pandas and numpy

pandas and numpy: Introduction

pandas is a Python module used for data manipulation and analysis.

* ability to read and write to CSV, XML, Excel files, etc.

* Excel-like numeric calculations, particularly *vectorization* (applying the same operation to multiple columns at once)

* emphasis on aligning data from multiple sources

* SQL-like merging, grouping and aggregating

numpy is a data analysis library that underlies pandas. We sometimes make direct calls to numpy - some of its variables (such as **np.nan**), variable-generating functions (such as **np.arange**) and some processing functions.

References

Various documentation sources will be necessary as the pandas library has many features.

```
import pandas as pd

help(pd.read_csv)
```

pandas official documentation
http://pandas.pydata.org/pandas-docs/stable

docs as a pdf:
http://pandas.pydata.org/pandas-docs/version/0.18.0/pandas.pdf

pandas textbook "Python for Data Analysis" by WesMcKinney
This is available as a paper textbook or a free pdf (if this link goes stale simply search **Python for Data Analysis pdf**

http://www3.canisius.edu/yany/python/Python4DataAnalysis.pdf

pandas objects

The *DataFrame* is the primary object in pandas; a DataFrame column or row can be isolated as a *Series* object.

DataFrame:
 * is like an Excel spreadsheet - rows, columns, and row and column labels
 * is like a "dict of dicts" in that it holds *column*-indexed Series
 * offers database-like and excel-like manipulations (merge, groupby, etc.)

Series
 * a "dictionary-like list" -- ordered values by associates them with an *index*
 * has a *dtype* attribute that holds its objects' common type

Index
 * an object that provides indexing for both Series (its index) and DataFrame (its columns or Series indices) objects

The DataFrame

A dataframe is a 2-dimensional structure that can be indexed like a list and subscripted like a dictionary. It is the central structure in most of our analysis.

```
import pandas as pd
import numpy as np

# initialize a new, empty DataFrame
df = pd.DataFrame()

# initialize a DataFrame with sample data
df = pd.DataFrame( {'a': [1, 2, 3, 4],
                    'b': [1.0, 1.5, 2.0, 2.5],
                    'c': ['a', 'b', 'c', 'd'] }, index=['r1', 'r2', 'r3', 'r4'] )

print df


# a DataFrame, printed
    a    b  c
r1  1  1.0  a
r2  2  1.5  b
r3  3  2.0  c
f4  4  2.5  d
```

Series objects as Columns or Rows of DataFrame

A Series is a list of values indexed by position as well as label. The index of a Series provides the labels.

```
import pandas as pd
import numpy as np

df = pd.DataFrame( {'a': [1, 2, 3, 4],
                    'b': [1.0, 1.5, 2.0, 2.5],
                    'c': ['a', 'b', 'c', 'd'] }, index=['r1', 'r2', 'r3', 'r4'] )

print df
    a    b  c
r1  1  1.0  a
r2  2  1.5  b
r3  3  2.0  c
f4  4  2.5  d


s1 = df['a']       # Series([1, 2, 3, 4]):  column in dataframe
s1 = df.a          # same

s2 = df.ix[0]      # Series([1, 1.0, 'a'])
```

Series: initializing with index and name

A *Series* is an ordered sequence with index labels. It is like a list in that the elements are ordered; it is like a dictionary in that its indices can be numbers or strings. So it can act like a dictionary as well as a list.

A Series can be initialized with any sequence, such as a tuple or list.

```
>>> import pandas as pd
>>> s = pd.Series([5, 6.5, 'hello', 8])
>>> s
0    5
1    6.5
2    hello
3    8
```

What we see here in printing the Series are the values we entered in the right column, paired with an auto-generated index in the left column. We can conclude that a Series *always has an index* - everything we do with Series and DataFrame object data will be index-aware. Data will be lined up on indexes - and in fact when indexes don't match, data will *not* be aligned.

