Call stack

May 24, 2020

1 Call stacks and recursion

In this notebook, we'll take a look at *call stacks*, which will provide an opportunity to apply some of the concepts we've learned about both stacks and recursion.

1.0.1 What is a *call stack*?

When we use functions in our code, the computer makes use of a data structure called a **call stack**. As the name suggests, a *call stack* is a type of stack—meaning that it is a *Last-In, First-Out* (LIFO) data structure.

So it's a type of stack—but a stack of *what*, exactly?

Essentially, a *call stack* is a stack of *frames* that are used for the *functions* that we are calling. When we call a function, say `print_integers(5)`, a *frame* is created in memory. All the variables local to the function are created in this memory frame. And as soon as this frame is created, it's pushed onto the call stack.

The frame that lies at the top of the call stack is executed first. And as soon as the function finishes executing, this frame is discarded from the *call stack*.

```
In [ ]: ### An example

        Let's consider the following function, which simply takes two integers and returns their
```

```
In [3]: def add(num_one, num_two):
            output = num_one + num_two
            return output
```

```
In [5]: result = add(5, 7)
        print(result)
```

```
12
```

Before understanding what happens when a function is executed, it is important to remind ourselves that whenever an expression such as `product = 5 * 7` is evaluated, the right hand side of the = sign is evaluted first. When the right-hand side is completely evaluated, the result is stored in the variable name mentioned in the left-hand side.

When Python executes line 1 in the previous cell (`result = add(5, 7)`), the following things happen in memory:

- A frame is created for the `add` function. This frame is then pushed onto the *call stack*. We do not have to worry about this because Python takes care of this for us.
- Next, the parameters `num_one` and `num_two` get the values `5` and `7`, respectively

If we run this code in Python tutor website http://pythontutor.com/ , we can get a nice visualization of what's happening "behind the scenes" in memory:

- Python then moves on to the first line of the function. The first line of the function is

  ```
  output = num_one + num_two
  ```

  Here an expression is being evaluated and the result is stored in a new variable. The expression here is sum of two numbers the result of which is stored in the variable `output`. We know that whenever an expression is evaluated, the right-hand side of the `= sign` is evaluated first. So, the numbers `5 and 7` will be added first.
- Once the right-hand side is completely evaluated, then the assignment operation happens i.e. now the result of `5 + 7` will be stored in the variable `output`.
- In the next line, we are returning this value.

  ```
  return output
  ```

  Python acknowledged this return statement.
- Now the last line of the function has been executed. Therefore, this function can now be discarded from the stack frame. Also, the right-hand side of the expression `result = add(5, 7)` has finished evaluation. Now, the result of this evaluation will be stored in the variable `result`.

Now the next question is how does this behave like a stack? The answer is pretty simple. We know that a stack is a Last-In First-Out (LIFO) structure, meaning the latest element inserted in the stack is the first to be removed.

You can play more with such "behind-the-scenes" of code execution on the Python tutor website: http://pythontutor.com/

1.0.2 Another example

Here's another example. Let's say we have a function `add()` which adds two integers and then prints a custom message for us using the `custom_print()` function.

```
In [6]: def add(num_one, num_two):
            output = num_one + num_two
            custom_print(output, num_one, num_two)
            return output

        def custom_print(output, num_one, num_two):
            print("The sum of {} and {} is: {}".format(num_one, num_two, output))

        result = add(5, 7)
```

What happens "behind-the-scenes" when `add()` is called, as in `result = add(5, 7)`?

Feel free to play with this on the Python tutor website. Here are a few points which might help aid the understanding.

- We know that when add function is called using `result = add(5, 7)`, a frame is created in the memory for the `add()` function. This frame is then pushed onto the call stack.
- Next, the two numbers are added and their result is stored in the variable `output`.
- On the next line we have a new function call - `custom_print(output, num_one, num_two)`. It's obvious that a new frame should be created for this function call as well. You must have realized that this new frame is now pushed into the call stack.
- We also know that the function which is at the top of the call stack is the one which Python executes. So, our `custom_print(output, num_one, num_two)` will now be executed.
- Python executes this function and as soon as it is finished with execution, the frame for `custom_print(output, num_one, num_two)` is discarded. If you recall, this is the LIFO behavior that we have discussed while studying stacks.
- Now, again the frame for `add()` function is at the top. Python resumes operation just after the line where it had left and returns the `output`.

1.0.3 Call Stack and Recursion

Problem Statement Consider the following problem:

Given a positive integer `n`, write a function, `print_integers`, that uses recursion to print all numbers from `n` to `1`.

For example, if `n` is `4`, the function shuld print `4 3 2 1`.

If we use iteration, the solution to the problem is simple. We can simply start at `4` and use a loop to print all numbers till `1`. However, instead of using an interative approach, our goal is to solve this problem using recursion.

```
In [10]: def print_integers(n):
             # TODO: Complete the function so that it uses recursion to print all integers from
             pass
```

Show Solution

```
In [11]: print_integers(5)
```

Now let's consider what happens in the call stack when `print_integers(5)` is called.

- As expected, a frame will be created for the `print_integers()` function and pushed onto the call stack.
- Next, the parameter `n` gets the value `5`.
- Following this, the function starts executing. The base condition is checked. For `n = 5`, the base case is `False`, so we move forward and print the value of `n`.

- In the next line, `print_integers()` is called again. This time it is called with the argument `n - 1`. The value of `n` in the current frame is `5`. So this new function call takes place with value `4`. Again, a new frame is created. **Note that for every new call a new frame will be created.** This frame is pushed onto the top of the stack.
- Python now starts executing this frame. Again the base case is checked. It's `False` for `n = 4`. Following this, the `n` is printed and then `print_integers()` is called with argument `n - 1 = 3`.
- The process keeps on like this until we hit the base case. When `n <= 0`, we return from the frame without calling the function `print_integers()` again. Because we have returned from the function call, the frame is discarded from the call stack and the next frame resumes execution right after the line where we left off.

In [ ]: