Lab 1

Code Challenges to review and refresh your Python skills

Part 1 - Strings

Copy all the following predefined strings into a new Mu editor tab. Save your code as lastname_lab1_part1.py

defined strings

```
capitals = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
lowers = "abcdefghijklmnopqrstuvwxyz"
specials = "~!@#$^&*()<>?{}[]|\/"
numbers = "0123456789"
```

Write the function count_caps that will count the total number of capital letters in a given input string and will return that number. If no capital letters are present the function should return -1.

e.g. `count_caps("CHEEsy tacos are YUMMY")` would return 9

`count_caps("cheesy tacos are yummy")` would return -1

```
def count_caps(my_string):
    < add your code>
```

Test the function count_caps by first prompting the user for a string of characters which they will type in. Make sure to output your result after it is returned from the function. A possible output might be:

```
Phrase: Eddie's Grill has the BEST burgers!
Contains: 6 capitals letters
```

How many of the 6 basic programming abilities did you need to combine to complete this part of the program?

List the basic abilities you used here:

Should have used all 6!

In the same python program you are going to make another function.

Write a function password_checker that will check to make sure a password is "strong".

For a password to be considered strong it needs to include the following:

- Password has at least one capital letter
- Password has at least one lower case letter
- Password has at least one special character - use the provided `specials` string
- Password has at least one number
- Password is at least 10 characters long

The function accepts a string of characters, i.e. password, as an argument and should return `True` if the password is strong or `False` otherwise.

Test the function password_checker multiple times with a variety of "passwords" to make sure it works properly. If password_checker detects a strong password, have the program output the response: "`That is one STRONG password`". Otherwise have the program output a response similar to: "`That password is very weak`".

SOLUTION CODE IN THE FILE: lab1_part1.py

That's all for Part 1. Before moving on you should tidy up your code a little. This process is called refactoring. You will not change the function of your code but it will be a bit more organized. Here is a skeleton file outline that you can follow:

```
"""
Filename:
Author(s):
Date:
Description:
"""
"""
I call the information above the "header".  You should try to add a header to
all your programs and include lots of inline comments too.  If you have to go
back to this code sometime in the future, the header description and the inline
comments could save you a lot of time trying to figure out what the program
does!

"""
# import - place all your module imports at the top.

# initialize variables / create objects

# def my_func():  function definitions come next.

# The actual "program" comes last.  Using the Feather Bluefruit Sense we will
# often have a loop that runs continuously.  For this lab you will just be
# testing your functions.
```

Refactor your code from Part 1 and save the file. DO NOT FORGET TO ADD INLINE COMMENTS. If you get in the habit of structuring your code this way, it will make it easier to understand and troubleshoot/debug in the future.

Show your commented code to your instructor.

Part 2

An avid runner has been logging his 5k running times and wants you to write some code to help analyze the data. Save your code as lastname_lab1_part2.py

- Import random and use it to help **create a list of fifty (50) 5k times** between 18 and 36 minutes. (The number 21.5 would represent 21 minutes, 30 seconds)
- **Create a function find_mean** which accepts a list as an argument and returns the average of all the run times in the list
- **Create a function find_max** which accepts a list as an argument and returns the maximum(i.e. slowest) of all the run times in the list
- **Create a function find_min** which accepts a list as an argument and returns the minimum(i.e. fastest) of all the run times in the list
- **Create a function find_median** which accepts a list as an argument and returns the median of all the run times in the list
- **Test all the functions** and output the results in the following format:

```
5k time  average = 25.96
5k time  minimum = 18.17
5k time  maximum = 35.73
5k time  median = 26.50
```

This program may be a good chance to work on debugging skills. When programming there are 3 types of errors:

1. Syntax Errors - The python interpreter will find your syntax errors. Most likely it has already found many syntax errors for you. You typed something incorrectly that the interpreter did not understand. You did not follow the rules of python.

2. Runtime Errors - Good news, you do not have to look for runtime errors. Bad news, when they find you your program will crash. There are some common runtime errors we can guard against such as ZeroDivisionError or TypeError. In those cases we can plan ahead and use a `try: … except:` construct.

3. Semantic/Logic Errors - Semantic errors are errors in meaning. The python interpreter finds no syntax errors so the program runs, it just does not do what we

want it to do. Somewhere in the code a logical error has been made. THESE are the errors that can drive you crazy since they can seem tricky to find at times.

It is common for programmers to just add lots of `print()` statements in their code when they are trying to find a semantic error. Sometimes that works. Many IDEs like the MU editor have built in some debugging help which can be even more helpful.

If you look at the top menu bar of the Mu editor, there is a Debug option. Clicking Debug will change the menu and give you some different options. It will also open up the Debug inspector frame. In this frame you can follow the variables in your program. A useful way to debug is to add a breakpoint in your code. You are going to pick a spot in your code where the Python interpreter will stop and give you a chance to see what is really going on. You can add multiple breakpoints to closely track variables in the Debug inspector. To add a breakpoint simply click on the line number on the left side of the editor window. You should see a red dot. Now when you click the Continue button, your code will run right up to your breakpoint and then halt, until you tell it to continue.

Add a few breakpoints to your current code and explore the Debug options. Pick one of your variables and watch as the value changes as you step through the program.

To clear any breakpoints simply click on the line number again and the breakpoint will be removed.

Refactor your lab1_part2 code. DO NOT FORGET TO ADD INLINE COMMENTS.

Show your working code to your instructor.

SOLUTION CODE IN THE FILE: lab1_part2.py

Part 3

Your running friend has been logging more than just 5k running times and suggests you use a python dictionary to store all the running data. Start with your code from part 2 and save it as lastname_lab1_part3.py

Here is the template for the dictionary.

```
times_dict = {"5k": [],"10k": [], "mile": [], "half": []}
```

The keys of the dictionary are “5k”, “10k”, “mile” and “half”. Each of these keys should access a list of run times for the respective distance.

Instead of filling only a 5k list with random times, you will need to fill all four lists in the dictionary with random times. Use the following ranges when you fill the lists:

For 5k - make 50 random times between 17.5 and 29.99 minutes
For 10k - make 50 random times between 45.0 and 62.99 minutes
For mile - make 25 random times between 5.0 and 8.25 minutes
For half - make 9 random times between 95 and 125 minutes

Make any modifications necessary to your functions from Part 2 to allow them to work with your updated data structure.

Use an enhanced for loop to traverse the `times_dict` and test the functions on all the different lists.

```
E.g. for key in time_dict:
          <more code>
```

Your program output should look something like this:

```
5k average = 24.00
5k minimum = 18.29
5k maximum = 29.45
5k median = 24.18
10k average = 55.22
10k minimum = 45.21
10k maximum = 62.60
10k median = 55.48
mile average = 6.65
mile minimum = 5.08
mile maximum = 8.13
mile median = 6.68
half average = 111.54
half minimum = 97.58
half maximum = 124.01
half median = 112.92
```

Use the debugger if necessary to help troubleshoot your code.

Refactor your code and add inline comments.

Show your working code to your instructor.

SOLUTION CODE IN THE FILE: lab1_part3.py