Python Functions

User-Defined Functions

Built-in functions

Built-in functions are those that are already defined in the Python library

User-defined functions

A function that you define yourself in a program is known as user defined function.

You can give any name to a user defined function.

QUICK TIP

You cannot use the Python keywords as function name.

The def keyword

In python, we define the user-defined function using def keyword, followed by the function name.

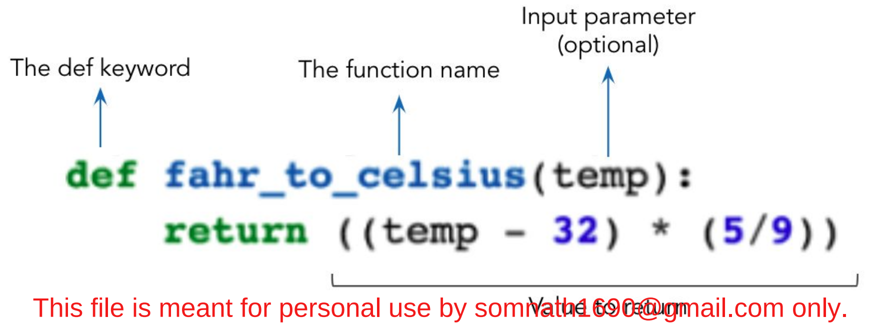
Defining the user-defined function

Step 1: Declare the function with the keyword def followed by the function name

Step 2: Write the arguments inside the opening and closing parentheses of the function, and end the declaration with a colon

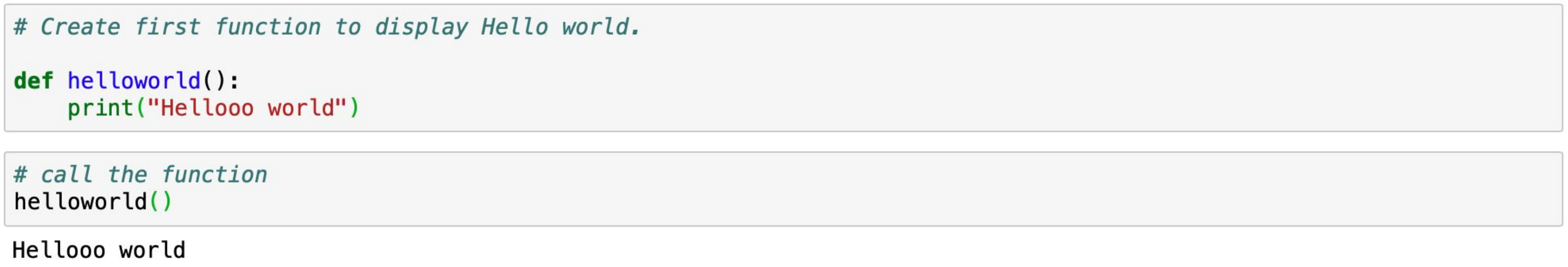
Step 3: Add the program statements to be executed

Step 4: End the function with/without return statement



```
def fahr_to_celsius(temp):
    return ((temp - 32) * (5/9))
```

Write your first function using the def keyword

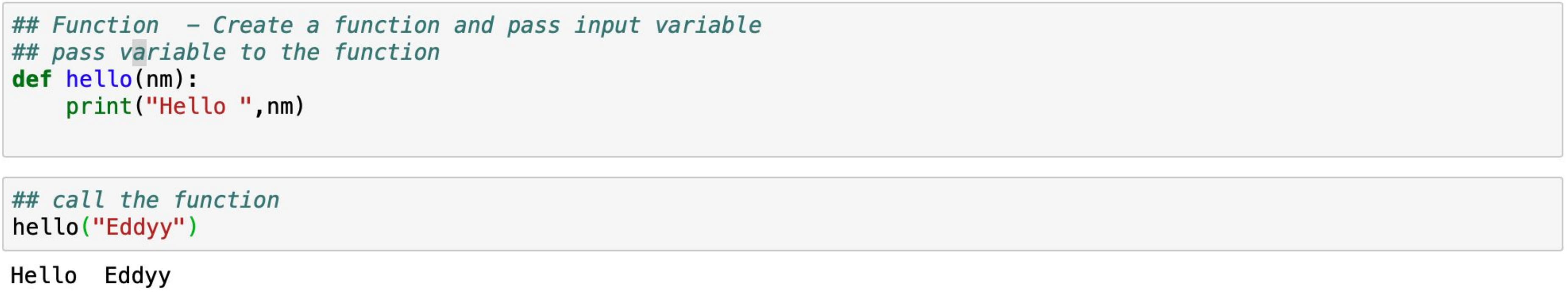


```
# Create first function to display Hello world.

def helloworld():
    print("Hellooo world")
```

```
# call the function
helloworld()
```

```
Hellooo world
```

