MSIS 5223: Tutorial 2 – Using Data in Python

Instructions

The purpose of this tutorial is to help you become familiar with using Python. The most fundamental functions in analytics using Python involve importing data, manipulating that data, and understanding how to access that data. Once you become familiar with manipulating the data, you should have no problem familiarizing yourself with it.

If you wish to see examples of the code used in this tutorial, please follow along and use the Python code titled *Data Derivation and Selection*.

1. Setting the Working Directory

By default, the working directory of Python in Windows is your Documents folder. To determine what your current directory is, simply import the os library and run the command getcwd(). See Figure 1 below.

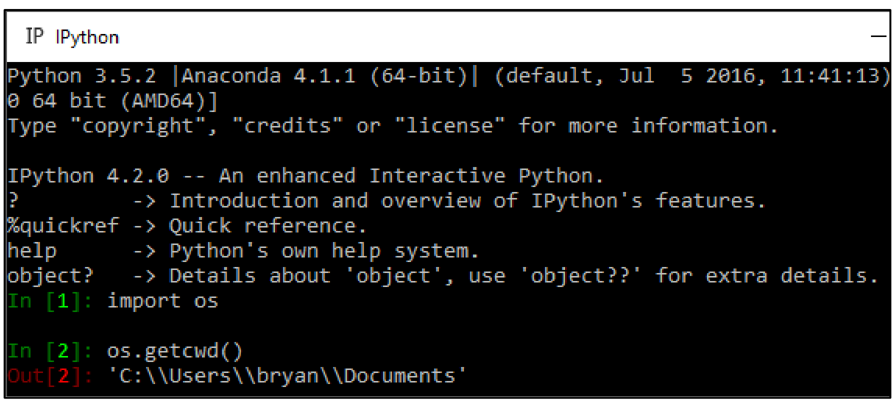


Figure 1 Obtaining the Working Directory

To change the working directory, simply use the chdir() function. See Figure 2 for a demonstration.

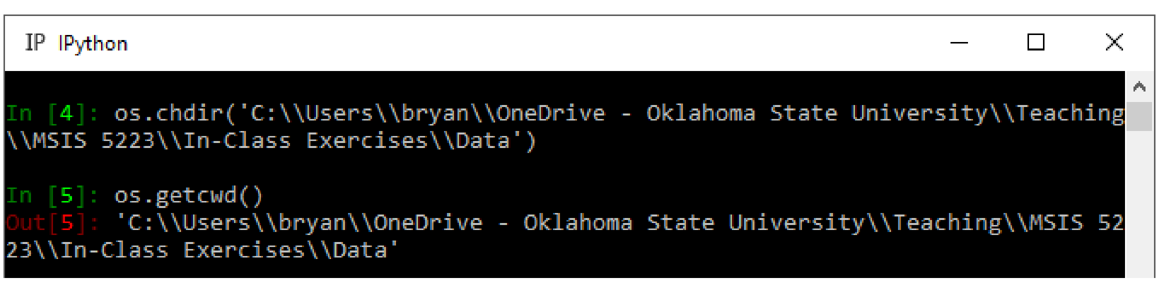


Figure 2 Changing the Working Directory

2. Data Input

Importing a data file into iPython is a simple process, much like it is in R. The function is simply pd.read_table() where pd is the pandas library. The Pandas library provides the ability to access data as a dataframe, similar to R. At the start of working on any project, always import the Pandas library (see Figure 1). Additionally, you should also import Numpy and matplotlib.pyplot.

```
IP IPython
Python 3.5.2 |Anaconda 4.1.1 (64-bit)| (default, Jul  5 2016, 11:41:13) [MSC v.
1900 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 4.2.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: import pandas as pd

In [2]: import numpy as np

In [3]: import matplotlib.pyplot as plt

In [4]:
```

