Module 6 - Overview

Introduction

Module Learning Outcomes

After successful completion of this module, you will be able to do the following:

1. Write recursive functions using a few different approaches.

Key question:

* What are some of the different options you have when writing a recursive function?

Explorations

Use the pages within this module to explore the following concepts:

* Exploration: [More recursion](https://canvas.oregonstate.edu/courses/1915078/pages/exploration-more-recursion) (CLO 5, MLO 1)
* Video Demo: [More Recursion](https://canvas.oregonstate.edu/courses/1915078/pages/video-demo-more-recursion) (CLO 5, MLO 1)
* [Module 6 exercise solutions](https://canvas.oregonstate.edu/courses/1915078/pages/module-6-exercise-solutions)

Optional Resources

* [*Think Python* chapter 5 sections 8 - 10 Links to an external site.](http://greenteapress.com/thinkpython2/html/thinkpython2006.html#sec62)
* [*Think Python* chapter 5 exercisesLinks to an external site.](http://greenteapress.com/thinkpython2/html/thinkpython2006.html#sec68)

Task List

Complete the following assignments and other tasks:

* Read the Exploration pages and do the interactive exercises on those pages (CLO 5, MLO 1).
* Do [Assignment 6](https://canvas.oregonstate.edu/courses/1915078/assignments/9227006), which gives you more practice with writing recursive functions (CLO 5, MLO 1).
* Take [Quiz 6](https://canvas.oregonstate.edu/courses/1915078/quizzes/2859162) (CLO 5, MLO 1).

Banner Art: [PixabayLinks to an external site.](https://pixabay.com/" \t "_blank)

# Exploration: More recursion

## More recursion

Recall that in order to be recursive, a function must call itself at least once. A recursive function must also have one or more base cases. A base case tells the function when it can stop making recursive calls, so that it doesn’t just keep going until the computer runs out of memory. You might remember the recursive factorial function, which I use here to find the factorial of 5:

A screenshot of a computer

Description automatically generated with medium confidence

The if statement that checks whether num is equal to zero is the base case. If that condition is true, then we don’t make a recursive call. The function call “factorial(num-1)” is a recursive call, because the function is calling itself, on a smaller-sized problem. Remember that the conceptual key to writing a recursive function is figuring out how to break it down into smaller problems of the same type. The problems need to get smaller so that we eventually reach the base case and stop making recursive calls, so that the function can finish.

[Python TutorLinks to an external site.](https://pythontutor.com/) is a helpful site for visualizing what's happening when you run Python code. As you go through this exploration, try running the examples there also.

### **A few different ways to write a recursive function**

It’s often the case that a recursive function will need “extra” parameters, such as a counter to keep track of which subproblem the function is currently working on. For example, here’s a function that reverses a string:

A screenshot of a computer program

Description automatically generated with low confidence

The pos parameter keeps track of the current position in the string, which means that the substring being reversed at each recursive level is the substring from position pos to the end of the string. To reverse "wombat", it reverses "ombat" and concatenates "w" to it: "tabmo" + "w" = "tabmow". To reverse "ombat", it reverses "mbat" and concatenates "o" to it: "tabm" + "o" = "tabmo". At the base case, we've run out of characters, so we're reversing the empty string, so it just returns "". The concatenation of all the characters happens on the way back up out of the recursion. I've included a print statement inside the function to print the intermediate results, so you can see how the string is being built.

Calling the function would look like this:

print(reverse\_str("elephant", 0))

With this implementation, the user has to know that they need to pass zero as the second parameter of the recursive function, which isn’t really a reasonable expectation. Implementation details should be hidden from the user. That way, not only is it easier for the user, but that allows us to change the implementation later without breaking the user’s code. We can improve things by adding a helper function:

A screenshot of a computer program

Description automatically generated with low confidence

First notice that we’ve renamed the recursive function as rec\_reverse\_str. That’s so that we could name the new helper function reverse\_str, since this is the function users will call now. The helper function only takes one parameter, the string to be reversed. The user doesn’t have to pass a cryptic number zero. Instead the helper function takes care of that by calling the recursive function and passing both the string to be reversed and the number zero for the position counter. When the recursive function finishes and returns the reversed string to the helper function, the helper function then returns that same string.

Besides using a helper function, another way to make it so the user doesn't need to know about extra parameters is to assign default arguments to them, like this:

A screenshot of a computer program

Description automatically generated with low confidence

The user can still call the function with just the string, and the default argument will fill in the starting position of zero. The recursive calls supply both arguments, so the default isn't used for those.

Here's a different version that uses two default arguments:

A key difference between this version and the previous ones is that the previous ones dive down to the base case and then build the reversed string on the way back up out of the recursion, whereas this version uses its third parameter to build up the reversed string on the way down to the base case, and does no additional work on the way back up out of the recursion. This is because the reversed string is already complete when we arrive at the base case. I've again included a print statement to print the intermediate results, but this time the print statement happens before the recursive call instead of after. Compare the output to the output of the second interactive window and make sure you understand why they're different.