We can (and often do) set the index ourselves with the *index* attribute, and the name with the *name* attribute.

```
>>> s1 = pd.Series([5, 6, 7, 8], index=['r1', 'r2', 'r3', 'r4'],
                                 name='numbers')
>>> s1
r1    5
r2    6
r3    7
r4    8
Name: numbers
```

We can also construct a **Series** object and then use **object.attribute** syntax to set the **index** or **name**.

```
>>> s2 = pd.Series([1, 2, 3])
>>> s2.index = ['this', 'that', 'other']
>>> s2.name = 'morenums'
s2
this    1
that    2
other   3
Name: morenums
```

Series: accessing elements with indexing and slicing

Since it's an ordered sequence, we can use standard integer indexing to access elements of a Series.

```
>>> s1 = pd.Series([5, 6, 7, 8], index=['r1', 'r2', 'r3', 'r4'], name='numbers')
>>> s1
r1    5
r2    6
r3    7
r4    8
Name:  numbers

>>> s1[0]
5

>>> s1[0:3]         # slice returns a new Series
r1    5
r2    6
r3    7
Name: numbers
```

We can also use the index labels, also for individual elements as well as slices.

```
print s1['r1']           # 5
s1[['r1', 'r3', 'r4']]
                         # r1    5
                         # r3    7
                         # r4    8
                         # Name:  numbers
```

Note this last slice: we specify a list of labels, then pass that list into **s1**'s subscript (square brackets) -- thus the nested square bracket syntax.

Series: setting element values and dtype

You can use the regular integer index to *set* element values in an existing Series. However, the new element value must be the same type as that defined in the Series.

```
>>> s1 = pd.Series([1.5, 2.4, 3.3, 4.2, 5.1], index=['r1', 'r2', 'r3', 'r4', 'r5'])
>>> s1[0] = 'hello'
ValueError: could not convert string to float: hello
```

Note that we never told pandas to store these values as floats. But since they are all floats, pandas decided to set the type - a little like Python setting the type when we first initialize a value (but *unlike* Python in that it does this for a whole container). If a heterogenous (i.e., differently-typed objects) Series is initialized, then type 'object' is applied.

We could easily have included 'hello' in the Series in an initialization:

```
>>> s2 = pd.Series(['hello', 2.4, 3.3, 4.2, 5.1])
>>> s2.dtype
dtype('object')
```

Or, we could create a new Series from the old, setting the type with the **astype** method:

```
s2 = pd.Series(['hello', 2.4, 3.3, 4.2, 5.1])
s1 = s2.astype('object')
s1[0] = 'hello'
```

Series: Vectorized Operations

Operations to Series are *vectorized*, meaning they are propagated across the Series.

```
si = pd.Series([1, 2, 3], index=['r1', 'r2', 'r3'])
                  # print si
                  # r1    1
                  # r2    2
                  # r3    3

sia = si + 1
print sia
                  # r1    2
                  # r2    3
                  # r3    4

sim = si * 2
print sim
                  # r1    2
                  # r2    4
                  # r3    6
```

Series: vectorization with two or more series

We can do computations with two Series, and pandas will match on the indices:

```
si = pd.Series([1, 2, 3], index=['r1', 'r2', 'r3'])
si2 = pd.Series([100, 200, 300], index=['r1', 'r2', 'r3'])

print si + si2
# r1    101
# r2    202
# r3    303
```

But note what happens when indices do not match:

```
si = pd.Series([1, 2, 3], index=['r1', 'r2', 'r3'])
si2 = pd.Series([100, 200, 300], index=['r2', 'r3', 'r4'])

>>> si + si2
r1    NaN          # 'Not a Number values where operation was impossible
r2    102
r3    203
r4    NaN
```

Because there was only a partially common index, pandas was unable to perform the requested operation. So we can see that operations between structures are not positional; they are all index-based.