Write a function with an argument



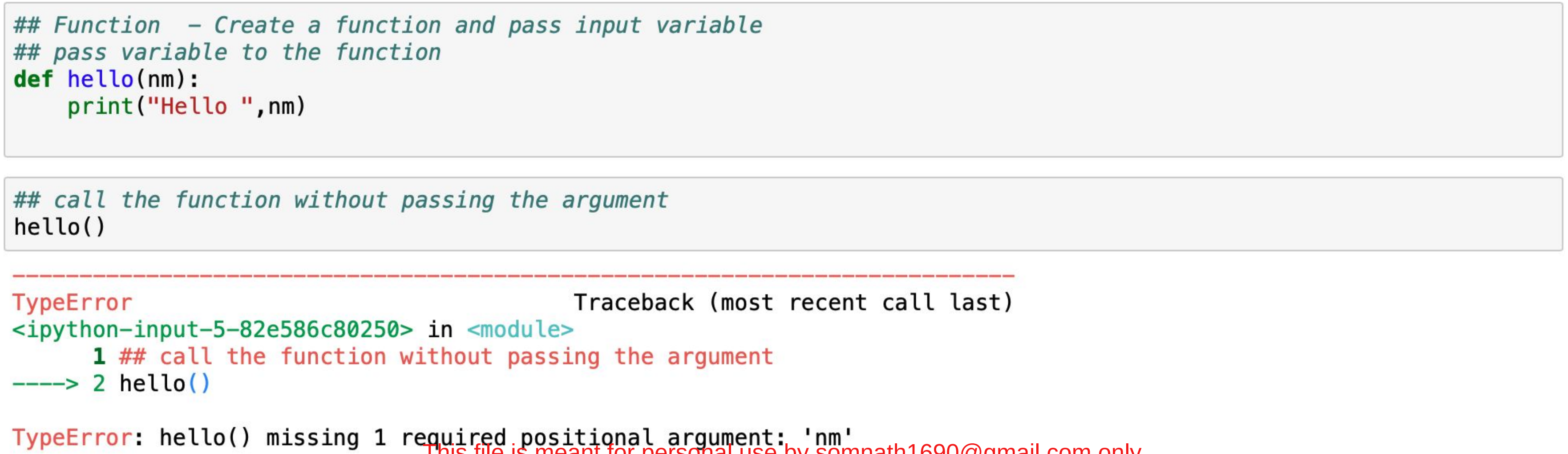
```
## Function  - Create a function and pass input variable
## pass variable to the function
def hello(nm):
    print("Hello ",nm)
```

```
## call the function
hello("Eddyy")
```

```
Hello  Eddyy
```

Calling the function without passing argument

When we declare a function that expects input argument, we should pass the required value. If we do not pass the required value, the function will throw an error.



```
## Function  - Create a function and pass input variable
## pass variable to the function
def hello(nm):
    print("Hello ",nm)
```

```
## call the function without passing the argument
hello()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-5-82e586c80250> in <module>
      1 ## call the function without passing the argument
----> 2 hello()

TypeError: hello() missing 1 required positional argument: 'nm'
```

A function without any return value

Note: We use the return keyword in the function but we do not mention what to return. Hence the function returns no value

```
# Functuion without return value
def empty_return(x,y):
    c = x + y
    return
```

```
result = empty_return(4,5)
print(result)
```

```
None
```

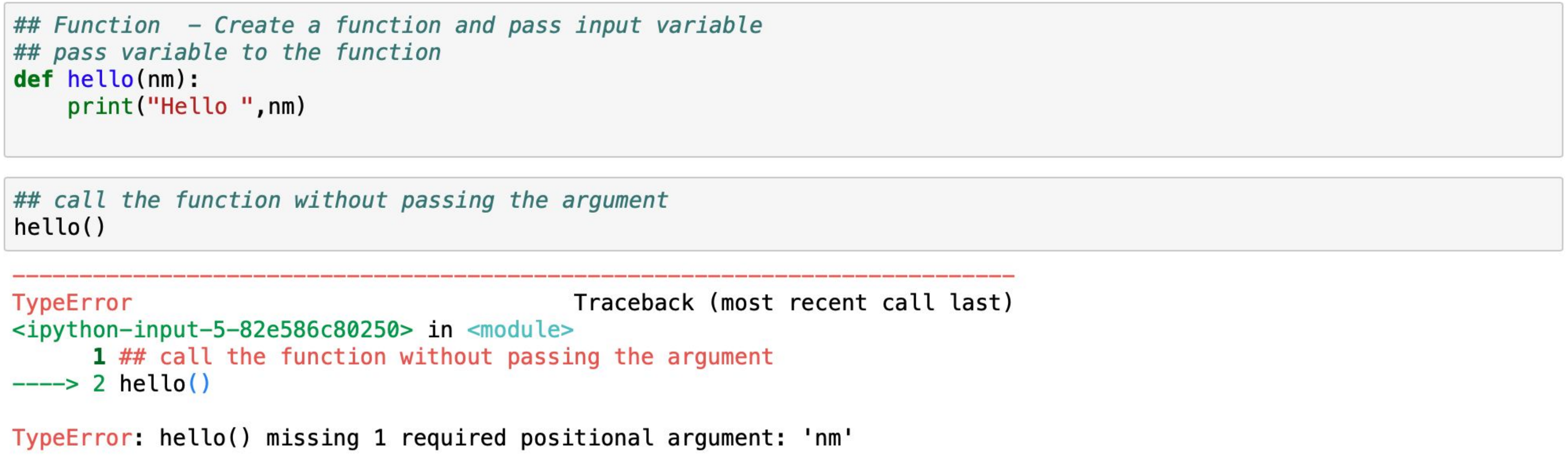
Function Arguments

Types of function arguments

1. Required Arguments
2. Keyword Arguments
3. Default Arguments
4. Variable-Length Arguments

Required Arguments

In this case, if the argument is not passed it will throw an error



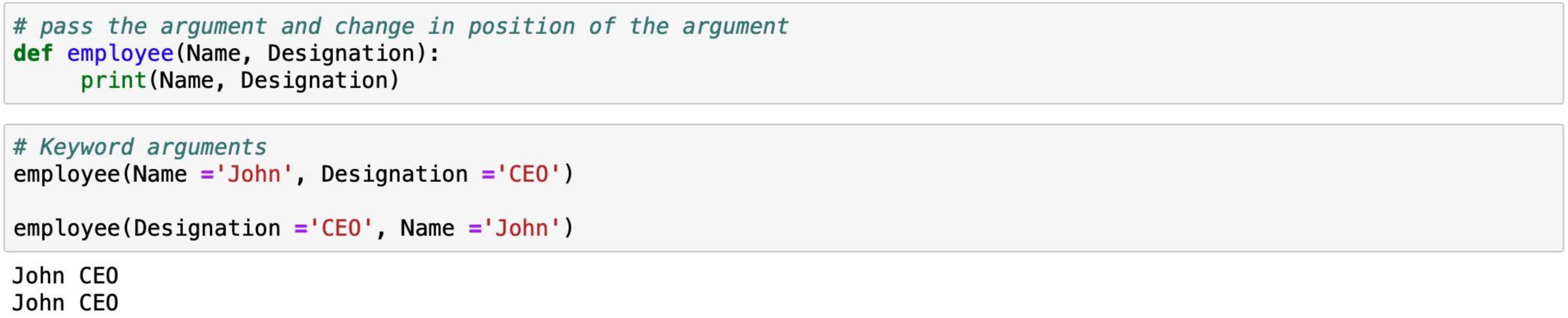
```
## Function  - Create a function and pass input variable
## pass variable to the function
def hello(nm):
    print("Hello ",nm)
```

```
## call the function without passing the argument
hello()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-5-82e586c80250> in <module>
      1 ## call the function without passing the argument
----> 2 hello()

TypeError: hello() missing 1 required positional argument: 'nm'
```

Keyword Arguments

The arguments have names assigned to them. In the below example, we have Name & Designation as the named parameters. We pass 'John' as Name and 'CEO' as Designation.



```
# pass the argument and change in position of the argument
def employee(Name, Designation):
     print(Name, Designation)
```

```
# Keyword arguments
employee(Name ='John', Designation ='CEO')

employee(Designation ='CEO', Name ='John')
```

```
John CEO
John CEO
```

Keyword Arguments

Note: Even if the wrong values are passed, it will NOT throw an error. For example, if we say 'CEO' as Name instead of 'John', the function will still work but with wrong values

```
# pass the argument and change in position of the argument
def employee(Name, Designation):
     print(Name, Designation)
```

```
# even if the wrong values are passed, it will runw ithout any error
employee(Name ='CEO', Designation ='John')
```

```
CEO John
```

Default Arguments

Note: The function expects 2 arguments - Name and Salary. We have passed a default value for salary. When we call the function, we pass only Name but not Salary. In this case, it will consider the default value for Salary that has been passed when the function was defined.

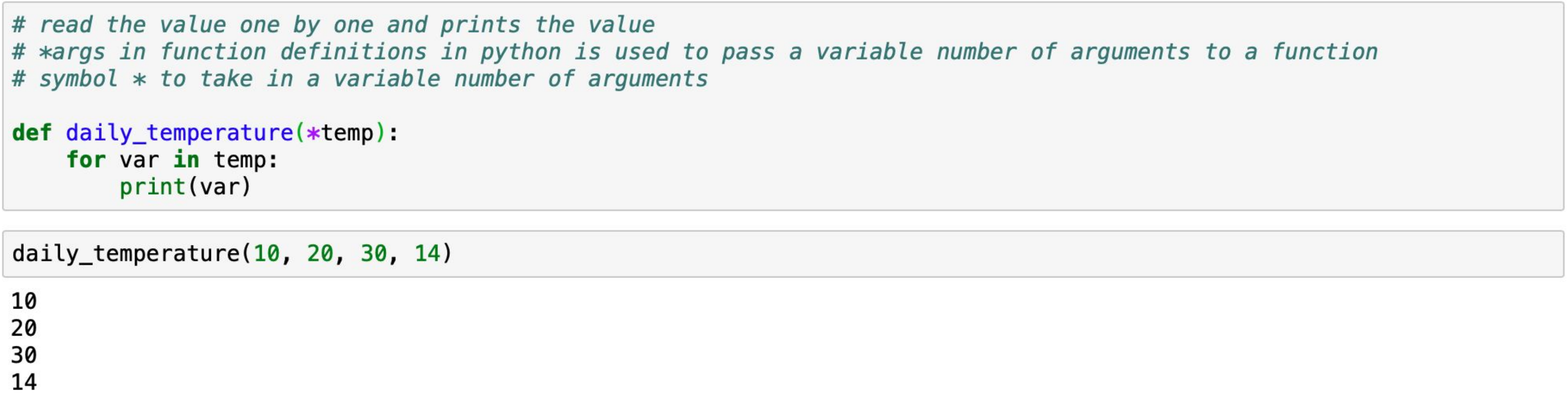
```
def employee(Name, Salary = 40000 ):

    print("Employee Name: ", Name)
    print("Employee Salary ", Salary)
    return;
```

```
employee( "Paul" )
```

```
Employee Name:  Paul
Employee Salary  40000
```

Variable-length arguments using *arg keyword

This helps you in passing variable number of arguments. This is especially helpful when you do not know how many arguments to pass to the function.



```python
# read the value one by one and prints the value
# *args in function definitions in python is used to pass a variable number of arguments to a function
# symbol * to take in a variable number of arguments

def daily_temperature(*temp):
    for var in temp:
        print(var)
```

```python
daily_temperature(10, 20, 30, 14)
```

```
10
20
30
14
```

Variable-length keyworded arguments using **kwargs

**kwargs allows you to pass keyworded variable length of arguments to a function.

You should use **kwargs if you want to handle named arguments in a function.

```
def my_function(**krgs):
    print(type(krgs))
    for k,v in krgs.items():
        print(k,"==", v)
```

```
my_function(firstname = "John", secondname = "Allen", salary = 20000, pf=345.75, goodperformer=True)
```

```
<class 'dict'>
firstname == John
secondname == Allen
salary == 20000
pf == 345.75
goodperformer == True
```

Variable Scope

Variable scope

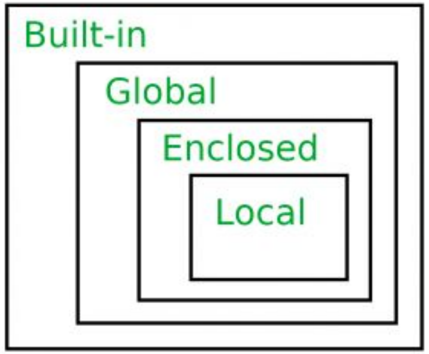
- A namespace is a container where names are mapped to objects (variables)
- A scope defined the hierarchical order in which the namespaces have to be searched in order to obtain the mapping name-to-object (variables)
- Scope defined the accessibility and lifetime of a variable

The LEGB rule

In Python, the LEGB rule is used to decide the order in which the namespaces are to be searched for scope resolution.

