pandas and numpy

Python 2

home (../handouts.html)

pandas and numpy: Introduction

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

* Excel-like numeric calculations, particularly column-wise and row-wise calculations (*vectorization*)

* SQL-like merging, grouping and aggregating

* emphasis on aligning data from multiple sources and cleaning and normalizing missing data

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

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.

pandas Reference

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

cheat sheet (Treehouse)

```
https://s3.amazonaws.com/assets.datacamp.com/blog_assets/PandasPythonForDataScience.pdf (https://s3.amazonaws.c
```

docs on any pandas function or DataFrame method

```
import pandas as pd

help(pd.read_csv)       # help on the read_csv function of pandas

df = pd.DataFrame()     # initialize a DataFrame
help(df.join)           # help on the join() method of a DataFrame
```

pandas official documentation

```
http://pandas.pydata.org/pandas-docs/stable (http://pandas.pydata.org/pandas-docs/stable)

http://pandas.pydata.org/pandas-docs/version/0.19.0/pandas.pdf (http://pandas.pydata.org/pandas-docs/version/0.
```

pandas cookbook

```
http://pandas.pydata.org/pandas-docs/stable/cookbook.html (http://pandas.pydata.org/pandas-docs/stable/cookbook
```

pandas textbook "Python for Data Analysis" by Wes McKinney

```
http://www3.canisius.edu/~yany/python/Python4DataAnalysis.pdf (http://www3.canisius.edu/~yany/python/Python4Data
```

(If the above link goes stale, simply search **Python for Data Analysis pdf**.)

Please keep in mind that pandas is in active developemnt (latest version: 0.19.0)

pandas objects

The *DataFrame* is the primary object in pandas; a DataFrame column or row can be isolated as a *Series* object; columns and rows are enumerated with the *Index* 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.)

```
import pandas as pd
df = pd.DataFrame({'a': [1, 2, 3], 'b': [10, 20, 30], 'c': [100, 200, 300]},
                  index=['r1', 'r2', 'r3'])

print df
            #      a   b    c
            # r1   1  10  100
            # r2   2  20  200
            # r3   3  30  300

print df['c']['r2']    # 200
```

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

```
# read a column as a Series
bcol = df['b']
print bcol
            # r1    10
            # r2    20
            # r3    30
            # Name: b, dtype: int64


# read a row as a Series (using .loc)
oneidx = df.loc['r2']
print oneidx
            # a      2
            # b     20
            # c    200
            # Name: r2, dtype: int64
```

Index
* an object that provides indexing for both the Series (its item index) and the DataFrame (its column or row index).

```
columns = df.columns   # Index([u'a',  u'b',  u'c'],  dtype='object')
idx = df.index         # Index([u'r1', u'r2', u'r3'], dtype='object')
```

Reading pandas DataFrames from Data Sources

Pandas can read in a Dataframe object from CSV, JSON, Excel and XML formats.

CSV

```
# read from file
df = pd.read_csv('quarterly_revenue_2017Q4.csv')


# write to file
wfh = open('output.csv', 'w')
df.to_csv(wfh, na_rep='NULL')


# reading from Fama-French file (the abbreviated file, no header)
# sep= indicates the delimiter on which to split() the fields
# names= indicates the column heads
df = pd.read_csv('FF_abbreviated.txt', sep='\s+',
                                       names=['date', 'MktRF', 'SMB', 'HML', 'RF'])


# reading from Fama-French non-abbreviated (the main file including headers and footers)
# skiprows=5:  start reading 5 rows down
df = pd.read_csv('F-F_Research_Data_Factors_daily.txt', skiprows=5, sep='\s+',
                                                         names=['date', 'MktRF', 'SMB', 'HML', 'RF'])
```

Excel

```
# reading from excel file (2 steps)
xls_file = pd.ExcelFile('data.xls')    # produce a file 'reader' object
df = xls_file.parse('Sheet1')          # parse a selected sheet to a DataFrames

# write to excel
df.to_excel('data2.xls', sheet_name='Sheet1')
```

JSON

```
# sample df for demo purposes
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'] )



# write dataframe to JSON
pd.json.dump(df, open('df.json', 'w'))

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

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'])
```

The DataFrame: Initializing and Slicing

THE *dataframe* is the pandas workhorse structure. It is a 2-dimensional structure with columns and rows (i.e., like a spreadsheet).

Initializing

```
import pandas as pd
import numpy as np

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


# init with list of lists
df = pd.DataFrame( [  [ 'a', 'b', 'c' ],
                      [  1,   2,   3  ],
                      [  10,  20, 30  ]   ]  )

print df

            #      0   1   2
            #  0   a   b   c
            #  1   1   2   3
            #  2  10  20  30


# init with dict of lists and index
df = pd.DataFrame( {'a': [1, 2, 3, 4],
                    'b': [1.0, 1.5, 2.0, 2.5],
                    'c': ['a', 'b', 'c', 'd'],
                    'd': [100, 200, 300, 400] },
                    index=['r1', 'r2', 'r3', 'r4'] )

print df

            #     a    b  c    d
            # r1  1  1.0  a  100
            # r2  2  1.5  b  200
            # r3  3  2.0  c  300
            # r4  4  2.5  d  400
```

Accessing columns or rows (Series objects)

```
cola = df['a']       # Series with [1, 2, 3, 4] and index ['r1', 'r2', 'r3', 'r4']
cola = df.a          # same

row2 = df.loc['r2']  # Series [2, 1.5, 'b', 200] and index ['a', 'b', 'c', 'd']
```

Accessing DataFrame slices (DataFrame objects)

```
dfslice1 = df[ ['a', 'b', 'c'] ]  # slice out 1st 3 columns
dfslice2 = df['r1': 'r2']         # slice out 1st 2 rows (label indexing upper bound is inclusive!)
dfslice3 = df[0:2]                # slice out same 1st 2 rows
```

The Series: subscripting and slicing

A *Series* is pandas object representing a column or row in a DataFrame.

A Series can be initialized on its own and made part of a DataFrame

```
s1 = pd.Series([1, 2, 3, 4])
s2 = pd.Series([1.0, 1.5, 2.0, 2.5])

df = pd.DataFrame({'a': s1, 'b': s2})
```

A DataFrame can be seen as a list of Series objects.

```
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
                  # r4  4  2.5  d
```

DataFrame subscript accesses a column

```
s1 = df['a']      # Series object (column)

                  # r1    1
                  # r2    2
                  # r3    3
                  # r4    4
                  # Name: a, dtype: int64
```

(Series name is the column head, index are the row labels)

DataFrame **.loc** indexer accesses the rows

```
r1 = df.loc['r2'] # Series object (row)

                  # a      2
                  # b    1.5
                  # c      b
                  # Name: r2, dtype: object
```

(Series name is the row label, index are the column heads)

Series items can be accessed through the index labels, or by index position.

```
print r1['a']        # 2
print r1.a           # 2
print r1[0]          # 2
```

Working with Series Objects as part of a DataFrame

We usually work with the DataFrame subscript directly, resulting in a double-subscript

Initialize a DataFrame

```
import pandas as pd

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
                  # r4  4  2.5  d
```

Access a Series (DataFrame column)

```
print df['b']

                  # r1    1.0
                  # r2    1.5
                  # r3    2.0
                  # r4    2.5
                  # Name: b, dtype: float64
```

Access a single value in a DataFrame

```
# by Series index
print df['b'][0]            # 1.0

# by Series row label
print df['b']['r2']         # 1.5
```

Access a slice in a DataFrame

```
# by Series indices
print df['c'][1:3]

                  # r2    b
                  # r3    c
                  # Name: c, dtype: object


# by Series row labels
print df['c'][['r2', 'r3', 'r4']]

                  # r2    b
                  # r3    c
                  # r4    d
                  # Name: c, dtype: object
```

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

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 = pd.DataFrame({'c1': [0,   1,  2, 3,   4],
                   'c2': [5,   6,  7, 8,   9],
                   'c3': [10, 11, 12, 13, 14],
                   'c4': [15, 16, 17, 18, 19],
                   'c5': [20, 21, 22, 23, 24],
                   'c6': [25, 26, 27, 28, 29] },
           index = ['r1', 'r2', 'r3', 'r4', 'r5'])

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


print dfi[['c1', 'c3']]
                  #     c1  c3
                  # r1   0  10
                  # r2   1  11
                  # r3   2  12
                  # r4   3  13
                  # r5   4  14
```

```
print dfi.ix[['r1', 'r3', 'r5']]

