Week: 04 Date: 2/10/2022

15-110 Recitation Week 4

Reminders

- HW2 due Monday 2/14 @ Noon EDT
- Recitation feedback form: https://forms.gle/L21iwvPZgr3BCaWZA

Overview

- While loops
- For Loops
- Strings

Problems

While Loops vs. For Loops + Q/A

For loops! These allow you to set a specific range of values to iterate through ahead of time:

- General format: for i in range(x, y, z):
  - `i` → loop variable - contains current value of iteration
    - Can use any variable name here as long as we keep it consistent
  - `range(x,y,z)` → start value, end value not inclusive, step size
    - Start value is inclusive, end value is exclusive, x and z are optional

While loops! These allow you to set a certain condition under which we keep iterating.

- General format: while (condition):
  - Condition will generally be some boolean expression. As long as this expression evaluates to `True`, we continue to iterate.
  - Make sure the expression evaluates to `False` at some point otherwise we end up in an infinite loop!

For loop example:

```
for i in range(0,10,2):
    print(i)
```

Now recall an example of a similar while loop:

```
i = 0
while i <= 10:
    print(i)
    i += 2
```

Note: these two examples are not equivalent! The for loop will not print 10, need to increase the end value to greater than 10. Also after the while loop, the value of i will be 12, whereas after the for loop (without changing the end value), the value of i will be 8.

While vs For Loops:

- For loops are used for a fixed number of iterations, help you avoid infinite loops
- While loops require declaring an iterator variable outside the loop and updating that variable within the loop. That is abstracted away in for loops by using range()
- While loops are more versatile, condition statement gives you more flexibility
- You can write any for loop as a while loop!

Nested For Loop Practice

1. Open the week 4 starter file and run the function `nestedFor()` using the function calls provided in the file. Pay close attention to the order of the print statements as that tells you the control flow through the two loops!

2. Next, write a function `oddsBefore(x)` that takes in a number x and prints, on separate lines, numbers 1 through x (inclusive) along with all the odd numbers that appear before each number, separated by a space. If there are no odds before a given number, just print the number. For example, if we call oddsBefore(4), we will get the following printout:

```
0:
1:
2: 1
3: 1
4: 1 3
```

Use the following algorithm:

1. Iterate over each of the numbers from 0 to x using a for loop, but be sure that the range calculated is inclusive at the upper bound (How might we ensure inclusivity at the upper bound or range()?)
    a. Inside this outer loop, create a variable, **odds**, for keeping track of odds in a string (Why do we do this inside the outer loop? What would happen if we put this variable outside the outer loop?)
    b. Create another loop to iterate over all the numbers less than the current value for the outer loop to check the previous numbers. For each number, we need to figure out how many numbers before it are odd. Thus we need an inner loop!
        i. Check if the number value of the inner iteration is odd or even. If it is odd, add it to the **odds** string with a space before it (Is there anything we need to do to the number value before adding it to the string?)
    c. Print the current value for the outer loop along with the **odds** string.

Quick String Questions

Given the string `s = "abcdefghi"`, answer the following short answer questions:

1) How do I access the character `"i"` from string `s`?
2) How do I create a string `x` which is equal to `"cdef"` using `s`?
3) How do I create a string `x` which is equal to "`beh`" using `s`?
4) How do I create a string `x` which is equal to the reverse of `s`?

Strings Functions Code Writing

1. Write a function `everyOther` that takes in a string s and prints every other letter of s.

2. Next, write a function **isSubString(sub, x)** that returns the index of the start of the left-most occurrence of a substring within a string. If the substring is not found, return -1. Assume len(sub) <= len(x). If the substring is the empty string, return 0. For example, if x = "hello", then:

`isSubString("", x)` returns 0
`isSubString("h", x)` returns 0
`isSubString("e", x)` returns 1
`isSubString("l", x)` returns 2 #Note, even with 2 l's, only the first (leftmost) is returned
`isSubString("ello", x)` returns 1

When slicing a string, you can slice using expressions as well. For example, if we want to slice the first three letters of the string out and have an index variable **y** set to the value 0, we can do the following:

```
s = "Goodbye" #String you want to slice the first three letters out
y = 0 #Our index variable
result = s[y : y + 3] #Will evaluate to "Goo"
```

Because slicing will never go out of bounds, we can imagine the following algorithmic approach to the problem:

1. Iterate over the string **x**
2. For every index, we check if the substring created starting from that index spanning to the index + the length of **sub** is equal to **sub** (Why do we check using the length of **sub** here? How might we use the length of **sub** in our slicing expression?)
    a. If the substring created above is equal to **sub** exactly, then we can return the index immediately
3. If the **sub** is not found in **x**, return -1 outside the loop (Why do we want to return -1 outside the outer loop?)