# Week #2 - Python Practice

## Some Ins and Outs with Python

There are many different kinds of data to be managed and analyzed today, and there are many ways to do it using Python.   Being able to manage and modify data isn't useful unless you can also get data into Python, and save your results from it. Beginning with this session we're going to review techniques for input and output from Python starting with the simplest file formats and some basic Python tools. We'll also take a first look at the pandas package.  Pandas has become very popular amongst "Pythonic" data scientists and is being used at the largest of the big data firms.  In the sessions that follow we'll consider more complicated file types and data "munging" tools and techniques.

Time (and the term) is a'wasting, so lets crank up your Canopy environment and get to work with some examples.   In what follows it's assumed that you are using the Canopy "Editor," which includes a version of IPython, and so the command prompts will look like In '[n]:' without the single quotes.  What "n" is depends on where you are in the current session.   As I'm sure you realize by now since you have most likely use Canopy or IPython before, for many "Ins" in IPython there is an "Out" of some kind which is a result of the "In" that precedes it.

So, let's start with flat files.  A flat file is just a file that's, well, flat. It's typically a string of characters that may include end of line markers like a newline or carriage return code.  Let's write a simple flat file out to disk by entering the following command:

In [1]:

```
outfile = open('myflatfile.txt','w')  # open to write to a text file
```

Note: "In [1]:" represents the command prompt in your IPython session.  Depending on what you've been doing in a session, the digit or digits you see in it will vary.  But you knew that, right?

In [2]:

```
outfile # outfile is an open file 'object' in write mode.  By default it's a text,
```

Out[2]:

```
<open file 'myflatfile.txt', mode 'w' at 0x103a84810>
```

In [3]:

```
type(outfile)
```

Out[3]:

```
file
```

Pardon the slight digression, above. It's purpose was to show what kind of Python 'object' outfile is. (Everything in Python is an object, right?) the .txt file name extension is optional. Now, let's create a text string and then write it to outfile:

In [4]:

```
iLikeButter='''Slather me toast with a bargefull of butter, and crown it with a buck
```

This is obviously a quote from a high cholesterol data science pirate. How many characters are in this string? Try the function len(iLikeButter).

In [5]:

```
len(iLikeButter)
```

Out[5]:

```
91
```

Write the string to outfile and then close outfile:

In [6]:

```
outfile.write(iLikeButter)
```

In [7]:

```
outfile.close()     # it's good practice to close whatever you open
```

In [8]:

```
import os
os.getcwd()
```

Out[8]:

```
'/Users/Zeeshan/Desktop/PREDICT 420/Week 2/Sync Session2-2/ExercisePra
ctice2'
```

You'll get as a response a character string indicating what Canopy considers to be your working directory. A third way to find out what your current directory is from IPython is to type ! (the "bang"), which escapes into your OS, followed by whatever command returns what the current directory is. On Linux and OS X, the command is pwd.

Let's read your file back in:

In [9]:

```
infile=open('myflatfile.txt','r')   #read as a text file. For a binary, you'd use '
doYouWantButter=infile.read()   #reads the file contents into a string variable
doYouWantButter         #this should give you he iLikeButter string value
type(doYouWantButter)       # this should give you "str"
type(_)             # should have the same result as above, right?
```

Out[9]:

```
str
```

When used as above, what does the underscore, "_", represent?

Next, let's read a text file with more than one line. The text file louielouie.txt has been provided to you. Pop it into the default directory for your session, the directory you identified before. Then, open it for reading:

In [10]:

```
kingsMenLouie=open('louielouie.txt','r')        #'r' since this is a text file.
```

In [11]:

```
louielouie=kingsMenLouie.read()     # take a look at louielouie by typing its name
```

In [12]:

```
type(louielouie)
```

Out[12]:

```
str
```

And, then you could split the lines in louielouie into a list of lines as strings:

In [13]:

```
louielist=louielouie.split()        # lists are your Python Friend. (One of them, a
```

In [14]:

```
type(louielist)
```

Out[14]:

```
list
```

You could also read this file line by line with readline(). For example, to get the file contents into louie a string variable (and assuming that the file has been closed and opened again after the foregoing):

In [15]:

```
louie=""    # string var where we're going to put the lines from the file
```

In [16]:

```
while True:
    line=kingsMenLouie.readline()
    if not line:
        break
    louie+=line
```

Give the above code a try to see what you get. Are louie and louielouie different? Try the command louie==louielouie.

Python usually does a good job closing files that have been opened, but it's good practice to do so explicitly whenever possible. This is expecially true when you are writing data out to a file, as explicitly closing a file written to forces any remaining write operation to finish. Did you close all the files you opened, above?

A simple way to close a file you've written to is as follows. Suppose you want to write the character string iLikeButter to a file called greaseitup.txt in your current directory. If you do:

In [17]:

```
type(louie)
```

Out[17]:

```
str
```

In [18]:

```
with open('greaseitup.txt','wt') as butterOut:
    butterOut.write(iLikeButter)
```

the file will be closed automatically for you when your write operation is completed. Note the 'wt' in the open statement. 't' is for text, but it's optional. if you include a 'b' instead, you'll have a binary file instead of a text file.