This orientation confers upon us the ability to work with different structures that share indices that may be out of order or even incomplete in one structure, and know that the values won't be misaligned. Accordingly, it requires us to handle our indices (and with DataFrames, columns as well) so that they represent the data we want.

mask with Series

Oftentimes we want to broadcast a computation conditionally, i.e. only for some elements based on their value. To do this, we can use a *mask*, which goes into subscript-like square brackets:

```
>>> si3 = pd.Series([1, 5, 100, 0, -6, -10, -100])
>>> si3
0      1
1      5
2    100
3      0
4     -6
5    -10
6   -100

>>> si3[ si3 < 0 ] = 0
4     -6
5    -10
6   -100
```

The mask by itself returns a *boolean Series*. This mask can of course be assigned to a name and used by name:

```
>>>si3 = pd.Series([1, 5, 100, 0, -6, -10, -100])
>>>mask = si3 < 0
>>>mask
0   False
1   False
2   False
3   False
4   True
5   True
6   True
>>>si3[ mask ]
4     -6
5    -10
6   -100
```

You can think of this mask as being placed over the Series in question, using the criteria **< 0** to determine whether the element is visible.

Series.apply()

Sometimes our computation is more complex than simple math, or we need to apply a function to each element. We can use **apply()**:

```
>>> ss = Series(['a', 'b', 'C', 'd'])
>>> ssc = ss.apply(str.upper)
>>> ssc
0   A
1   B
2   C
3   D
```

Usually though, we use a custom named function or a lambda, because we usually wanted some custom work done:

```
si = Series([1, 2, 3, 4, 5])
sj = si.apply(lambda x: 'num_' + str(x))
sj
>>> 0   num_1
>>> 1   num_2
>>> 2   num_3
>>> 3   num_4
>>> 4   num_5
```

Notice in all of these operations, pandas returns a new Series. That is the default behavior for most operations on Series or DataFrame.

DataFrame as a container of Series objects

We can think of a DataFrame as a collection of like-indexed Series objects. We can access a column as a Series using a label index:

```
dfi = pd.DataFrame(np.arange(30).reshape(6,5))
dfi.columns = ['c1', 'c2', 'c3', 'c4', 'c5']
dfi.index = ['r1', 'r2', 'r3', 'r4', 'r5', 'r6']

print dfi

          c1  c2  c3  c4  c5
      r1   0   1   2   3   4
      r2   5   6   7   8   9
      r3  10  11  12  13  14
      r4  15  16  17  18  19
      r5  20  21  22  23  24
      r6  25  26  27  28  29
```

We use the column head in a subscript to specify a particular Series:

```
print dfi['c1']

      r1     0
      r2     5
      r3    10
      r4    15
      r5    20
      r6    25
      Name: c1

print type(dfi['c1'])          #
```

So we see that **dfi['c1']** is a Series, which means we can apply all the previously discussed features of a Series to a column (or a row, if needed) in a DataFrame.

DataFrame initializations

There are several ways to initialize a DataFrame in code.

```
df6 = pd.DataFrame(  {'a': [1, 2, 3, 4],
                      'b': [1.0, 1.5, 2.0, 2.5],
                      'c': ['a', 'b', 'c', 'd'] },
                     columns=['a', 'b', 'c']  )


print df6

   a    b  c
0  1  1.0  a
1  2  1.5  b
2  3  2.0  c
3  4  2.5  d

# initializing with a list of lists
dflol = pd.DataFrame([ [1, 0.5, 'a'],
                       [2, 0.6, 'b'],
                       [3, 0.7, 'c'] ], columns=['col1', 'col2', 'col3'],
                                        index=['r1', 'r2', 'r3'])

print dflol

    col1  col2 col3
r1     1   0.5    a
r2     2   0.6    b
r3     3   0.7    c


df6 = pd.DataFrame({'Nevada': {2001: 2.4, 2002: 2.9},
                    'Ohio': {2000: 1.5, 2001: 1.7, 2002: 3.6}} )

print df6

      Nevada  Ohio
2000     NaN   1.5
2001     2.4   1.7
2002     2.9   3.6
```

We will look at acquiring DataFrames from text shortly.

