COMP 1405B
Fall 2019 – Practice Problems #5

Objectives

- Continue to practice using the programming concepts we have covered already
- Practice writing functions
- Practice using functions to organize code
- Practice following a precise specification

Remember to test your solutions to verify their correctness and discuss your solutions with both other students and TAs in the course. Verifying your solutions with TAs will be a good way of ensuring that you are on a path to success in the course.

Problem 1 (Common Multiples - *)

Write a function called **ismultiple** which takes 2 integer arguments (*a* and *b*). This function must return True if *b* is a multiple of *a* (i.e., *a* divides into *b* evenly) and False otherwise. Write a second function called **commonmultiple** which takes 3 integer arguments (*a*, *b*, and *c*). This function must return True if *c* is a multiple of both *a* and *b*, and False otherwise. **Additionally, the commonmultiple function must use two calls to the ismultiple function to compute its return value.** Once you have implemented both functions, include testing code that asks the user to enter two numbers (*a* and *b*) and prints out all numbers between 1 and 100 (inclusive) that are common multiples of *a* and *b*.

Problem 2 (Number of Digits Function - *)

This problem is a function-based version of a problem from a previous set of practice problems. If you completed the previous problem, you can probably use most of your code again. Create a function called numDigits(int) that takes an integer argument and prints the number of digits in that integer. You should **not** use the len() function to get the length of the string. Test the function by calling it with several integer values of varying length.

Problem 3 (Prime Number Function- *)

Write a function called isPrime(int) that takes a single integer argument. The function should return True if the given integer input is a prime number, and False otherwise. If you solved this problem with a loop in the previous set of practice problems, you can re-use most a lot of your code. Test the function by using a loop to call the function and print the result for all integers from 1-100.

Problem 4 (Binary Quiz Game - **)

Write and test a function called pow2() that takes one positive integer number as input and returns the largest power of 2 that's less than or equal to that number. Some examples include:

pow2(2) → 2
pow2(12) → 8
pow2(20) → 16
pow2(284) → 256

Write and test a function called decToBin() that takes in a decimal number as an argument and returns a string representing that number in binary. Your solution should *not* use the built-in bin() function, though it may be helpful when testing your function to verify your answers.

The design for this algorithm should work as follows:

```
Let x be the input value,
Let p b the largest power of 2 less than or equal to x,
and let output be "".

Repeat until p is 0:
     If p ≤ x,
     Then subtract p from x and add a "1" to the output.
     Otherwise add a "0" to the output

     Divide p by 2
Return output
```

Some examples:

decToBin(2) → "10"
decToBin(10) → "1010"
decToBin(16) → "10000"

Write a function called userBin() that takes an integer as an argument. This function should display the integer to the user and prompt them to enter the corresponding binary value. Have your function simply return the user's input (it will be used in the next step).

Finally, create a main() function for your program. The purpose of this function is to control the game-logic for a Binary Quiz game. The steps of this game are as follows:

- Generate a random integer in the range [0,256]
- Ask the user to enter the corresponding binary number.
- Check if their answer is correct or not
    - If so, add 1 to their score

The above steps should repeat until the user types "stop", at which point your program should print out their final score and finish. Your program should not use the built-in bin() function. Instead, use the decToBin function you created. The user's input, and the result of decToBin are both strings, so don't convert the binary numbers to integers.

Problem 5 (Grade Analysis – **)

The studentinfo.txt file available in the PP5-Resources.zip contains information for many randomly generated students. The student information follows the structure given below:

```
Student1_first_name
Student1_last_name
Student1_student_number
Student1_assignment_grade
Student1_midterm_grade
Student1_exam_grade
Student2_first_name
Student2_last_name
Student2_student_number
Student2_assignment_grade
Student2_midterm_grade
Student2_exam_grade
…..
```

Download the studentinfo.txt file and put it in the same directory as your Python code. Write and test a function that takes the first name, last name, and grade information for a student as input and adds that student to the file with a random student number between 10000 and 99999. Write and test a function that returns the average assignment grade, average midterm grade, and average exam grade, calculated across all students. If you accidentally introduce an error into the file (i.e., break the structure some how), you can redownload it to get a fresh copy. Write a function that returns the

name of the student with the highest overall final grade. For the final grade calculation, you can use the weighting of assignments=25%, midterm=25%, and exam=50%.

Problem 6 (Order/Inventory System Progressive Part 2 - ****)

A problem within the looping and files practice problem set developed an ordering system for an electronics store. The last part of that problem discussed storing inventory information in files. Additionally, we could have saved sales information to files, so the store could track and remember all transactions. For this set of problems, we will assume that somebody has added that functionality and generated several files recording sales information for different time periods. You will develop a module for performing some sales analysis by processing these files. For this set of problems, you should assume that each order saved to a file will consist of 6 lines of information in the following pattern: Customer Name, Number of Desktops Purchased, Number of Laptops Purchased, Number of Tablets Purchased, Number of Toasters Purchased, Total Cost of Purchase. A single file will consist of many orders, resulting in a file with the structure:

Purchase #1 Customer Name
Purchase #1 Number of Desktops
Purchase #1 Number of Laptops
Purchase #1 Number of Tablets
Purchase #1 Number of Toasters
Purchase #1 Total Cost
Purchase #2 Customer Name
Purchase #2 Number of Desktops
Purchase #2 Number of Laptops
Purchase #2 Number of Tablets
Purchase #2 Number of Toasters
Purchase #2 Total Cost
…etc…

The functions you write for this set of problems must work with any file that follows this format, regardless of the number of orders contained in the file. You do not need to create the files. You can use the salesinfoX.txt test files and the sales-stats-tester.py file included in the PP5-Resources.zip to see if your code is producing the correct values. You should read through the entire problem description first and plan accordingly. If you approach this problem correctly, you will not have to write a lot of code.

Note: Your module should only have function code. There should be no print statements and no code outside of the function definitions. Save all your functions inside a single file called salesstats.py.

The first function you write must be called get_number_purchases(filename). This function must process the file with the name specified in the filename argument and return an integer representing the total number of purchases that have occurred (that is, the total number of orders, not the total number of products).

Add another function to the salesstats.py file called get_total_purchases(filename). This function must process the file with the name specified in the filename argument and return the total cost of all purchases that have occurred (that is, the sum of the last lines of each purchase record).

Add another function to the salesstats.py file called get_average_purchases(filename). This function must process the file with the name specified in the filename argument and return the average cost of all purchases that have occurred.

Add a new function to salesstats.py called get_number_customer_purchases(filename, customer). This function must process the file with the name specified in the filename argument and return an integer representing the number of purchases that have been made by a person with the name specified in the customer argument.

Add a new function to salesstats.py called get_total_customer_purchases(filename, customer). This function must process the file with the name specified in the filename argument and return the total cost of all purchases that have been made by a person with the name specified in the customer argument.

Add a new function to salesstats.py called get_average_customer_purchases(filename, customer). This function must process the file with the name specified in the filename argument and return the average cost of all purchases that have been made by a person with the name specified in the customer argument.

Add a final function to the salesstats.py file called get_most_popular_product(filename). This function must process the file with the name specified in the filename argument and return a string value representing the product that sold the highest number of units (“Desktop”, “Laptop”, “Tablet”, or “Toaster”). In the case of a tie, your function can return any of the highest selling products.

Note: If it makes it easier to program, you can assume that no customer will ever have the name “*”. It may be worth thinking about how this could allow you to generalize some of your functions and eliminate duplicated code.