Recursion

CMSC 201Section 60
October 23, 2024

Administrative Notes

Project 1 is out

A note about this week's lab - why don't you need to 'return' a dictionary from a function?

Recursion

When a function calls itself, that’s called “recursion” or “recursive programming.” And it turns out to be a very useful way to solve certain problems. Like, those where certain things have to be done over and over, with only slight differences.

Up until now, when we’ve had things that needed to be done repeatedly, we used loops. Either for loops, or while loops.

Using loops is called “iteration” or “iterative programming.”

Every programming problem that can be solved with recursion can also be solved with iteration!!

So why use recursion? Sometimes it just makes the problem easier to solve

So how does recursion work?

Somebody gives you a task to do.

If the task is easy enough, you just do the task yourself. *This is called the "base case"*

If the task is too hard to do directly, you offer to do a part of the task, then get somebody else to do the rest of it for you. *This is called the recursive case.*

This is only gonna work if you actually do part of the task yourself, and get other people to do a simpler version of the task. Otherwise we're in an infinite loop.

A software example: computing a factorial

Mathematically, n factorial (written as n!) for any positive integer n is just the product n * (n-1) * (n-2)*...* 1.

We know how to solve it iteratively. Now let's do it recursively

Iterative
```
def fact(n):
   prod = 1
   for i in range(n,1, -1):
      prod *= i
   return prod
```

Recursive
```
def fact(n):
    If n == 1:
        return 1
    else:
        return (n * fact(n-1))
```

Visualizing recursion

Let’s look visually at what’s happening when the recursive function is executing:

http://www.pythontutor.com/visualize.html#mode=edit

Another example - the Fibonacci sequence

1, 1, 2, 3, 5, 8, 13,...

After the first two numbers, each number is the sum of the previous two numbers

That is, f(n) = f(n-1) + f(n-2)

Iterative

```
def fib(n):
  If n<= 3:
      return n
  Else:
      fib = [1,1]
      for i in range(3,n+1)
         fib.append(fib[i-1] + fib[i-2])
      return (fib[n])
```

Recursive

```
def fib(n):
  if n < 3:
    return n
  Else:
    X = fib(n-1) + fib(n-2)
    return X
    return (fib(n-1) + fib(n-2))
```

Tracing the recursive routine

http://www.pythontutor.com/visualize.html#mode=display

How Python Works

Python, like many (but not all) programming languages, uses a stack of frames to keep track of where it is in the program

- What instruction gets executed next
- What variables are in scope and can be accessed
- Where to go when this function finishes

Stack - stack of plates. Put a plate on top (‘push’); take a plate off the top (‘pop’)
LIFO - last in, first out

This determines exactly how the program will work, and what statement will be executed after the current statement is executed

Only the frame on TOP of the stack can be executing!!

A stack is a data structure

A way of organizing data, with a defined set of rules about what can be done.

- Similar to a list or a dictionary, but with different operations

With a stack, you can only get the most recently added item from it.

- You can think of it more literally as a stack of papers where the most recently called functions information can be found on top.
- Here’s an illustration

https://www.cs.usfca.edu/~galles/visualization/SimpleStack.html

Frames and the Python Stack

A ***frame*** is the set of all symbols (variables, constants, function names) currently in scope - currently known to the Python interpreter

When the program starts, it pushes the main program’s frame onto the stack, and the main program executes.

When the main program calls a function, Python creates a new frame for that function and pushes that frame onto the stack

- Since the function’s frame is on top of the stack, that function is now executing

When a Function Ends

When a function ends, its frame is popped off the stack

- “Pop” in this sense means remove the top element from the stack
- All its parameters and local variables disappear
- Program Control returns to the frame that’s now on top of the stack - that is, whoever called that function!!

When does a function end?

- When a return statement is executed
- When there is no return statement, but all code in the function has been executed

When does it all end?

When the main program’s code has been executed, the main program’s frame is popped off the stack and that’s the end of it. There’s no place to return control, so the program is over

- This assumes there were no errors that ended your program prematurely

Back to Recursion - the Fibonacci sequence

1, 1, 2, 3, 5, 8, 13,...

After the first two numbers, each number is the sum of the previous two numbers

That is, f(n) = f(n-1) + f(n-2)

Iterative

```
def fib(n):
   If n<= 3:
        return n
   Else:
        fib = [1,1]
        for i in range(3,n+1)
            fib.append(fib[i-1] + fib[i-2])
        return (fib[n])
```

Recursive

```
def fib(n):
   if n < 3:
      return 1
   else:
      return (fib(n-1) + fib(n-2))
```

Tracing the recursive routine

http://www.pythontutor.com/visualize.html#mode=display

How do you solve a problem using recursion?

1. What is the base case? #there may be more than one
2. How do I describe a subproblem of my problem
   a. If I repeat making that subproblem, do I get a base case?
   b. At this point we should TRUST the recursive calls
3. Assuming I have the solution to the subproblem, ***HOW DO I SOLVE MY PROBLEM WITH IT?***
   a. I should be careful to make sure I'm returning the answer to MY PROBLEM

Palindromes

Calculate whether a string is a palindrome using recursion

What’s the base case?

- A string that is zero characters long - an empty string - IS a palindrome
- A string that is one character long IS a palindrome
- A string where the first character is DIFFERENT from the last character is NOT a palindrome
    - “cat” is NOT a palindrome
-

Palindromes - recursive case

What about the recursive case?

- IF the first character is the SAME as the last character, the string IS a palindrome if what's left when you throw away the first and last characters is a palindrome

yay - remove first character; remove last character; look at what's left

- a - IS a palindrome

Pseudocode

x=’yay’ IS a palindrome

First character equals last character

- Create the substring by throwing away the first and last character
    - x[1:-1] or (x[1:len(x)-1])
- We’re left with the string ‘a’

y = ‘tt’ IS a palindrome

Throw away first and last character

- we have an empty string left
- An empty string is a palindrome by definition (base case)

Now, let’s write this code

Battab - atta - tt - empty string - IS a palindrome

The rules of recursion: when a function calls itself

1. Any problem that can be solved by recursion can also be solved by iteration. The reverse is not true.
2. Recursion is always more expensive than iteration. But you use recursion because it’s easier for the programmer to solve the problem correctly
3. There must be at least one base case, which can be solved directly. There can be more than one base case, but there has to be at least one
4. There must be at least one recursive case. The recursive case must make the problem simpler; that is, it must be closer to a base case

Limits on recursion

Most Python implementations limit you to a certain number of recursions to ensure you don’t consume too many resources

- It’s kind of an “infinite loop” check

Now some more recursive examples

- Summing a list of numbers
- Summing the digits of a number
- Calculating x**y

129472309472390580923850