

Deliverables:

- Submit a single zip-compressed file that has the name: YourLastName_Exercise_1 that has the following files:
 1. Your **PDF document** that has your Source code and output
 2. Your **ipynb script** that has your Source code and output

Objectives:

In this exercise, you will:

- Use Jupyter notebook to run ipynb script
- Experiment with file processing (reading and writing files in different formats)
- Use different data types to store and process data

Submission Formats :

Create a folder or directory with all supplementary files with your last name at the beginning of the folder name, compress that folder with zip compression, and post the zip-archived folder under the assignment link in Canvas. The following files should be included in an archive folder/directory that is uploaded as a single zip-compressed file. (Use zip, not Stuffit or any 7z or any other compression method.)

1. Complete IPYNB script that has the source code in Python used to access and analyze the data. The code should be submitted as an IPYNB script that can be loaded and run in Jupyter Notebook for Python
2. Output from the program, such as console listing/logs, text files, and graphics output for visualizations. If you use the Data Science Computing Cluster or School of Professional Studies database servers or systems, include Linux logs of your sessions as plain text files. Linux logs may be generated by using the script process at the beginning of your session, as demonstrated in tutorial handouts for the DSCC servers.
3. List file names and descriptions of files in the zip-compressed folder/directory.

Formatting Python Code When programming in Python, refer to Kenneth Reitz' PEP 8: The Style Guide for Python Code: <http://pep8.org/> (<http://pep8.org/>) (Links to an external site.)Links to an external site. There is the Google style guide for Python at <https://google.github.io/styleguide/pyguide.html> (<https://google.github.io/styleguide/pyguide.html>) (Links to an external site.)Links to an external site. Comment often and in detail.

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.

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
```

```
Out[2]: <_io.TextIOWrapper name='myflatfile.txt' mode='w' encoding='UTF-8'>
```

```
In [3]: type(outfile)
```

```
Out[3]: _io.TextIOWrapper
```

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 wi
```

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)
```

```
Out[6]: 91
```

```
In [7]: outfile.close()      # it's good practice to close whatever you open
```

```
In [8]: import os
os.getcwd()
```

```
Out[8]: '/Users/zachtsouprakos/Documents/MSDS/MSDS-420/Exercise_1'
```

Let's read your file back in:

```
In [9]: infile=open('myflatfile.txt','r')    #read as a text file. For a binary, you
doYouWantButter=infile.read()    #reads the file contents into a string vari
doYouWantButter                  #this should give you the 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 f
```

```
In [11]: louielouie=kingsMenLouie.read()    # take a look at louielouie by typing i
```

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

```
In [12]: louielist=louielouie.split()    # lists are your Python Friend. (One of
```

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

```
In [14]: 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 especially 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 [15]: 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 python object might need to be written/read in binary representation/format into/from file for further/later processing. To do so, we need to import the pickle package:

```
In [16]: import pickle
```

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

```
In [17]: pickle.dump(louielist, open('louielist.p', 'wb'))
```

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

```
In [18]: 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 [19]: 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.

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.

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.

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 [20]: xyzcust10=pd.read_csv('xyzcust10.csv')
```

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 [21]: xyzcust10.dtypes
```

```
Out[21]: ACCTNO                object
        ZIP                  int64
        ZIP4                 int64
        LTD_SALES            float64
        LTD_TRANSACTIONS      int64
        YTD_SALES_2009        float64
        YTD_TRANSACTIONS_2009 int64
        CHANNEL_ACQUISITION   object
        BUYER_STATUS          object
        ZIP9_Supercode         int64
        ZIP9_SUPERCODE         int64
        dtype: object
```

Look at the first and last rows in xyzcust10:

```
In [22]: xyzcust10.head()
```

```
Out[22]:
```

	ACCTNO	ZIP	ZIP4	LTD_SALES	LTD_TRANSACTIONS	YTD_SALES_2009	YTD_TRANSACTIONS
0	WDQQLLDQL	60084	5016	90.0	1	0.0	
1	WQWAYHYLA	60091	1750	4227.0	9	1263.0	
2	GSHAPLHAW	60067	900	420.0	3	129.0	
3	PGGYDYWAD	60068	3838	6552.0	6	0.0	
4	LWPSGPLLS	60090	3932	189.0	3	72.0	