Figure 3 Importing Pandas Library

For this part of the tutorial, open the ozone.data.txt file. Again, this is similar to R. See Figure 4 below for the syntax. Like in R, the dataframe automatically assigns index values to each of the records. Alternatively, you can assign one of the columns to be the index by using the argument index_col='column_name' where column_name is the name of the column within your data.

```
IP IPython
In [8]: ozone_data = pd.read_table('ozone.data.txt', sep='\t')

In [9]: ozone_data
Out[9]:
    rad  temp  wind  ozone
0   190    67   7.4     41
1   118    72   8.0     36
2   149    74  12.6     12
3   313    62  11.5     18
4   299    65   8.6     23
5    99    59  13.8     19
6    19    61  20.1      8
7   256    69   9.7     16
8   290    66   9.2     11
9   274    68  10.9     14
10   65    58  13.2     18
11  334    64  11.5     14
12  307    66  12.0     34
13   78    57  18.4      6
```

Figure 4 Opening a Data File in Python

Referencing columns in Python is a little different than in R. In R, the dollar symbol $ is used to denote a column; in Python, simply use a period between the dataframe and the column name. To lookup the names of the columns within your dataframe, use the columns attribute. See the figure below for both examples.

```
IP IPython
Out[11]: Index(['rad', 'temp', 'wind', 'ozone'],
 dtype='object')

In [12]: ozone_data.rad
Out[12]:
0       190
1       118
2       149
3       313
4       299
5        99
6        19
7       256
8       290
```

Figure 5 Column Data and Metadata

As mentioned in another document, iPython provides tab-completion as a function. This can be helpful, especially when your dataset contains over 50 columns of data. Try this out on your own. Type `ozone_data.r` and hit the tab key. You should see a short list of attributes belonging to the dataframe that begin with the letter "R." The column `rad` is listed first. Try the exercise again, this time type in `ozone_data.ra`, then hit tab. Now only two values appear in the output.

To quickly view your data, you can use the `head(num)` or `tail(num)` functions, where `num` is the number of records you wish returned. See Figure 6 below for an example.

```
IP IPython

In [14]: ozone_data.head(5)
Out[14]:
   rad  temp  wind  ozone
0  190    67   7.4     41
1  118    72   8.0     36
2  149    74  12.6     12
3  313    62  11.5     18
4  299    65   8.6     23
```

Figure 6 Quick View of Rows in Dataframe

3. Changing Column Data

Removing columns in Python is a straightforward process. After importing your data into a dataframe, use the function drop() to remove the columns of interest. For this portion of the tutorial, use the car.test.frame.txt file. The dataset contains the following variables, in order: *Price*, *Country*, *Reliability*, *Mileage*, *Type*, *Weight*, *Disp.*, and *HP*.

In this situation, you are not interested in the column *Mileage*, *Type*, and *Weight*. These columns are the fourth, fifth, and sixth respectively. To remove the columns, type the following:

```
car_data.drop(['Mileage','Type','Weight'], axis=1, inplace=True)
```

or

```
car_data.drop(car_data.columns[[3,4,5]], axis=1, inplace=True)
```

The first example is a way to remove individual columns of data using the name of the column. If you have a long list of columns to remove, typing out individual column names would become tedious. The second example shows a way in which to remove columns using their index value. It should be noted that in Python, the indexing starts at 0 for both columns and rows; in R, indexing values start at 1. Thus, the fourth column in Python has an index value of 3, not 4.

Renaming columns is a simple process. Python provides many possible ways to change a column header. The first requires you to type in all of the column names, even the ones you are not changing. This can be tedious, however, if you have a lot of columns. If that is the case, the second method is better; also, it is recommended that you remove unwanted columns or create a subset of your data prior to renaming columns.

For the first method, assume you would like to rename the column *Mileage* to *Total_Mileage*. Use the following syntax:

```
car_data.columns = ['Price', 'Country', 'Reliability', 'Total_Mileage', 'Type'
, 'Weight', 'Disp.', 'HP']
```

The second method uses the function rename(). This method will only allow the renaming of a single column of data. This avoids having to write out all the names of each column. Rename the column *Total_Mileage* back to *Mileage*. See the figure below.

```
car_data.rename(columns={'Total_Mileage':'Mileage'}, inplace=True)
```

```
IP IPython
In [94]: car_data.columns
Out[94]:
Index(['Price', 'Country', 'Reliability', 'Mileage', 'Type', 'Weight', 'Disp.',
       'HP'],
      dtype='object')

In [95]: car_data.columns = ['Price', 'Country', 'Reliability', 'Total_Mileage', 'Type'
, 'Weight', 'Disp.', 'HP']

In [96]: car_data.columns
Out[96]:
Index(['Price', 'Country', 'Reliability', 'Total_Mileage', 'Type', 'Weight',
       'Disp.', 'HP'],
      dtype='object')

In [97]: car_data.rename(columns={'Total_Mileage':'Mileage'}, inplace=True)

In [98]: car_data.columns
Out[98]:
Index(['Price', 'Country', 'Reliability', 'Mileage', 'Type', 'Weight', 'Disp.',
       'HP'],
      dtype='object')
```

