cs4_section4

February 23, 2019

What is recursion?

Answer: Recursion is a *problem solving strategy* in which you solve a big problem using subproblems until you cannot break the problem down any further. A fun example of this in real life is figuring out what row you're sitting in during lecture; instead of having to count how many rows there are in front of you, you can deduce that the row your sitting in is 1 + the row the person in front of you is sitting in. So you can solve this problem by asking the person in front of you, who will ask the person in front of them, and so on until you reach the person in the front row, who knows where they're sitting!

What does this look like in code?

Let's try out a different example. Below is a function called `powerOf2` which takes in an int n, where $n \geq 0$ and *recursively* computes 2^n. Note that we *could* just write `2**n` but then we'd never learn recursion!

```
def powerOfTwo( n ):
    # return the nth power of 2 for n >= 0
    # Base case
    if n == 0:
        return 1
    # Recursive Case
    return 2 * powerOfTwo( n - 1 )
```

There are 2 primary components of this code:

1. **The Recursive case:** `return 2 * powerOfTwo( n - 1 )` This step is the meat of the recursion. What it does is calls the same function, but on a smaller subset of the data. Without this line of code, we haven't really done anything recursive. Without making the problem smaller, our code will never finish execution.

2. **The Base case:** `if n == 0: return 1` Making a problem smaller is great! But you cant make it smaller forever and ever. At some point, you should reach a point where your problem cannot be broken down any further; this point is your base case. In your base case, you should be returning some constant value. Be careful here! Just because you know how to compute something doesn't mean it is the simplest a problem can be; n == 1 cannot be your base case because 0 is a valid input to your program!

What is map?

Answer: map is a higher order python function that takes in a function and a list and applies that function to every element of the list. It's super cool and very useful for simplifying code, and you're going to write your own implementation of it for hw04

You can play around with python's implementation to see how it works below:

```
In [ ]: # Lets try it out with our powerOfTwo code
        def powerOfTwo( n ):
            # return the nth power of 2 for n >= 0
            # Base case
            if n == 0:
                return 1
            # Recursive Case
            return 2 * powerOfTwo( n - 1 )

        test_list = [] #fill this in with a list of ints of your choosing

        # Note that map returns a map object! We need to cast that to a list
        awesome_result = list(map(powerOfTwo, test_list))

        # Feel free to try it out with a different function or list!
        print(awesome_result)
```

What is Abstract Testing?

Not every piece of code/functions is easily testable with assert statements. For instance, a function that approximates the square root of a number may use toy tests of perfect squares because you can't figure out what the output necessarily may be for more complex numbers. you can get lucky based on how complicated your toy test is i.e. for perfect squares your square root function might work great but for other numbers it might not work at all.

How do you test for these complex values? Sanity checks! Sanity checks basically tell you if your output makes sense- for instance, in the square root example, the square root of 40 should fall between 6 and 7. So you don't have to actually know the value of square root of 40. You can test if your output is between 6 and 7 and use that as a proxy to know if your value is correct.

This is very useful for functions that won't give the same output every time - random functions. Use the generate_text method: we can write a toy output that will give us the same thing every time- look at our solution to understand this.

We don't know what the output will be every time, but we know a couple of things about the text that will be generated- for every preceding word, the current word should be in its list in the dictionary. So what can we do? We can iterate through the generated text and check if the next word is in the current word's dictionary. If it is, we update a variable that keeps track of how many words are correct- the total number of correct words should equal the total number of words present in the passage.

Here's some pseudocde

```
def test_gentext(generated_text, dict):
    true_count=0
      for each index in generated_text:
          if word(index) is in dict[word(index-1)]
            true_count += 1
     assert(true_count == len(generated_text))
```

While this won't necessarily tell us if our method truly worked, it at least gives us some idea or a sanity check that our solution is plausible.

```
In [ ]:
```