12.Return_of_the_Function

January 20, 2019

1 Return of the Functions

1.1 Some examples before start

We have used several functions (and methods which are essentially functions) previously, let's look at them

```
In [3]: # Assign result of function calling to variable
        result = list((1, 2, 3))

In [5]: # Look what it is
        result

Out[5]: [1, 2, 3]

In [10]: # Assign result of function calling to variable
         new_result = result.count(2)

In [11]: # Look what it is
         new_result

Out[11]: 1

In [6]: # Assign result of function calling to variable
        another_result = result.append(4)

In [7]: # Look what it is
        another_result

In [9]: result

Out[9]: [1, 2, 3, 4]
```

`append()` method doesn't return something, though do some work. Let's investigate the reason of this behaviour

1.2 Function schema

1 type of functions is procedure - a function which doesn't return something

```
def function_name(arguments):
    body...
```

Our previous functions were so - they can output something, but they didn't give result of operation

```
In [12]: def greet(name):
             print('Hi there', name)

In [13]: a = greet('Sasha')

Hi there Sasha


In [15]: a, type(a)

Out[15]: (None, NoneType)

In [16]: def sum_of_2(a, b):
             print(a + b)

In [17]: a = sum_of_2(3, 4)

7


In [18]: a
```

1.3 Do we need functions which can return us result (not just print it)?

Yes, cause it is very convenient. To achieve it we use `return` keyword

```
In [19]: def sum_of_2(a, b):
             return a + b

In [21]: summa = sum_of_2(3, 4)
         print(summa)

7


In [23]: # Main profit is in usage result of functions afterwards
         summa * 3

Out[23]: 21
```

1.3.1 Practice

Create a function which returns a product of 3 numeric arguments

```
In [24]: def product_of_3(a, b, c):
             return a * b * c

In [25]: product_of_3(3, 4, 10)

Out[25]: 120
```

1.4 Nested function

Yes, they can be nested too

```
def my_function(args):
    body...;
    def helper(args):
        body2...;
    body...
```

```
In [31]: # Not really necessary in this case
         def declare_some_primes():
             primes = [1, 2, 3, 5, 7, 11, 13, 17, 19]
             def speak_up(prime):
                 print(f'I\'m a prime {prime}')
             for prime in primes:
                 speak_up(prime)

In [34]: declare_some_primes()
```

```
I'm a prime 1
I'm a prime 2
I'm a prime 3
I'm a prime 5
I'm a prime 7
I'm a prime 11
I'm a prime 13
I'm a prime 17
I'm a prime 19
```

```
In [35]: speak_up()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)

<ipython-input-35-cb665b8330f8> in <module>()
----> 1 speak_up()


NameError: name 'speak_up' is not defined
```

1.5 Scopes

Here comes the scope concept - places where python looks for requested variables

- local - variables dwell in the function definition block
- global - variables dwell in the 1st level of script (not in defined functions)

1.6 LEGB rule

Variabes are searched in this order (bottom-up) * **L**ocal - variables on the same level of operation * **E**mbedding - variables 1 level higher function definition * **G**lobal - variables at the top level of python script * **B**uilt-in - built-in names which are present in every python script

```
In [52]: # Variable at the global level
         # I.e. global level is local for this variable
         n = 10

         # print is a built-in
         print(n)

         def declare_some_primes():
             # Variable at local level
             n = [1, 2, 3, 5, 7, 11, 13, 17, 19]
             print(n)
```

```
        def speak_up(prime):
            # Here n is an embedding variable
            print(n)
            print(f'I\'m a prime {prime}')
            # If we assign n to some value here it would be a local for this function

        for prime in n:
            speak_up(prime)
```

```
10
```

```
In [53]: declare_some_primes()
```

```
[1, 2, 3, 5, 7, 11, 13, 17, 19]
[1, 2, 3, 5, 7, 11, 13, 17, 19]
I'm a prime 1
[1, 2, 3, 5, 7, 11, 13, 17, 19]
I'm a prime 2
[1, 2, 3, 5, 7, 11, 13, 17, 19]
I'm a prime 3
[1, 2, 3, 5, 7, 11, 13, 17, 19]
I'm a prime 5
[1, 2, 3, 5, 7, 11, 13, 17, 19]
I'm a prime 7
[1, 2, 3, 5, 7, 11, 13, 17, 19]
I'm a prime 11
[1, 2, 3, 5, 7, 11, 13, 17, 19]
I'm a prime 13
[1, 2, 3, 5, 7, 11, 13, 17, 19]
I'm a prime 17
[1, 2, 3, 5, 7, 11, 13, 17, 19]
I'm a prime 19
```

By default variables are local to the level where they were assigned. It can be overriden by `global` and `local` keywords

1.7 Interesting function techniques

Let's look at recursion

Recurrent definition is a definition which use name of something to explain it, i.e. there is the same part in left and right parts

Hofstadter's Law: It always takes longer than you expect, even when you take into account *Hofstadter's Law*

Function use itself to make something `python def function(): bla-bla function() bla-bla-bla`


Do Not Use, Copy, or Redistribute

1.7.1 Factorial

$n! = n \cdot (n-1)$
$0! = 1$

```
In [62]: def factorial(n):
             if n == 0:
                 return 1
             return n * factorial(n - 1)

In [65]: factorial(5)

Out[65]: 120
```