                  #     c1  c2  c3  c4  c5  c6
                  # r1   0   5  10  15  20  25
                  # r3   2   7  12  17  22  27
                  # r5   4   9  14  19  24  29


print dfi.ix[['r1':'r3']]

                  #     c1  c2  c3  c4  c5  c6
                  # r1   0   5  10  15  20  25
                  # r2   1   6  11  16  21  26
                  # r3   2   7  12  17  22  27
```

2-dimensional slicing

```
print dfi[['c1', 'c2']]['r1': 'r2']

                  #     c1  c2
                  # r1   0   5
                  # r2   1   6
```

Slicing columns by index

```
dfslice = dfi.iloc[:, [0, 1, 2, 3]]   # 1st 4 columns of data frame
print dfslice

                  #     c1  c2  c3  c4
                  # r1   0   5  10  15
                  # r2   1   6  11  16
                  # r3   2   7  12  17
                  # r4   3   8  13  18
                  # r5   4   9  14  19
```

The Index

An Index object is used to specify a DataFrame's columns or index, or a Series' index.

Columns and Indices

A DataFrame makes use of two Index objects: one to represent the columns, and one to represent the rows.

```
df = pd.DataFrame( {'a': [1, 2, 3, 4],
                    'b': [1.0, 1.5, 2.0, 2.5],
                    'c': ['a', 'b', 'c', 'd'],
                    'd': [100, 200, 300, 400] },
                    index=['r1', 'r2', 'r3', 'r4'] )
```

Columns or index labels can be reset using the dataframe's **rename()** method.

```
df = df.rename(columns={'a': 'A', 'b': 'B', 'c': 'C', 'd': 'D'},
               index={'r1': 'R1', 'r2': 'R2', 'r3': 'R3', 'r4': 'R4'})
```

The columns or index can also be set directly using the dataframe's attributes (although this is more prone to error).

```
df.columns = ['A', 'B', 'C', 'D']

# 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))
```

Dataframe and Series dtypes

Unlike core Python containers (but similar to a database table), pandas cares about object type. Wherever possible, pandas will assign a type to a column Series and attempt to maintain the type's integrity.

This is done for the same reason it is done with database tables: speed and space efficiency.

In the below DataFrame, Python "sniffs out" the type of a column Series. If all values match, pandas will set the type.

```
import pandas as pd

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
                  # r4  4  2.5  d


print df.dtypes

                  # a      int64       # note special pandas types int64 and float64
                  # b    float64
                  # c     object       # 'object' is general-purpose type, covers strings or mixed-type columns
                  # dtype: object
```

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.

```
df['b'][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.

We can change a dtype for a Series:

```
df.a = df.a.astype('object')         # or df['a'] = df['a'].astype('object')

df['a'][0] = 'hello'
```

Vectorized Operations

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

```
import pandas as pd

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
                  # r4  4  2.5  d

# 'single value':  assign the same value to all cells in a column Series
df.a = 0
print df

                  #     a    b  c
                  # r1  0  1.0  a
                  # r2  0  1.5  b
                  # r3  0  2.0  c
                  # r4  0  2.5  d


# 'calculation':  compute a new value for all cells in a column Series
df.b = df.b * 2

print df

                  #     a    b  c
                  # r1  0  2.0  a
                  # r2  0  3.0  b
                  # r3  0  4.0  c
                  # r4  0  5.0  d
```

Adding New Columns with Vectorized Values

We can also add a new column to the Dataframe based on values or computations:

```
df = pd.DataFrame( {'a': [0, 0, 0, 0],
                    'b': [2.0, 3.0, 4.0, 5.0],
                    'c': ['a', 'b', 'c', 'd'] }, index=['r1', 'r2', 'r3', 'r4'] )


df['d'] = 3.14           # new column, each field set to same value

print df

                  #     a    b  c  d
                  # r1  1  2.0  a  3.14
                  # r2  2  3.0  b  3.14
                  # r3  3  4.0  c  3.14
                  # r4  4  5.0  d  3.14


df['e'] = df.a + df.b    # vectorized computation to new column

print df

                  #     a    b  c     d  e
                  # r1  1  2.0  a  3.14  3.0
                  # r2  2  3.0  b  3.14  5.0
                  # r3  3  4.0  c  3.14  7.0
                  # r4  4  5.0  d  3.14  9.0
```

mask: conditional vectorization

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

```
import pandas as pd

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 -20  a
                    # r2  2 -10  b
                    # r3  3  10  c
                    # r4  4  20  d

df['b'][ df['b'] < 0 ] = 0    # for each 'b' value that is < 0, set to 0

print df

                    #     a   b  c
                    # r1  1   0  a
                    # r2  2   0  b
                    # r3  3  10  c
                    # r4  4  20  d
```

