2009 and 2012 (constant 2005 US$) and the growth rates for the same years as well as a variable containing the region of the state. The data is loaded directly into a DataFrame using `read_excel`.

```
>>> from pandas import read_excel
>>> state_gdp = read_excel('US_state_GDP.xls','Sheet1')
>>> state_gdp.head()
state_code state gdp_2009 gdp_2010 gdp_2011 gdp_2012
0 AK Alaska 44215 43472 44232 44732
1 AL Alabama 149843 153839 155390 157272
2 AR Arkansas 89776 92075 92684 93892
3 AZ Arizona 221405 221016 224787 230641
4 CA California 1667152 1672473 1692301 1751002
gdp_growth_2009 gdp_growth_2010 gdp_growth_2011 gdp_growth_2012 region
0 7.7 1.7
1.7 1.1 FW
1 3.9
2.7 1.0 1.2 SE
2 2.0
2.6 0.7 1.3 SE
3 8.2
0.2
1.7 2.6 SW
4 5.1
0.3 1.2 3.5 FW
```

2.1 Selecting Columns

Single columns are selectable using the columnname, as in `state_gdp['state']`, and the value returned in a Series. Multiple columns are similarly selected using a list of column names as in `state_gdp [['state_code', 'state']]`, or equivalently using an Index object. Note that these two methods are slightly different – selecting a single column returns a Series while selecting multiple columns returns a DataFrame. This is similar to how NumPy's scalar selection returns an array with a lower dimension. Use a list of column names containing a single name to return a DataFrame with a single column.

```
>>> state_gdp['state_code'].head() # Series
0 AK
```

```
1 AL
2 AR
3 AZ
4 CA
Name: state_code, dtype: object
>>> state_gdp[['state_code']].head() # DataFrame
state_code
0 AK
1 AL
2 AR
3 AZ
4 CA
>>> state_gdp[['state_code','state']].head()
state_code state
0 AL Alabama
1 AK Alaska
2 AZ Arizona
3 AR Arkansas
4 CA California
>>> index = state_gdp.index
>>> state_gdp[index[1:3]].head() # Elements 1 and 2 (0based
counting)
state gdp_2009
0 Alabama 149843
1 Alaska 44215
2 Arizona 221405
3 Arkansas 89776
4 California 1667152
```

Finally, single columns can also be selected using dot-notation and the column name.2 This is identical to using df['column'] and so the value returned is a Series.

```
>>> state_gdp.state_code.head()
0 AL
1 AK
2 AZ
3 AR
4 CA
Name: state_code, dtype: object
```

The column name must be a legal Python variable name, and so cannot contain spaces or reserved notation.

```
>>> type(state_gdp.state_code)
pandas.core.series.Series
```

Selecting Rows Rows can be selected using standard numerical slices.

```
>>> state_gdp[1:3]
state_code state gdp_2009 gdp_2010 gdp_2011 gdp_2012
1 AL Alabama 149843 153839 155390 157272
2 AR Arkansas 89776 92075 92684 93892
gdp_growth_2009 gdp_growth_2010 gdp_growth_2011 gdp_growth_2012 region
1 3.9
2.7 1.0 1.2 SE
2 2.0
2.6 0.7 1.3 SE
```

A function version is also available using iloc[rows] which is identical to the standard slicing syntax. Labeled rows can also be selected using the method loc[label] or loc[list of labels] to elect multiple rows using their label . Finally, rows can also be selected using logical selection using a Boolean array with the same number of elements as the number of rows as the DataFrame.

```
>>> state_long_recession = state_gdp['gdp_growth_2010']<0
>>> state_gdp[state_long_recession].head()
state_code state gdp_2009 gdp_2010 gdp_2011 gdp_2012
1 AK Alaska 44215 43472 44232 44732
2 AZ Arizona 221405 221016 224787 230641
28 NV Nevada 110001 109610 111574 113197
50 WY Wyoming 32439 32004 31231 31302
gdp_growth_2009 gdp_growth_2010 gdp_growth_2011 gdp_growth_2012
1 7.7 1.7
1.7 1.1
2 8.2
0.2
```

```
1.7 2.6
28 8.2
0.4
1.8 1.5
50 3.4 1.3
2.4
0.2
```

