CSC 242 NOTES
Spring 2019-2020: Anthony Zoko
Week of Tuesday, May 12th, 2020

Announcements

- Assignment 7 is due next week
- Lab 7 is this week

Function Behavior Review

CallStack Revisit:

```
def a(x):
    print(x)
    ret = b(x+1)
    print(x)
    return ret+1

def b(x):
    print(x)
    ret = c(x+1)
    print(x)
    return ret+1

def c(x):
    print(x)
    return 1

print('Final output: ', a(10))
```

Draw the ***call stack*** and look at the behavior of the code at http://pythontutor.com/. Notice, that the first part of the values printed to the screen happens as we call the functions and the second half of the values printed to the screen happens as we are about to exit the functions. The final line contains the return value of the call to function a.

```
10
11
12
11
10
Final output:  3
```

Recursion

What is the definition of recursion? See recursion. :-)

A **recursive method** is a method that solves a problem by making one or more calls to itself.

Recursion is a particularly helpful tool for certain problems.

Some programming languages are even primarily recursive in nature (e.g. Lisp and Scheme), where loops are almost completely absent from programs written in them.

Any recursive method consists of:

- **One or more base cases**: these are the portions of the problem for which an immediate solution is known
- **One or more recursive calls**: to avoid infinite recursion these subproblems must be in some way smaller than the original problem

The best way to learn recursion is to practice, a LOT. So, we will see many examples during this part of the course.

Counting down

Consider the problem of printing the numbers from n to 1 for a positive value of n.

Input: A non-negative integer n
Output: The numbers from n down to 1, printed one per line

Sample runs:

```
>>> countdown(10)
10
9
8
7
6
5
4
3
2
1
Blast off!
>>> countdown(1)
1
Blast off!
>>> countdown(0)
Blast off!
```

```
>>> countdown(-3)
Blast off!
>>>
```

We will consider a **recursive algorithm** to solve the problem.

To do that we need to think recursively, meaning that we need to construct subproblems that when solved and combined can produce the solution to the original problem. We also need to find what the base cases are.

Let's first think about the **base case**. When would the problem be easy?

If the number passed into the method were less than or equal to 0, then we would just print blast off.

How can we use that to find a **recursive solution**?

- We could print n
- Then we could print n-1 down to 1 using a recursive call.

Write a Python implementation of this. See the file **count.py** for the solution.

Printing vertically

Consider the problem of printing an integer's digits vertically:

Input: A non-negative integer n
Output: The integer n, with its digits stacked vertically

Example: If the input is 1234, the output is:

1
2
3
4

We will write a **recursive algorithm** to solve the problem.

To do that we need to think recursively, meaning that we need to construct subproblems that when solved and combined can produce the solution to the original problem. We also need to find what the base cases are.

Let's first think about the **base case**. When would the problem be easy?

If we had just one digit to print, then we would print it on a line by itself and would be done.

How can we use that to find a **recursive solution**?

- We could print all but the last digit by making a recursive call. (Why the last digit and not the first one?)

- Then we could print the last digit. (Why not print the last digit before the recursive call?)

Write a Python implementation of this. See the file **vertical.py** for the solution.

Why and how does this solution work?

What happens when we call vertical on input n = 361?

Executing **vertical(361)**:

Since 361 is not less than 10, the call to vertical(36) is made.

Before this call gets executed the computer must store all the information necessary to complete the original call vertical(361) after vertical(36) is completed.

The necessary information includes the values of the variables and where execution should resume. It is stored in an **activation record**.

So before executing vertical(36), we store the activation record:

n = 361
return at step 6

Executing **vertical(36)**:

Since 36 is not less than 10, the call to vertical(3) is made.

Before doing this we store the activation record:

n = 36
return at step 6

Executing **vertical(3)**:

Since 3 is less than 10, we output 3 and return … where?

Back to line 6 in the execution of vertical(36).

Executing **vertical(36)**:

We output 6 and return to line 6 of the execution of vertical(361).

Executing **vertical(361)**:

We output 1 and stop.

Exercises

Problem 1: Consider a reversed version of vertical print.

Input: A non-negative integer n
Output: The integer n, with its digits stacked vertically, in reverse order.

Example: If n = 361, then the program should output:

1
6
3

Hint: How can we modify the recursive solution for the previous problem to get the solution to this problem?

Exercises

Problem 2: Write a recursive function printLst() that takes a list as a parameter and prints the items in the list, one per line.

The function cannot use any loops!

```
>>> printLst([1, 2, 3])
1
2
3
>>> printLst([1])
1
>>>
```

Problem 3: Write a recursive function cheers() that on an integer parameter n, will output n-1 strings “Hip” followed by “Hurrah”.

```
>>> cheer(5)
Hip
Hip
Hip
Hip
Hurrah
>>> cheer(3)
```

```
Hip
Hip
Hurrah
>>> cheer(1)
Hurrah
>>> cheer(2)
Hip
Hurrah
>>> cheer(0)
Hurrah
```

See the solutions in the file **exercises.py**.

The factorial function

For a given non-negative integer n, n! (read "n factorial") is defined as the product of all the positive integers up to n: n (n-1) (n-2) (n-3) … 1.

It's easy to write a loop that solves this. What about a recursive definition? How can we write n! in terms of a smaller factorial?

Exercise: Find a recursive definition and write it in Python.

How many calls to factorial are made for a parameter n?

One for n
One for n-1
One for n-2
…
One for 1 → A total of n calls

The exponent function

a to the power n is just a * a * a *… * a, i.e. a multiplied by itself n times.

This can be easily implemented iteratively using a loop:

```
def loop(a, n):
      ans = 1
      for i in range(n):
            ans = ans * a
      return ans
```

How many multiplications are done?

Using recursion and the addition rule for exponents we can reduce the number of multiplications.

What is **the addition rule for exponents**?

$a^{2n} = a^n * a^n = a^{n+n}$

$a^{2n+1} = a^n * a^n * a = a^{n + n + 1}$

See the solution in **exponent.py**.

How many multiplications are done? How does it compare with the iterative definition?

List printing, redux

What if we want to print a list that contains sublists? How can we modify the solution we developed to work in that case?

We need to add several bases cases to handle various cases for the first element.

What values can the first element be? Which of those values is relevant for making recursive calls?

Exercise: Modify the solution produced previously to work on multi-dimensional lists. It should work as follows:

```
>>> listPrint([1, 2, 3, 4])
1
2
3
4
>>> listPrint([[[1, [2], [3], [[[4]]]], [5], [[6, 7],
8]], 9])
1
2
3
4
5
6
7
8
9
>>> listPrint([])
>>>
```