LEARNING AI-DS-ML SKILLS

PYTHON PROGRAMMING

CHAPTER 3: RECURSION

BY FRED ANNEXSTEIN, PHD

PYTHON PROGRAMMING

3.
RECURSION

BY FRED ANNEXSTEIN, PHD

In this module we will be covering the basics of using recursion in computational problem solving.

CHAPTER 3 OBJECTIVES

By the end of this module, you will be able to...

- Understand what recursion is and how it is handled by the interpreter.
- Identify the base case and the recursive step in a recursive function.
- Correctly use recursion in problem solving.
- Understand the mathematical significance of classical recursive functions, such as factorial, harmonic numbers.
- Compare and contrast recursive and iterative solutions to problems.
- Identify the contexts in which recursive solution are efficient or inefficient.

RECURSION

A recursive function is one that can call itself before it returns. Evaluating a call to a recursive function is not essentially different from evaluating a non-recursive function. Since every function call statement sets up a new environment frame with local variables.

Recursion lends itself to an important problem-solving approach. This approach structures a solution with a base case and an inductive case. Base cases are typically trivial as they solve the simplest cases, usually the solution is direct without further processing. If you call the function satisfying a base case, it immediately returns with a result. If you call the function with the inductive case (more complex than a base case), the function will typically divide the problem into pieces. The pieces must be slightly simpler or a smaller version of the original problem. Since each problem piece resembles the original problem, the function calls a fresh copy of itself to work on the smaller problem piece(s)—this is called the recursion step. The recursion step executes while the original function call is still active

and not yet complete. This situation can result in many more recursive calls as the function divides each new subproblem until a base case is reached.

For proper execution the recursion must eventually terminate, thus converging on a base case. When the function recognizes the base case, it returns a result to the environment associated with the previous copy of the function. A sequence of returns ensues until the original function call returns the final result to the original caller - the main program. It is possible however to misuse recursion resulting in an infinite loop of recursive calls and thus never return.

Understanding and using recursion correctly is essential for any programmer. When applying recursion in problem solving, consider how a solution to a simpler version of the problem can be used to solve a more complex version. Remember to trust the recursion to work as designed: you can do this by assuming that your solution to the simpler problem works correctly without worrying about the details of how it accomplishes the task.

Begin your problem solution by thinking about what the answer would be in the simplest possible case(s). These will be your base cases - the stopping points for your recursive calls. Make sure to consider the possibility that you are missing base cases for this is a common way recursive solutions fail. You may also find it helpful to write the iterative version first before constructing a recursive solution.

COUNTING UP AND DOWN WITH RECURSION

Here are two simple recursive functions, one that counts up to a number given as an argument n, and the other counts down from n to 0. Try to see if you can understand the logic and see if you can argue that these are correct implementations.

```
def countup(n):
    if n == 0:
        print(0, end = ' '))
    else:
        countup(n-1)
        print(n, end = ' ')

def countdown(n):
```

```
    if n == 0: print(0, end = ' ')
    else:
        print(n, end = ' ')
        countdown(n-1)

>>> countup(10)
0 1 2 3 4 5 6 7 8 9 10

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

One can argue that these functions are correct implementations by using the following type of argument. Let us consider the countup() function. We know the base case is when n=0 and so our function is correct on that input. Now we assume that our function countup() works for all values up to k >=0. Let us show it is correct on input value k+1. This is called the inductive step. In this case the function will execute the else clause, and since we have assumed countup(k) is correct we know that it will print out the numbers 0, 1, up to k. Finally it prints the final value k+1, and terminates correctly. Try to construct a similar argument for the countdown function.

LIMIT ON RECURSION DEPTH

The depth of recursion is the number of times a recursive function calls itself. Python interpreters set a fixed limit on how many times one function can call itself before an exception is raised. This is used to prevent infinite recursion. We can see the current limit and test where our recursive function exceeds that depth, say using countup(1500). However, we can reset that limit using a module called sys. Typically the default limit on recursion is 1000 calls. In the following example we show how to double that limit to 2000.

