UsingPandas

February 21, 2015

```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt
```

```
In [2]: import warnings
        warnings.filterwarnings("ignore")
```

0.1 Using Pandas

The `numpy` module is excellent for numerical computations, but to handle missing data or arrays with mixed types takes more work. The `pandas` module provides objects similar to R's data frames, and these are more convenient for most statistical analysis. The `pandas` module also provides many mehtods for data import and manipulaiton that we will explore in this section.

Pandas for R Users

```
In [3]: import pandas as pd
        import statsmodels.api as sm
        from pandas import Series, DataFrame, Panel
        from string import ascii_lowercase as letters
        from scipy.stats import chisqprob
```

0.1.1 Series

Series is a 1D array with axis labels.

```
In [4]: # Creating a series and extracting elements.

        xs = Series(np.arange(10), index=tuple(letters[:10]))
        print xs[:3],'\n'
        print xs[7:], '\n'
        print xs[::3], '\n'
        print xs[['d', 'f', 'h']], '\n'
        print xs.d, xs.f, xs.h
```

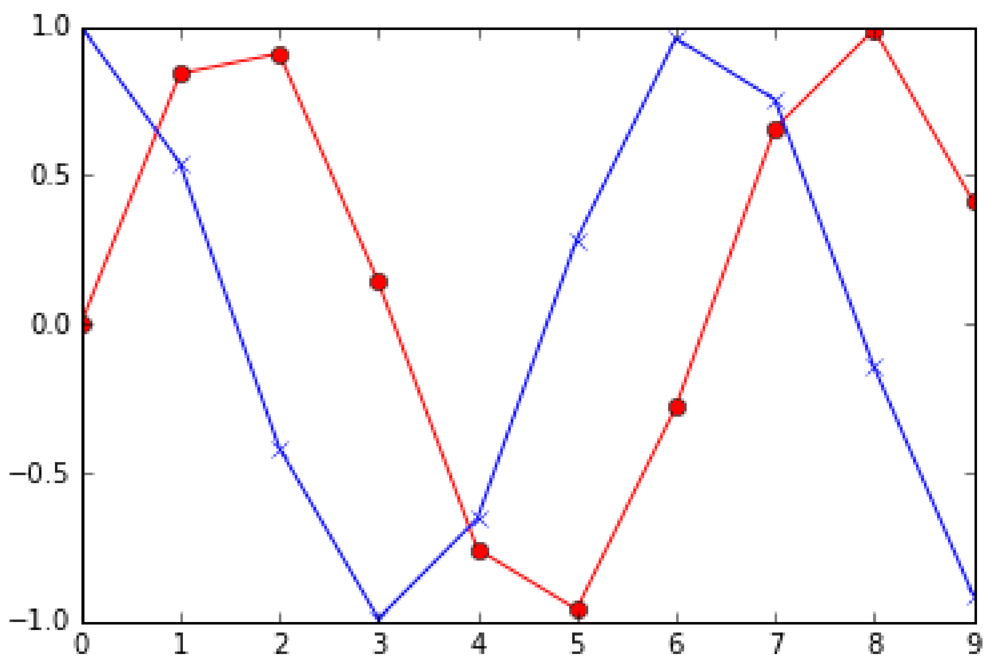
```
a    0
b    1
c    2
dtype: int64

h    7
i    8
j    9
dtype: int64
```

```
a    0
d    3
g    6
j    9
dtype: int64

d    3
f    5
h    7
dtype: int64

3 5 7
```

In [5]:
```
# All the numpy functions wiill work with Series objects, and return another Series

y1, y2 = np.mean(xs), np.var(xs)
y1, y2
```

Out[5]:
```
(4.5, 8.25)
```

In [6]:
```
# Matplotlib will work on Series objects too
plt.plot(xs, np.sin(xs), 'r-o', xs, np.cos(xs), 'b-x');
```