Figure 7 Changing Column Names

In addition to changing the columns within your dataframe, you may wish to change the actual data within your columns. For example, say the value of *Price* for the first record was entered incorrectly. The correct value is 111. Using the index values of your record and column, you can assign the new value. The first record has the index value of 0 and *Price* is the first column and has an index value of 0.

```
car_data.ix[0,0] = 111
```

4. Working with a Dataframe

This next section deals with working with the data within a dataframe. Prior to even performing simple descriptive statistics, it may be beneficial to familiarize yourself with the data contained in the dataframe. This entails selecting specific columns or rows, sorting data, and selecting data based on conditions.

Sometimes it is useful to select specific values within your data. Like R, Python uses an indexing system, like the majority of statistical packages, for both rows and columns. In Python, just like R, because there is no GUI, you must specify the value using the indices, or subscripts as they are sometimes called.

The indices look like this

[*r*, *c*]

where *r* is the row value and *c* is the column value. Recall the car data contains eight columns of data. For this portion of the tutorial, use the car.test.frame.txt file.

Prior to starting, a few points should be made.

- In Python, the indexing starts at 0 for both columns and rows; in R, indexing values start at 1
- Pandas is not inclusive with index ranges. In R, for example, if you type in `car_data[1:5,:]` the first five rows would be returned. In Python, however, only the first four columns would be returned. To obtain the first five rows in Python, type `car_data.ix[0:5,:]` (remember, 5 actually is the 6th record in Python).
- In R, you can leave a blank space in the brackets to represent all values. For example, to obtain data for row 4, you type car_data[4,]. In Python, you cannot leave a blank space; instead, you need the colon: car_data.ix[3,:]
- If you want to find the data for the fourth row of data, for transmission (which is the third column), you would type the following: `car_data.ix[3,2]`

It is also possible to pull the data for a single row. To do so, you would leave the column index value "blank" by leaving a colon, like so:

```
car_data.ix[36,:]
```

This brings up all of the data for row 37 for all columns of data. Likewise, you can pull data for a single column for all rows of data. If you would like to pull data for *type*, then you would leave a colon in the row index value, like so:

```
car_data.ix[:,4]
```

It is quite possible that you would like specific columns for a specific row. Assume you would like only columns 2, 3, and 4 for row 57. There are two ways to do this. The first method uses a range of columns, not inclusive; the second method specifies each column using indices or column names:

Method 1: `car_data.ix[56,1:4]`

Method 2b: `car_data.ix[56,[1,2,3]]`

Method 2b: `car_data.ix[56,['`*Country*`', '`*Reliability*`', '`*Mileage*`']]`

The first method is useful if you have a long range of columns and you do not want to type out each one individually. The second method gives you the freedom to specify the columns of interest. If you only wanted columns 2 and 4 without 3, you could not use the first method; only the second method would allow this:

```
car_data.ix[56,[1,3]]
car_data.ix[56,['Country', 'Mileage')]
```

You can also save an entire column of data as a new dataframe like so:

```
mileage_data = car_data.ix[:,3]
mileage_data = car_data.ix[:,['Mileage']]
```

Of course, a simple way of doing this would be referencing the column of data from the dataframe itself. Simply type in the following code:

```
mileage_data = car_data.Mileage
```

This returns only the data for the column *Mileage*. This is useful for referencing the column to be used in various equations and functions. This will only work for a single column of data. If you would like to save two or more columns of data into a new dataframe, you would have to use the previously shown methods using index values.

