**Data Focused Python**

**Homework 3**

**Mini 3 2023**

***Due at 11:59 pm on Monday, Sept. 16***

***You will lose 10 points per hour after that time***

1. **(30 points)** **Lists, Tuples, Sets, Dicts, and Comprehensions**

**expenses.txt** is a small text file describing business expenses. Each line (after the header) gives the money amount, category, date, and description of an expense.

1. Create a Python script file named **hw3.1.py**. In this script, define an empty list named **records**, then read the lines from **expenses.txt** and **append** each line (*excluding* its terminating newline character) to the **records** list. Add this code to display the lines from **records**:

**for line in records:**

**print(line)**

Confirm that the output is not double-spaced; that is, confirm that each line (string) in the **records** list does not include a terminating newline.

1. Close the open **expenses.txt** file, then open **expenses.txt** again. Use ***list*** *comprehension* notation to create and initialize a new list, **records2**, from the lines in the **expenses.txt** file, excluding the terminating newline characters. Confirm that you have done this correctly, by adding this code at the end of the script:

**print("\nrecords == records2:",**

**records == records2, '\n')**

This should display **records == records2: True**.

1. Close the open **expenses.txt** file, and open **expenses.txt** again. Learn about the **str** class’s **split** function. Fields in the **expenses.txt** file are separated with colon characters, **‘:’**, since expense descriptions often contain commas. Use *nested* ***tuple*** *conversion* notation to create and initialize a new *tuple* *of tuples*, **records3**, in which each “inner” tuple has the form **(***amount***,***category***,***date***,***description***)**, and the “outer” tuple contains one “inner” tuple for each line of input. We use a tuple of tuples because tuples are *immutable*, and we want to protect the input data from accidental change.

Add this code to display the tuple of tuples **records3**:

**for tup in records3:**

**print(tup)**

The output from this loop should look like:

**('Amount', 'Category', 'Date', 'Description')**

**('5.25', 'supply', '20170222', 'box of staples')**

**...**

**('8.98', 'supply', '20170325', 'Flair pens')**

1. A function is a mapping from arguments to values. A sequence or map (**dict**) can also be thought of as a mapping from arguments to values. Creation of sequences/maps from data can simplify function definitions, or even eliminate the need for some of them. A **list** or **tuple** is a mapping from an integer subscript to a value; a **set** is a mapping from a value to **in == True** or **in == False**; and a **dict** is a mapping from a key to a value.

Using ***set*** *comprehension* notation with **records3**, define: **cat\_set**, the set of categories (do not include the string **'Category'**) in the expense records; and, **date\_set**, the set of dates (again, do not include the string **'Date'**) in the expense records. Add this code to display these two sets:

**print('Categories:', cat\_set, '\n')**

**print('Dates: ', date\_set, '\n')**

Since **set**s are unordered, your exact output may differ, but the output should look something like:

**Categories: {'supply', 'meal', 'travel', 'util'}**

**Dates: {'20170222', '20170223', …, '20170325'}**

1. Using ***dict*** *comprehension* notation with **records3**, define a **dict** named **rec\_num\_to\_record** in which each entry’s *key* is the record (line) number, and each entry’s *value* is the tuple representing the data. *Hint*: use a combination of **range()** and **zip**() along with **records3**. In **rec\_num\_to\_record**, store the field names as record number **0**.

Add this code to display **rec\_num\_to\_record**:

**for rn in range(len(rec\_num\_to\_record)):**

**print('{:3d}: {}'.format(rn,**

**rec\_num\_to\_record[rn]))**

The output from this loop should look like:

**0: ('Amount', 'Category', 'Date', 'Description')**

**1: ('5.25', 'supply', '20170222', 'box of staples')**

**...**

**22: ('212.06', 'util', '20170308', 'Duquesne Light')**

Add this code, using the **items()** iterable, to display **rec\_num\_to\_record**:

**for i in rec\_num\_to\_record.items():**

**print('{:3d}: {}'.format(i[0], i[1]))**

Alternatively, using *tuple unpacking* into two loop variables, you can use (for example):

**for k, v in rec\_num\_to\_record.items():**

**print('{:3d}: {}'.format(k, v))**

1. **(30 points) Variadic Functions**

You will need to modify the file **mystats.py** for this part of the homework, since we will use **mystats.py** as a module to **import** in part 3 of the homework. Your **mystats.py** file must be included in the **.zip** archive that you upload when you have completed this homework.

***HINT:*** It is cheating to implement your **mean** function by calling the **mean** function from some other library/module. Write your own code! Check Wikipedia, Wolfram, or elsewhere to make sure you have the correct formulas.

1. Open the **mystats.py** file (uploaded to Canvas as part of this homework) for editing and running with Python. You will see that **mystats.py** mostly consists of comments describing functions that you will need to write, with a single uncommented **print()** function call that displays the name of the module itself.

**print('The current module is:', \_\_name\_\_)**

Run the module, and add a comment after the **print()** function call describing the output when **mystats.py** is the main module, that is, the module that you are currently editing and testing.

1. Define the function **mean()** below the relevant comment. The **mean()** function should take *one or more* positional parameters, and return the arithmetic mean of its arguments when called. Uncomment these test statements in **mystats.py** to test that your **mean()** function is working correctly:

**print('mean(1) should be 1.0, and is:', mean(1))**

**print('mean(1,2,3,4) should be 2.5, and is:',**

**mean(1,2,3,4))**

**print('mean(2.4,3.1) should be 2.75, and is:',**

**mean(2.4,3.1))**

**print('mean() should FAIL:', mean())**

Run the module and confirm success. (Comment out the last of these four **print()** calls after you have confirmed that calling **mean()** with no argument fails.)

1. Improve your **mean()** function so that it accepts numeric *iterables* as well as individual numeric values as arguments. To test whether a variable refers to an *iterable*, you can call the **iter()** function. **iter()** will fail and *raise an exception* if its argument is not iterable. Here is a function that uses **try**/**except** to test whether an object is iterable without crashing your program. (You can copy this function code into your own program.)

**def is\_iter(v):**

**v\_is\_iter = True**

**try:**

**iter(v)**

**except:**

**v\_is\_iter = False**

**return v\_is\_iter**

**v1 = {1, 2, 3} # a set is iterable**

**print(v1, "is iterable:", is\_iter(v1))**

**v2 = 123**

**print(v2, "is iterable:", is\_iter(v2))**

*Hint*: If **v** refers to an iterable, then **sum(v)** is the sum of the values in the iterable, and **len(v)** is the length of the iterable. You can use these to help with your implementation of your improved **mean()** function.

Uncomment these test statements in **mystats.py** to test that your improved **mean()** function is working correctly. (The test **print()** functions from part (b) should also continue to work, except of course for the call of **main()** with no argument, which should be commented out.)

**print('mean([1,1,1,2]) should be 1.25, and is:',**

**mean([1,1,1,2]))**

**print('mean((1,), 2, 3, [4,6]) should be 3.2,' +**

**'and is:', mean((1,), 2, 3, [4,6]))**

Run the module and confirm success.

1. At the top of your **mystats.py** file, insert the line:

**import numpy as np**

so that we can use NumPy’s random number generator functions. (We will do more

with NumPy in part (3) below.)

**np.random.randn()** with no argument will return a single random draw from the

standard Normal distribution (mean 0.0, standard deviation 1.0, often represented in

writing as **Norm(0,1)**). Add this code to the end of your program to display ten

draws from Norm(0,1):

**for i in range(10):**

**print("Draw", i, "from Norm(0,1):",**

**np.random.randn())**

At the end of your program, use **list** comprehension to create a **list** of 50 draws from

Norm(0,1). Then, test your **mean()** function on this **list**, like this:

**ls50 =** *...* **list** *comprehension ...*

**print("Mean of", len(ls50), "values from Norm(0,1):",**

**mean(ls50))**

The displayed mean should be “close to” 0.0. If the random number generator is

“truly random" then the mean should tend closer to 0.0 as the number of draws in

the sample increases. Perform another test using 10,000 draws, like this:

**ls10000 =** *...* **list** *comprehension ...*

**print("Mean of", len(ls10000), "values from " +**

**"Norm(0,1):", mean(ls10000))**

Is the displayed mean “closer to” 0.0 than before? Of course, this is a casual test, not

a rigorous test.