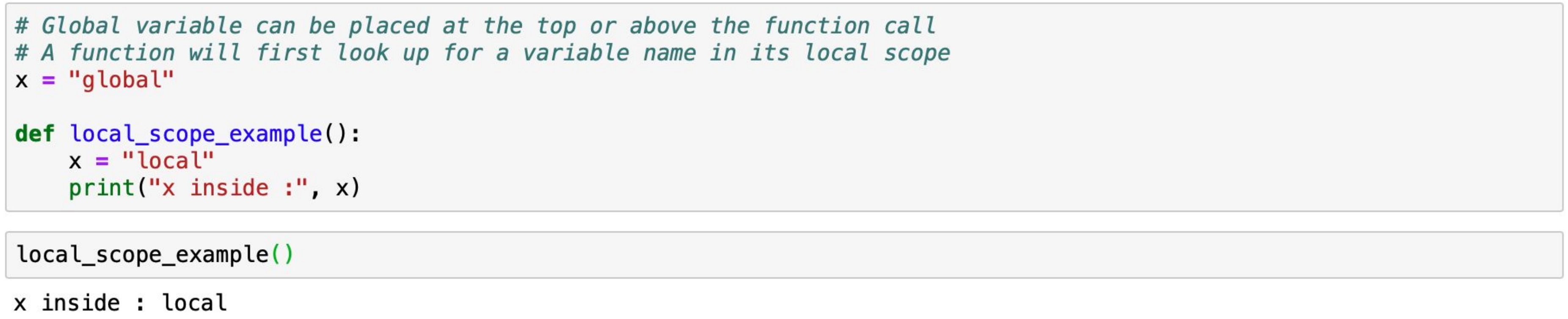
Variable scope hierarchy:

1. Built-In (B): Reserved names in Python
2. Global Variable (G): Defined at the uppermost level
3. Enclosed (E): Defined inside enclosing or nested functions
4. Local Variable (L): Defined inside a function



Local Scope

Local scope refers to variables defined in current function. A function will first look up for a variable name in its local scope. If it does not find it there, only then the outer scopes are searched for.



```
# Global variable can be placed at the top or above the function call
# A function will first look up for a variable name in its local scope
x = "global"

def local_scope_example():
    x = "local"
    print("x inside :", x)
```

```
local_scope_example()
```

```
x inside : local
```

Global Scope

If a variable is not defined in local scope, then, it is checked for in the higher scope, in this case, the global scope.

```
# Global variable can be placed at the top or above the function call
# A function will first look up for a variable name in its local scope
x = "global"

def global_scope_example():
    x = "local"
    print("x inside :", x)
```

```
# Local scope output
global_scope_example()
```

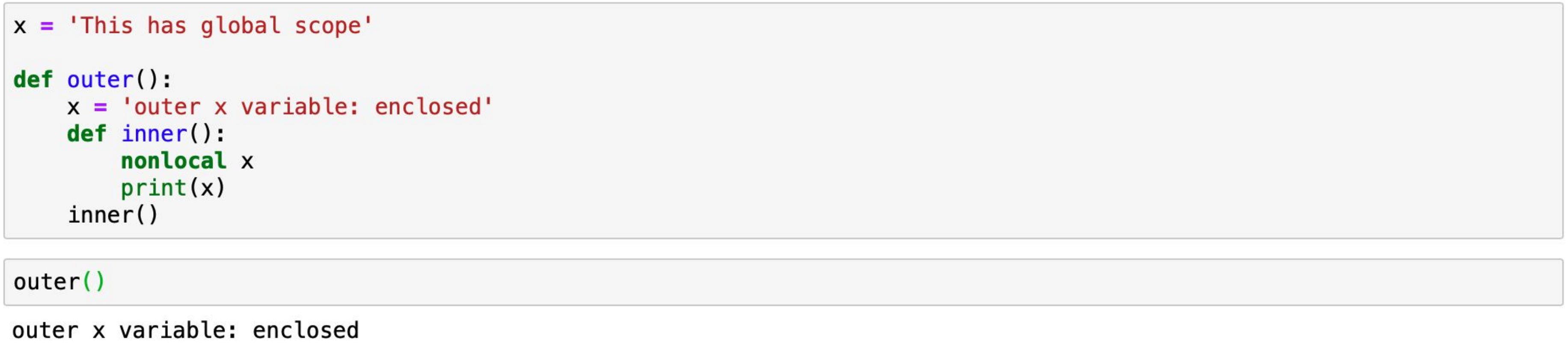
```
x inside : local
```

```
# Global scope output
print(x)
```

```
global
```

Enclosed Scope

For the enclosed scope, we need to define an outer function enclosing the inner function. Refer to the variable using the nonlocal keyword.



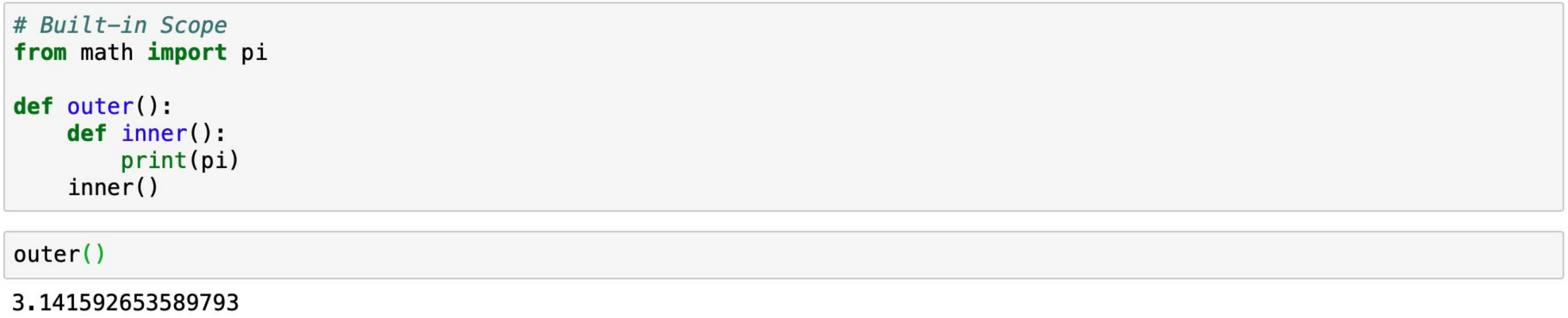
```python
x = 'This has global scope'

def outer():
    x = 'outer x variable: enclosed'
    def inner():
        nonlocal x
        print(x)
    inner()
```

```python
outer()
```

```
outer x variable: enclosed
```

Built-In Scope

If we have not defined a variable and the name of the variable matches with a built-in function from an existing Python module, the function will use the built-in function.



```
# Built-in Scope
from math import pi

def outer():
    def inner():
        print(pi)
    inner()
```

```
outer()
```

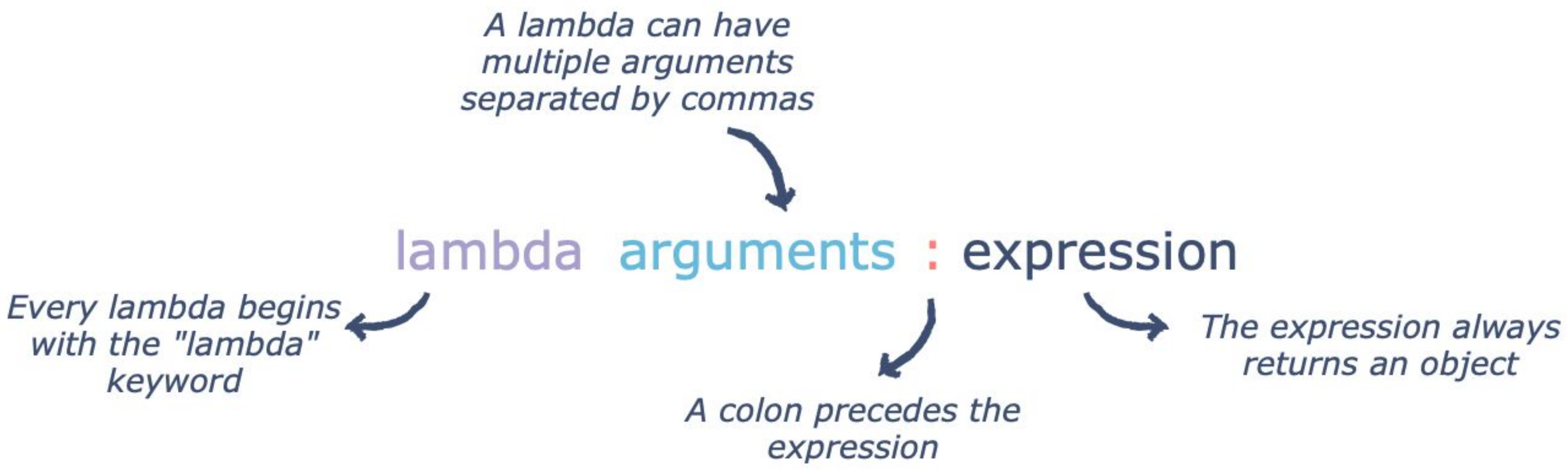
```
3.141592653589793
```

Lambda Function

Lambda function

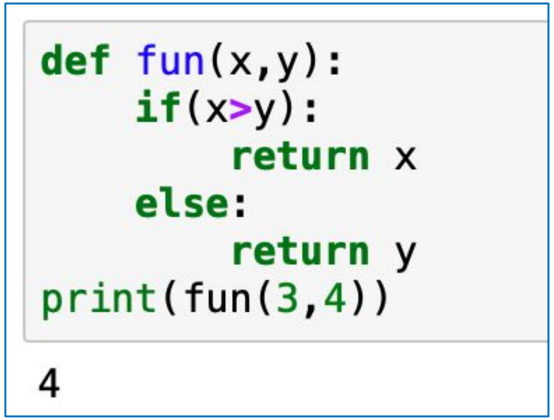
- Lambda functions are anonymous, i.e. to say they have no names
- The lambda is a keyword
- It is a simple one-line function
- No def or return keyword to be used with a lambda function

The structure of the lambda function


A lambda can have multiple arguments separated by commas
lambda arguments : expression
Every lambda begins with the "lambda" keyword
A colon precedes the expression
The expression always returns an object

Using lambda function to reduce code size

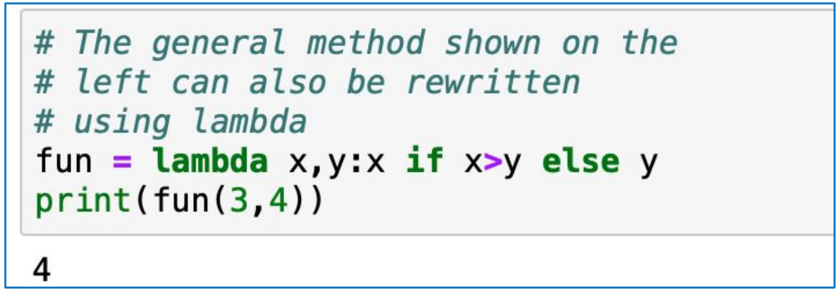
Normal Python Code



```
def fun(x,y):
    if(x>y):
        return x
    else:
        return y
print(fun(3,4))
```

```
4
```

