EECS 1015 Lab 8

Goal

- Be able to write a script that contains conditional statements and loops

Tasks

1. Guide you through the process of planning to write a script that uses loops and multiple functions
2. learn to debug a script that contains a for-loop and if statement
3. Learn to write a script with loop
4. Learn to write a script with loop and if statement
5. Learn to write a script with loop
6. Learn to write a script with loop and if statement

Total Credit: 100 pts

You are very welcome to ask clarification questions to TAs. But please read the document carefully before you ask questions.

It is an academic offense to copy code from other students, provide code to other students, let other people (including the teaching staff) debug your code, or debug other students' code.

We will check code similarities at the end of the semester.

Questions 3 – 6 may not be arranged by the difficulty level. You are highly recommended to take a look and decide the order of completing the questions. You will experience something similar in the exams and it is important to learn how to triage your tasks.

Note: For question 3 – 6, you must come up with a function description in your own words, and cannot copy the words from handout.

Task 1: Follow the Steps (30 pts)

For this task, we are going to write a script that can count the number of prime numbers in a given range. A prime number is a number that is only divisible by 1 and itself. For example, 13 is a prime number because it can be divided by 1 and itself (e.g., 13). For this task, you will write the count_primes() function. count_primes() takes two arguments, *start* and *stop*. It will look at every number in the range of [*start*, *stop*] inclusive and return the number of prime numbers within that range.

You are highly recommended to attempt the questions yourself before you look at the solution as a way to learn.

Q1.1 Based on the description above, declare the function and write the function recipe including test cases. You can fill the rest of the body of the function with a pass statement for now.

Solution:

```
def count_primes(start: int, stop: int) -> int:
    """
    Counts the total of prime numbers in the range [start, stop]

    >>> count_primes(3, 10)
    3
    >>> count_primes(2, 12)
    5
    >>> count_primes(1, 100)
    25
    """
    pass
```

Q1.2 Planning out each step you need to take to achieve the function goal.

Solution:

```
def count_primes(start: int, stop: int) -> int:
    """
    Counts the total of prime numbers in the range [start, stop]

    >>> count_primes(3, 10)
    3
    >>> count_primes(2, 12)
    5
    >>> count_primes(1, 100)
    25
    """
    # check preconditions

    # create a counter variable

    # loop from [start, stop]
    #   if prime add 1 to counter

    # return total number of primes
```

Q1.3 Check pre-conditions, i.e., make sure the user passed in positive integers.

Solution:

```
def count_primes(start: int, stop: int) -> int:
    """
    Counts the total of prime numbers in the range [start, stop]

    >>> count_primes(3, 10)
    3
    >>> count_primes(2, 12)
    5
    >>> count_primes(1, 100)
    25
    """
    # check preconditions
    # start must a positive int
    assert type(start) == int and start > 0, "start must be a positive int"
    # stop must a positive int
    assert type(stop) == int and stop > 0, "stop must be a positive int"

    # create a counter variable

    # loop from [start, stop]
    #   if prime add 1 to counter

    # return total number of primes
```

Q1.4 Create a counter variable so we can keep track of the number of primes we have seen. Also, Write the for-loop. Leave the loop body as a pass statement for now

Solution:

```
def count_primes(start: int, stop: int) -> int:
    """
    Counts the total of prime numbers in the range [start, stop]

    >>> count_primes(3, 10)
    3
    >>> count_primes(2, 12)
    5
    >>> count_primes(1, 100)
    25
    """
    # check preconditions
    # start must a positive int
    assert type(start) == int and start > 0, "start must be a positive int"
    # stop must a positive int
    assert type(stop) == int and stop > 0, "stop must be a positive int"

    # create a counter variable
    counter = 0

    # loop from [start, stop]
    for num in range(start, stop + 1):
        pass
    #   if prime add 1 to counter

    # return total number of primes
```

Q1.5 Think about how we can calculate if a number is prime. Should we create a new function to do it?

Solution:

```
def count_primes(start: int, stop: int) -> int:
    """
    Counts the total of prime numbers in the range [start, stop]

    >>> count_primes(3, 10)
    3
    >>> count_primes(2, 12)
    5
    >>> count_primes(1, 100)
    25
    """
    # check preconditions
    # start must a positive int
    assert type(start) == int and start > 0, "start must be a positive int"
    # stop must a positive int
    assert type(stop) == int and stop > 0, "stop must be a positive int"

    # create a counter variable
    counter = 0

    # loop from [start, stop]
    for num in range(start, stop + 1):
    #   if prime add 1 to counter
        if is_prime(num):
            counter += 1

    # return total number of primes
```

Let's create a function that will check if the number is prime. The reason we will do this is to simplify our code within the count_primes() function. A function should only do one thing, so if this function's goal is to count the number of primes then we should have another function that will check if a number is prime. We can think of is_prime() as a helper function in this case.