In [7]:
```
# Convert to numpy arrays with values

print xs.values
```

```
[0 1 2 3 4 5 6 7 8 9]
```

In [8]:
```
# The Series datatype can also be used to represent time series
```

```
import datetime as dt
from pandas import date_range

# today = dt.date.today()
today = dt.datetime.strptime('Jan 21 2015', '%b %d %Y')
print today, '\n'
days = date_range(today, periods=35, freq='D')
ts = Series(np.random.normal(10, 1, len(days)), index=days)

# Extracting elements
print ts[0:4], '\n'
print ts['2015-01-21':'2015-01-28'], '\n' # Note - includes end time
```

```
2015-01-21 00:00:00

2015-01-21     9.719261
2015-01-22     8.894461
2015-01-23    10.074521
2015-01-24    10.769334
Freq: D, dtype: float64

2015-01-21     9.719261
2015-01-22     8.894461
2015-01-23    10.074521
2015-01-24    10.769334
2015-01-25    10.159401
2015-01-26     8.992754
2015-01-27     9.681121
2015-01-28     9.908445
Freq: D, dtype: float64
```

```
In [9]: # We can geenerate statistics for time ranges with the resample method
        # For example, suppose we are interested in weekly means, standard deviations and sum-of-square

        df = ts.resample(rule='W', how=('mean', 'std', lambda x: sum(x*x)))
        df
```

```
Out[9]:                  mean       std    <lambda>
        2015-01-25   9.923396  0.688209  494.263430
        2015-02-01  10.357088  0.848930  755.208973
        2015-02-08  10.224806  0.869441  736.362134
        2015-02-15  10.672230  0.942680  802.607338
        2015-02-22   9.785174  1.012906  676.403270
        2015-03-01   9.495084  1.472653  182.481942
```

0.1.2 DataFrame

For statisticians, a DataFrame is similar to the R dataframe object. For everyone else, it is like a simple tabular spreadsheet. Each column is a Series object.

```
In [10]: # Note that the df object in the previous cell is a DataFrame
         print type(df)
```

```
<class 'pandas.core.frame.DataFrame'>
```

```
In [11]: # Renaming columns
         # The use of mean and std are problmeatic because there are also methods in dataframe with tho
         # Also, <lambda> is unifnormative
         # So we would like to give better names to the columns of df

         df.columns = ('mu', 'sigma', 'sum_of_sq')
         print df
```

```
mu     sigma   sum_of_sq
2015-01-25   9.923396  0.688209  494.263430
2015-02-01  10.357088  0.848930  755.208973
2015-02-08  10.224806  0.869441  736.362134
2015-02-15  10.672230  0.942680  802.607338
2015-02-22   9.785174  1.012906  676.403270
2015-03-01   9.495084  1.472653  182.481942
```

```
In [12]: # Extracitng columns from a DataFrame

         print df.mu, '\n' # by attribute
         print df['sigma'], '\n' # by column name
```

```
2015-01-25     9.923396
2015-02-01    10.357088
2015-02-08    10.224806
2015-02-15    10.672230
2015-02-22     9.785174
2015-03-01     9.495084
Freq: W-SUN, Name: mu, dtype: float64

2015-01-25    0.688209
2015-02-01    0.848930
2015-02-08    0.869441
2015-02-15    0.942680
2015-02-22    1.012906
2015-03-01    1.472653
Freq: W-SUN, Name: sigma, dtype: float64
```

```
In [13]: # Extracting rows from a DataFrame

         print df[1:3], '\n'
         print df['2015-01-21'::2]
```

```
mu     sigma   sum_of_sq
2015-02-01  10.357088  0.848930  755.208973
2015-02-08  10.224806  0.869441  736.362134

                   mu     sigma   sum_of_sq
2015-01-25   9.923396  0.688209  494.263430
2015-02-08  10.224806  0.869441  736.362134
2015-02-22   9.785174  1.012906  676.403270
```

```
In [14]: # Extracting blocks and scalars

         print df.iat[2, 2], '\n' # extract an element with iat()
         print df.loc['2015-01-25':'2015-03-01', 'sum_of_sq'], '\n' # indexing by label
         print df.iloc[:3, 2], '\n'  # indexing by position
         print df.ix[:3, 'sum_of_sq'], '\n' # by label OR position
```

```
736.362134378

2015-01-25    494.263430
2015-02-01    755.208973
2015-02-08    736.362134
2015-02-15    802.607338
2015-02-22    676.403270
2015-03-01    182.481942
Freq: W-SUN, Name: sum_of_sq, dtype: float64

2015-01-25    494.263430
2015-02-01    755.208973
2015-02-08    736.362134
Freq: W-SUN, Name: sum_of_sq, dtype: float64

2015-01-25    494.263430
2015-02-01    755.208973
2015-02-08    736.362134
Freq: W-SUN, Name: sum_of_sq, dtype: float64
```

```
In [15]: # Using Boolean conditions for selecting eleements

         print df[(df.sigma < 1) & (df.sum_of_sq < 700)], '\n' # need parenthesis because of operator p
         print df.query('sigma < 1 and sum_of_sq < 700') # the query() method allows more readable quer
```

```
mu     sigma  sum_of_sq
2015-01-25  9.923396  0.688209  494.26343

                  mu     sigma  sum_of_sq
2015-01-25  9.923396  0.688209  494.26343
```

