

Recursion and Memory Visualization

In this section, we will learn how memory is allocated in recursive functions.

We'll cover the following



- Memory Allocation in Recursion
 - Memory Allocation of Recursive Functions
 - Calculating Factorial of a Number

Memory Allocation in Recursion

When a function is called, its memory is allocated on a **stack**. Stacks in computing architectures are the regions of memory where data is added or removed in a **last-in-first-out (LIFO)** process. Each program has a reserved region of memory referred to as its *stack*. When a function executes, it adds its **state** data to the **top** of the stack. When the function exits, this data is removed from the stack.

Suppose we have a program as follows:

```
def function1(<parameters>) :  
    <create some variables>  
    return(<some data>)  
  
def function2(<parameters>) :  
    <create some variables>  
    return(<some data>)  
  
# Driver Code  
function1()  
function2()
```

We have created some dummy example and called them in our program. Our memory stack will now look like this:

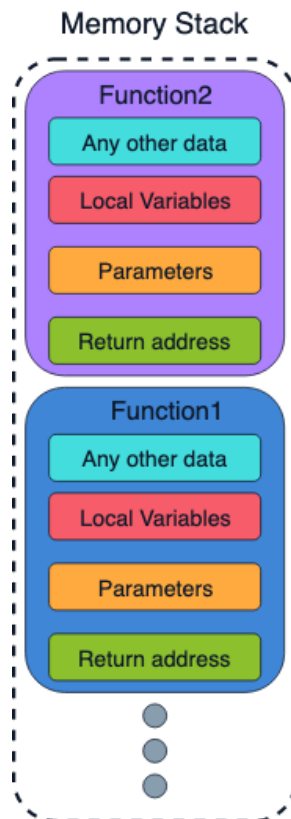


Illustration of memory stack

Memory Allocation of Recursive Functions

A recursive function calls itself, so the memory for a called function is allocated on top of the memory allocated for **calling the function**.

Remember, a different copy of local variables is created for each function call. When the base case is reached, the function returns its value to the function that it was called from, and its memory is de-allocated. This process continues until the parent function is returned.

Calculating Factorial of a Number

Let's take a look at a recursive function that calculates the factorial of a number.

A **factorial** is the product of an integer and all the positive integers less than it. A factorial is denoted by the symbol $!$

For example, $4!$ (read as **four factorial**) represents the following:

$$0! = 1$$

$$1! = 1$$

$$2! = 2 \times 1 = 2$$

$$3! = 3 \times 2 \times 1 = 6$$

$$4! = 4 \times 3 \times 2 \times 1 = 24$$

Therefore, if *targetNumber* = *n*

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


Here, the smallest starting value is 1!. Therefore, this is our base case.

Notice that some part of the previous task is being replicated in the current task. This is illustrated below:

$$1! = 1$$

1 of 4


$$2! = 2 \times \boxed{1} = 2$$

1! 

2 of 4




$$3! = 3 \times 2 \times 1 = 6$$

$2!$ 

3 of 4

$$4! = 4 \times 3 \times 2 \times 1 = 24$$

$3!$ 

4 of 4

— []

Let's take a look at the code:

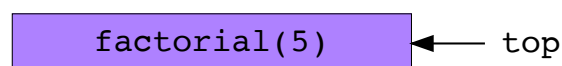
```
1 def factorial(targetNumber) :
2     # Base case
3     if targetNumber == 1 : # Factorial of 1 is 1
4         return 1
5
6     # Recursive case
7     else :
8         return (targetNumber * factorial(targetNumber - 1)) # Factorial of any other number is
9                                                                # number multiplied by factorial of numb
10
11 # Driver Code
12 targetNumber = 5
13 result = factorial(targetNumber)
14 print("The factorial of " + str(targetNumber) + " is: " + str(result))
```



Now, let's visualize the code stack of this code:

The first call to the function `factorial()` lies at the bottom of the stack.

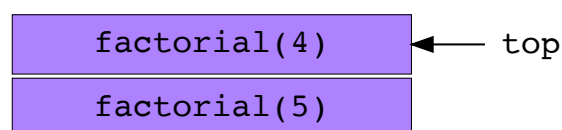
This is also the function that will return at the very end.



`factorial(5)` ← top

1 of 9

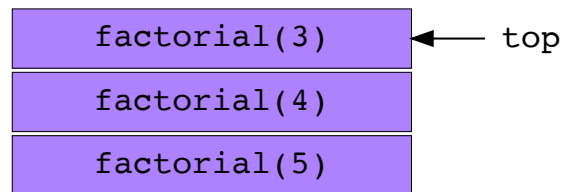
`factorial(5)` function calls another instance of the same function, but this time `target - 1` is passed to the function i.e., `factorial(4)`



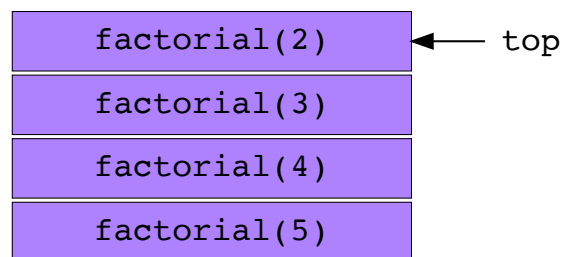
`factorial(4)` ← top
`factorial(5)`

2 of 9

Each time a function calls another instance of the function it is pushed at the top of the stack.



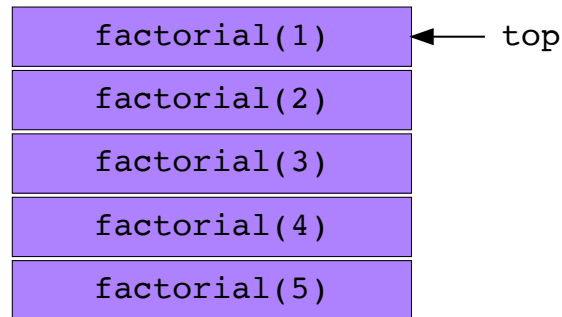
3 of 9



4 of 9

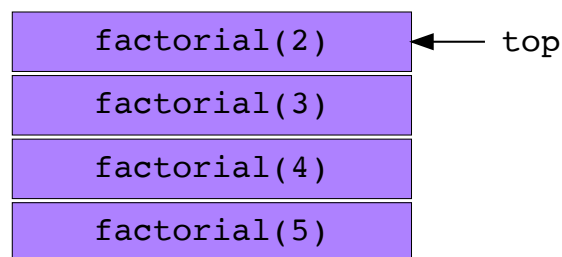


From here onwards, i.e., after factorial(1) is called there will be no further function calls. Instead, the base case is satisfied and the function returns 1.

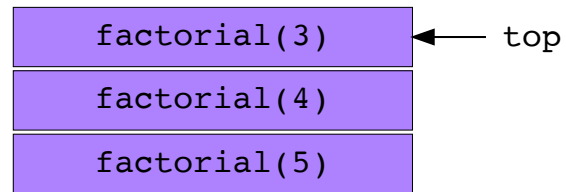


5 of 9

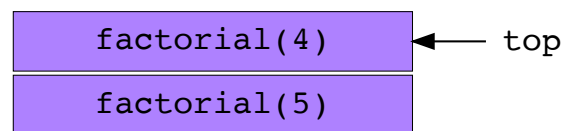
Each function will now return its respective value.



6 of 9



7 of 9



8 of 9

This is the beauty of recursion.

Each child function call returns result to its parent function.
In the end the result is accumulated and returned.



`factorial(5)` ← top

9 of 9

— []

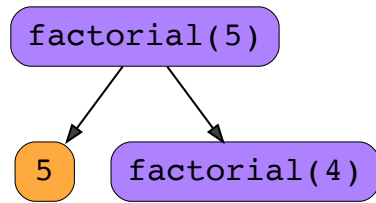
For the first 5 iterations call the `factorial()` function until it is added at the top of the stack, i.e. on top of the previous function call.

Each child function call returns the value to its parent call.

`factorial(5)`

Let's dry run.

1 of 10

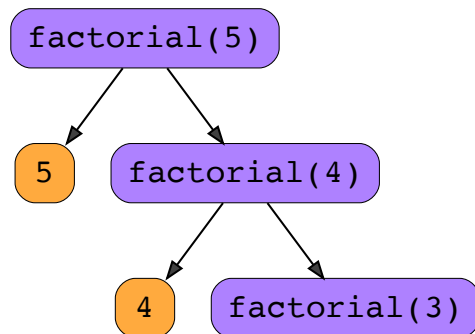


There are two parts to this function:
multiplication of the target with the
returned value of the child function.

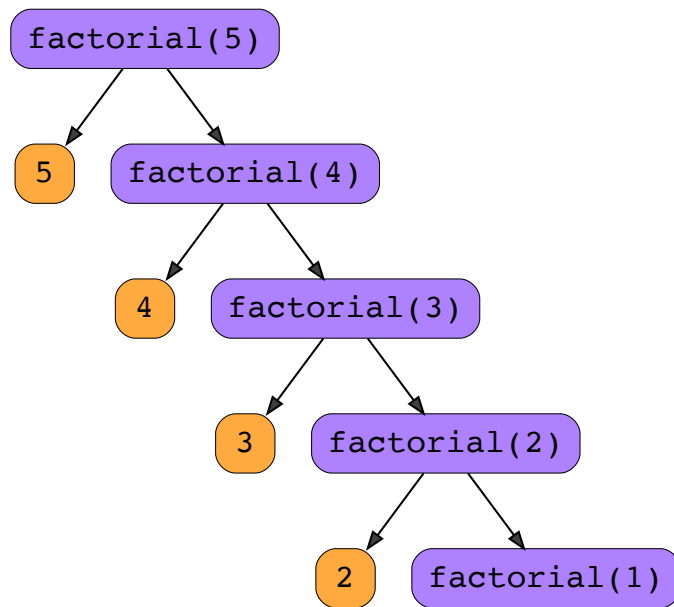
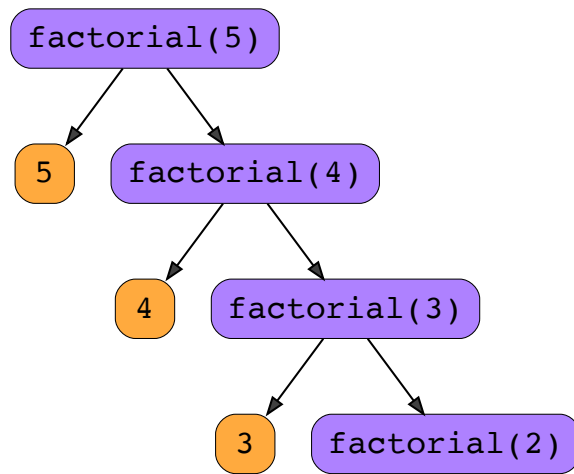
We cannot multiply unless we have the returned value of the child function.
That is why we move to the child function next.