```
In [23]: xyzcust10.tail()
```

```
Out[23]:
```

	ACCTNO	ZIP	ZIP4	LTD_SALES	LTD_TRANSACTIONS	YTD_SALES_2009	YTD_TRANSACTIONS
30466	SYDQYLSWH	60098	3951	2736.0	9	96.0	
30467	SAPDQHQLP	60098	9681	2412.0	8	108.0	
30468	SASYAPDSY	60098	0	429.0	1	0.0	
30469	PWQPDWHA	60098	7927	651.0	1	0.0	
30470	SQQHDYHWH	60098	4160	4527.0	16	672.0	

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.

Requirements :

1. Produce a list of Zip values in xyzcust10 along with their frequencies
2. How many records with missing ZIP in xyzcust10?
3. How many active and inactive BUYER_STATUS in xyzcust10?

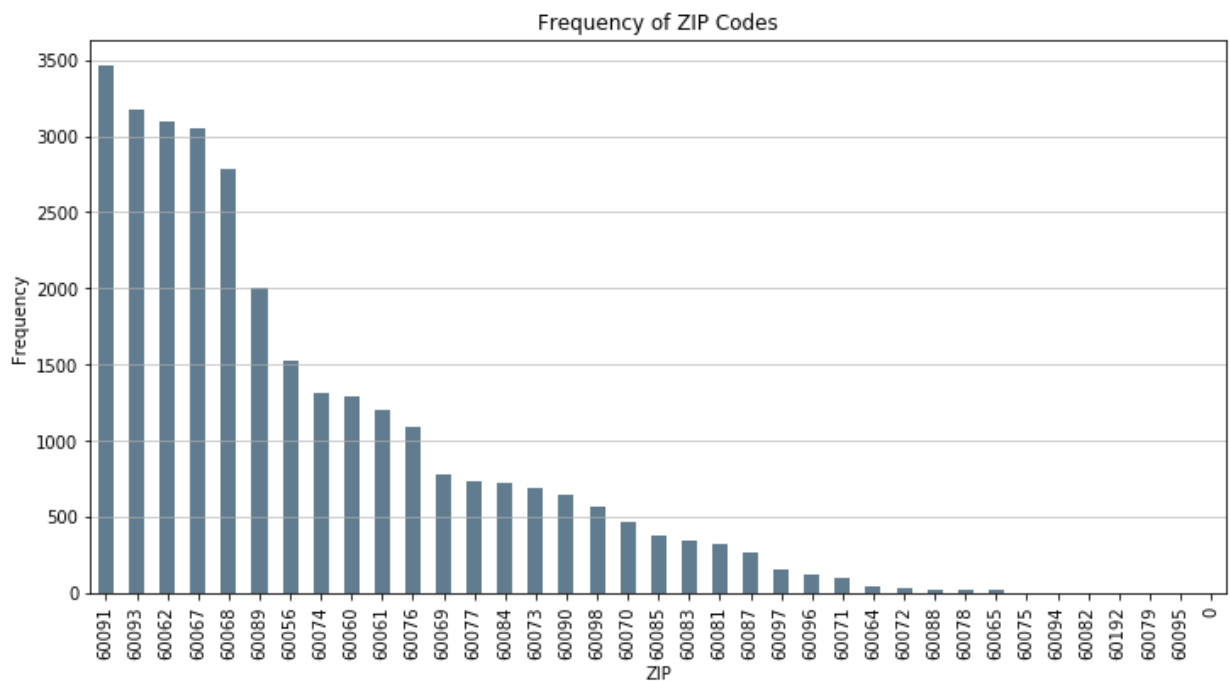
In [24]: *# Write your python code that meets the above requirements in this cell*
1. Produce a list of Zip values in xyzcust10 along with their frequencies

```
import matplotlib.pyplot as plt
from matplotlib.pyplot import figure

zipFreq = xyzcust10.groupby('ZIP')['ZIP'].count().sort_values(ascending=False)
print(zipFreq)

figure(figsize=(12,6))
zipFreq.plot.bar(color='#607c8e', title = 'Frequency of ZIP Codes')
plt.ylabel('Frequency')
plt.grid(axis='y', alpha=0.75)
```

```
ZIP
60091    3458
60093    3178
60062    3099
60067    3050
60068    2781
60089    2007
60056    1529
60074    1313
60060    1296
60061    1207
60076    1090
60069     784
60077     740
60084     723
60073     686
60090     648
60098     564
60070     463
60085     379
60083     344
60081     322
60087     268
60097     151
60096     125
60071      98
60064      42
60072      34
60088      28
60078      25
60065      21
60075       5
60094       4
60082       3
60192       2
60079       2
60095       1
0           1
Name: ZIP, dtype: int64
```

```
In [25]: # 2. How many records with missing ZIP in xyzcust10?
# What is 'missing'?
# Lets check for nulls and zip codes that equal 0. The group by statement h

print("Count of null zip codes: " + str(sum(xyzcust10['ZIP'].isnull())))
print("Cout of zip codes that equal '0': " + str(len(xyzcust10[xyzcust10['Z
```

```
Count of null zip codes: 0
Cout of zip codes that equal '0': 1
```

```
In [26]: # 3. How many active and inactive BUYER_STATUS in xyzcust10?
```

```
buyer_status = xyzcust10.groupby('BUYER_STATUS')['BUYER_STATUS'].count()
print(buyer_status)

figure(figsize=(12,6))
buyer_status.plot.barh(color='#607c8e', title = 'Frequency of Buyers by Sta
plt.ylabel('Status')
plt.xlabel('Frequency')
plt.grid(axis='y', alpha=0.75)
```

```
BUYER_STATUS
ACTIVE      13465
INACTIVE     9078
LAPSED       7928
Name: BUYER_STATUS, dtype: int64
```