0.1.3 Panels

Panels are 3D arrays - they can be thought of as dictionaries of DataFrames.

```
In [16]: df= np.random.binomial(100, 0.95, (9,2))
         dm = np.random.binomial(100, 0.9, [12,2])
         dff = DataFrame(df, columns = ['Physics', 'Math'])
         dfm = DataFrame(dm, columns = ['Physics', 'Math'])
         score_panel = Panel({'Girls': dff, 'Boys': dfm})
         print score_panel, '\n'
```

```
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 12 (major_axis) x 2 (minor_axis)
Items axis: Boys to Girls
Major_axis axis: 0 to 11
Minor_axis axis: Physics to Math
```

```
In [17]: score_panel['Girls'].transpose()
```

Out[17]:

	0	1	2	3	4	5	6	7	8	9	10	11
Physics	95	95	96	95	93	95	96	94	96	NaN	NaN	NaN
Math	95	95	94	92	91	92	96	95	97	NaN	NaN	NaN

```
In [18]: # find physics and math scores of girls who scored >= 93 in math
         # a DataFrame is returned
         score_panel.ix['Girls', score_panel.Girls.Math >= 93, :]
```

```
Out[18]:    Physics  Math
         0       95    95
         1       95    95
         2       96    94
         6       96    96
         7       94    95
         8       96    97
```

0.1.4 Split-Apply-Combine

Many statistical summaries are in the form of split along some property, then apply a funciton to each subgroup and finally combine the results into some object. This is known as the 'split-apply-combine' pattern and implemnented in Pandas via groupby() and a function that can be applied to each subgroup.

```
In [19]: # import a DataFrame to play with
         try:
             tips = pd.read_pickle('tips.pic')
         except:
             tips = pd.read_csv('https://raw.github.com/vincentarelbundock/Rdatasets/master/csv/reshape2
             tips.to_pickle('tips.pic')
```

```
In [20]: tips.head(n=4)
```

```
Out[20]:    Unnamed: 0  total_bill   tip     sex smoker  day    time  size
         0           1       16.99  1.01  Female     No  Sun  Dinner     2
         1           2       10.34  1.66    Male     No  Sun  Dinner     3
         2           3       21.01  3.50    Male     No  Sun  Dinner     3
         3           4       23.68  3.31    Male     No  Sun  Dinner     2
```

```
In [21]: # We have an extra set of indices in the first column
         # Let's get rid of it

         tips = tips.ix[:, 1:]
         tips.head(n=4)
```

```
Out[21]:    total_bill   tip     sex smoker  day    time  size
         0       16.99  1.01  Female     No  Sun  Dinner     2
         1       10.34  1.66    Male     No  Sun  Dinner     3
         2       21.01  3.50    Male     No  Sun  Dinner     3
         3       23.68  3.31    Male     No  Sun  Dinner     2
```

```
In [22]: # For an example of the split-apply-combine pattern, we want to see counts by sex and smoker s
         # In other words, we split by sex and smoker status to get 2x2 groups,
         # then apply the size function to count the number of entries per group
         # and finally combine the results into a new multi-index Series

         grouped = tips.groupby(['sex', 'smoker'])
         grouped.size()
```

```
Out[22]: sex     smoker
         Female  No        54
                 Yes       33
         Male    No        97
                 Yes       60
         dtype: int64
```

In [37]:
```
# If you need the margins, use the crosstab function

pd.crosstab(tips.sex, tips.smoker, margins=True)
```

Out[37]:
```
smoker   No  Yes  All
sex
Female   54   33   87
Male     97   60  157
All     151   93  244
```

In [23]:
```
# If more than 1 column of resutls is generated, a DataFrame is returned

grouped.mean()
```

Out[23]:
```
               total_bill       tip      size
sex    smoker
Female No       18.105185  2.773519  2.592593
       Yes      17.977879  2.931515  2.242424
Male   No       19.791237  3.113402  2.711340
       Yes      22.284500  3.051167  2.500000
```

In [24]:
```
# The returned results can be further manipulated via apply()
# For example, suppose the bill and tips are in USD but we want EUR

import json
import urllib

# get current conversion rate
converter = json.loads(urllib.urlopen('http://rate-exchange.appspot.com/currency?from=USD&to=E
print converter
grouped['total_bill', 'tip'].mean().apply(lambda x: x*converter['rate'])
```

```
{u'to': u'EUR', u'rate': 0.879191, u'from': u'USD'}
```

Out[24]:
```
               total_bill       tip
sex    smoker
Female No       15.917916  2.438453
       Yes      15.805989  2.577362
Male   No       17.400278  2.737275
       Yes      19.592332  2.682558
```

In [25]:
```
# We can also transform the original data for more convenient analysis
# For example, suppose we want standardized units for total bill and tips

zscore = lambda x: (x - x.mean())/x.std()

std_grouped = grouped['total_bill', 'tip'].transform(zscore)
std_grouped.head(n=4)
```

Out[25]:
```
   total_bill       tip
0   -0.153049 -1.562813
1   -1.083042 -0.975727
2    0.139661  0.259539
3    0.445623  0.131984
```

In [26]:
```
# Suppose we want to apply a set of functions to only some columns
grouped['total_bill', 'tip'].agg(['mean', 'min', 'max'])
```

```
Out[26]:                total_bill                      tip
                                mean   min    max       mean   min   max
         sex    smoker
         Female No       18.105185  7.25  35.83  2.773519  1.00   5.2
                Yes      17.977879  3.07  44.30  2.931515  1.00   6.5
         Male   No       19.791237  7.51  48.33  3.113402  1.25   9.0
                Yes      22.284500  7.25  50.81  3.051167  1.00  10.0
```

```
In [27]: # We can also apply specific functions to specific columns
         df = grouped.agg({'total_bill': (min, max), 'tip': sum})
         df
```

```
Out[27]:                    tip total_bill
                            sum        min    max
         sex    smoker
         Female No       149.77       7.25  35.83
                Yes       96.74       3.07  44.30
         Male   No       302.00       7.51  48.33
                Yes      183.07       7.25  50.81
```