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

```
mask = df['a'] > 2
print mask          # printing just for illustration

                    # r1    False
                    # r2    False
                    # r3    True
                    # r4    True
                    # Name: a, dtype: bool
```

Of course a vector operation can be filtered with a mask:

```
mask = df['a'] > 2
df['a'][ mask ] = df['b'] * 2

print df
                  #      a   b  c
                  # r1   1   0  a      # 'a' not > 3:  no effect
                  # r2   2   0  b      # 'a' not > 3:  no effect
                  # r3  20  10  c      # 'a' > 3, so now a == b * 2
                  # r4  40  20  d      # 'a' > 3, so now a == b * 2
```

You can think of this mask as being placed over the Series in question ('a'), using the criteria **< 3** to determine whether the element is visible.

negating a mask

a tilde (**~**) in front of a mask creates its inverse:

```
mask = df['a'] > 2
df['a'][ ~mask ] = 0

print df
                  #      a   b  c
                  # r1   0   0  a
                  # r2   0   0  b
                  # r3  20  10  c
                  # r4  40  20  d
```

Series.apply(): vectorize a function call over a column

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

```
import pandas as pd

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
                  # r4  4  2.5  d


df['d'] = df.c.apply(str.upper)

print df
                  #     a    b  c  d
                  # r1  1  1.0  a  A
                  # r2  2  1.5  b  B
                  # r3  3  2.0  c  C
                  # r4  4  2.5  d  D
```

Many times, though, we use a custom named function or a lambda, because we want some custom work done:

```
df['e'] = df['a'].apply(lambda x: '$' + str(x * 1000) )


print df
                  #     a    b  c  d      e
                  # r1  1  1.0  a  A  $1000
                  # r2  2  1.5  b  B  $2000
                  # r3  3  2.0  c  C  $3000
                  # r4  4  2.5  d  D  $4000
```

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.