Three more functions that are useful should be noted here. The first one identifies the unique values contained within your data. If you have categorical data, this can be helpful in determining the various values contained in your variable (in R, the term "factor" is used instead of categorical). For example, *Country* contains eight unique values. To see a list of these values, type in the following code:

```
pd.unique(car_data.Country)
```

The second function, or set of functions, provides basic information on row and column size. This may be useful if you do not know the number of records your data contains or the number of columns. Two methods are provided as follows:

Rows and columns: `car_data.shape`

Rows: `len(car_data.index)`

Columns: `len(car_data.columns)`

The third set of functions allows you to sort your data. You can sort ascending, descending, select multiple columns to sort by, include only certain columns in your results, and many other combinations. A simple sort would look like this:

```
car_data.sort_values(by='Reliability')
```

This sorts the data only on the column *Reliability*. Notice the missing values in the column are listed toward the bottom of the sort. If you would like the missing values at the top, type in the following:

```
car_data.sort_values(by='Reliability',na_position='first')
```

Sometimes you are only interested in the largest or smallest value within a column of data. Using the functions nlargest() and nsmallest(), you can obtain those values respectively. For example, say you want the six largest values for Reliability.

```
car_data.nlargest(6,'Reliability')
```

Perhaps you would like the six largest values for both *Reliability* and *Mileage*. The code is not too much of an extension from the previous line.

```
car_data.nlargest(6,['Reliability','Mileage'])
```

Looking at the next figure, the output from the two different lines of code reveal subtle differences. While the same records appear in both outputs, the order they appear differs. In the second output where *Mileage* is added as a condition, the record with an index value of 11 is listed first. This is because it has the largest value for both *Reliability* and *Mileage*.

```
IP IPython
In [69]: car_data.nlargest(6,'Reliability')
Out[69]:
    Price    Country  Reliability  Mileage   Type  Weight  Disp.   HP
3    6635  Japan/USA          5.0       32  Small    2260     91   92
4    6599      Japan          5.0       32  Small    2440    113  103
6    7399  Japan/USA          5.0       33  Small    2275     97   90
8    9599      Japan          5.0       25  Small    2295    109   90
10   8748  Japan/USA          5.0       29  Small    2390     97  102
11   6488      Japan          5.0       35  Small    2075     89   78

In [70]: car_data.nlargest(6,['Reliability','Mileage'])
Out[70]:
    Price    Country  Reliability  Mileage   Type  Weight  Disp.   HP
11   6488      Japan          5.0       35  Small    2075     89   78
6    7399  Japan/USA          5.0       33  Small    2275     97   90
3    6635  Japan/USA          5.0       32  Small    2260     91   92
4    6599      Japan          5.0       32  Small    2440    113  103
10   8748  Japan/USA          5.0       29  Small    2390     97  102
8    9599      Japan          5.0       25  Small    2295    109   90
```

Figure 8 Obtaining Six Largest Values

Sorting multiple columns is also straight forward. Just add the additional columns, in the order that you would like them sorted. If you want to sort by *Reliability* and *Mileage* in that order, you would list *Reliability* first; if you want *Mileage* sorted first, then list *Mileage* first. See the following for the examples:

car_data.sort_values(by=['Reliability','Mileage'])

car_data.sort_values(by=['Mileage','Reliability'])

Observe that the sorting code is entered in the index for row; no column index was specified. One of the issues in sorting the data this way is R includes all of the other columns in your dataset. What if you don't want all the other columns included in your sort? Unfortunately, this is not as straight forward as it is in R. The columns you want need to be pulled out into a new dataframe and then sorted.

5. Subsampling in Python

As mentioned in the R tutorial, one basic approach to sampling is basing your subsample on a percentage of the overall sample size. For example, say you would like to sample 60% of your original data and perform an analysis on it. As a refresher, the steps include

- determine how many rows is 60% of your data,
- find out how many rows are in your dataframe,
- determine the range of your sample,
- and perform the splitting.

Use the ozone data to perform this operation. You can look at the figure below to follow the steps within Python. On line 26, notice the function at the end of the line, astype(int). The function np.round() returns a number with an decimal point, even when you request zero decimal points. To change the datatype from float to an integer, you have to recast the object using astype(int).

```
IP IPython

In [26]: splitnum = np.round((len(ozone_data.index) * 0.6), 0).astype(int)

In [27]: splitnum
Out[27]: 67

In [28]: ozone_data_sample = ozone_data.sample(n=splitnum,replace=False)

In [29]: len(ozone_data_sample.index)
Out[29]: 67
```

Figure 9 Sampling 60% of Data

Another option is to skip the steps calculating the number of rows to sample. The function `sample()` provides the argument `frac` that allows you to specify the percentage you would like sampled. This is a quicker method than that previously shown.

```
IP IPython
In [35]: ozone_data_sample = ozone_data.sample(frac=0.6,replace=False)

In [36]: len(ozone_data_sample.index)
Out[36]: 67
```

Figure 10 Using Fraction Within Sample()

Often you will want to select a subsample based on certain conditions or criteria given the data you have. For this example, the `seedlings_data` will be used. The seedlings data contains three columns, *cohort*, *death*, and *gapsize*. *Cohort* contains two unique values as shown below.

```
IP IPython
In [31]: seedlings_data.head()
Out[31]:
      cohort  death  gapsize
0  September      7   0.5889
1  September      3   0.6869
2  September     12   0.9800
3  September      1   0.1921
4  September      4   0.2798

In [32]: seedlings_data.columns
Out[32]: Index(['cohort', 'death', 'gapsize'], dtype='object')

In [33]: seedlings_data.cohort.unique()
Out[33]: array(['September', 'October'], dtype=object)
```

Figure 11 Seedlings Data

Assume you would like to perform an analysis only on seedlings planted in September. Or, put another way, you want data not obtained in October. The code below shows how you perform both of these operations.

```
IP IPython
In [41]: seedlings_data[seedlings_data.cohort=='September']
Out[41]:
       cohort  death  gapsize
0   September      7   0.5889
1   September      3   0.6869
2   September     12   0.9800
3   September      1   0.1921
4   September      4   0.2798
5   September      2   0.2607
6   September      6   0.9467
7   September      6   0.6375
8   September      8   0.9000
9   September      3   0.8237
10  September      1   0.5979
```

Figure 12 Data in September

```
IP IPython
In [42]: seedlings_data[seedlings_data.cohort!='October']
Out[42]:
      cohort  death  gapsize
0  September      7   0.5889
1  September      3   0.6869
2  September     12   0.9800
3  September      1   0.1921
4  September      4   0.2798
5  September      2   0.2607
6  September      6   0.9467
7  September      6   0.6375
8  September      8   0.9000
9  September      3   0.8237
```

Figure 13 Data Not in October

After perusing your September-data, you realize that you only want data with a death value less than or equal to 10. This is another simple process. You just append additional conditions using the symbol &.

```
IP IPython
In [51]: seedlings_data[(seedlings_data.cohort=='September')&(seedlings_data.death<=10)]
Out[51]:
       cohort  death  gapsize
0   September      7   0.5889
1   September      3   0.6869
3   September      1   0.1921
4   September      4   0.2798
5   September      2   0.2607
6   September      6   0.9467
7   September      6   0.6375
8   September      8   0.9000
9   September      3   0.8237
10  September      1   0.5979
11  September      1   0.2914
```

Figure 14 Selection of Seedlings Based on Cohort and Death

What if you want to find data that is for September or October? You would type in both conditions and separate them using the OR operator, which is |.

```
IP IPython
In [52]: seedlings_data[(seedlings_data.cohort=='September')|(seedlings_data.cohort=='October')]
Out[52]:
      cohort  death  gapsize
0  September      7   0.5889
1  September      3   0.6869
2  September     12   0.9800
3  September      1   0.1921
4  September      4   0.2798
5  September      2   0.2607
6  September      6   0.9467
```

Figure 15 The OR Operator in Selection

Taking this one step further, you can select data for September or October and has a death value less than or equal to 5. Here are two different statements that will select two different datasets:

- seedlings_data[((seedlings_data.cohort=='September')|(seedlings_data.cohort=='October'))&(seedlings_data.death<=5)]

- seedlings_data[(seedlings_data.cohort=='September')|((seedlings_data.cohort=='October')&(seedlings_data.death<=5))]

Can you spot the difference betwee these two? Here is a hint: Look at the placement of the parentheses. In statement 1, the parentheses surround the OR operator; in the second statement, they surround the AND operator. The first statement selects data that is 1) September and less than 10 or 2) October and less than 10. The second statement selects data that is 1) September or 2) October and less than 10. What kind of data would you select if you didn't use any parentheses? Think about that for a while.

You can also test your dataframe for missing values. The first method utilizes two functions and returns a TRUE-FALSE value based on whether it is complete; i.e. TRUE indicates no missing values whereas FALSE indicates missing values. This function is notnull().

```
IP IPython
In [71]: pd.notnull(seedlings_data)
Out[71]:
   cohort death gapsize
0    True  True    True
1    True  True    True
2    True  True    True
3    True  True    True
4    True  True    True
5    True  True    True
6    True  True    True
7    True  True    True
8    True  True    True
9    True  True    True
10   True  True    True
```

Figure 16 Detecting Missing Values: notnull()

To perform the opposite test, use the function isnull(). This returns TRUE for missing values and FALSE when no data missing.

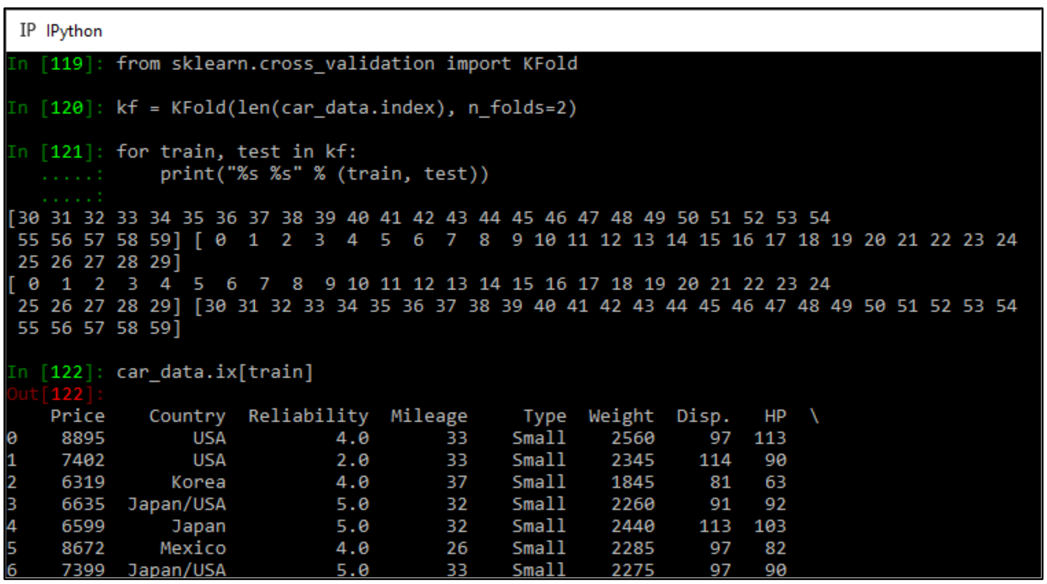
```
IP IPython
In [70]: pd.isnull(seedlings_data)
Out[70]:
   cohort  death gapsize
0   False  False   False
1   False  False   False
2   False  False   False
3   False  False   False
4   False  False   False
5   False  False   False
6   False  False   False
7   False  False   False
8   False  False   False
9   False  False   False
10  False  False   False
```

Figure 17 Detecting Missing Values: isnull()

An important note should be provided here. Datetime datatypes, specifically dattime64[ns] types, NaT represents missing values, whereas NaN is typically used in numeric datatypes. Object datatypes will use the value provided them. Pandas objects are intercompatible between NaT and NaN.

6. More on Subsampling

As an extension of the previous section on subsampling, you may wish to create subsamples to perform techniques requiring training, testing, validation data or even *k*-fold cross validation data. This is a simple extension of what was covered in the previous section. This part of the tutorial will use the car.test.frame.txt file. The code is presented below. For more information on using the library and its functions, please review this webpage: http://scikit-learn.org/stable/modules/cross_validation.html#k-fold.



```
IP IPython
In [119]: from sklearn.cross_validation import KFold

In [120]: kf = KFold(len(car_data.index), n_folds=2)

In [121]: for train, test in kf:
     .....:     print("%s %s" % (train, test))
     .....:
[30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54
 55 56 57 58 59] [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
 25 26 27 28 29]
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
 25 26 27 28 29] [30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54
 55 56 57 58 59]

In [122]: car_data.ix[train]
Out[122]:
    Price    Country  Reliability  Mileage   Type  Weight  Disp.   HP  \
0    8895        USA          4.0       33  Small    2560     97  113
1    7402        USA          2.0       33  Small    2345    114   90
2    6319      Korea          4.0       37  Small    1845     81   63
3    6635  Japan/USA          5.0       32  Small    2260     91   92
4    6599      Japan          5.0       32  Small    2440    113  103
5    8672     Mexico          4.0       26  Small    2285     97   82
6    7399  Japan/USA          5.0       33  Small    2275     97   90
```