0.1.5 Using statsmodels

Many of the basic statistical tools available in R are replicted in the `statsmodels` package. We will only show one example.

```
In [28]: # Simulate the genotype for 4 SNPs in a case-control study using an additive genetic model

         n = 1000
         status = np.random.choice([0,1], n )
         genotype = np.random.choice([0,1,2], (n,4))
         genotype[status==0] = np.random.choice([0,1,2], (sum(status==0), 4), p=[0.33, 0.33, 0.34])
         genotype[status==1] = np.random.choice([0,1,2], (sum(status==1), 4), p=[0.2, 0.3, 0.5])
         df = DataFrame(np.hstack([status[:, np.newaxis], genotype]), columns=['status', 'SNP1', 'SNP2'
         df.head(6)
```

```
Out[28]:    status  SNP1  SNP2  SNP3  SNP4
         0       0     2     1     2     0
         1       1     1     0     2     2
         2       1     0     1     2     1
         3       1     2     2     1     2
         4       1     1     2     0     1
         5       1     0     0     1     2
```

```
In [29]: # Use statsmodels to fit a logistic regression to  the data
         fit1 = sm.Logit.from_formula('status ~ %s' % '+'.join(df.columns[1:]), data=df).fit()
         fit1.summary()
```

```
Optimization terminated successfully.
         Current function value: 0.642824
         Iterations 5
```

```
Out[29]: <class 'statsmodels.iolib.summary.Summary'>
         """
                                    Logit Regression Results
         ==============================================================================
         Dep. Variable:                 status   No. Observations:                 1000
```

```
Model:                          Logit   Df Residuals:                      995
Method:                           MLE   Df Model:                            4
Date:                Thu, 22 Jan 2015   Pseudo R-squ.:                 0.07259
Time:                        15:34:43   Log-Likelihood:                -642.82
converged:                       True   LL-Null:                       -693.14
                                        LLR p-value:                 7.222e-21
==============================================================================
                 coef    std err          z      P>|z|      [95.0% Conf. Int.]
------------------------------------------------------------------------------
Intercept     -1.7409      0.203     -8.560      0.000        -2.140    -1.342
SNP1           0.4306      0.083      5.173      0.000         0.267     0.594
SNP2           0.3155      0.081      3.882      0.000         0.156     0.475
SNP3           0.2255      0.082      2.750      0.006         0.065     0.386
SNP4           0.5341      0.083      6.404      0.000         0.371     0.698
==============================================================================
"""
```

In [30]:
```
# Alternative using GLM - similar to R
fit2 = sm.GLM.from_formula('status ~ SNP1 + SNP2 + SNP3 + SNP4', data=df, family=sm.families.Bi
print fit2.summary()
print chisqprob(fit2.null_deviance - fit2.deviance, fit2.df_model)
print(fit2.null_deviance - fit2.deviance, fit2.df_model)
```

```
Generalized Linear Model Regression Results
==============================================================================
Dep. Variable:                 status   No. Observations:                 1000
Model:                            GLM   Df Residuals:                      995
Model Family:                Binomial   Df Model:                            4
Link Function:                  logit   Scale:                             1.0
Method:                          IRLS   Log-Likelihood:                -642.82
Date:                Thu, 22 Jan 2015   Deviance:                       1285.6
Time:                        15:34:43   Pearson chi2:                 1.01e+03
No. Iterations:                     5
==============================================================================
                 coef    std err          t      P>|t|      [95.0% Conf. Int.]
------------------------------------------------------------------------------
Intercept     -1.7409      0.203     -8.560      0.000        -2.140    -1.342
SNP1           0.4306      0.083      5.173      0.000         0.267     0.594
SNP2           0.3155      0.081      3.882      0.000         0.156     0.475
SNP3           0.2255      0.082      2.750      0.006         0.065     0.386
SNP4           0.5341      0.083      6.404      0.000         0.371     0.698
==============================================================================
7.22229516479e-21
(100.63019840179481, 4)
```

0.2 Using R from IPython

While Python support for statstical computing is rapidly improving (especially with the pandas, statsmodels and scikit-learn modules), the R ecosystem is staill vastly larger. However, we can have our cake and eat it too, since IPyhton allows us to run R (almost) seamlessly with the Rmagic (rpy2.ipython) extension.