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:

```
print df
                  #     c1  c2   c3
                  # 0    6  NaN   2
                  # 0    6    1   2
                  # 0  NaN    3   2

df = df.fillna(0)
print 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.

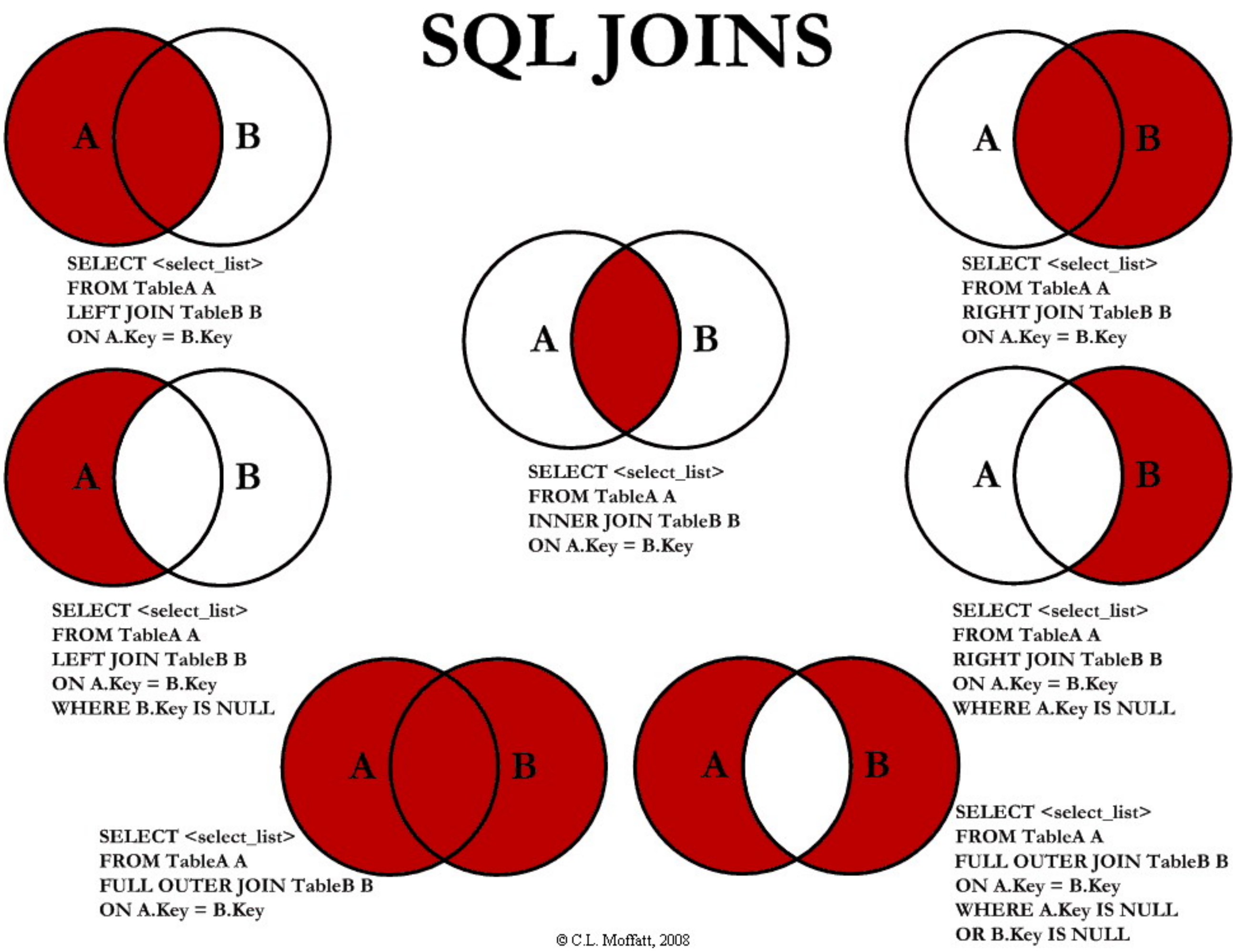
```
print dfi
                  #     c1  c2  c3  c4  c5
                  # r1   0   1   2   3   4
                  # r2   5   6   7   8   9
                  # r3  10  11  12  13  14
                  # r4  15  16  17  18  19
                  # r5  20  21  22  23  24
                  # r6  25  26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

print 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 perform aggregate sums, counts or custom aggregations.

This simple hypothetical table shows client names, regions, revenue values and type of revenue.

```
df = pd.DataFrame( { 'company': ['Alpha', 'ALPHA', 'ALPHA', 'BETA', 'Beta', 'Beta', 'Gamma', 'Gamma', 'Gamma'],
                     'region':  ['NE', 'NW', 'SW', 'NW', 'SW', 'NE', 'NE', 'SW', 'NW'],
                     'revenue': [10, 9, 2, 15, 8, 2, 16, 3, 9],
                     'revtype': ['retail', 'retail', 'wholesale', 'wholesale', 'wholesale',
                                 'retail', 'wholesale', 'retail', 'retail'] } )

print df

                  #   company region  revenue    revtype
                  # 0   Alpha     NE       10     retail
                  # 1   ALPHA     NW        9     retail
                  # 2   ALPHA     SW        2  wholesale
                  # 3    BETA     NW       15  wholesale
                  # 4    Beta     SW        8  wholesale
                  # 5    Beta     NE        2     retail
                  # 6   Gamma     NE       16  wholesale
                  # 7   Gamma     SW        3     retail
                  # 8   Gamma     NW        9     retail
```

(Due to a quirk in the data, the client names are either all uppercase or titlecase, and we're choosing not to normalize these values - we'll need to correct for this in one of our aggregations.)

Built-in Aggregation Functions

Aggregations are provided by the DataFrame **groupby()** method, which returns a special **groupby** object. If we'd like to see revenue aggregated by region, we can simply select the column to aggregate and call an aggregation function on this object:

```
# revenue sum by region
rsbyr = df.groupby('region').sum()   # call sum() on the groupby object
print rsbyr

                  #         revenue
                  # region
                  # NE           28
                  # NW           33
                  # SW           13


# revenue average by region
rabyr = df.groupby('region').mean()
print rabyr

                  #           revenue
                  # region
                  # NE       9.333333
                  # NW      11.000000
                  # SW       4.333333
```

The result is dataframe with the 'region' as the index and 'revenue' as the sole column.