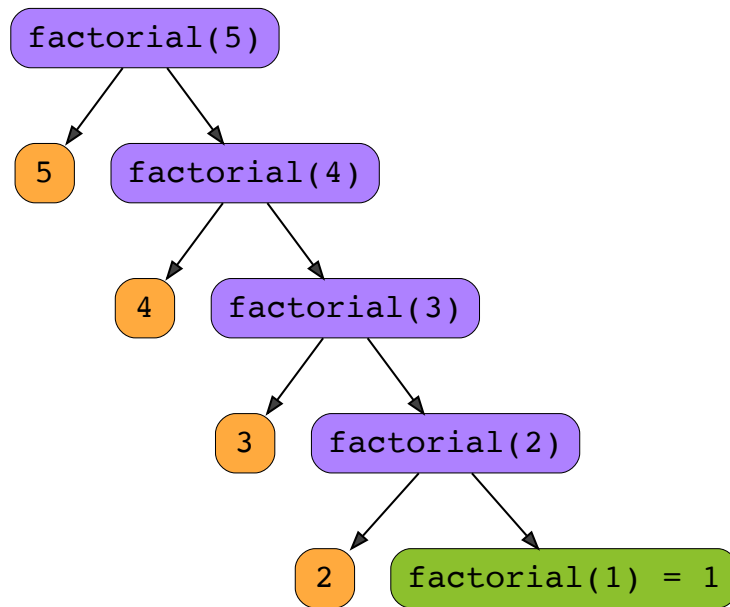
For the function factorial(5) the recursive case is executed.

2 of 10



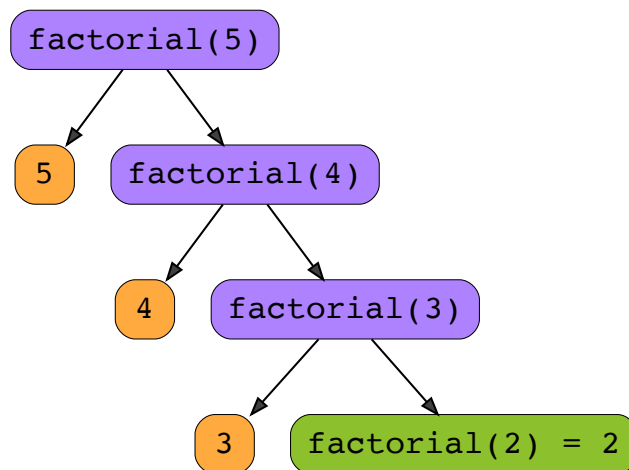
3 of 10





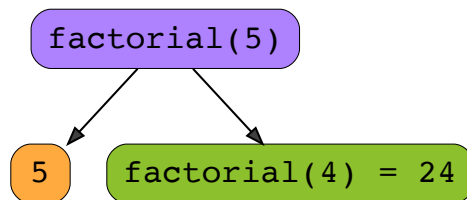
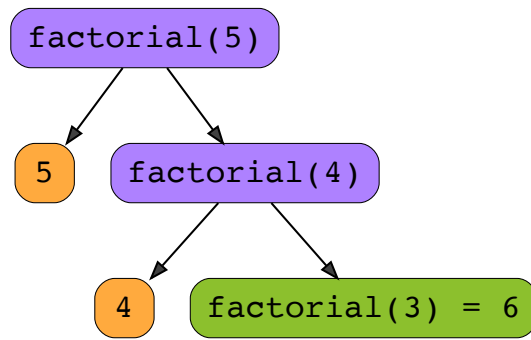
Here the base case is satisfied and the function `factorial(1)` returns 1.

6 of 10



Now, that we have the value of the child function, we can multiply it with current function's target value which is 3 in this case.

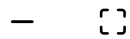
7 of 10





```
factorial(5) = 120
```

10 of 10



The above sequence represents the sequence of function calls.

In the next lesson, we will be learning two different types of recursion.

← Back

What is Recursion?

Next →

Direct Vs. Indirect Recursion

✓ Completed



Report an
Issue



Ask a Question

(https://discuss.educative.io/tag/recursion-and-memory-visualization__recursion-fundamentals__recursion-for-coding-interviews-in-python)