Figure 18 K-Fold Cross Validation in Python

7. Dates and Times in Python

All modern statistical packages provide functions that perform mathematical operations and dates and times. For example, say you have data on employee work hours and you need to calculate pay for hourly employees. For each day over a five-day span you have the time the employee clocked in and the time the employee clocked out. You need to calculate the total number of hours the employee worked by using one of two methods: 1) you convert the date-time values into hours-minutes-seconds and sum up the values or 2) use a date-time function that will automatically convert for you and provide you the total hours.

The second option is the obvious choice as it requires minimal computational skills on your part. One of the downsides to most statistical packages is they do not convert date-time values into date-time objects. That is, date-time values are read as categorical values made up of character strings; you cannot perform math on character strings. Converting date-time values in any statistical program requires work. For this portion of the tutorial you will be given an example of how to convert date-time values from object datatypes to date-time datatypes. Use the `afib_data.txt` file to follow along. The figure below contains the column names of the dataset. This contains data on atrial fibrillation patients.

```
IP IPython
In [125]: afib_data.dtypes
Out[125]:
patient_sk                  int64
race                       object
gender                     object
age_in_years                int64
weight                    float64
marital_status             object
patient_type_desc          object
census_region              object
payer_code                 object
payer_code_desc            object
total_charges             float64
CARESETTING_DESC           object
admitted_dt_tm             object
discharged_dt_tm           object
dischg_disp_code_desc      object
diagnosis_type_display     object
dtype: object
```

Figure 19 Datatypes in Afib_Data

The process of converting to a date-time object is simpler in Python than it is in R because the Pandas library provides powerful tools. Focus on the column *admitted_dt_tm* for this example. Looking at the data itself reveals the formatting is a string character:

```
3        Inpatient     Northeast         MC        Medicare      33889.719
4        Inpatient     Northeast         MC        Medicare      37571.289

  CARESETTING_DESC               admitted_dt_tm              discharged_dt_tm  \
0 Medical/Surgical  2006-12-12 21:04:00.0000000  2006-12-19 17:04:00.0000000
1 Medical/Surgical  2006-09-27 18:31:00.0000000  2006-10-01 11:35:00.0000000
2 Medical/Surgical  2006-09-12 13:49:00.0000000  2006-09-18 15:20:00.0000000
3 Medical/Surgical  2007-04-19 04:01:00.0000000  2007-04-25 13:58:00.0000000
4 Medical/Surgical  2006-07-20 11:02:00.0000000  2006-08-01 14:23:00.0000000

                         dischg_disp_code_desc diagnosis_type_display
```

Figure 20 Datetime as a String

To convert the column to a date-time format, simply use the pd.to_datetime() function as shown below. Notice the actual data is slightly different, with less trailing zeroes than before. The datatype is now datetime64[ns] instead of object.

```
IP IPython
In [138]: afib_data['admitted_dt_tm'] = pd.to_datetime(afib_data['admitted_dt_tm'])

In [139]: afib_data.admitted_dt_tm.head()
Out[139]:
0   2006-12-12 21:04:00
1   2006-09-27 18:31:00
2   2006-09-12 13:49:00
3   2007-04-19 04:01:00
4   2006-07-20 11:02:00
Name: admitted_dt_tm, dtype: datetime64[ns]

In [140]: afib_data.dtypes
Out[140]:
patient_sk                          int64
race                               object
gender                             object
age_in_years                        int64
weight                            float64
marital_status                     object
patient_type_desc                  object
census_region                      object
payer_code                         object
payer_code_desc                    object
total_charges                     float64
CARESETTING_DESC                   object
admitted_dt_tm             datetime64[ns]
discharged_dt_tm                   object
dischg_disp_code_desc              object
diagnosis_type_display             object
dtype: object
```

Figure 21 Converting to a Date-Time Datetype