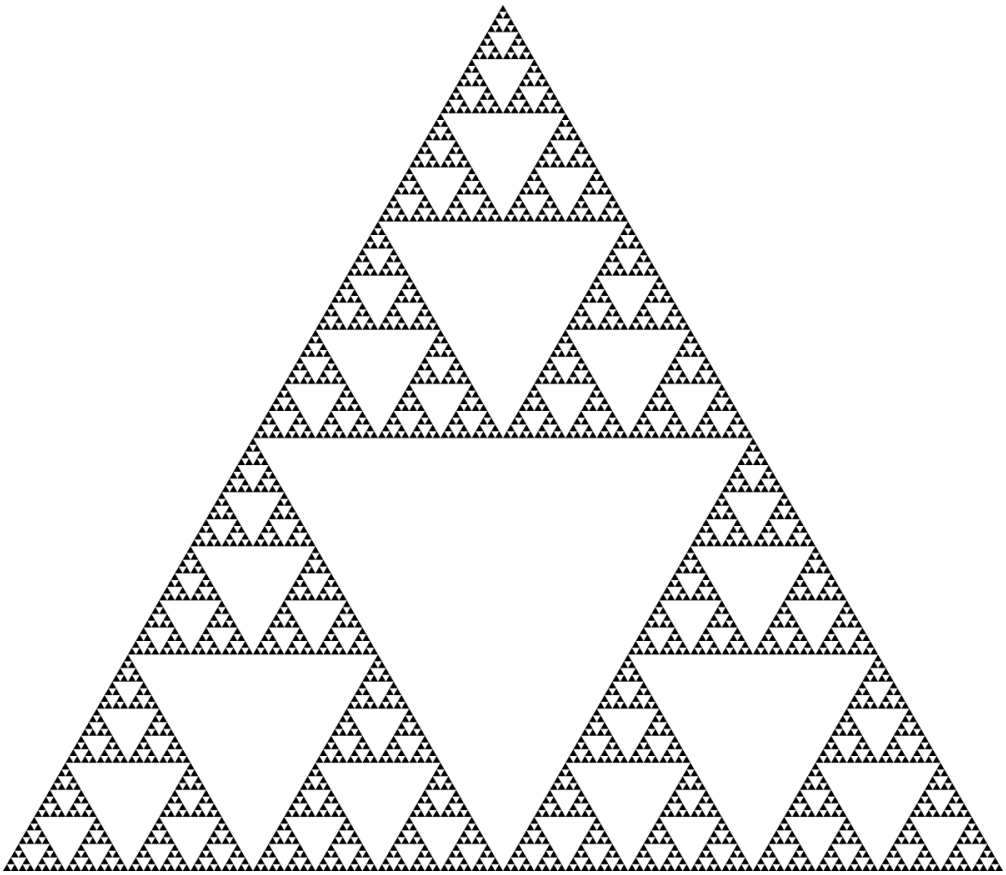
Recursion is good for work with objects which have same schema. It is an intuitive way of solving some problems.

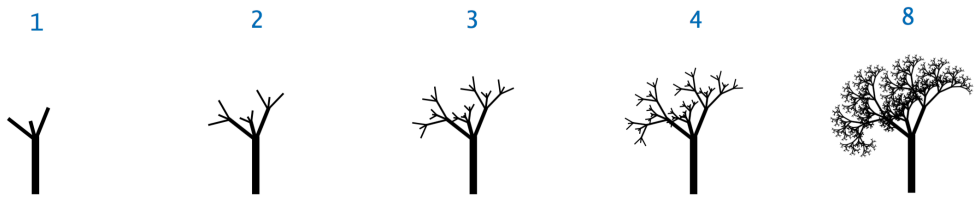
Though recursion have drawbacks - requires place at stack for each call, there is no automatic recursion call optimization (tail recursion) in python (but it is present in some languages)

1.7.2 Practice

How could you rewrite factorial function to avoid recursion?

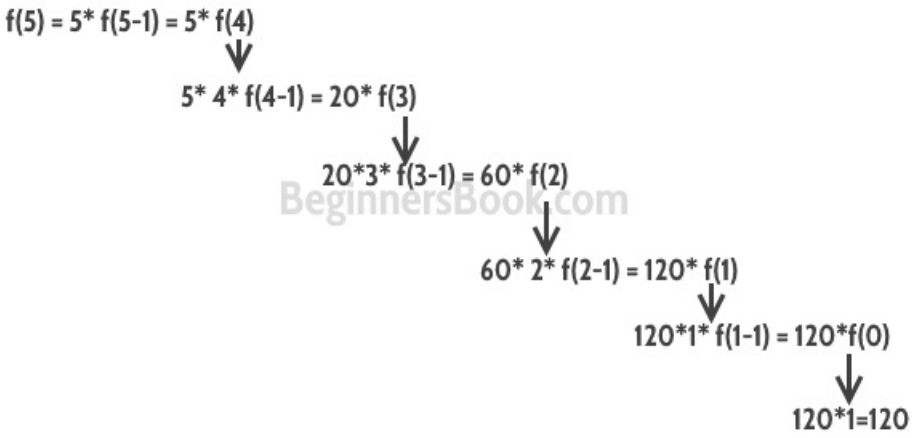
```
In [78]: def factorial(n):
             result = 1
```




1
2
3
4
8

Factorial function: f(n) = n*f(n-1)

Lets say we want to find out the factorial of 5 which means n =5




```
    for i in range(1, n + 1):
        result *= i
    return result
```

```
In [79]: factorial(5)
```

```
Out[79]: 120
```