**Week 4 In-Class Exercises**

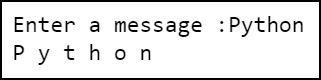
**(Strings and for-Loops)**

**Q1: Print a Message with Separators**

**Part I [ \* ]**

Prompt the user for a message. Display the message character by character but **separate every two adjacent characters by a space**.

Your code should produce the following output:

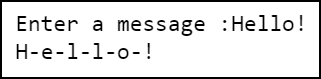
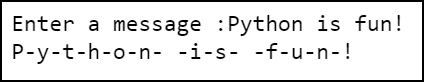
**Hint:**

* Use a for-loop to go through all the characters in the message one by one.
* Use print(s, end='') to print out s and still keep the cursor in the same row.

**Part II [ \*\* ]**

Now write another piece of code that displays the message from the user character by character but separate every two adjacent characters **by a hyphen**.

Your code should produce the following output:

**Hint:** The last character should be handled differently from the other characters.

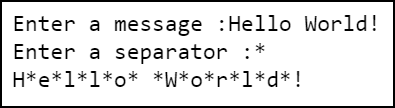
**Part III [ \*\* ]**

Now let us make the code above re-usable and more general.

Define a function called print\_message\_with\_separators that does the following:

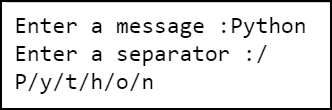
* The function takes in two parameters:
  + msg (of type str): a message string (e.g., 'Hello!')
  + sep (of type str): a separator string (e.g., '-')
* The function **prints out** msg character by character, but it inserts sep between every two adjacent characters of msg.
* The function **does not return any value**.

For example, when print\_message\_with\_separators("Hello World!", '\*') is called, we should see the following output:



We can see that the separator '\*' is inserted everywhere between two adjacent characters in the message.

When print\_message\_with\_separators("Python", '/') is called, we should see the following output:



**Q2: Display Numbers [ \*\* ]**

Define a function called display\_numbers. The function takes in two parameters: **m** and **n**. It is assumed that both **m** and **n** are integers, and **m** is less than or equal to **n**. The function displays all the integers between **m** and **n** (both inclusive) one by one, separated by spaces. However, the function handles the following numbers in a special way:

* If the number is a multiple of 3 but not a multiple of 5, the function displays a hyphen instead of the number.
* If the number is a multiple of 5 but not a multiple of 3, the function displays an asterisk instead of the number.
* If the number is a multiple of both 3 and 5, the function displays a # instead of the number.

For example, calling display\_numbers(4, 16) gives the following output to the screen:



**Q3: Compute Sum [ \*\* ]**

Inside a file called sum.py, write a function called compute\_sum that takes in two parameters, **m** and **n**. The function returns the sum of all the integers between **m** and **n** (both inclusive).

Add the following code to sum.py, after your own code:

my\_sum = compute\_sum(4, 10)  
print("The sum is " + str(my\_sum))

We should see the following output when sum.py is executed:



**Q4: Palindrome [ \*\*\* ]**

A word is called a palindrome if it reads the same backwards as forwards. For example, “**madam**” and “**rotator**” are examples of palindromes.

Write a program that prompts the user for a word and checks whether the word is a palindrome. Your code should produce the following output:



**Q5: Buying Books [ \*\*\* ]**

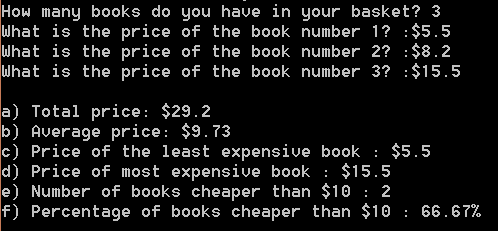
Write a simple program called books.py that helps a bookstore’s customer to gather some statistics of the books she has picked.

The program first asks the customer how many books she has in her basket. The program then prompts the customer for the price of each book. After that, the program does the following:

1. The program displays the total price of all the books.
2. The program displays the average price of all the books.
3. The program displays the price of the least expensive book
4. The program displays the price of the most expensive book.
5. The program displays how many books are cheaper than $10.00.
6. The program displays the percentage of books cheaper than $10.00.

**Note:** You may want to use the round() function to limit the number of digits after the decimal point to 2.

Your code should produce the following output:

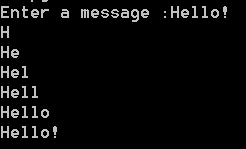


**Q6: Playing with Strings [ \*\*\* ]**

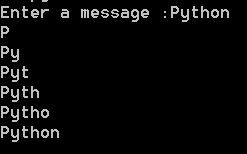
**Part I**

Write a program that prompts the user for a string as a message. The program displays a triangle that gradually reveals the whole message, as shown below.

Suppose the message is “Hello!”. The program displays the following output:



Suppose the message is “Python”. The program displays the following output:



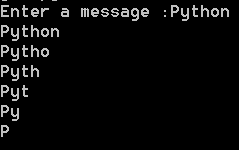
**Part II**

Write a similar program that displays the triangle upside down, as shown below.

Suppose the message is “Hello!”. The program displays the following output:



Suppose the message is “Python”. The program displays the following output:



**Q7: Password Strength [ \*\*\* ]**

A strong password should have the following criteria:

* Be at least 8 characters long
* Contain a lower case letter (a-z)
* Contain an upper case letter (A-Z)
* Contain a number (0-9)
* Contain a special character (!@#$)

Write a code snippet that asks the user for a password, and then checks for the strength of the password. You should print True if the password is strong, and False otherwise.

Here is an example of how the output may look like with a strong password.

Input a password: ThisIsAStrongPassword123!

True

Here is an example of how the output may look like with a weak password.

Input a password: ThisIsAStrongPassword

False

You should be able to complete your solution in 30 lines of code (or less).