Using lambda function

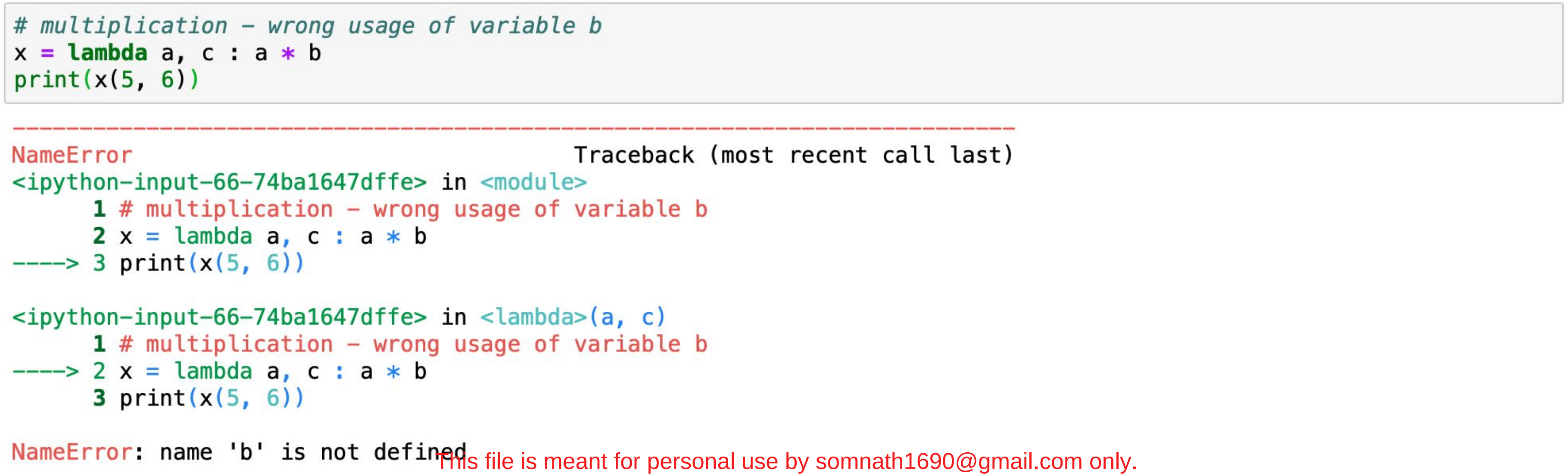


```
# The general method shown on the
# left can also be rewritten
# using lambda
fun = lambda x,y:x if x>y else y
print(fun(3,4))
```

```
4
```

Wrong usage of lambda

You will need to declare the variable which you may use inside the lambda function. In the below example, we use variable, b, without declaring it.



```
# multiplication - wrong usage of variable b
x = lambda a, c : a * b
print(x(5, 6))
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-66-74ba1647dffe> in <module>
      1 # multiplication - wrong usage of variable b
      2 x = lambda a, c : a * b
----> 3 print(x(5, 6))

<ipython-input-66-74ba1647dffe> in <lambda>(a, c)
      1 # multiplication - wrong usage of variable b
----> 2 x = lambda a, c : a * b
      3 print(x(5, 6))

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

The lambda() with map()

map() functions expect a function_object, in our case a lambda function,and a sequence (iterables, such as list, dictionary, etc.)

It executes the function_object for each element in the sequence and returns a sequence of the elements modified by the function object.

lambda expression (You can also consider any other function)

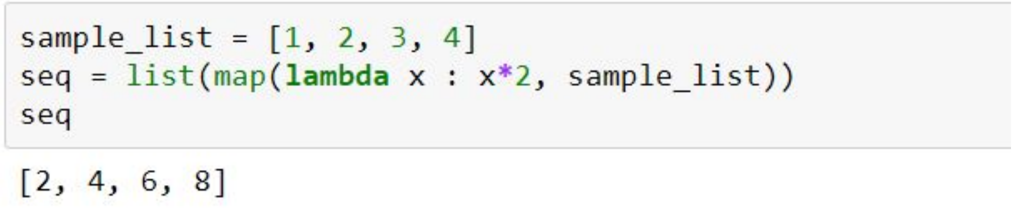
map(lambda_expression, sequence)

map() function

iterables such as list, tuples, etc. (you can pass multiple iterables)

The lambda() with map()

The output is often type-casted into a seq type, as follow:

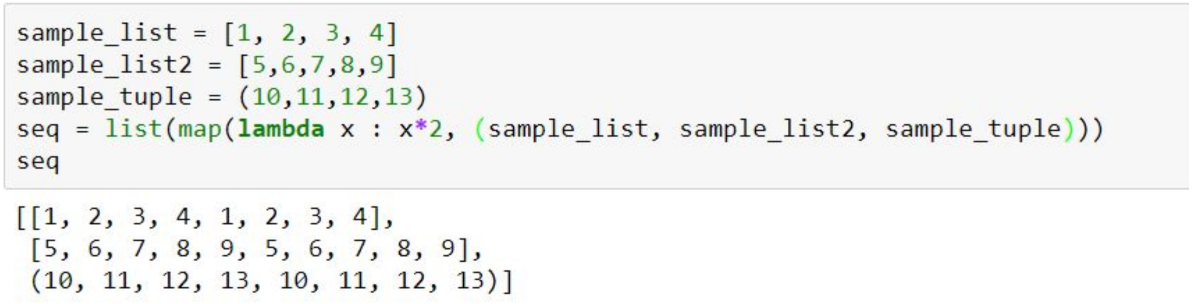


```
sample_list = [1, 2, 3, 4]
seq = list(map(lambda x : x*2, sample_list))
seq
```