2.2 Selecting Rows and Columns

Since the behavior of slicing depends on whether the input is text (selects columns) or numeric/Boolean (selects rows), it isn't possible to use standard slicing to select both rows and columns.
Instead, the selector method `ix[rowselector,colselector]` allows joint selection where `rowselector` is either a scalar selector, a slice selector, a Boolean array, a numeric selector or a row label or list of row labels and colselector is a scalar selector, a slice selector, a Boolean array, a numeric selector or a column name or list of column names.

```
>>> state_gdp.ix[state_long_recession,'state']
1 Alaska
2 Arizona
28 Nevada
50 Wyoming
181
Name: state, dtype: object
>>> state_gdp.ix[state_long_recession,['state','gdp_growth_2009','gdp_growth_2010']]
state gdp_growth_2009 gdp_growth_2010
1 Alaska 7.7 1.7
2 Arizona 8.2
0.2
28 Nevada 8.2
0.4
50 Wyoming 3.4 1.3
>>> state_gdp.ix[10:15,0] # Slice and scalar
10 GA
11 HI
12 IA
13 ID
14 IL
15 IN
```

```
>>> state_gdp.ix[10:15,:2] # Slice and slice
state_code state
10 GA Georgia
11 HI Hawaii
12 IA Iowa
13 ID Idaho
14 IL Illinois
15 IN Indiana
```

2.3 Adding Columns

Columns are added using one of three methods. The most obvious is to add a Series merging along the index using a dictionary-like syntax.

```
>>> state_gdp_2012 = state_gdp[['state','gdp_2012']]
>>> state_gdp_2012.head()
state gdp_2012
0 Alabama 157272
1 Alaska 44732
2 Arizona 230641
3 Arkansas 93892
4 California 1751002
>>> state_gdp_2012['gdp_growth_2012'] = state_gdp['gdp_growth_2012']
>>> state_gdp_2012.head()
state gdp_2012 gdp_growth_2012
0 Alabama 157272 1.2
1 Alaska 44732 1.1
2 Arizona 230641 2.6
3 Arkansas 93892 1.3
```

This syntax always adds the column at the end. `insert(location,column_name,series)` inserts a Series at an specified location, where location uses 0-based indexing (i.e. 0 places the column first, 1 places it 182 second, etc.), `column_name` is the name of the column to be added and `series` is the series data. series is either a Series or another object that is readily convertible into a Series such as a NumPy array.

```
>>> state_gdp_2012 = state_gdp[['state','gdp_2012']]
>>> state_gdp_2012.insert(1,'gdp_growth_2012',state_gdp['gdp_growth_2012'])
>>> state_gdp_2012.head()
state gdp_growth_2012 gdp_2012
0 Alabama 1.2 157272
1 Alaska 1.1 44732
2 Arizona 2.6 230641
3 Arkansas 1.3 93892
4 California 3.5 1751002
```

Formally this type of join performs a left join which means that only index values in the base DataFrame will appear in the combined DataFrame, and so inserting columns

with different indices or fewer items than the DataFrame results in a DataFrame with the original indices with `NaN`-filled missing values in the new Series.

```
>>> state_gdp_2012 = state_gdp.ix[0:2,['state','gdp_2012']]
>>> state_gdp_2012
state gdp_2012
0 Alabama 157272
1 Alaska 44732
2 Arizona 230641
>>> gdp_2011 = state_gdp.ix[1:4,'gdp_2011']
>>> state_gdp_2012['gdp_2011'] = gdp_2011
state gdp_2012 gdp_2011
0 Alabama 157272 NaN
1 Alaska 44732 44232
2 Arizona 230641 224787
```

2.4 Deleting Columns

Columns are deleted using the del keyword, using pop(column) on the DataFrame or by calling `drop(list of columns,axis=1)` . The behavior of these differs slightly: `del` will simply delete the Series from the DataFrame.
`pop()` will both delete the Series and return the Series as an output, and `drop()` will return a DataFrame with the Series dropped by will not modify the original DataFrame.

```
>>> state_gdp_copy = state_gdp.copy()
>>> state_gdp_copy = state_gdp_copy[['state_code',
 'gdp_growth_2011','gdp_growth_2012']]
>>> state_gdp_copy.index = state_gdp['state_code']
>>> state_gdp_copy.head()
>>>
>>> gdp_growth_2012 = state_gdp_copy.pop('gdp_growth_2012')
>>> gdp_growth_2012.head()
>>>
>>> state_gdp_copy.head()
>>> del state_gdp_copy['gdp_growth_2011']
>>> state_gdp_copy.head()
>>> state_gdp_copy = state_gdp.copy()
>>> state_gdp_copy = state_gdp_copy[['state_code',
 'gdp_growth_2011','gdp_growth_2012']]
```

```
>>> state_gdp_dropped = state_gdp_copy.drop(['state_code'
 ,'gdp_growth_2011'],axis=1)
>>> state_gdp_dropped.head()
```

2.5 Notable Properties and Methods

drop, dropna and drop_duplicates

`drop()`, `dropna()` and `drop_duplicates()` can all be used to drop rows or columns from a DataFrame. drop(labels) drops rows based on the rowlabels in a label or list labels. drop(column_name,axis=1) drops columns based on a column name or list column names.

`dropna()` drops rows with anyNaN(or null) values. It can be used with the keyword argument dropna(how='all') to only drop rows which have missing values for all variables. It can also be used with the keyword argument dropna(axis=1) to drop columns with missing values. Finally, `drop_duplicates()` removes rows which are duplicates or other rows, and is used with the keyword argument `drop_duplicates(cols=col_list)` to only consider a subset of all columns when checking for duplicates.

values and index

values retrieves a the NumPy array (structured if the data columns are heterogeneous) underlying the DataFrame, and index returns the index of the DataFrame or can be assigned to to set the index.

fillna

`fillna()` fills NaN or other null values with other values. The simplest use fill all `NaN`s with a single value and is called `fillna(value=value )`. Using a dictionary allows for more sophisticated na-filling with column names as the keys and the replacements as the values.

```
>>> df = DataFrame(array([[1, nan],[nan, 2]]))
>>> df.columns = ['one','two']
>>> replacements = {'one':1,
'two':2}
>>> df.fillna(value=replacements)
```

T and transpose

T and transpose are identical – both swap rows and columns of a DataFrame. T operates like a property, while transpose is used as a method.

sort and sort_index

sort and sort_index are identical in their outcome and only differ in the inputs. The default behavior of sort is to sort using the index. Using a keyword argument axis=1 sorts the DataFrame by the column names.

Both can also be used to sort by the data in the DataFrame. sort does this using the keyword argument columns, which is either a single column name or a list of column names, and using a list of column names produces a nested sort. sort_index uses the keyword argument by to do the same.

Another keyword argument determines the direction of the sort (ascending by default). sort(ascending=False) will produce a descending sort, and when using a nested sort, the sort direction is specified using a list
sort(columns=['one','two'], ascending=[True,False])
where each entry corresponds to the columns used to sort.

```
>>> df = DataFrame(array([[1, 3],[1, 2],[3, 2],[2,1]]),
 columns=['one','two'])
>>> df.sort(columns='one')
>>>
>>> df.sort(columns=['one','two'])
>>>
>>> df.sort(columns=['one','two'], ascending=[0,1])
```

The default behavior is to not sort in-place and so it is necessary to assign the output of a sort. Using the keyword argument inplace=True will change the default behavior.

pivot

pivot reshapes a table using column values when reshaping. pivot takes three inputs. The first, index, defines the column to use as the index of the pivoted table. The second, columns, defines the column to use to formthe column names, and values defines the columns to for the data in the constructed DataFrame.
The following example showhowa flatDataFrame with repeated values is transformed into a more meaningful representation.

```
>>> prices = [101.0,102.0,103.0]
>>> tickers = ['GOOG','AAPL']
>>> import itertools
>>> data = [v for v in itertools.product(tickers,prices)]
>>> import pandas as pd
>>> dates = pd.date_range('20130103',
periods=3)
>>> df = DataFrame(data, columns=['ticker','price'])
>>> df['dates'] = dates.append(dates)
>>> df
>>>
>>> df.pivot(index='dates',columns='ticker',values='price')
>>>
```

`stack and unstack`

`stack` and `unstack` transform a DataFrame to a Series (stack) and back to a DataFrame (`unstack`). The stacked DataFrame (a Series) uses an index containing both the original row and column labels.

