CSC110 Tutorial 5: Loops and Mutation

Print this handout

In this tutorial, you'll get more practice using loops and working with mutable objects. You'll also revisit using the PyCharm debugger to help investigate and identify some common errors students make when writing code involving loops and mutation. And finally, we'll build on the "printing and drawing" exercise from last week's tutorial by showing how loops can be used to create small interactive programs.

Exercise 1: Practice with loops and mutation

First, download the starter file `tutorial5_part1.py`. Inside the file are three functions for you to implement. These functions will give you the opportunity to practice writing for loops that mutate an accumulator variable, and using nested loops!

After you finish implementing these functions, we encourage you to compare your three function bodies: What is similar about each one? What's different? Drawing comparisons and contrasts between them will help you identify common patterns that you'll use in your for loops going forward in CSC110.

Exercise 2: Debugging corner

This week's *Debugging corner* will be based on common errors you can encounter when writing all kinds of loops (using the accumulator pattern, early return pattern, nested loops), and code involving mutation.

Download the starter file `tutorial5_part2.py`. In the file, we've included a few functions that each have an error, and a doctest that fails because of the error. This exercise will get you to practice stepping through the function code to identify the error using the PyCharm debugger, like you did in Tutorial 4 last week.

1. First, run the file in the Python console, and then attach the PyCharm debugger (by clicking the green "bug" icon on the left side of the Python console pane).
2. In the `tutorial5_part2.py` file, scroll to the first function (`total_string_size`) and click on the margin to the left of the first assignment statement to set a breakpoint.
3. Then, copy the doctest example into the Python console and press Enter to start the PyCharm debugger.
4. Like last week's tutorial, use the "Step Over" button to step through the function code line by line, inspecting the values of the local variables at each step. Use this to identify the place where the error occurs.
5. Fix the error, then re-run the file in the Python console and call the doctest example again to verify that it now returns the correct value.
6. Repeat Steps 2-5 for each of the other functions in the file.

Exercise 3: Enabling user interactions

In Tutorial 4, you saw one application of loops that involved printing text in the Python console and drawing shapes in a Pygame window. In the final exercise of today's tutorial, you'll extend that work to use loops to create small interactive programs that wait for some kind of *user input* and then respond to that input.

> *Definition.* We define a **round** of interaction as follows:
>
> 1. The program calls a function that waits for some kind of user input.
>   - For example, we'll use the built-in `input` function to wait for the user to type in some text.
> 2. The program calls another function that responds to that input.
>   - For example, we'll use the built-in `print` function to display some text back to the user.
>
> Our interactive programs will use a for loop to repeat the above for a given number of rounds. (This isn't the only way we could set up these rounds, but it's the simplest using what we've covered in this course!)

Let's see this in action. To begin, download the `tutorial5_part3.py` and `tutorial5_pygame.py` starter files and open them in PyCharm.

1. Find the `input_demo` function and read through its documentation and code. Then, run the file in the Python console and call `input_demo(5)` to try it out.
2. After you're confident you understand `input_demo`, complete the two functions `yell_at_user` and `report_input_lengths`. You can use the same structure as `input_demo`, but will need to modify the code to perform the necessary tasks.

Pretty cool! Now, we'll see how to take the same principle of "interaction rounds" and apply it to a graphical context using Pygame.

3. Find the `input_pygame_mouse_demo` function and read through its documentation and code. Then, run the file in the Python console and call `input_pygame_mouse_demo(5)` to try it out.
4. After you're confident you understand ths function, complete the two functions `draw_circles` and `draw_increasingly_large_circles`. Once again, you can use the same structure as `input_pygame_mouse_demo`, but will need to modify the code to perform the necessary tasks.

   You will find it helpful to review the documentation for the `pygame.draw.circle` function! (Hint: you used the same function in Tutorial 4 last week.)

Additional exercises

1. *More debugging.* Each of the following functions has an error. Follow the same process as Exercise 2 to practice debugging each function to find the error.

```
def monkeys_in_barrel(strings: list[str]) -> int:
    """Return the number of time 'monkey' appears in the given strings.

    >>> monkeys_in_barrel(['monkey', 'David', 'monkey'])
    2
    """
    for item in strings:
        if item == 'monkey':
            monkey_count_so_far = monkey_count_so_far + 1

    return monkey_count_so_far
```

```
def total_vowels(strings: set[str]) -> int:
"""Return the total number of vowels found in the given strings.

>>> total_string_size({'Hello', 'a', 'David is cool'})
8
"""
for s in strings:
    vowel_count_so_far = 0
    for char in s:
        if char in 'aeiou':
            vowel_count_so_far += 1

return vowel_count_so_far
```

```
def square_list(numbers: list[int]) -> list[int]:
    """Return a new list of every element in numbers squared.

    >>> square_list([1, 2, 3])
    [1, 4, 9]
    """
    squares_so_far = []
    for number in numbers:
        list.extend(squares_so_far, number)
    return squares_so_far
```

```
# The annotation Any (case-sensitive) can be used to indicate "any type"
from typing import Any


def swap_values(d: dict, key1: Any, key2: Any) -> None:
    """Update d by swapping the values for key1 and key2.

    Preconditions:
        - key1 in d
        - key2 in d

    >>> food_prices = {'Apple': 2.25, 'Orange': 2.5}
    >>> swap_values(food_prices, 'Apple', 'Orange')
    >>> food_prices == {'Apple': 2.5, 'Orange': 2.25}
    True
    """
    d[key1] = d[key2]
    d[key2] = d[key1]
```

2. There are many ways of extending the simple user interaction functions you developed in Exercise 3. Here are some ideas for Pygame:

   - Explore drawing shapes other than circles (e.g., rectangles).
   - Explore using the `random` module to draw circles with random colours and radii.
   - Change the shape properties (e.g., colour) based on the coordinates of the click position. For example, make the circles near the top brighter, and the circles near the bottom darker.
   - Use an **accumulator** to keep track of past clicks, and then explore what you can do with that information!