Q1.6 Write the function design recipe for the is_prime() function and check the pre-conditions. is_prime() takes the argument *num*, and the function will return True or False whether *num* was prime

Solution:

```
def is_prime(num: int) -> bool:
    """
    Determines if a number is prime
    >>> is_prime(5)
    True
    >>> is_prime(10)
    False
    >>> is_prime(23)
    True
    """
    # check preconditions
    # num must a positive int
    assert type(num) == int and num > 0, "num must be a positive int"

    if num == 1:
        return False
```

2 is the smallest prime number so we can return False if the number we are checking is 1. We won't put it in the assert statement so that we can still pass 1 to this function while looping through the range in the count_primes() function.

Q1.7 Planning for the steps of implementing the is_prime() function

Solution:

```
def is_prime(num: int) -> bool:
    """
    Determines if a number is prime
    >>> is_prime(5)
    True
    >>> is_prime(10)
    False
    >>> is_prime(23)
    True
    """
    # check preconditions
    # num must a positive int
    assert type(num) == int and num > 0, "num must be a positive int"

    if num == 1:
        return False

    # create boolean flag for if the num is prime

    # loop through divisors
    #   if divisible by any number other than 1 and itself then num is not prime

    # return if the number is prime
```

Q1.8 Write the for loop of the is_prime() function. You can leave the loop body as pass statement for now.

Solution:

```
def is_prime(num: int) -> bool:
    """
    Determines if a number is prime
    >>> is_prime(5)
    True
    >>> is_prime(10)
    False
    >>> is_prime(23)
    True
    """
    # check preconditions
    # num must a positive int
    assert type(num) == int and num > 0, "num must be a positive int"

    if num == 1:
        return False

    # loop through all numbers from [2, num)
    for i in range(2, num):
        pass
    #   if divisible by any number in this range then num is not prime

    # return if the number is prime
```

We will loop from [2, num). We are skipping 0 because you can't divide by zero, and also 1 because everything is divisible by 1. The point of the for loop is to check if the number is not prime so we want to start dividing by 2.

Q1.9 Finish the is_prime() function

Solution:

```
def is_prime(num: int) -> bool:
    """
    Determines if a number is prime
    >>> is_prime(5)
    True
    >>> is_prime(10)
    False
    >>> is_prime(23)
    True
    """
    # check preconditions
    # num must a positive int
    assert type(num) == int and num > 0, "num must be a positive int"

    if num == 1:
        return False

    # loop through all numbers from [2, num)
    for i in range(2, num):
    #   if divisible by any number in this range then num is not prime
        if num % i == 0:
            return False

    # return if the number is prime
    return True
```

An easy way to tell if a number is divisible by a divisor is to check if there is a remainder. If x/y has no remainder, then x is divisible by y. To check the remainder when dividing x by y, you can use the modulo operator in Python (%). For example, 4%2 equals 0 and 5%2 equals 1. This shows that 4 and 2 are divisible, and 5 is not divisible by 2. Use the modulo operator to check if *num* is divisible by each divisor. If it is not divisible by any divisor in the range of [2, num) then num is prime, else it is not prime.

Q1.10 Finish the count_primes() function

Solution:

To complete count_primes() we just have to return the counter

```
def count_primes(start: int, stop: int) -> int:
    """
    Counts the total of prime numbers in the range [start, stop]

    >>> count_primes(3, 10)
    3
    >>> count_primes(2, 12)
    5
    >>> count_primes(1, 100)
    25
    """
    # check preconditions
    # start must a positive int
    assert type(start) == int and start > 0, "start must be a positive int"
    # stop must a positive int
    assert type(stop) == int and stop > 0, "stop must be a positive int"

    # create a counter variable
    counter = 0

    # loop from [start, stop]
    for num in range(start, stop + 1):
    #   if prime add 1 to counter
        if is_prime(num):
            counter += 1

    # return total number of primes
    return counter
```

Submission

- Go to eClass -> our course -> Week 8 -> Practice -> Lab -> Lab 8 Submission
- Copy your code to Task 1
- You can resubmit your code as many times as you need, but you need to wait for 5 minutes before submission.

Rubrics:

- You will get full marks for this question if you submit the solution (with no typo) to the required location on Prairielearn before the deadline.