Standard Python operations with DataFrame

DataFrames behave as you might expect

```
df = pd.DataFrame( {'a': [1, 2, 3, 4],
                    'b': [1.0, 1.5, 2.0, 2.5],
                    'c': ['a', 'b', 'c', 'd'] }, index=['r1', 'r2', 'r3', 'r4'] )


print len(df)             # 4

print len(df.columns)     # 3

print max(df['a'])        # 4

print list(df['a'])       # [1, 2, 3, 4]      (column for 'a')

print list(df.ix['r2'])   # [2, 1.5, 'b']     (row for 'r2')

print set(df['a'])        # set([1, 2, 3, 4])


# looping - loops through columns
for colname in df:
    print '{}:  {}'.format(colname, df[colname])

                          # 'a':  pandas.core.series.Series
                          # 'b':  pandas.core.series.Series
                          # 'c':  pandas.core.series.Series


# looping with iterrows -- loops through rows
for index, row in df.iterrows():
    print 'row {}:  {}'.format(index, list(row))


                          # row r1:  [1, 1.0, 'a']
                          # row r2:  [2, 1.5, 'b']
                          # row r3:  [3, 2.0, 'c']
                          # row r4:  [4, 2.5, 'd']
```

Although keep in mind that we generally prefer vectorized operations across columns or rows to looping.

Index and Column Manipulation

Column labels and Index labels are readily manipulable.

```
# rename individual columns
df = df.rename(columns={'a': 'A'})
df = df.rename(index={'alpha': 'affa'})

# change labels wholesale
df.columns=['col1', 'col2', 'col3']
df.index=['a', 'b', 'c']

# reset indices to integer starting with 0
df.reset_index()

# set name for index and columns
df.index.name = 'year'
df.columns.name = 'state'

# reindex ordering by index:
df = df.reindex(reversed(df.index))

df.reindex(columns=reversed(df.columns))
```

Access a Series object through DataFrame column or index labels

Again, we can apply any Series operation on any of the Series within a DataFrame - slice, access by Index, etc.

```
print dfi['c5']
r1     4
r2     9
r3    14
r4    19
r5    24
r6    29
Name: c5

dfi['c5'][0:3]
r1     4
r2     9
r3    14

dfi['c5']['r1']
4
```

Create a DataFrame as a portion of another DataFrame

Oftentimes we want to eliminate one or more columns from our DataFrame. We do this by slicing Series out of the DataFrame, to produce a new DataFrame:

```
>>> dfi[['c1', 'c3']]
    c1  c3
r1   0   2
r2   5   7
r3  10  12
r4  15  17
r5  20  22
r6  25  27
```

Far less often we may want to isolate a row from a DataFrame - this is also returned to us as a Series. Note the column labels have become the Series index, and the row label becomes the Series Name.

```
dfi.ix['r1']
c1    0
c2    1
c3    2
c4    3
c5    4
Name: r1
```

2-dimensional slicing

```
df[['a', 'b']]['alpha': 'gamma']
df.ix[['alpha', 'beta', 'gamma']][['a', 'b']]
```

Slicing columns by index (workaround)

```
dfslice = df.icol(range(4))   # 1st 4 columns of data frame
```

Vectorized operations on DataFrame and Series

As with Series, any operation made on a DataFrame will *broadcast* across *all* elements:

```
>>> dfi
    c1  c2  c3  c4  c5
r1   0   1   2   3   4
r2   5   6   7   8   9
r3  10  11  12  13  14
r4  15  16  17  18  19
r5  20  21  22  23  24
r6  25  26  27  28  29

>>> dfi * 2
    c1  c2  c3  c4  c5
r1   0   2   4   6   8
r2  10  12  14  16  18
r3  20  22  24  26  28
r4  30  32  34  36  38
r5  40  42  44  46  48
r6  50  52  54  56  58
```

Of course we can operate on a single Series within a DataFrame - which translates to applying an operation to an entire column at once:

```
>>> dfi['c1'] * 100
r1       0
r2     500
r3    1000
r4    1500
r5    2000
r6    2500
Name: c1
```