Here’s yet another possible implementation:

A screenshot of a computer program

Description automatically generated with low confidence

The previous versions pass the same entire string to each recursive call, keeping track of where in the string they are with the pos parameter. This version does something different. It uses a slice to make a substring that doesn’t have the first character, essentially chopping off the first character at each level of recursion. If we initially pass “elephant”, then the first recursive call will pass “lephant”, the second recursive call will pass “ephant”, etc. When we get down to the empty string, that is the base case that tells us we’re done. The reversed string is built up on the way back out of the recursion. This version has the advantage that no extra parameters are needed. However, it has the disadvantage that it uses more memory than the other versions. Each new substring that’s created takes up additional memory. It’s not really a problem for small input sizes, but you wouldn’t want to use this approach for large input sizes.

### **Recursive binary search function**

Let's look at a recursive implementation of the binary search algorithm. We'll pass the whole list with each recursive call to avoid using up memory by creating new sublists.

Here the recursive function needs extra parameters, but we can't use default arguments because the initial value of the last parameter needs to make use of the value passed for the a\_list parameter, and you can't do that with a default argument. So instead we use a helper function.

### **Recursive list merge**

Here's a recursive algorithm for merging two lists. It takes the smaller head element of the two lists to be the head element of the merged list, and makes a recursive call to merge the rest of the two lists.

A screenshot of a computer program

Description automatically generated with low confidence

**Recursive Fibonacci function**

The Fibonacci sequence is a well-known mathematical sequence where the first two numbers are both 1, and each successive number is the sum of the two preceding numbers: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, etc. Let's write a recursive function named fib that takes some integer n as its parameter and returns the nth Fibonacci number. Our first attempt might look like this:

A screenshot of a computer

Description automatically generated with medium confidence

Although short and simple, this implementation has the disadvantage that it's wildly inefficient. The reason is that it wastes time repeatedly solving the same problems. I inserted a print statement in the function above so you can see every time a recursive call is made. Notice how many duplicate calls there are.

A diagram of a fibonacci with Great Pyramid of Giza in the background

Description automatically generated with medium confidence

To compute fib(5), we end up computing fib(3) twice and fib(2) three times, as you can see in the above diagram. To compute fib(6), we would end up computing fib(4) twice, fib(3) three times, and fib(2) five times. The time complexity for this implementation is O(2n), which is terrible, especially since it's easy to write an iterative solution that is O(n). Occasionally someone will claim this shows that recursion is less efficient than iteration for problems like this. However, it's entirely possible to write a recursive solution that is O(n) - we just have to be a little more clever in how we go about it.

This implementation computes the nth Fibonacci number in O(n) time. The base case checks whether we've gone far enough in the sequence. The successive terms of the sequence are computed on the way down to the base case. When it reaches the base case, the answer has been found, which then just gets passed up as the return value of the recursive calls. Again I've inserted a print statement so you can see every time a recursive call is made. The difference is drastic.

What makes this implementation more efficient? By starting at the beginning of the sequence and passing the previous value along with the current value, it doesn't need to make a separate recursive call to compute that previous value. This avoids recomputing subproblems.

Another way to avoid recomputing subproblems is to use a technique called **memoization**, in which the function keeps a record of solutions to subproblems. If a subproblem has already been computed, it just looks it up in the record. Otherwise, it computes it and adds it to the record. The following example uses a dictionary, which is empty when the function is first called.



We give memo a default value of None to avoid the pitfalls of mutable default arguments (which you hopefully remember from CS 161). This implementation also computes the nth Fibonacci number in O(n) time.

### **Recursive maze solver**

Here's a recursive algorithm for finding the gold hidden a maze. Notice that it doesn't "look ahead" before making recursive calls - it lets the base cases handle any issues.

This function is an example of recursive backtracking. Whenever it finds that the current route it's exploring won't work, it rewinds back to the most recent decision point and tries the next possibility. It keeps going like this until it finds a route that works. Note that when there are multiple possible routes, this approach won't necessarily return the most efficient one.

## Exercises

Try these out on your computer using PyCharm:

1. Write a **recursive** function named summer that takes a list as its parameter and returns the sum of the values in the list. Have it do the addition on the way up out of the recursive calls.

Then test it using this unit test file in PyCharm: [more\_recursion\_1\_exercise\_tests.py](https://canvas.oregonstate.edu/courses/1915078/files/98542013?wrap=1)[Download more\_recursion\_1\_exercise\_tests.py](https://canvas.oregonstate.edu/courses/1915078/files/98542013/download?download_frd=1)

2. Same as #1, but this time have it do the addition on the way down into the recursive calls.

Then test it using this unit test file in PyCharm: [more\_recursion\_2\_exercise\_tests.py](https://canvas.oregonstate.edu/courses/1915078/files/98542004?wrap=1)[Download more\_recursion\_2\_exercise\_tests.py](https://canvas.oregonstate.edu/courses/1915078/files/98542004/download?download_frd=1)

3. Write a recursive function named bin\_convert that takes a non-negative integer as its parameter and returns a string giving the binary representation of that number.

Then test it using this unit test file in PyCharm: [more\_recursion\_3\_exercise\_tests.py](https://canvas.oregonstate.edu/courses/1915078/files/98541852?wrap=1)