```
>>> import sys
>>> sys.getrecursionlimit()
1000

>>> countup(1500)
RecursionError: maximum recursion depth exceeded
in comparison

>>> sys.setrecursionlimit(2000)
>>> sys.getrecursionlimit()
2000

>>> countup(1500)
. . .
1486 1487 1488 1489 1490 1491 1492 1493 1494
1495 1496 1497 1498 1499 1500
```

RECURSIVE FUNCTIONS FOR FACTORIAL AND FIBONACCI

You can arrive at a recursive factorial representation by observing that n factorial or n! can be written as:

```
n! = n · (n – 1)!
```

For example, 5! is equal to 5 · 4!, as in:

```
5! = 5 · (4 · 3 · 2 · 1)
5! = 5 · (4!)
```

We can define the function factorial recursively in python as follows:

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n – 1)
```

We know by its definition that 0! is 1. So we choose n = 0 as our base case. The recursive step also follows from the definition of factorial, i.e. n! = n * (n-1)!.

Let us turn now to the fibonacci function which is defined recursively as follows:

```
Fib(0) = Fib(1) = 1 ,  and
Fib(n) = Fib(n-1) + Fib(n-2), for all n>=2.
```

We can write this function is Python as follows.

```
def fib(n):
    if n >= 2: return fib(n-1) + fib(n-2)
    else:  return 1
```

USING RECURSIVE FUNCTIONS IN LIST COMPREHENSIONS

As we will see over the course, Python *list comprehensions* are a concise notation for creating list of values. List comprehensions use special for construction and can replace many multi-line for-loops that create new lists. More about lists will be covered in the next chapter. Here we construct two lists using the functions discussed above each within a list comprehension and display the results.

```
>>> one = [fac(i) for i in range(9)]
```

```
>>> two = [fib(i) for i in range(9)]
>>> one
[1, 1, 2, 6, 24, 120, 720, 5040, 40320]
>>> two
[1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Fibonacci is in a different class of recursive functions from Factorial. The fibonacci function exhibits a branching in their recursive calls, in this case, in form of binary tree of recursion calls. This type of tree recursion can be very costly in execution time. This is demonstrate in the `lucas()` function from the first lab assignment. As a challenge, try to determine the largest m for which `fib(m)` will terminate within few seconds.

ENVIRONMENTS TO EVALUATE RECURSIVE FUNCTIONS

Evaluating a call to a recursive function is not essentially different from evaluating a non-recursive function. Recall that Python implements function calls with a stack-frame containing information on local data. Each recursive function call gets its own stack-

frame on the function-call stack. When a given call completes, the system pops the function's stack frame from the stack and control returns to the caller, which is possibly another copy of the same function. So each recursive call extends the current environment of stack-frames adding any new local objects referenced.

RECURSIVE FUNCTION FOR DIGIT SUMS

Suppose that we want a function that will break a number into a collection of single digits, and then produce the sum of all the digits. To design such a function we can use two arithmetic operations to break up a given number into two. The mod operator %10 will produce the last or low order digit. Integer division by 10 (using //10) will produce the number that has all but the low-order digit, which is dropped.

The logic used to design the `sumdigits()` function will be to check the base case for when the number is just a single digit. If the number does not have a single digit, then we can recursively call the

function on the remaining digits and add the low-order digit to the result as follows:

```
def sumdigits(n):
    """
    >>>sumdigits(134)
    8
    >>> sumdigits(5679)
    27
    """
    if n < 10:  return n
    else:
        last = n % 10
        prefix = n //10
    return last + sumdigits(prefix)
```

THE HARMONIC SERIES

Consider the function that is the sum of the first n unit fractions.

H(n) = 1 + 1/2 + 1/3 + 1/4 + ... + 1/n

In mathematics this sum is called the nth harmonic number. As n grows large this series, called the harmonic series, is divergent — that is, the infinite harmonic series grows without bound. The name harmonic series derives from the concept of overtones in music.

The difference between the nth

harmonic number and the natural logarithm of n (denoted ln n) converges to a small constant called the Euler–Mascheroni constant = 0.57721…..

Here is recursive code to test this difference and watch it converge.

```
def harmonic(n):
    if n>1:  return 1/n + harmonic(n-1)
    else: return 1

import math
for n in range(1,1000):
    print(n, ':',  harmonic(n) - math.log(n))
```

CHAPTER 3 EXERCISES

1. Run the example above for the harmonic series. Change the number of iterations from 1000 to 5000 and run it again. What happens and why?

2. Fix the error above to find differences up to the term harmonic(5000) - log(5000).

3. After 5000 iterations how close are last two differences, that is what is the value of

[harm(5001) - log(5001)] -
[harm(5000) - log(5000)]. ?