```
[2, 4, 6, 8]
```

The lambda() with map()

You can pass multiple sequences to the map function as follow:



```
sample_list = [1, 2, 3, 4]
sample_list2 = [5,6,7,8,9]
sample_tuple = (10,11,12,13)
seq = list(map(lambda x : x*2, (sample_list, sample_list2, sample_tuple)))
seq
```

```
[[1, 2, 3, 4, 1, 2, 3, 4],
 [5, 6, 7, 8, 9, 5, 6, 7, 8, 9],
 (10, 11, 12, 13, 10, 11, 12, 13)]
```

map(f, li1, li2, li3, …) → applies f to all lists in parallel. That is, first element of result would be f(e1,e2,e3) where its args are first element of each list.

The lambda() with filter()

The *filter()* function expects two arguments: *function_object(lambda)* and an iterable.

function_object returns a boolean value and is called for each element of the iterable.

It returns only those elements for which the function_object returns *true*.

lambda expression (You can also consider any other function)

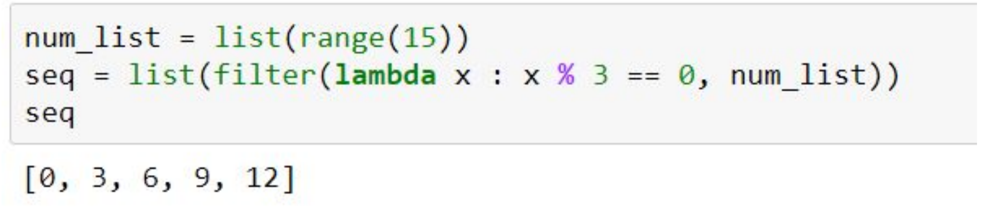
filter(lambda_expression, sequence)

filter() function

iterables such as list, tuples, etc. (only one iterable as input)

The lambda() with filter()

The output is often type-casted into a seq type, as follow:



```
num_list = list(range(15))
seq = list(filter(lambda x : x % 3 == 0, num_list))
seq
```

```
[0, 3, 6, 9, 12]
```

Unlike *map(), the filter()* function can only have one iterable as input.

The lambda() with reduce()

The reduce() function in Python takes in a function and a sequence as argument.

The function is called with a lambda function and a seq and a new reduced result is returned.

This performs a repetitive operation over the pairs of the seq.

lambda expression (You can also consider any other function)

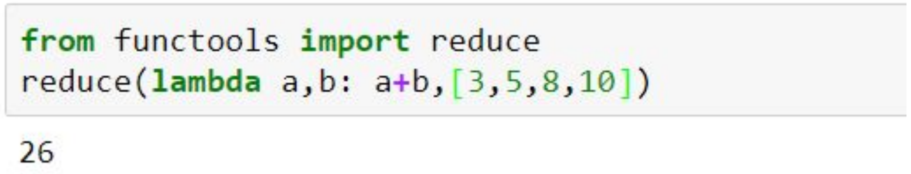
reduce(lambda_expression, sequence)

reduce() function

iterables such as list, tuples, etc. (you can also pass multiple iterables of same type)

The lambda() with reduce()

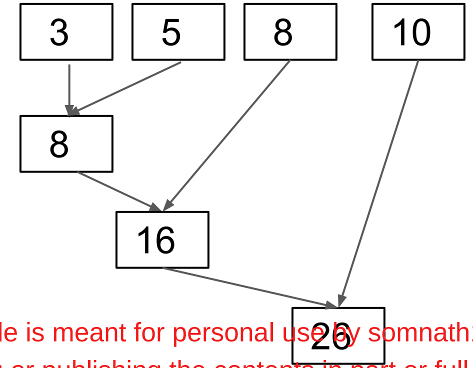
Determining the summation of all elements of a list of numerical values by using reduce:



```
from functools import reduce
reduce(lambda a,b: a+b,[3,5,8,10])
```

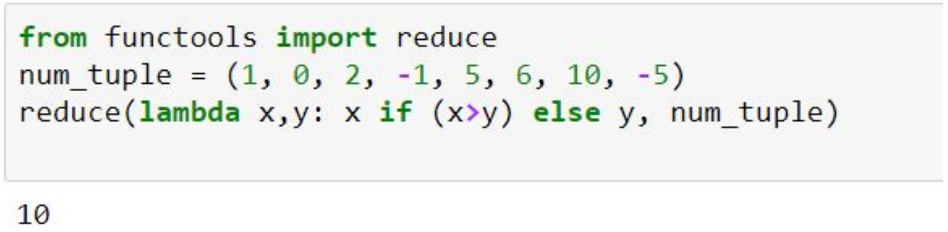
```
26
```

Working:



The lambda() with reduce()

Determining the maximum of a list of numerical values by using reduce:



```
from functools import reduce
num_tuple = (1, 0, 2, -1, 5, 6, 10, -5)
reduce(lambda x,y: x if (x>y) else y, num_tuple)
```

```
10
```

Note : reduce() can only have iterables of same type as input.

The lambda() with accumulate()

The accumulate() function in Python takes in a function and a sequence as argument.

The function is called with a lambda function and a seq and a new reduced result is returned.

Unlike reduce(), it returns a sequence containing the intermediate results

lambda expression (You can also consider any other function)

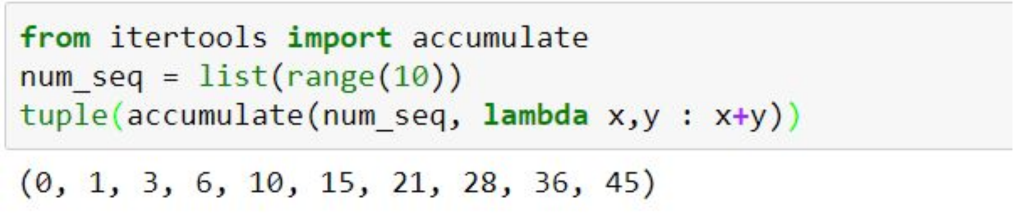
accumulate(sequence, lambda_expression)

accumulate() function

iterables such as list, tuples, etc. (you can also pass multiple iterables of same type)

The lambda() with accumulate()

Determining the summation of all elements of a list of numerical values by using accumulate:



```
from itertools import accumulate
num_seq = list(range(10))
tuple(accumulate(num_seq, lambda x,y : x+y))
```

```
(0, 1, 3, 6, 10, 15, 21, 28, 36, 45)
```

Difference between reduce() and accumulate()

reduce()	accumulate()
The reduce() stores the intermediate result and only returns the final summation value	The accumulate() returns a list containing the intermediate results. The last number of the list returned is summation value of the list
The reduce(fun,seq) takes function as 1st and sequence as 2nd argument	The accumulate(seq,fun) takes sequence as 1st argument and function as 2nd argument
The reduce() is defined in “functools” module	The accumulate() is defined in “itertools” module

Recursive Functions

Recursive Function

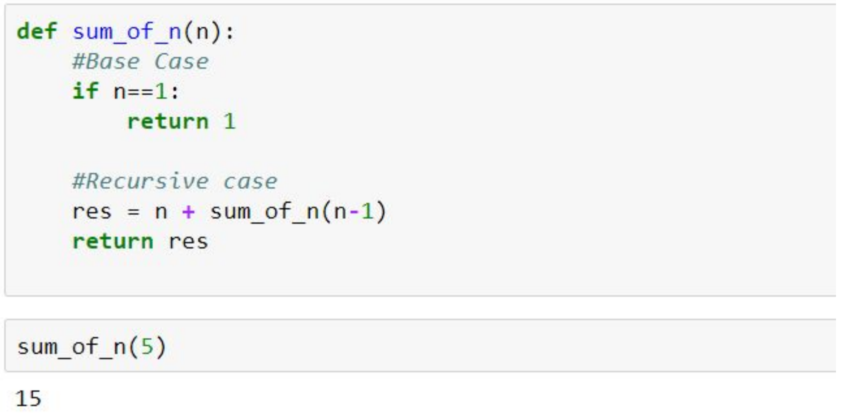
A recursive function is a function defined in terms of itself via self-referential expressions.

The function will continue to call itself and repeat its behavior until some condition is met to return a result.

All recursive functions share a common structure made up of two parts: base case and recursive case.

Recursive Function

For example:



```
def sum_of_n(n):
    #Base Case
    if n==1:
        return 1

    #Recursive case
    res = n + sum_of_n(n-1)
    return res
```

```
sum_of_n(5)
```

```
15
```

A base case is a case, where the problem can be solved without further recursion.

Recursive Function

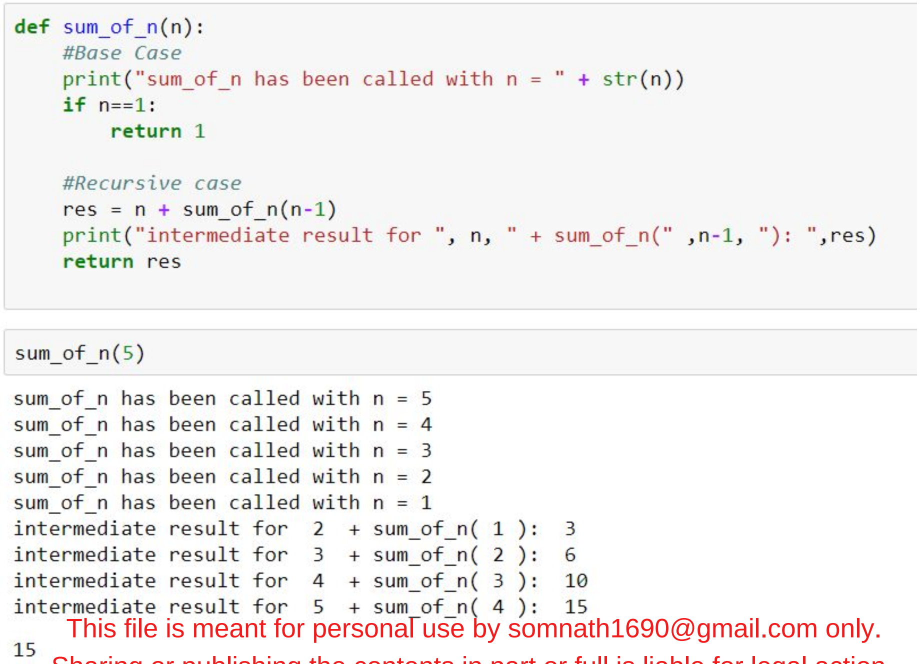
A recursive function has to fulfil an important condition to be used in a program: *it has to terminate.*

A recursive function terminates, if with every recursive call the solution of the problem is downsized and moves towards a base case.

A recursion can end up in an infinite loop, if the base case is not met in the calls.

Recursive Function

Let us track how the previously defined recursive function, sum_of_n works by adding two print functions:



```
def sum_of_n(n):
    #Base Case
    print("sum_of_n has been called with n = " + str(n))
    if n==1:
        return 1

    #Recursive case
    res = n + sum_of_n(n-1)
    print("intermediate result for ", n, " + sum_of_n(" ,n-1, "): ",res)
    return res
```

```
sum_of_n(5)
```

```
sum_of_n has been called with n = 5
sum_of_n has been called with n = 4
sum_of_n has been called with n = 3
sum_of_n has been called with n = 2
sum_of_n has been called with n = 1
intermediate result for  2  + sum_of_n( 1 ):  3
intermediate result for  3  + sum_of_n( 2 ):  6
intermediate result for  4  + sum_of_n( 3 ):  10
intermediate result for  5  + sum_of_n( 4 ):  15

15
```

Thank You