Column-to-column DataFrame Operations

DataFrames as Series containers along with Series vectorization leads us to arguably the single most useful feature of pandas: column-to-column vectorized operations:

```
>>> dfi
    c1  c2  c3  c4  c5
r1   0   1   2   3   4
r2   5   6   7   8   9
r3  10  11  12  13  14
r4  15  16  17  18  19
r5  20  21  22  23  24
r6  25  26  27  28  29

>>> dfi['c1'] = dfi['c3'] * 100  # change an existing column
                                 # based on another column

>>> dfi['c6'] = dfi['c5']        # create a new column
                                 # based on another column

>>> dfi
      c1  c2  c3  c4  c5  c6     # c6 has same values as c5
r1   200   1   2   3   4   4
r2   700   6   7   8   9   9
r3  1200  11  12  13  14  14
r4  1700  16  17  18  19  19
r5  2200  21  22  23  24  24
r6  2800  26  27  28  29  29
```

These types of operations are simply the same Series operations we discussed earlier, but expanded to a DataFrame. The principal purpose and conceptual advantage of a DataFrame is that is lines up Series by index, and allows these operations to be applied across them, each operation vectorized column-wise.

apply() and applymap()

apply() applies a function to a Series in a vectorized operation

```
>>> dfm
   floats  ints strs
0     1.3     1    a
1     2.3     2    b
2     3.3     3    c
3     4.3     4    d

>>> dfm['y'] = dfm['strs'] + dfm['ints'].apply(str)
   floats  ints strs   y
0     1.3     1    a  a1
1     2.3     2    b  b2
2     3.3     3    c  c3
3     4.3     4    d  d4
```

applymap simply works across all cells in a DataFrame - the same way element vectorization does:

```
>>> dfi
    c1  c2  c3  c4  c5
r1   0   1   2   3   4
r2   5   6   7   8   9
r3  10  11  12  13  14
r4  15  16  17  18  19
r5  20  21  22  23  24
r6  25  26  27  28  29

>>> dfi = dfi * 100
>>> dfi
      c1    c2    c3    c4    c5
r1     0   100   200   300   400
r2   500   600   700   800   900
r3  1000  1100  1200  1300  1400
r4  1500  1600  1700  1800  1900
r5  2000  2100  2200  2300  2400
r6  2500  2600  2700  2800  2900

>>> dfi.applymap(lambda x: len(str(x)))
    c1  c2  c3  c4  c5
r1   1   3   3   3   3
r2   3   3   3   3   3
r3   4   4   4   4   4
r4   4   4   4   4   4
r5   4   4   4   4   4
r6   4   4   4   4   4
```

mask

A mask specifies a condition under which a vectorized operation will be applied. We use a conditional against the value in a column, and change its value if the condition is met:

```
>>> dfi
    c1  c2  c3  c4  c5
r1   0   1   2   3   4
r2   5   6   7   8   9
r3  10  11  12  13  14
r4  15  16  17  18  19
r5  20  21  22  23  24
r6  25  26  27  28  29

>>> mask = dfi['c1'] < 20
>>> dfi['c1'][ mask ] = 0
    c1  c2  c3  c4  c5
r1   0   1   2   3   4
r2   0   6   7   8   9
r3   0  11  12  13  14
r4   0  16  17  18  19
r5  20  21  22  23  24
r6  25  26  27  28  29
```

...but take special note - if you refer to a column as part of the assignment value, you must mask that column too:

```
>>> mask = dfi['c2'] > 10
dfi['c6'][ mask ] = dfi['c5'][ mask ]
    c1  c2  c3  c4  c5  c6
r1   0   1   2   3   4
r2   5   6   7   8   9
r3  10  11  12  13  14  14
r4  15  16  17  18  19  19
r5  20  21  22  23  24  24
r6  25  26  27  28  29  29
```

nan and fillna()

If pandas can't insert a value (because indexes are misaligned or for other reasons), it inserts a special value call **NaN** (not a number) in its place.

If we wish to fill the dataframe with an alternate value, we can use **fillna()**, which like all operations vectorizes across the structure:

```
>>> df
    c1  c2   c3
0    6  NaN   2
0    6    1   2
0  NaN    3   2
>>> df = df.fillna(0)
>>> df
    c1  c2  c3
0    6   0   2
0    6   1   2
0    0   3   2
```

Concatenating / Appending

concat() can join dataframes either horizontally or vertically.

```
df3 = pd.concat([df, df2])            # horizontal concat
df4 = pd.concat([df, df2], axis=1)    # vertical concat
```

merge

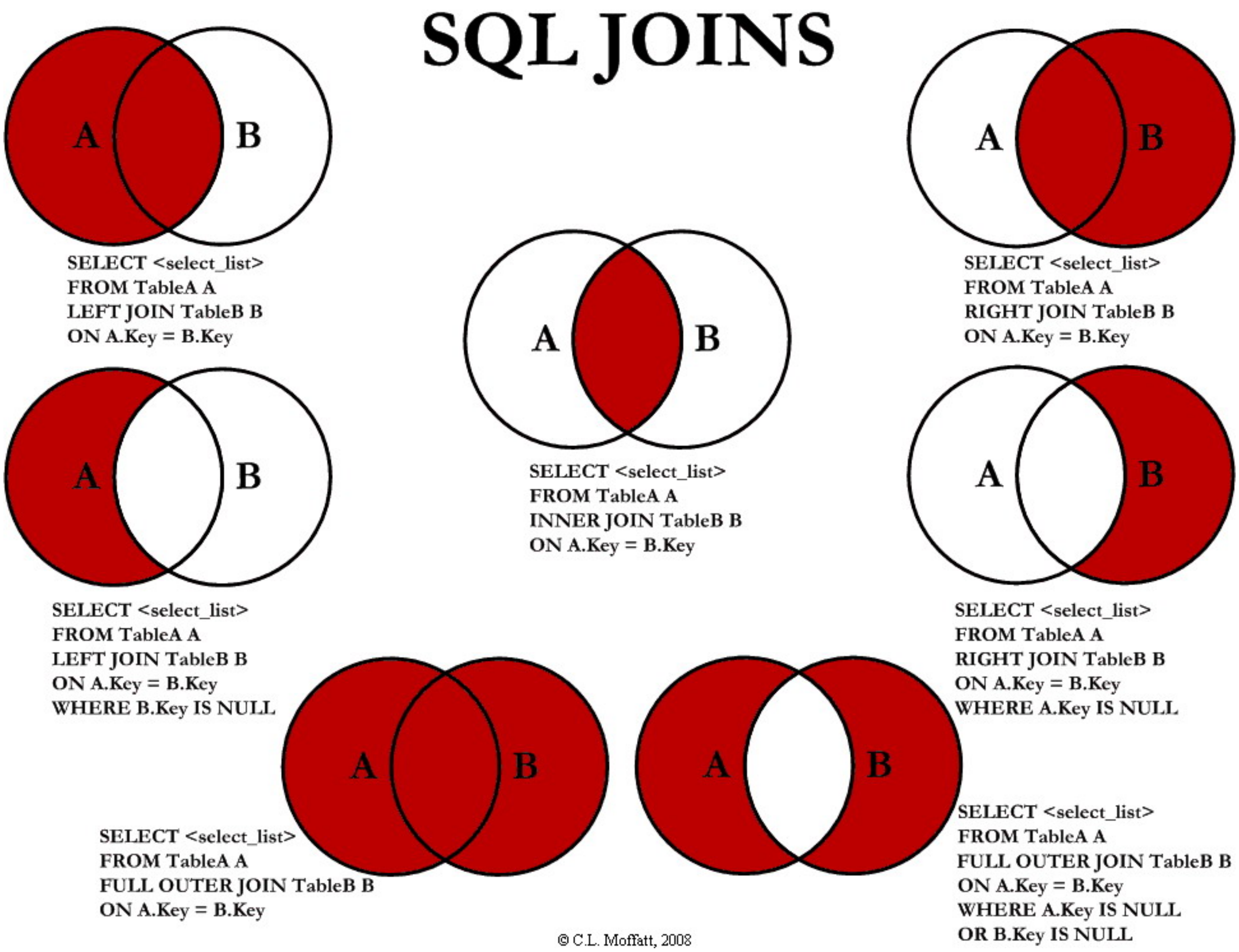
Merge performs a relational database-like **join** on two dataframes. We can join on a particular field and the other fields will align accordingly.

```
>>> dfi
    c1  c2  c3  c4  c5
r1   0   1   2   3   4
r2   5   6   7   8   9
r3  10  11  12  13  14
r4  15  16  17  18  19
r5  20  21  22  23  24
r6  25  26  27  28  29
>>> dfi2
    c1  c6  c7
r1   0  41  42
r2   5  51  52
r3  10  61  62
r4  15  71  72
r5  20  81  82
r6  25  91  92

>>> dfi.merge(dfi2, on='c1', how='left')
    c1  c2  c3  c4  c5  c6  c7
r1   0   1   2   3   4  41  42
r2   5   6   7   8   9  51  52
r3  10  11  12  13  14  61  62
r4  15  16  17  18  19  71  72
r5  20  21  22  23  24  81  82
r6  25  26  27  28  29  91  92
```

