Assignment #1: Functional Overload

Due Date:

1. **Objective**

   The aim of this assignment is to have you practice on how to define functions, as well as beginning the development of critical thinking skills that will be essential for the remainder of the course. You are required to write ***twelve*** functions as specified below. However, it will be up to you on how to implement the required function and make sure that it works as intended. Do note that names of the function ***MUST*** be spelled as they are shown, along with any given parameter as well.

2. **Required Functions**

   return_five()

   Description: Will return the integer **5**.

   Output: This function should not print anything to the screen.

   Return Value: Return **5**.

   return_pi()

   Description: Will return the floating-point number **3.14**.

   Output: This function should not print anything to the screen.

   Return Value: Return **3.14**.

   return_hello()

   Description: Will return the string “Hello”, excluding the quotation marks.

Output: This function should not print anything to the screen.

Return Value: Return “Hello”.

return_True()

Description: Will return the Boolean value **True**.

Output: This function should not print anything to the screen.

Return Value: Return **True**.

determine_data_type(data)

Description: For a passed parameter *data*, the function will determine its data type and print the result. So, for example, if *data = 5,* then the function should print “Integer”.

Output: This function should print one of the following, all dependent on the data type of the parameter *data*:

- Integer
- Float
- String
- Boolean

Your output must match the sample output file exactly.

Return Value: This function should not return anything.

powTen(p)

Description: Given a specified power *p*, the function will compute a power of ten. In other words, if *p = 2*, then that means $10^2 = 100$.

Output: This function should not print anything to the screen.

Return Value: Return the calculated power of ten.

cubed(base)

Description: Given a specified base *base*, the function will compute its cube. In other words, if *base = 2*, then that means $2^3 = 8$.

Output: This function should not print anything to the screen.

Return Value: Return the calculated cube for the given base.

fahrenheit_to_celcius(f)

Description: Given a temperature *f* that will be in degrees Fahrenheit, convert it to its equivalent Celsius temperature. The conversion is given as the following: $c = (f - 32) * (5.0 / 9.0)$

Output: The function should print the passed Fahrenheit temperature *f* (up to two decimal places) and the calculated Celsius temperature, also up to two decimal places. For example, if *f* = 45, the output should be "Fahrenheit temperature: 45.00; Celsius temperature: 7.22"

Return Value: This function should not return anything.

print_median_of_three(a, b, c)

Description: Given three integers *a*, *b*, and *c*, determine the median of all three. Once determined, the function will print out which integer is the median.

Output: This function should print "The median is: X", where X is the value of the integer that is the median. Your output must match the sample output file exactly.

Return Value: This function should not return anything.

print_military_to_standard(mil_time)

Description: Military time is used with a 24-hour format. To elaborate, if it is 11:00 PM, then in military time that will be 2300 hours, or simply 23:00. Thus, given a military time, implement a function that will convert it to standard time and print out the time, indicating whether it is AM or PM. The input will always be an integer whose value will be within the range of 0 and 23, with 0 being 12:00 AM, 1 being 1:00 AM, so on and so forth.

Output: This function should print the standard time while also indicating whether it is AM or PM. From the example provided in the **Description**, if *mil_time = 23*, then the function should print "11:00 PM".

Return Value: This function should not return anything.

get_ordinal_day(month, day)

Description: Assuming a regular year, that is, a non-leap year, determine the corresponding ordinal day for a given month and day. For example, if the entered arguments are March 15th, then the corresponding ordinal day will be 74. Note that *month* will be an integer whose value will be within the range of 1 and 12, and *day* will also be an integer whose value will be within the range of 1 and 31. During testing, correct values will be passed so there will be no need for error-checking.

Output: This function should not print anything to the screen.

Return Value: Return the corresponding ordinal day for the passed date.

ordinal_day_error_checking(month, day)

Description: This function will have all of the functionality of the *get_ordinal_day()* function, but now with error-checking. Suppose that the arguments *-1, 4* are passed. The latter argument is valid, as it falls within the range of acceptable values for *day*, but the former argument is invalid as it falls outside of the range of acceptable values for *month*. An important thing to consider is that not all months have 31 days: April only has 30 days, whereas February only has 28! Thus, in the development of

this function, make sure to consider those special cases. Whenever an invalid argument is encountered, return a -1. Otherwise, if the arguments are valid, return the corresponding ordinal day.

Output: This function should not print anything to the screen.

Return Value: -1 if an invalid argument is encountered, otherwise return the corresponding ordinal day.

3. Testing

In testing your implementations of the above functions, you are allowed to test the functions individually using your own test cases.

However, it is highly recommended that you make use of the provided functionsTester.py program to obtain immediate feedback on your implemented functions. Please note that the functionsTester.py program makes use of the functionsTesterGenOutput.py script. As such, it is suggested that a new folder is created which contains the following three files: your program which contains the implemented functions (your program MUST be named functions.py, otherwise the functionsTester.py program will NOT work), the functionsTester.py script, and the functionsTesterGenOutput.py script.

To further elaborate on what the functionsTester.py script does, it will conduct a total of **32** tests on your implemented functions according to the following breakdown:

- return_five() – Test Case 1
- return_pi() – Test Case 2
- return_hello() – Test Case 3
- return_True() – Test Case 4
- determine_data_type() – Test Cases 5, 6, 7, and 8
- powTen() – Test Cases 9 and 10
- cubed() – Test Cases 11 and 12
- fahrenheit_to_celcius() – Test Cases 13, 14, 15, and 16
- print_median_of_three() – Test Cases 17, 18, 19, and 20
- print_military_to_standard() – Test Cases 21, 22, 23, and 24
- get_ordinal_day() – Test Cases 25, 26, 27, and 28
- ordinal_day_error_checking() – Test Cases 29, 30, 31, and 32

The functionsTester.py program will attempt to import your own program (functions.py), and the functionsTesterGenOutput.py program. If the import fails, the program will notify you and terminate testing.

If the import is successful, for each of the above functions, an attempt will be made on calling the function. If the function is misspelled or not yet defined, that particular test case will crash, and you will be notified. If the function is "improperly" [1] defined, then you will fail that test case.

If none of the above occurs, and your implemented function was "properly" [2] defined, then a success will occur, and you pass that test case.

Once the program completes all 32 test cases, then a summary report will be displayed, and a summary report file (under the name of results.txt) will be created. The results.txt file contains information such as the test values that were used during testing, along with the number of test cases that were passed, the number of test cases that failed, and the number of test cases that crashed. The results.txt does NOT indicate specifically which test case was a pass, a fail, or crashed; the functionsTester.py program will output that information for your convenience so that you may adjust your efforts towards the successful implementation of the functions.

4. Requirements

Your program MUST be named as functions.py.

5. Help & Advice

If you feel overwhelmed by this assignment, don't be. A good starting point is to first create skeleton definitions of the required functions that returns dummy values. From there, work on the function that you feel is the easiest to implement. Once defined, run the functionsTester.py program and see if you were able to successfully implement it. From this point, rinse and repeat.

If you need any kind of clarification to anything on this assignment, such as if you're confused about the requirements for one of the functions, or if you encounter a bug error that you are unable to figure out on your own, please send an email to **chsprogvol@gmail.com**. You may expect a response within 24 to 48 hours

[1] Please keep in mind that there is no right or wrong way of implementing something in programming.
[2] Reference the above.