CPS 109 - Lab 5

Agenda

Last lab was functions. This lab, let's talk about *recursive functions*.

I know some of you have encountered "classes" with testing. We will go over that next week.

Recursive Functions

What's a recursive function? Well, first let's talk about what it means to be recursive.

For something to be recursive, it has to reference itself in order to work.

Recursive Functions

In all seriousness, for something to be recursive, it has to call on itself in order to work.

Recursive Definitions

Let's look at an example from an assignment I was given last year:

(60 marks) Let w be a string of brackets { and }. Then w is called balanced if it satisfies the following recursive definition:

- w is the empty string, or
- w is of the form { x } where x is a balanced string, or
- w is of the form xy where x and y are balanced strings.

Recursive Functions

So what does this mean to you? Well, we can use this concept to write functions!

Recursive Functions

Why would this ever be useful? Well, there are some instances where recursion is the only way to solve a problem.

Can any of you think of a problem that may require recursive functionality?

Base Cases

First, however, we need to talk about base cases. These are super important for designing recursive functions, since they keep your program from recursing indefinity.

Base Cases

What is a base case?

Simply put: it’s the default behaviour you need to account for when writing a recursive function. Base cases are reserved for when the function’s input is very simple, like 1.

Recursive Functions

One application that you guys have already gone over is the Fibonacci sequence. Another one we can approach with recursion is factorials.

Recursive Functions

```
def factorial(num):
    if(num < 1):
        return 1
    else:
        new_num = num * factorial(num - 1)
        return new_num
```

A Word On Mutability

I’m sure that by now, you’ve heard one of us mention that lists are “mutable” and strings are “immutable”. What do these mean?

A Word On Mutability

Something is **mutable** if its contents can be changed (like a list).

Something is **immutable** if its contents cannot be changed (like a string or a tuple).

But what about str.replace()?

You'll notice that for string manipulation methods (.replace(), .lower(), etc.) the method returns a new string.

You need to overwrite your old string with the new string returned by the method.

Mutability Example 2

```python
# Additional Mutability Example

if __name__ == "__main__":

    teenage_mutable_ninja_turtles = ["Raphael", "Donatello", "Leonardo", "Michelangelo"] # Remember, lists are mutable - strings are not
    teenage_mutable_ninja_turtles.append("Splinter") # We can add new elements
    teenage_mutable_ninja_turtles.pop() # We can also remove elements (Shoo Splinter, you aren't a turtle!)

    for immutable_turtle in teenage_mutable_ninja_turtles:
        immutable_turtle = "B" + immutable_turtle[1:]
    print(teenage_mutable_ninja_turtles) # Notice how it doesn't change. This is due to how for loops work (we're modifying the
                                         # immutable_turtle variable, which is local to the for loop).

    for i in range(len(teenage_mutable_ninja_turtles)):
        teenage_mutable_ninja_turtles[i] = "B" + teenage_mutable_ninja_turtles[i][1:]
    print(teenage_mutable_ninja_turtles) # Notice how it does change, but we need an equals sign to do so (we are making a new string)

    #teenage_mutable_ninja_turtles[0][0] = "R" # We can't change strings like this!
    teenage_mutable_ninja_turtles[0] = "Baphael!" # But we can change list items to whole new strings

    teenage_mutable_ninja_turtles[0].replace("B", "R", 1)
    print(teenage_mutable_ninja_turtles[0]) # Notice how it doesn't change
    teenage_mutable_ninja_turtles[0] = teenage_mutable_ninja_turtles[0].replace("B", "R", 1)
    print(teenage_mutable_ninja_turtles[0]) # Notice how it does change, but we are setting it to the new string returned by the .replace() method
```

```
['Raphael', 'Donatello', 'Leonardo', 'Michelangelo']
['Baphael', 'Bonatello', 'Beonardo', 'Bichelangelo']
Baphael!
Raphael!
```