The merge joins the table. You can choose to join on the index, or one or more columns. **how=** describes the type of join, and the choices are similar to that in relationship databases:

```
Merge method    SQL Join Name          Description
left            LEFT OUTER JOIN        Use keys from left frame only
right           RIGHT OUTER JOIN       Use keys from right frame only
outer           FULL OUTER JOIN        Use union of keys from both frames
inner           INNER JOIN             Use intersection of keys from both frames
```



groupby

A groupby operation performs the same type of operation as the database GROUP BY. Grouping rows of the table by the value in a particular column, you can do things like sum or count the values found in another column.

```
>>> dfi
    c1  c2
r1  'a'  1
r2  'a'  6
r3  'b' 11
r4  'b' 16
r5  'c' 21
r6  'c' 26

>>> dfi.groupby('c1').sum()
    c2
c1
a    7
b   27
c   47

>>> dfi.groupby('c1').mean()
```

List of selected groupby functions

```
count
mean
sum
size
describe
min
max
```

Working with Data Sources

Pandas has native support for CSV, JSON, Excel and XML

CSV

```
# reading from Fama-French file (abbreviated, no header)
df = pd.read_csv('FF_abbreviated.txt', sep='\s+',
                                       names=['date', 'MktRF', 'SMB', 'HML', 'RF'])


# Fama-French non-abbreviated (with headers and footers)
df = pd.read_csv('F-F_Research_Data_Factors_daily.txt', skiprows=5, sep='\s+',
                                                        names=['date', 'MktRF', 'SMB', 'HML', 'RF'])


# write to STDOUT
df.to_csv(sys.stdout, na_rep='NULL')
```

JSON

```
df = pd.DataFrame( {'a': [1, 2, 3, 4],
                    'b': [1.0, 1.5, 2.0, 2.5],
                    'c': ['a', 'b', 'c', 'd'] }, index=['r1', 'r2', 'r3', 'r4'] )

pd.json.dump(df, open('df.json', 'w'))

mydict = pd.json.load(open('df.json'))
new_df = pd.DataFrame(mydict)
```

Excel

```
xls_file = pd.ExcelFile('data.xls')

table = xls_file.parse('Sheet1')
```

From Clipboard

This option is excellent for cutting and pasting data from websites

```
df = pd.read_clipboard(skiprows=5, sep='\s+',
                       names=['date', 'MktRF', 'SMB', 'HML', 'RF'])
```