Note that although we didn't specify the revenue column, pandas noticed that the other columns were not numbers and therefore should not be included in a sum or mean.

If we ask for a count, python counts each column (which will be the same for each). So if we'd like the analysis to be limited to one or more coluns, we can simply slice the dataframe first:

```
# count of all columns by region
print df.groupby('region').count()

                  #         company  revenue  revtype
                  # region
                  # NE            3        3        3
                  # NW            3        3        3
                  # SW            3        3        3


# count of companies by region
dfcr = df[['company', 'region']]       # dataframe slice:  only 'company' and 'region'
print dfcr.groupby('region').count()

                  #         company
                  # region
                  # NE            3
                  # NW            3
                  # SW            3
```

Multi-column aggregation

To aggregate by values in two combined columns, simply pass a list of columns by which to aggregate -- the result is called a "multi-column aggregation":

```
print df.groupby(['region', 'revtype']).sum()

                 #                   revenue
                 # region revtype
                 # NE     retail          12
                 #        wholesale       16
                 # NW     retail          18
                 #        wholesale       15
                 # SW     retail           3
                 #        wholesale       10
```

List of selected built-in groupby functions

```
              count()
              mean()
              sum()
              min()
              max()
              describe() (prints out several columns including sum, mean, min, max)
```

Custom groupby functions

We can design our own custom functions -- we simply use **apply()** and pass a function (you might remember similarly passing a function from the **key=** argument to **sorted()**). Here is the equivalent of the **sum()** function, written as a custom function:

```
def get_sum(df_slice):
    return sum(df_slice['revenue'])

print df.groupby('region').apply(get_sum)  # custom function: same as groupby('region').sum()

                 # region
                 # NE    28
                 # NW    33
                 # SW    13
                 # dtype: int64
```

As was done with **sorted()**, pandas calls our groupby function multiple times, once with each group. The argument that Python passes to our custom function is a dataframe slice containing just the rows from a single grouping -- in this case, a specific region (i.e., it will be called once with a silce of **NE** rows, once with **NW** rows, etc. The function should be made to return the desired value for that slice -- in this case, we want to see the sum of the revenue column (as mentioned, this is simply illustrating a function that does the same work as the built-in **.sum()** function).

(For a better view on what is happening with the function, print **df_slice** inside the function -- you will see the values in each slice printed.)

Here is a custom function that returns the median ("middle value") for each region:

```
def get_median(df):
    listvals = sorted(list(df['revenue']))
    lenvals = len(listvals)
    midval = listvals[ lenvals / 2 ]
    return midval

print df.groupby('region').apply(get_median)

                  # region
                  # NE     10
                  # NW      9
                  # SW      3
                  # dtype: int64
```

Custom aggregator function

Most aggregations aggregate based on a column value or a combination of column values. If more work is needed to identify a group, we can supply a custom function for this operation as well.

In this case, remember that our quirky dataset has company names that are uppercase or lowercase -- thus, aggregating on the company name will treat different casing as a different company:

```
print df.groupby('company').sum()

                  #          revenue
                  # company
                  # ALPHA         11
                  # Alpha         10
                  # BETA          15
                  # Beta          10
                  # Gamma         28
```

So, we can process this column value (or even include other column values) by referencing a function in the call to **groupby()**:

```
def get_lowercase(index):
    row = df.ix[index]              # a Series with the row values for this index
    return row['company'].lower()   # returning this row's company name, lowercased

print df.groupby(get_lowercase).sum()

                  #        revenue
                  # alpha       21
                  # beta        25
                  # gamma       28
```

The value passed to the function is the index of a row. We can thus use the **ix** attribute with the index value to access the row. This function isolates the company name within the row and returns its lowercased value.

using lambdas

Of course any of these simple functions can be rewritten as a lambda (and in many cases, should be, as in the above case since the function references the dataframe directly):

```
def get_lowercase(index):
    row = df.ix[index]              # a Series with the row values for this index
    return row['company'].lower()   # returning this row's company name, lowercased

print df.groupby(lambda index:  df.ix[index]['company'].lower()).sum()
```