The procedures for reading and writing binary files using open, .read, etc. are for the most part the same as for text files, and so we're not going to spend time here on binary file input and output. We're shortly going to move on to reading and writing csv files, but before that let's take a look at the classic method for “serializing” (storing with permanency) python objects called pickling.

A pickle file includes one or more Python objects that can be read back into Python that has a Python-specific, environment independent format. Two pickling packages in Python 2.7 are pickle and cPickle. cPickle is the faster of the two, and is the default algorithm in Python 3. So let's tinker with it here. First, import cPickle:

In [19]:

```
import cPickle as pickle        # we'll just call it what it is, whether dill or swe
```

Now let's pickle our louielist from above in a file in the current working directory.

In [20]:

```
pickle.dump(louielist,open('louielist.p','wb'))
```

The above writes a binary pickle file. You can read the file back into Python like:

In [21]:

```
louielsBack=pickle.load(open('louielist.p','rb'))
```

Did you get louielist back unchanged? Try louielist == louielsBack from the command prompt.

We're going to move ahead to consider csv files, but to do so we're going to make use of the pandas package. So let's import pandas first, and then look at a simple example of a very useful panda object, the DataFrame.

In [22]:

```
import pandas as pd # panda's nickname is pd

import numpy as np  # numpy as np

from pandas import DataFrame, Series        # for convenience
```

My guess is that you have used the numpy package before in your work or in a previous course.   DataFrame and Series are very handy pandas data structures that can do yeoman work for you in your data management efforts.

By way of introduction, let's first read a little pickled pandas DataFrame.  Put the file littleDF2.p in your default directory (or somewhere you can find it from in Canopy), and do:

In [23]:

```
littleDF=pickle.load(open('littleDF2.p','rb'))  # if it's in you default dir
```

In [24]:

```
type(littleDF)
```

Out[24]:

```
pandas.core.frame.DataFrame
```

If you then ask littleDF to type itself out:

In [25]:

```
In [26]: littleDF
```

Out[25]:

| | var1 | var2 | var3 | var4 |
|---|---|---|---|---|
| **obs1** | -1.228611 | 0.309656 | 0.963380 | 0.392899 |
| **obs2** | -0.560940 | -0.962692 | 0.042021 | -0.289916 |
| **obs3** | 0.781060 | 1.140318 | 0.621084 | 0.682519 |
| **obs4** | -0.092004 | -1.178608 | 1.854705 | 1.011108 |
| **obs5** | 0.041939 | 1.290624 | -0.313368 | -0.749832 |
| **obs6** | -0.156378 | 0.670368 | -0.415693 | -0.813018 |

you'll get a pretty table-like graphical thing with labelled rows and columns.
And if you do:

In [26]:

```
print littleDF
```

```
          var1      var2      var3      var4
obs1 -1.228611  0.309656  0.963380  0.392899
obs2 -0.560940 -0.962692  0.042021 -0.289916
obs3  0.781060  1.140318  0.621084  0.682519
obs4 -0.092004 -1.178608  1.854705  1.011108
obs5  0.041939  1.290624 -0.313368 -0.749832
obs6 -0.156378  0.670368 -0.415693 -0.813018
```

littleDF is a pandas DataFrame:

In [27]:

```
type(littleDF)
```

Out[27]:

```
pandas.core.frame.DataFrame
```

A pandas DataFrame is a table-like data structure with columns that can be of different data types, and that has both row and column indices. The row index of littleDF is in the leftmost column, above, with index values of obs1 through obs6. The column index is across the top, var1 through var4. A DataFrame can have more complicated, hierarchical or “nested' indices.

A Series is like one column of a DataFrame. It's a kind of vector that has an associated index. A DataFrame can be thought of as a set of Series in the columns that share a single index, the row index.

littleDF is a pandas DataFrame, and DataFrames are Python objects. So, littleDF has attributes, and lots of them. To see them you can use the IPython tab completion feature:

```
littleDF.<tab>
```

You'll ses a long list of different attributes (methods and data) that littleDF has. A second <tab> should allow you to scroll through the attributes. try littleDF.values to see the data, and littleDF.describe() to get summary statistics for the variables in the columns littleDF.columns.

DataFrame methods include different ways of converting to a different kind of data structure. For example, you can convert littleDF into a dict with :

In [28]:

```
littleDF.to_dict('records')
```

Out[28]:

```
[{'var1': -1.2286113016207898,
  'var2': 0.30965619163770047,
  'var3': 0.96338021151319131,
  'var4': 0.39289876033736942},
 {'var1': -0.56094030295428543,
  'var2': -0.96269174995277529,
  'var3': 0.042020936959569238,
  'var4': -0.28991632733800704},
 {'var1': 0.78106045003984292,
  'var2': 1.1403182932997518,
  'var3': 0.62108363926714805,
  'var4': 0.68251944983814727},
 {'var1': -0.092004064802186564,
  'var2': -1.178607577083874,
  'var3': 1.8547052279781786,
  'var4': 1.0111083188039576},
 {'var1': 0.041938987884194652,
  'var2': 1.2906243535240169,
  'var3': -0.31336832091555672,
  'var4': -0.74983162877031817},
 {'var1': -0.15637751256752078,
  'var2': 0.67036800193846668,
  'var3': -0.41569296335492445,
  'var4': -0.81301815241617248}]
```

or into csv:

In [29]:

```
littleDF.to_csv()
```

Out[29]:

```
',var1,var2,var3,var4\nobs1,-1.22861130162,0.309656191638,0.9633802115
13,0.392898760337\nobs2,-0.560940302954,-0.962691749953,0.042020936959
6,-0.289916327338\nobs3,0.78106045004,1.1403182933,0.621083639267,0.68
2519449838\nobs4,-0.0920040648022,-1.17860757708,1.85470522798,1.01110
83188\nobs5,0.0419389878842,1.29062435352,-0.313368320916,-0.749831628
77\nobs6,-0.156377512568,0.670368001938,-0.415692963355,-0.81301815241
6\n'
```

or into several other formats. Note that all these methods have options set by arguments when calling them. To see what's available, do:

```
littleDF.to_<tab>
```

Not surprisingly, you can covert other kinds of data structures into a DataFrame. Take a look:

```
littleDF.from_<tab>
```

Next, let's try writing out littleDF to a csv file called “littleDFcsv.csv:”

In [30]:

```
csvOutFile=open('littleDFcsv.csv','wt') # this will be a text file

csvOutFile.write(littleDF.to_csv())

csvOutFile.close()
```

Now, take a look at the file you written using the editor of your choice. You can use the editor in Canopy, of course. Does your file have a header record? Are character values enclosed in quotes?

DataFrames and Series have many useful attributes and features, some of which we'll explore in upcoming exercises. But now let's try reading a less trivial csv file into a DataFrame. The file is xyzcust10.csv, and it should be available to you on Canvas. Take a look at it with your favorite text editor. Then, put it in a place you can find it from Canvas, and input it into a DataFrame:

In [31]:

```
myvacation = r'/Users/Zeeshan/Desktop/PREDICT 420/Week 1/Sync Session1-2/ExercisePra

#df = pd.DataFrame(data = myvacation, columns=['CityNames', 'StateNames', 'DaysSpen
#df = pd.read_csv(myvacation, names=['CityNames' ,'StateNames', 'DaysSpent', 'YearV
df = pd.read_csv(myvacation, header = None)
df.columns = ['CityNames', 'StateNames', 'DaysSpent', 'YearVisited']




#myvacation=pd.read_csv('VacationHistory.csv')
```

In [32]:

```
type(df)
```

Out[32]:

```
pandas.core.frame.DataFrame
```

The file has 10 variables in it. The rows, or records, are XYZ customers. How many records are in xyzcust10?

What types of variables are in the columns of xyzcust10? To find out:

In [33]:

```
df.dtypes
```

Out[33]:

```
CityNames      object
StateNames     object
DaysSpent       int64
YearVisited     int64
dtype: object
```

Look at the first and last rows in xyzcust10:

In [34]:

```
df.head()
```

Out[34]:

| | CityNames | StateNames | DaysSpent | YearVisited |
|---|---|---|---|---|
| **0** | Atlanta | GA | 4 | 2017 |
| **1** | Houston | TX | 5 | 2013 |
| **2** | Columbus | OH | 3 | 2016 |
| **3** | SanFrancisco | CA | 7 | 2016 |
| **4** | LosAngeles | CA | 2 | 2010 |

In [35]:

```
df.tail()
```

Out[35]:

| | CityNames | StateNames | DaysSpent | YearVisited |
|---|---|---|---|---|
| **0** | Atlanta | GA | 4 | 2017 |
| **1** | Houston | TX | 5 | 2013 |
| **2** | Columbus | OH | 3 | 2016 |
| **3** | SanFrancisco | CA | 7 | 2016 |
| **4** | LosAngeles | CA | 2 | 2010 |

Note that in this file missing values for ZIP, ZIP4, and the nine digit ZIP are represented with zeros, “0's.” The ZIPs could really also be coded as strings, rather than as integers, couldn't they? Also, it looks like there might be two nine digit ZIP code variables. Are they the same?

That is are the values in these two variables the same for every row of data? How would you locate the rows in xyzcust10 that have a zero for ZIP or for ZIP4? We'll see in the next session's Python Practice.

Last but not least, be sure to pickle your xyzcust10 Data Frame so you can use it in the next Practice.

## Deliverable :

## Update the pd.read_csv('xyzcust10.csv') to use the VactaionHistory.csv file you created in Exercise #1 and run this script

In [36]:

```
df
```

Out[36]:

| | CityNames | StateNames | DaysSpent | YearVisited |
|---|---|---|---|---|
| **0** | Atlanta | GA | 4 | 2017 |
| **1** | Houston | TX | 5 | 2013 |
| **2** | Columbus | OH | 3 | 2016 |
| **3** | SanFrancisco | CA | 7 | 2016 |
| **4** | LosAngeles | CA | 2 | 2010 |

In [ ]: