ECS 32A: Programming Assignment #3

Instructor: Aaron Kaloti

Summer Session #1 2020

Contents

1 Changelog

You should always refer to the latest version of this document.

- v.1: Initial version.
- v.2: Updated hint for part #8.
- v.3: Fixed typo in part #5 directions.
- v.4: Added a note about `break` for part #3.

2 Due Date

This assignment is due the night of Saturday, July 11. Gradescope will say 12:30 AM on Sunday, July 12, due to the "grace period" (as described in the syllabus). *Do not rely on the grace period for extra time; this is risky.*

Some students tend to email me very close to the deadline. This is also a bad idea. There is no guarantee that I will check my email right before the deadline.

3 General Submission Requirements

Use whichever environment/editor that you prefer, so long as it produces a Python file. Make sure that you submit the Python files with the correct names to Gradescope; the Python file names are given in each part below. These names must match *exactly*. You may submit infinitely many times to Gradescope, before the deadline.

Once the deadline occurs, whatever submission is your **active submission** will be the one that dictates your final score. By default, your active submission will be your latest submission. However, you can change which submission is your active submission, which I talked about during the Thursday (6/25) lecture.

The output of your programs needs to match the expected output ***exactly***.

4 Grading Breakdown

This assignment is worth 9% of your final grade and will be worth 90 points in the autograder. Here is the breakdown of the 90 points by part:

- Part #1: 5
- Part #2: 5
- Part #3: 10
- Part #4: 20
- Part #5: 10
- Part #6: 10
- Part #7: 10
- Part #8: 20

5 Problems

For any given part, if I do not require you to use a specific kind of loop (i.e. `while` loop vs. `for` loop), then you can use any kind that you want. Do note, however, that on the exams, I may ask that you use a specific kind of loop, so you may want to avoid strictly using one kind of loop instead of the other. Not every part needs to be solved with a loop.

5.1 Part #1

File: `part1.py`

Write a program that prompts the user for an integer N and computes the sum of squares from 1 to N, i.e. computes $\sum_{i=1}^{N} i^2$.

Your approach for this part must be driven by a `while` loop, *not* a `for` loop.

Below are examples of how your program should behave. In this first example, the answer 14 comes from $1^2 + 2^2 + 3^2 = 1 + 4 + 9 = 14$.

```
Enter N: 3
The sum is: 14
```

For this second example, the answer 55 comes from $1^2 + 2^2 + 3^2 + 4^2 + 5^2 = 1 + 4 + 9 + 16 + 25 = 55$.

```
Enter N: 5
The sum is: 55
```

```
Enter N: 12
The sum is: 650
```

5.2 Part #2

File: `part2.py`

Redo part #1 but with an approach that is driven by a `for` loop and *not* a `while` loop.

5.3 Part #3

File: `part3.py`

Write a program that keeps asking the user to enter an integer until the sum of all integers entered so far becomes at least 12 or less than -20, at which point the program should print "End of program" and end. You may assume the user will only enter integers.

Which is better to use here: a `while` loop? Or a `for` loop? (You don't need to submit an answer to this question.)

A few students have asked me if it is okay to use `break` for this part. That is fine. This is arguably an example of a problem in which `break` feels "natural". When I say in lecture that `break` and `continue` are rarely used, that is not meant to

be interpreted as, "You should not use `break` or `continue`." At this level, you should not worry too much about which Python language features are OK to use, unless I prohibit that specific feature.

Below are examples of how your program should behave. Do not print anything within square brackets "[...]"; those are meant to clarify the examples.

```
Enter integer: -2
Enter integer: -3
Enter integer: 8
Enter integer: 9
End of program
```

```
Enter integer: -8
Enter integer: 6
Enter integer: -3
Enter integer: 0
Enter integer: -20   [the sum became -25, so stop asking for user input]
End of program
```

```
Enter integer: 53    [the sum became 53, so stop asking for user input]
End of program
```

5.4 Part #4

File: `part4.py`

You will write a soda machine simulator. Write a program that iterates and asks for user input on each iteration. Each time, the user input falls under one of the following scenarios (and you may assume it always will):

- **q**: The user wishes to enter a quarter.
- **s**:
  - If the user has at least one dollar in the machine, print a message telling them they will receive a soda *and exit the program.*
  - Otherwise, tell them how many more quarters they need to enter in order to be able to get a soda.
- **Done**: The user is done. Exit the program.

Which is better to use here: a `while` loop? Or a `for` loop? (Again, you don't need to submit an answer to this question.)

Below are examples of how your program should behave.

```
Enter: q
Enter: q
Enter: q
Enter: q
Enter: q
Enter: s
Enjoy your soda!
```

```
Enter: q
Enter: q
Enter: q
Enter: s
You must enter 1 more quarters.
Enter: q
Enter: s
Enjoy your soda!
```

```
Enter: s
You must enter 4 more quarters.
Enter: q
Enter: s
You must enter 3 more quarters.
Enter: q
Enter: s
You must enter 2 more quarters.
Enter: q
Enter: s
You must enter 1 more quarters.
Enter: q
Enter: Done
```

```
Enter: q
Enter: s
You must enter 3 more quarters.
Enter: q
Enter: Done
```

Hint (or Anecdote?): This was the problem that, after I assigned it last time I taught this class, taught me that prior programming experience is overrated. This problem turned out to be difficult for experienced[1] and new coders alike, but for different reasons. The new coders found it hard to think of an approach. However, the experienced coders had no issues thinking of approaches; it's just that they tended to think of overly complicated approaches that were much harder to get working (e.g. using loops within loops, which is totally unnecessary for this part).

5.5 Part #5

File: `part5.py`

Write a program that prompts the user to enter a string and that prints a certain message based on which of the below conditions (if either) is true:

- The third character of the given string is "x".
- The string has less than three characters.

Below are some examples of how your program should behave. Be careful about copy/pasting parts of these to your code (if you do), as the fact that this document is a PDF means certain characters (such as single or double quotation marks) may get distorted, unfortunately.

```
Enter: a
The string "a" has less than three characters.
```

```
Enter: ab
The string "ab" has less than three characters.
```

```
Enter: abx
The third character of "abx" is 'x'.
```

```
Enter: axc
The string "axc" has at least three characters.
```

```
Enter: abxaby
The third character of "abxaby" is 'x'.
```

```
Enter: abcdef
The string "abcdef" has at least three characters.
```

For this example, you should not have to do anything special if your program already works; `input()` can properly handle two words being entered on the same line and does combine them into one string (in this case, `"Hi there"`).

```
Enter: Hi there
The string "Hi there" has at least three characters.
```

5.6 Part #6

File: `functions.py`

Write a function called `subtract_eight` that takes as argument an integer and returns the result of subtracting 8 from that integer.

You may assume that this function will always be passed an integer (e.g. not a string or a boolean value).

Below are some examples of how your program should behave. I ran these examples on the Python IDLE interpreter, which is why you see the `>>>` prompt.

```
>>> subtract_eight(17)
9
>>> subtract_eight(5)
-3
>>> subtract_eight(8)
0
>>> subtract_eight(0)
-8
```

[1] When I say "experienced", I mean "having some experience", not necessarily a ton of experience. If you have years of experience and are in this class (for whatever reason), then this part shouldn't be an issue.

5.7 Part #7

File: `functions.py`

(Note that for this part, you are adding a function to the same file that you wrote your function in for part #6.)

Write a function called `foo` that takes four arguments: a string, an index, a second string, and a second index. The function should return `True` if the character at the first index in the first string is the same as the character at the second index in the second string. Otherwise, the function should return `False`. If either index is out of bounds for their resepctive string, then the function should return `False`.

Below are some examples of how your program should behave. I ran these examples on the Python IDLE interpreter, which is why you see the `>>>` prompt.

```
>>> foo("abc", 1, "bob", 0)
True
>>> foo("abc", 1, "bob", 1)
False
>>> foo("abc", 1, "bob", 2)
True
>>> foo("abc", 2, "bob", 2)
False
>>> foo("abc", 4, "bob", 2)
False
>>> foo("abc", 1, "bob", 8)
False
>>> foo("Hello, there", 5, "Hi, how are you?", 2)
True
>>> foo("Hello, there", 1, "Hi, how are you?", 4)
False
>>> foo("Hello, there", 1, "Hi, how are you?", 10)
True
>>> foo("Hello, there", 1, "Hi, how are you?", 11)
False
```

5.8 Part #8

File: `functions.py`

(Note that for this part, you are adding a function to the same file that you wrote your functions in for parts #6 and #7.)

Write a function called `repeats_same` that takes a string as its only argument and returns `True` if any character in the string occurs twice in a row and `False` otherwise.

Hint: When iterating through the string (which you will have to do), you may find it helpful to use a variable to keep track of the last character that was seen, unless you are using a `range()`-based `for` loop.

Below are some examples of how your program should behave. I ran these examples on the Python IDLE interpreter, which is why you see the `>>>` prompt.

```
>>> repeats_same("abcd")
False
>>> repeats_same("abccd")
True
>>> repeats_same("abcccd")
True
>>> repeats_same("abcCcd")
False
>>> repeats_same("Hi there")
False
>>> repeats_same("So trueee")
True
>>> repeats_same("rsttxyyz")
True
```

6 Autograder Details

TBA.