There are two ways to use Rmagic - using %R (appleis to single line) and %%R (applies to entire cell). Python objects can be passed into R with the -i flag and R objects pased out with the -o flag.

In [31]:
```
! pip install ggplot &> /dev/null
```

0.2.1 Using Rmagic

```
In [32]: %load_ext rpy2.ipython
```

```
In [33]: %%R -i df,status -o fit

         fit <- glm(status ~ ., data=df)
         print(summary(fit))
         print(fit$null.deviance - fit$deviance)
         print(fit$df.null - fit$df.residual)
         with(fit, pchisq(null.deviance - deviance, df.null - df.residual, lower.tail = FALSE))
```

```
Call:
glm(formula = status ~ ., data = df)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-0.7927  -0.4464   0.2073   0.4301   0.8999

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  0.10014    0.04323   2.316  0.02075 *
SNP1         0.09904    0.01874   5.285 1.55e-07 ***
SNP2         0.07217    0.01836   3.932 9.01e-05 ***
SNP3         0.05135    0.01856   2.767  0.00576 **
SNP4         0.12372    0.01869   6.620 5.86e-11 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for gaussian family taken to be 0.2269642)

    Null deviance: 250.00  on 999  degrees of freedom
Residual deviance: 225.83  on 995  degrees of freedom
AIC: 1361.9

Number of Fisher Scoring iterations: 2

[1] 24.16657
[1] 4
[1] 7.396261e-05
```

Using rpy2 directly

```
In [34]: import rpy2.robjects as ro
         from rpy2.robjects.packages import importr

         base = importr('base')

         fit_full = ro.r("lm('mpg ~ wt + cyl', data=mtcars)")
         print(base.summary(fit_full))
```

```
Call:
lm(formula = "mpg ~ wt + cyl", data = mtcars)
```

```
Residuals:
    Min      1Q  Median      3Q     Max
-4.2893 -1.5512 -0.4684  1.5743  6.1004

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  39.6863     1.7150  23.141  < 2e-16 ***
wt           -3.1910     0.7569  -4.216 0.000222 ***
cyl          -1.5078     0.4147  -3.636 0.001064 **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 2.568 on 29 degrees of freedom
Multiple R-squared:  0.8302,	Adjusted R-squared:  0.8185
F-statistic: 70.91 on 2 and 29 DF,  p-value: 6.809e-12
```

0.2.2 Using R from pandas

Reading R dataset into Python

In [35]:
```
import pandas.rpy.common as com

df = com.load_data('mtcars')
print df.head(n=6)
```

```
mpg  cyl  disp   hp  drat     wt   qsec  vs  am  gear  carb
0  21.0    6   160  110  3.90  2.620  16.46   0   1     4     4
1  21.0    6   160  110  3.90  2.875  17.02   0   1     4     4
2  22.8    4   108   93  3.85  2.320  18.61   1   1     4     1
3  21.4    6   258  110  3.08  3.215  19.44   1   0     3     1
4  18.7    8   360  175  3.15  3.440  17.02   0   0     3     2
5  18.1    6   225  105  2.76  3.460  20.22   1   0     3     1
```

In [36]:
```
%load_ext version_information

%version_information numpy, matplotlib, pandas, statsmodels
```

Out[36]:

Software	Version
Python	2.7.9 64bit [GCC 4.2.1 (Apple Inc. build 5577)]
IPython	2.3.1
OS	Darwin 13.4.0 x86_64 i386 64bit
numpy	1.9.1
matplotlib	1.4.2
pandas	0.15.1
statsmodels	0.5.0
Thu Jan 22 15:34:45 2015 EST	