# Recursion in Python

Guillaume Macneil

November 6, 2020

**Abstract**

This is the fifth lesson on the 'Introduction to Python' course. This course is loosely based around the 'MIT Introduction to Computer Science - Fall 2016' course. Our course is more focused towards the Python programming side of things.

## 1 Introduction to Recursion

There are two main ways to define recursion:

- ***Semantically***, recursion is a programming technique where a function calls itself. This is a confusing concept for many, "How can a piece of code tell the interpreter to run that very code, all while the code itself is running?" This is a good question, and can be explained in many ways - my favourite of which using acronyms.

  In the book 'Gödel, Escher, Bach: An Eternal Golden Braid', Douglas R. Hofstadter describes recursion using the acronym '**GOD**' (not to be confused with the deity). Two of the characters in the book encounter a being called a Djinn and one of them asks what '**GOD**' stands for. The Djinn replies that '**GOD**' stands for '**GOD** **o**f **D**jinn', which in turn stands for '**GOD** **o**f **D**jinn **o**f **D**jinn', which in turn stands for '**GOD** **o**f **D**jinn **o**f **D**jinn **o**f **D**jinn' and so on. Though the book discusses recursion in much greater detail, I feel that this describes recursion quite well, with '**GOD**' being a function and '**G**' being the code that recursively calls the '**GOD**' function.[1]

- ***Algorithmically***, recursion is a way to design solutions to problems by the *divide-and-conquer* technique - a way of reducing a problem into simpler versions of itself which can, in turn be solved either individually or by reducing the problem again.

[1]This section on the book 'GEB' has become a bit of a tangent, so I will finish it by saying that 'GEB' is a very good book to read for anyone interested in Computer Science, Mathematics or to some degree Psychology. I will warn you though it is a **substantial** book.

It is important to note that the '**GOD**' acronym example isn't entirely representative of how recursion works in general programming. The acronym example is an example of what is called *infinite recursion*, meaning that the function will continue to call itself forever, with no end in sight. In programming, the goal is to avoid this because otherwise the program would never end. So, in programming the recursive function must have one or more *base cases* - reduced versions of the problem that are easily solvable.

## 2 Recursion in Python

Let's start by considering multiplication. Multiplication is the act of adding '$a$' repeatedly to itself '$b$' times. There is a way a multiplication function can be written without recursion:

```
def multiply(a, b):
    output = 0
        while b > 0:
                output += a
                b -= 1
        return output

multiply(3, 5)
# Returns 15
```

Now there's nothing inherently wrong with this function, it's actually quite a neat solution (in my opinion), but there is a simpler way of doing it using recursion. Let's visualise what we are doing conceptually:

$$a \times b = a_1 + a_2 + a_3 + a_4 + ... + a_{b-3} + a_{b-2} + a_{b-1} + a_b$$

If we think about it closely we could vastly simplify this by using some multiplication. "But wait a minute! How can we use multiplication when we are defining the process of multiplication?" By using recursion. Look at what the equation becomes:

$$a \times b = a + a(b-1)$$

Now this seems much more manageable, so here is the multiplication function written recursively using Python:

```
def multiply(a, b):
        if b == 1:
                return a
        else:
                return a + multiply(a, b-1)

multiply(3, 5)
# Returns 15
```

Now this function is much less easy to understand than the last one. You can see that it takes *a* and *b* as arguments - the two numbers to multiply. You can also see that the first part of the if statement immediately returns the number *a* if *b* is equal to 1. This makes sense because any number multiplied by 1 is left unchanged. In the else block though, the function returns the number *a* added to a new instance of the function which takes *a* and *b-1* as arguments. If you look carefully, you can see that we have actually written out the simplified equation here. By this recursive process, you can also see that the function will keep calling itself until *b* is equal to 1. This is the *base case*. Once *b* is equal to 1, the function will just output the summation so far - which turns out to be the product of the two arguments.

Now after all that, you might be wondering: "Why would I bother doing that? The first, non-recursive function is so much easier." This is a valid question which is made even more valid when you consider that recursion isn't even always more efficient than just simple iteration. Recursion is, however, generally more intuitive than iteration - even if it may not seem so currently. It is important to remember that recursion is just a tool in your programming toolbox, just because you can use recursion for a problem doesn't always make it worthwhile.

Anyway, here is a *factorial* implementation as a final example of recursion. *Factorial* is a mathematical operation that multiplies an integer **n** by every integer before it, not including **0**. Here it is in mathematical notation:

$$n! = n \times (n-1) \times (n-2) \times (n-3) \times (n-4) \times ... \times 1$$

As you can see, this is a very similar operation to the multiplication example, here is how it could be written in python:

```python
def fact(n):
    if n == 1:
        return 1
    else:
        return n*fact(n-1)
```