2 Recursion

To understand recursion, one must first understand recursion.

Anonymous

An **iterative algorithm** solves problems by repeating steps over and over, typically using a loop. Most of the algorithms you've written in your programming journey so far are likely iterative algorithms. **Recursion** is a method of problem-solving where you solve smaller instances of the problem until you arrive at a solution. Recursive algorithms rely on functions that call themselves. Any problem you can solve with an iterative algorithm, you can also solve with a recursive one; however, sometimes, a recursive algorithm is a more elegant solution.

You write a recursive algorithm inside of a function or method that calls itself. The code inside the function changes the input and passes in a new, different input the next time the function calls itself. Because of this, the function must have a **base case**: a condition that ends a recursive algorithm to stop it from continuing forever. Each time the function calls itself, it moves closer to the base case. Eventually, the base case condition is satisfied, the problem is solved, and the function stops calling itself. An algorithm that follows these rules satisfies the three laws of recursion:

- A recursive algorithm must have a base case.
- A recursive algorithm must change its state and move toward the base case.
- A recursive algorithm must call itself recursively.

To help you understand how a recursive algorithm works, let's take a look at finding the factorial of a number using both a recursive and iterative algorithm. The **factorial** of a number is the product of all positive integers less than or equal to the number. For example, the factorial of 5 is $5 \times 4 \times 3 \times 2 \times 1$.

```
5! = 5 * 4 * 3 * 2 * 1
```

Here is an iterative algorithm that calculates the factorial of a number, `n`:

```
def factorial(n):
```

```
    the_product = 1
    while n > 0:
        the_product *= n
        n = n - 1
    return the_product
```

Your function, `factorial`, accepts the number, `n`, that you are using in your calculation.

```
def factorial(n):
```

Inside your function, you define a variable, `the_product`, and set it to 1. You use `the_product` to keep track of the product as you multiply `n` by the numbers preceding it, for example, 5 * 4 * 3 * 2 * 1.

Next, you use a `while` loop to iterate backward from `n` to 1 while keeping track of the product.

```
while n > 0:
    the_product *= n
    n = n - 1
```

At the end of your `while` loop, you return `the_product`, which contains the factorial of `n`.

```
return the_product
```

Here is how to write the same algorithm recursively:

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

First, you define a function called `factorial` that accepts the number, `n`, as a parameter. Next comes your base case. Your function will call itself repeatedly until `n` is 0, at which point it will return `1` and will stop calling itself.

```
if n == 0:
    return 1
```

Whenever the base case is not satisfied, this line of code executes:

```
return n * factorial(n - 1)
```

As you can see, your code calls the `factorial` function, which is itself. If this is your first time seeing a recursive algorithm, this probably looks strange to you, and it might even look like this code could not possibly work. But I promise you it does work. In this case, your `factorial` function calls itself and returns the result. However, it does not call itself with the value of `n`; rather, it calls it with the value of `n` – 1. Eventually, `n` will be less than 1, which will satisfy your base case:

```
if n == 0:
    return 1
```

That is all the code you have to write for your recursive algorithm, which is only four lines of code:

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

So, how does it work? Internally, each time your function hits a return statement, it puts it on a stack. A stack is a type of data structure you will learn more about in Part II. It is like a list in Python, but you remove items in the same order you added them. Let's say you call your recursive `factorial` function like this:

```
factorial(3)
```

Your variable, `n`, starts as 3. Your function tests your base case, which is `False`, so Python executes this line of code:

```
return n * factorial(n - 1)
```

Python does not know the result of `n * factorial(n – 1)` yet, so it puts it on the stack.

```
# Internal stack (do not try to run this code)

# n = 3
[return n * factorial( n - 1)]
```

Then, your function calls itself again after decrementing `n` by 1:

```
factorial(2)
```

Your function tests your base case again, which evaluates to `False`, so Python executes this line of code:

```
return n * factorial(n - 1)
```

Python does not know the result of `n * factorial(n – 1)` yet, so it puts it on the stack.

```
# Internal stack

# n = 3                          # n = 2
[return n * factorial( n - 1), return n * factorial( n - 1),]
```

Once again, your function calls itself after decrementing `n` by 1:

```
factorial(1)
```

Python does not know the result of `n * factorial(n – 1)` yet, so it puts it on the stack.

```
# Internal stack

# n = 3                          # n = 2                          # n = 1
[return n * factorial( n - 1), return n * factorial( n - 1), return n *
factorial( n - 1),]
```

Again, your function calls itself after decrementing `n` by 1, but this time `n` is 0, which means your base case is satisfied, so you return `1`.

```
if n == 0:
    return 1
```

Python puts the return value on the stack again, but this time it knows what it is returning: the number 1. Now, Python's internal stack looks like this:

```
# Internal stack

# n = 3                          # n = 2                          # n = 1
[return n * factorial( n - 1), return n * factorial( n - 1), return n * factorial(
n - 1), 1]
```

Because Python knows the last return result, it can now calculate the previous return result and remove the previous result from the stack. In other words, Python multiples `1 * n`, and `n` is 1.

```
1 * 1 = 1
```

Now, Python's internal stack looks like this:

```
# Internal stack

# n = 3                          # n = 2
[return n * factorial( n - 1), return n * factorial( n - 1), 1]
```

Once again, because Python knows the last return result, it can calculate the previous return result and remove the previous result from the stack.

```
2 * 1 = 2
```

Now, Python's internal stack looks like this:

```
# Internal stack

# n = 3
[return n * factorial( n - 1), 2]
```

Finally, because Python knows the last return result, it can calculate the previous return result, remove the previous result from the stack, and return the answer.

```
3 * 2 = 6

# Internal stack

[return 6]
```

As you can now see, calculating the factorial of a number is a perfect example of a problem you can solve by finding solutions to smaller instances of the same problem. By recognizing that and writing a recursive algorithm, you created an elegant solution to calculate a number's factorial.

When to Use Recursion

How often you want to use recursion in your algorithms is up to you. Any algorithm you can write recursively, you can also write iteratively. The main advantage of recursion is how elegant it is. As you saw earlier, your iterative solution to calculate factorials took six lines of code, whereas your recursive solution took only four. A disadvantage of recursive algorithms is that they often take up more memory because they have to hold data on Python's internal stack. Recursive functions can also be more difficult than iterative algorithms to read and debug because it can be harder to follow what is happening in a recursive algorithm.

Whether or not you use recursion to solve a problem depends on the specific situation, for example, how important memory usage is versus how much more elegant your recursive algorithm will be than a corresponding iterative algorithm. Later in the book, you will see more examples where recursion offers a more elegant solution than an iterative algorithm, like traversing a binary tree.

Vocabulary

iterative algorithm: An algorithm that solves problems by repeating steps over and over, typically using a loop.

recursion: A method of solving a problem where the solution depends on solutions to smaller instances of the same problem.

base case: A condition that ends a recursive algorithm to stop it from continuing forever.

factorial: The product of all positive integers less than or equal to a number.

Challenge

1. Print the numbers from 1 to 10 recursively.