`concat and append`

append appends rows of another DataFrame to the end of an existing DataFrame. If the data appended has a different set of columns, missing values are `NaN`-filled. The keyword argument `ignore_index=True` instructs append to ignore the existing index in the appendedDataFrame. This is useful when index values are not meaningful, such as when they are simple numeric values.

`pd.concat` is a core function which concatenates two or more DataFrames using an outer join by default. An outer join is a method of joining DataFrames which will returns a DataFrame using the union of the indices of input Data Frames. This differs fromthe left join that is used when adding a Series to an existing DataFrame using dictionary syntax.

The keyword argument `join='inner'` can be used to perform an inner join, which will return a DataFrame using the intersection of the indices in the input DataFrames. Be default pd.concat will concatenate using column names, and the keyword argument axis=1 can be used to join using index labels.

```
>>> df1 = DataFrame([1,2,3],index=['a','b','c'],columns=['one'])
```

```
>>> df2 = DataFrame([4,5,6],index=['c','d','e'],columns=['two'])
>>> pd.concat((df1,df2), axis=1)
one two
a 1 NaN
b 2 NaN
c 3 4
d NaN 5
e NaN 6
>>> pd.concat((df1,df2), axis=1, join='inner')
one two
c 3 4
```

`reindex`, `reindex_like` and `reindex_axis`

`reindex` changes the labels while null-filling any missing values, which is useful for selecting subsets of a DataFrameor re-ordering rows. `reindex_like` behaves similarly, but uses the index from another DataFrame. The keyword argument axis directs `reindex_axis` to alter either rows or columns.

```
>>> original = DataFrame([[1,1],[2,2],[3.0,3]],index=['a','b','c'],
columns=['one','two'])
>>> original.reindex(index=['b','c','d'])
>>>
>>> different = DataFrame([[1,1],[2,2],[3.0,3]],index=['c','d','e'],
columns=['one','two'])
>>> original.reindex_like(different)
>>>
>>> original.reindex_axis(['two','one'], axis = 1)
```

`merge and join`

`merge` and `join` provide SQL-like operations for merging the DataFrames using row labels or the contents of columns.
The primary difference between the two is that `merge` defaults to using column contents while `join` defaults to using index labels. Both commands take a large number of optional inputs. The important keyword arguments are:

- how, which must be one of `'left', 'right', 'outer', 'inner'` describes which set of indices to use when performing the join.
- '`left`' uses the indices of the DataFrame that is used to call the method and '`right`' uses the DataFrame input into merge or join.
- '`outer`' uses a union of all indices from both DataFrames and '`inner`' uses an intersection from the two DataFrames.
- `on` is a single column name of list of column names to use in the merge. on assumes the names are common. If no value is given for on or `left_on/right_on`, then the common column names are used.
- `left_on` and `right_on` allow for a merge using columns with different names. When left_on and right_on contains the same column names, the behavior is the same as on.
- `left_index` and `right_index` indicate that the index labels are the join key for the left and right DataFrames.

```
>>> left = DataFrame([[1,2],[3,4],[5,6]],columns=['one','two'])
>>> right = DataFrame([[1,2],[3,4],[7,8]],columns=['one','three'])
>>> left.merge(right,on='one') # Same as how='inner'
>>>
>>> left.merge(right,on='one', how='left')
>>>
>>> left.merge(right,on='one', how='right')
>>>
>>> left.merge(right,on='one', how='outer')
>>>
```

`update`

`update` updates the values in one DataFrame using the non-null values from another DataFrame, using the index labels to determine which records to update.

```
>>> left = DataFrame([[1,2],[3,4],[5,6]],columns=['one','two'])
>>> left
>>> right = DataFrame([[nan,12],[13,nan],[nan,8]],
 columns=['one','two'],index=[1,2,3])
>>> right
>>>
>>> left.update(right) # Updates values in left
>>> left
```

`groupby`

`groupby` produces a `DataFrameGroupBy` object which is a `groupedDataFrame`, and is useful when a DataFrame has columns containing group data (e.g. sex or race in cross-sectional data). By itself, `groupby` does not produce any output, and so it is necessary to use other functions on the output `DataFrameGroupBy`.

```
>>> subset = state_gdp[['gdp_growth_2009','gdp_growth_2010','region']]
>>> subset.head()
>>>
>>> grouped_data = subset.groupby(by='region')
>>> grouped_data.groups # Lists group names and index labels for group
>>>
>>> grouped_data.mean()
>>> grouped_data.std() # Can use other methods
```

`apply`

`apply` executes a function along the columns or rows of a DataFrame. The following example applies the mean function both down columns and across rows, which is a trivial since mean could be executed on the DataFrame directly. `apply` is more general since it allows custom functions to be applied to a DataFrame.

```
>>> subset = state_gdp[['gdp_growth_2009','gdp_growth_2010','gdp_growth_2011','gdp_grow
>>> subset.index = state_gdp['state_code'].values
>>> subset.head()
gdp_growth_2009 gdp_growth_2010 gdp_growth_2011 gdp_growth_2012
AK 7.7 1.7
1.7 1.1
AL 3.9
2.7 1.0 1.2
AR 2.0
2.6 0.7 1.3
AZ 8.2
0.2
1.7 2.6
CA 5.1
0.3 1.2 3.5
>>> subset.apply(mean) # Same as subset.mean()
gdp_growth_2009 2.313725
gdp_growth_2010 2.462745
gdp_growth_2011 1.590196
gdp_growth_2012 2.103922
dtype: float64
>>> subset.apply(mean, axis=1).head() # Same as subset.mean(axis=1)
```

`applymap`

`applymap` is similar to apply, only that it applies element-by-element rather than column- or row-wise.

`pivot_table`

`pivot_table` provides a method to summarize data by groups. A pivot table first forms groups based using the keyword argument index and then returns an aggregate of all values within the group (using mean by default). The keyword argument `aggfun` allows for other aggregation function.

```
>>> subset = state_gdp[['gdp_growth_2009','gdp_growth_2010','region']]
>>> subset.head()
gdp_growth_2009 gdp_growth_2010 region
0 7.7 1.7
```

```
FW
1 3.9
2.7 SE
2 2.0
2.6 SE
3 8.2
0.2
SW
4 5.1
0.3 FW
>>> subset.pivot_table(index='region')
gdp_growth_2009 gdp_growth_2010
region
FW 2.483333
1.550000
GL 5.400000
3.660000
MW 1.250000
2.433333
NE 2.350000
2.783333
PL 1.357143
2.900000
RM 0.940000
1.380000
SE 2.633333
2.850000
SW 2.175000
1.325000
```

`pivot_table` differs from pivot since an aggregation function is used when transforming the data.

2.6 Statistical Function

pandas Series and DataFrame are derived from NumPy arrays and so the vast majority of simple statistical functions are available. This list includes `sum, mean, std, var, skew, kurt, prod, median, quantile, abs, cumsum`, and `cumprod`. DataFrame also supports `cov` and `corr` – the keyword argument axis determines the direction of the operation (0 for down columns, 1 for across rows). Some Particular statistical routines are described below.

count

`count` returns number of non-null values – that is, those which are not `NaN` or another null value such as `None` or `NaT` (not a time, for datetimes).

describe

`describe` provides a summary of the Series or DataFrame.

```
state_gdp.describe()
```

value_counts

`value_counts` performs frequency counting ("histogramming") of a Series or DataFrame.

```
>>> state_gdp.region.value_counts()
SE 12
PL 7
NE 6
FW 6
MW 6
GL 5
RM 5
SW 4
dtype: int64
```