1. We can set the *seed* value for the random number generator with:

**np.random.seed(***seed***)**

Set the seed to **0**, then create an **ndarray** of 10 random draws from Norm(0,1) as we

illustrated in lecture. Add this code at the end of your program; run the module and

confirm success.

**a1 = np.random.randn(10)**

**print("a1:", a1) # should display an ndarray of 10 values**

An **ndarray** object is iterable, so your **mean()** functions should process **a1** just fine.

Add this code at the end of your program, then run the module.

**print("the mean of a1 is:", mean(a1))**

Since you set the seed to 0.0 prior to this, you should get the same value that I do:

**0.7380231707288347**

1. Define the **stddev()** function to compute the *sample standard deviation* of its arguments, where the arguments can be the same as those passed to the **mean()** function. Add this code at the end of your program, then run the module.

**print("the stddev of a1 is:", stddev(a1))**

If you have defined **stddev()** correctly, you should get the same value that I do:

**1.0193909949356386**

1. Define the **median()** function to compute the *median* of its arguments, where the arguments can be the same as those passed to the **mean()** function. Add this code at the end of your program, then run the module.

**print("the median of a1 is:", median(a1))**

If you have defined **median()** correctly, you should get the same value that I do:

**0.6803434597319808**

Then, add this code at the end of your program, and run the module.

**print("median(3, 1, 5, 9, 2):", median(3, 1, 5, 9, 2))**

If you have defined **median()** correctly, you should get **3**.

1. Define the **mode()** function to compute the *mode* of its arguments, where the arguments can be the same as those passed to the **mean()** function. The mode is usually used with integer counts, rather than floating point values. Since a data set can be *multimodal* (that is, have two or more values that each occur “most often”), your **mode()** function should return a **tuple** of modal values. *Hint:* You may find a **dict** useful for keeping track of how often each value occurs. You can use *v* **in** *d* to test whether *v* is a key in dictionary *d*.

Add this code at the end of your program, then run the module.

**print("mode(1, 2, (1, 3), 3, [1, 3, 4]) is:",**

**mode(1, 2, (1, 3), 3, [1, 3, 4]))**

If you have defined **mode()** correctly, you should get either **(1, 3)** or **(3, 1)**, since

1 and 3 each occur 3 times in the argument list, whereas 2 and 4 only occur one

time each.

1. **(10 points) Modules**
2. Open the **my\_stat\_test.py** file (uploaded to Canvas as part of this homework) for editing and running with Python. You will see that **my\_stat\_test.py** contains several tests of the functions that you developed in **mystats.py**, such as:

**print("mean of int\_list1:", ms.mean(int\_list1))**

It also makes use of some more random number generator functions, **rand()** and **randint()**, that you should learn about in the Python NumPy documentation.

Add an appropriate **import** statement at the top of **my\_stat\_test.py** so that **mystats.py** is imported with the abbreviated name **ms**. Run the **my\_stat\_test.py** module. You should see that it works, but all of the testing code in **mystats.py** is also executed when **mystats.py** is imported. This is not the behavior that an importer of a library expects.

1. Modify **mystats.py** so that its testing code is only executed when **mystats.py** is the main module. Then, run **my\_stat\_test.py** as the main module, and confirm that although the functions defined in **mystats.py** are available in **my\_stat\_test.py**, the testing code in **mystats.py** is *not* executed when **mystats.py** is imported.
2. **(30 points) NumPy**
3. In lecture, we introduced NumPy and its N-dimensional array type, **ndarray**. We looked at ways to create **ndarray** objects, and at three ways for accessing rows/columns/sections of an **ndarray**: slices, Boolean indexes, and “fancy” or integer indexes.

There is *much* more to NumPy than we have time to talk about. In Python for Data Analysis, 3rd Ed., read through Chapter 4: NumPy Basics: Arrays and Vectorized Computation, and try out the examples in a Jupyter notebook.

You can access Wes McKinney’s book at:

https://wesmckinney.com/book/

***When finished, put your hw3.1.py, mystats.py and my\_stat\_test.py source code files and your Jupyter notebook into a zip archive named* Team***N***\_HW3.zip *file, where*** *N* ***is your team number, then upload your .zip archive to Canvas.***