Task 2: Debugging (30 pts)

For this task, we will debug a function called repeat_sum(). This function takes one argument *num*

- If num is an even number, then the function should return the sum of all even numbers from [1, num] inclusive
- if num is an odd number, then the function should return the sum of all odd numbers from [1, num] inclusive

For example, when *num* is 8, the output should be 20 (e.g., 2 + 4 + 6 + 8). When *num* is 7, the output should be 16 (e.g., 1 + 3 + 5 + 7).

The code to debug (lab8_task2.py) may have syntax errors, run-time errors, semantic errors or errors in test cases. To help with debugging, you can use the debugger to execute the code line by line (you should fix the syntax errors before doing this).

Submission

- Copy the Python code to Lab 8 -> Task 2 on Prairielearn.
- You can resubmit your code as many times as you need, but you need to wait for 5 minutes before submission
- Copy the Python code to Lab 8 -> Task 2 on Prairielearn.
- Note: In order for the autograder to work properly:
    - You must NOT change the name of the function
    - You must NOT change the order of the arguments

Rubric

- You will not receive any mark if you do not submit it to the designated location on Prairielearn before the deadline
- You will not receive any mark if your code violates the above requirements
- Otherwise
    - you will receive 5 points if your script correctly provides a solution for repeat_sum(10), ANS: 30
    - you will receive 5 points if your script correctly provides a solution for repeat_sum(9), ANS: 25
    - you will receive 5 points if your script correctly provides a solution for repeat_sum(20), ANS: 110
    - you will receive 5 points if your script correctly provides a solution for repeat_sum(33), ANS: 289
    - you will receive 10 points for providing a solution for two other test cases (5 pts each)

Task 3: Implementation (10 pts)

π is an irrational number - this means that it can't be expressed as a fraction, so how do we compute it? We can do so by using the Leibniz formula, named after the mathematician Gottfried Wilhelm von Leibniz.

$$\pi = 4 * \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1}$$

Using the Leibniz formula, the more terms we compute in the series, the more accurate our calculation of π. Your task is to compute π up to a user-specified number of terms (i.e., up to M terms)

$$\pi = 4 \times \sum_{n=0}^{M} \frac{(-1)^n}{2n+1}$$

Let's walk through the formula to understand what's going on. The ∑ (sigma) symbol denotes a sum of multiple terms. You can think of it as a mathematical way of expressing a for loop from n up to and including M, where n and M are variables. In this case, the term we are referring to is the highlighted blue portion of our formula.

An example if n=0 and M=2:

$$4 * \sum_{n=0}^{2} \frac{-1^n}{2n+1} = 4 * ((\frac{-1^{(0)}}{2(0)+1}) + (\frac{-1^{(1)}}{2(1)+1}) + (\frac{-1^{(2)}}{2(2)+1})) = 3.466\overline{6}$$

To complete this task you will finish the implementation of a function called approximate_pi that has been started for you in the file lab8_task3.py. This function has one argument *m* which specifies how many terms we will calculate in our series. The function will return the approximation of pi based on m. You should notice that as you call the function with a higher and higher value for m the approximation of pi will get closer and closer to 3.1415…

Some example outputs would be:

approximate_pi(1) → 2.6667
approximate_pi(5) → 2.976
approximate_pi(8) → 3.2524
approximate_pi(20) → 3.1892

Floating point numbers in Python are not the most accurate, so the last line of the function will round the approximation to four decimal places. Do not change this or you will not pass the tests.

Requirement

- You must use a for-loop or while-loop to do the summation for your calculation
- Do not change the last line of the function that rounds the result
- You cannot use any other libraries or packages to get the answer for you

Submission

- Copy the Python code to Lab 8 -> Task 3 on Prairielearn.
- You can resubmit your code as many times as you need, but you need to wait for 5 minutes before submission.
- Note: In order for the autograder to work properly:
    - You must NOT change the name of the function
    - You must NOT change the order of the arguments

Rubric

- You will not receive any mark if you do not submit it to the designated location on Prairielearn before the deadline
- You will not receive any mark if your code violates the above requirements
- Otherwise
    - You will receive 0.5 pt for function signature
    - You will receive 0.5 pt for function description
    - You will receive 0.5 pt for doctest (At least 4, and they cannot be the cases below)
    - You will receive 1 pt if your script correctly provides the solution to: `approximate_pi(1), ANS: 2.6667`
    - You will receive 1 pt if your script correctly provides the solution to: `approximate_pi(5), ANS: 2.976`
    - You will receive 1 pt if your script correctly provides the solution to: `approximate_pi(8), ANS: 3.2524`
    - You will receive 1 pt if your script correctly provides the solution to: `approximate_pi(20), ANS: 3.1892`
    - You will receive 4.5 pt if your script correctly provides the solution to other examples
        - Hidden case 1: 1 pt
        - Hidden case 2: 1 pt
        - Hidden case 3: 2.5 pts

Task 4: Implementation (10 pts)

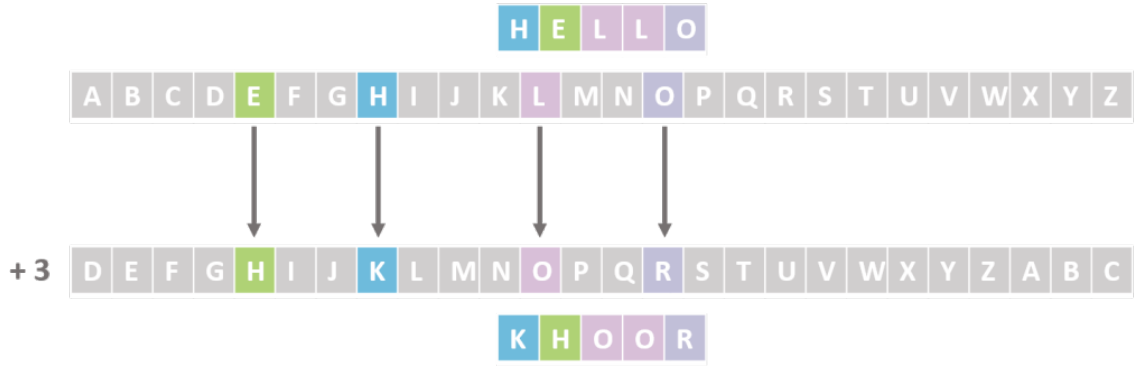
In cryptography, a cipher is an algorithm for performing encryption or decryption. For this task, you will be implementing a simple cipher from around 58 BC called the Caesar Cipher. Used by Julius Caesar, this substitution cipher shifts letters in a message to make it unreadable. To decrypt the message, one must simply reverse the shift. Here is an example:

message: ATTACK AT NIGHTFALL
shift: 4
encrypted message: EXXEGO EX RMKLXJEPP

Here is how encryption works. For every letter in the original message, you replace the letter with another letter some number of positions down the alphabet. The number of positions down the alphabet is specified by the ***shift*** value. For example, if your shift value is 1, then A is shifted to B. Here are more examples:

ENCRYPT:
shift=2: B → D
shift=10: G → Q
shift=8: A → I

Here is an image to show how to find the encrypted message if the shift is 3:



(image source: https://girlscodetoo.co.uk/encode-secrets-with-the-caesar-cipher/)

When you are encrypting, you can think of it as shifting to the *right*. What happens if the shift is greater than the number of letters in the alphabet? Or what happens if you try to shift a letter but it goes past Z? In that case, you loop back around. So for example:

ENCRYPT:
shift=1: Z → A
shift=5: W → B
shift=26: A → A

For this task, you will implement the encrypt function. The encrypt function has been started for you in lab8_task4.py. encrypt will receive two arguments - message and shift. It should return a string containing the encrypted message. A decrypt function that depends on your encrypt function will also be provided. This function is there just so you can check your work. If

you pass the message returned by the encrypt function to the decrypt function and use the same shift value, then you should get the original message back.

HINT:

In Python, You can convert string characters to numbers (e.g., Unicode representation) using the `ord()` function. You can convert a number back to a character using the `chr()` function. For example:

ord('A') → 65
chr(90) → Z

The uppercase alphabet is associated with the range of numbers [65, 90]. The ord function will return the number of the letter in the same order as the alphabet. You can use these functions as well as some math operations to handle the shifting of letters in your message.

You may assume all letters in the messages will be in uppercase and the only thing you need to shift are letters. Numbers and punctuation marks do not have to be accounted for.

Requirement

- You must use a for loop or while loop during your encryption and decryption process

Submission

- Copy the Python code to Lab 8 -> Task 4 on Prairielearn.
- You can resubmit your code as many times as you need, but you need to wait for 5 minutes before submission
- Note: In order for the autograder to work properly:
  - You must NOT change the name of the function
  - You must NOT change the order of the arguments

Rubric

- You will not receive any mark if you do not submit it to the designated location on Prairielearn before the deadline
- You will not receive any mark if your code violates the above requirements
- Otherwise
  - You will receive 0.5 pt for function signature
  - You will receive 0.5 pt for function description
  - You will receive 0.5 pt for doctest (At least 4, and they cannot be the cases below)
  - You will receive 1 pt if your script correctly provides the solution to: `encrypt("ABCDEFG", 4), ANS: EFGHIJK`
  - You will receive 1 pt if your script correctly provides the solution to: `encrypt("SEND 5 PIZZAS", 12), ANS: EQZP 5 BULLME`

- You will receive 1 pt if your script correctly provides the solution to: `encrypt("READY THE TROOPS!!", 20), ANS: LYUXS NBY NLIIJM!!`
- You will receive 1 pt if your script correctly provides the solution to: `encrypt("SUPER SECRET MESSAGE 123", 13), ANS: FHCRE FRPERG ZRFFNTR 123`
- You will receive 4.5 pts if your script correctly provides the solution to other examples
    - Hidden case 1: 1 pt
    - Hidden case 2: 1 pt
    - Hidden case 3: 2.5 pts

Task 5: Implementation (10 pts)

In this task, you will implement a function and no starter code is provided. The function should be called reverse_str and accept one argument. Given a string, the function should return the reversed string. For example:

```
>>> reverse_str("abcde")
'edcba'
>>> reverse_str("a")
'a'
```

The parameter can be any string and there is no other specifications of the format of the string

Submission

Copy the Python code to Lab 8 -> Task 5 on Prairielearn.

You can resubmit your code as many times as you need, but you need to wait for 5 minutes before submission (You need to spend some time debugging!).

Rubrics:

- You will not receive any mark if you do not submit it to the designated location on Prairielearn before the deadline
- You will not receive any mark if your code violates the above requirements
- Otherwise
    - You will receive 0.25 pts if you provide a description of the function.
    - You will receive 0.25 pts if you provide the argument annotations correctly
    - You will receive 0.5 pt if you include at least 4 test cases (and they cannot be the same as test cases provided) and your code pass those test cases
    - You will receive 1 pts if your function passes general test (e.g., reverse_str("abcdefgh"), expected output: "hgfedcba")
    - You will receive 1 pts if your function passes the corner test - Singleton data (e.g., reverse_str("c"), expected output: "c")
    - You will receive 1 pts if your function passes the corner test - zero data (e.g., reverse_str(""), expected output: "")
    - You will receive 1 pts if your function passes the condition test - string contains digits (e.g., reverse_str("123"), expected output: "321")
    - You will receive 1 pts if your function passes the condition test - string contains special characters (e.g., reverse_str(" a@54! "), expected output: " !45@a ")
    - You will receive 1 pts if your function passes the contract violation test (e.g., reverse_str(True), expected output: raise error)

- You will receive 3 pts if your function passes three more hidden test (1 point each)

Task 6: Implementation (10 pts)

In this task, you will implement a function and no starter code is provided. The function should be called remove_vowels and accept one argument. Given a string, the function should return a string with vowels removed (e.g., a, e, i, o, u). For example:

```
>>> remove_vowels("Python Programming")
'Pythn Prgrmmng'
>>> remove_vowels("abc")
'bc'
```

The parameter may only contain letters and spaces. There are no other specifications of the format of the string. When you handle the preconditions, think of using string methods, rather than writing everything from scratch yourself. If you write everything yourself, you will need to write code and debug this part as well. The string methods provided by Python have been tested which saves your time and effort. Please note that you can call existing functions, this is not cheating or plagiarism. But copy and paste code is cheating and/or plagiarism.

Submission

Copy the Python code to Lab 8 -> Task 6 on Prairielearn.

You can resubmit your code as many times as you need, but you need to wait for 5 minutes before submission (You need to spend some time debugging!).

Rubrics:

- You will not receive any mark if you do not submit it to the designated location on Prairielearn before the deadline
- You will not receive any mark if your code violates the above requirements
- Otherwise
    - You will receive 0.25 pts if you provide a description of the function.
    - You will receive 0.25 pts if you provide the argument annotations correctly
    - You will receive 0.5 pt if you include at least 4 test cases and your code pass those test cases
    - You will receive 1 pts if your function passes general test (e.g., remove_vowels("moving"), expected output: "mvng")
    - You will receive 1 pts if your function passes the corner test - Singleton data (e.g., remove_vowels("a"), expected output: "")
    - You will receive 1 pts if your function passes the corner test - zero data (e.g., remove_vowels(""), expected output: "")

- You will receive 1 pts if your function passes the condition test - string contains upper leters (e.g., remove_vowels("APple"), expected output: "Ppl")
- You will receive 1 pts if your function passes the condition test - string contains spaces (e.g., remove_vowels(" ch erry"), expected output: " ch rry")
- You will receive 1 pts if your function passes the contract violation test - string contains digits (e.g., remove_vowels(" number 1 "), expected output: raise error)
- You will receive 3 pts if your function passes three more hidden test (1 point each).