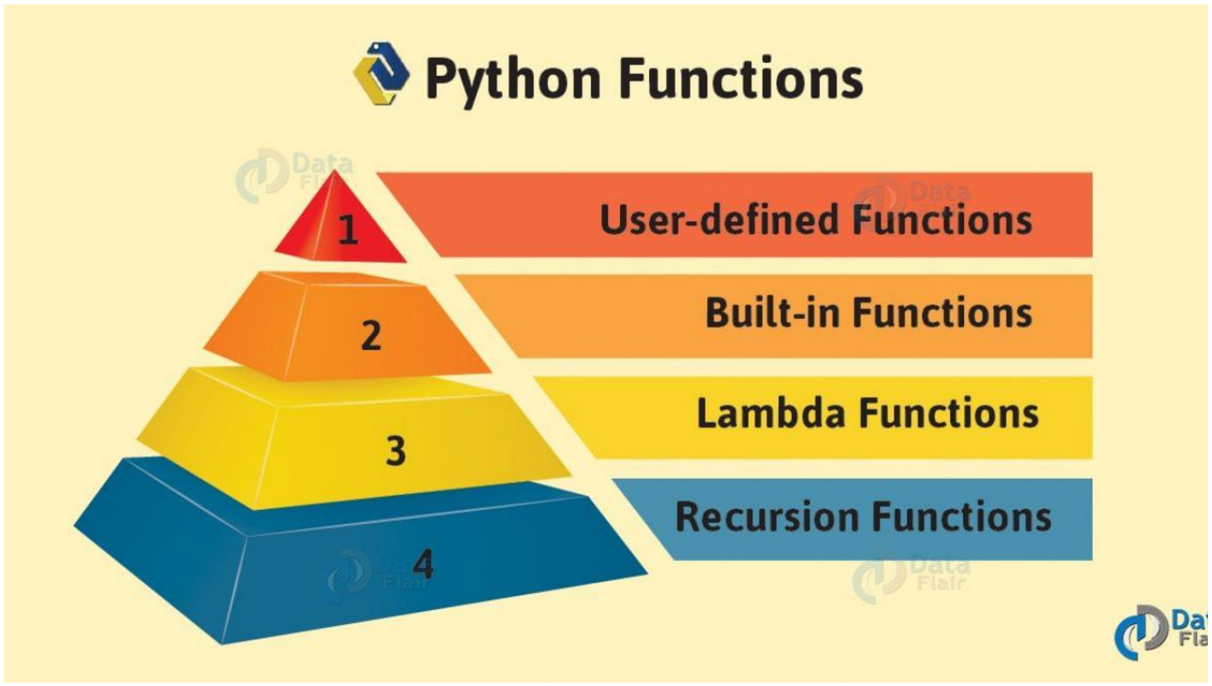
FUNCTION IN PYTHON



A function is a block of organized and reusable code that performs a single, specific and well-defined task. Every function encapsulates a set of operations and when called it returns the information to the calling program. Functions provide better modularity for your application and a high degree of code reuse.

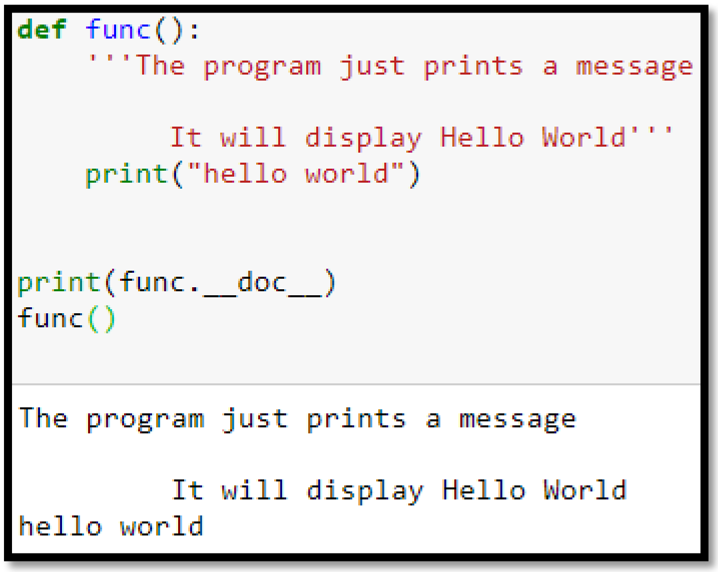
DOCSTRINGS:

Documentation strings serve the purpose of explaining the block of code and unlike comments they are more specific and have a proper syntax. They are started by using two/three quotes (‘’’) and ended by the same.

Key Points to remember:

- The first line should be short and concise highlighting the summary of the code’s purpose
- It should begin with a capital letter and end with a period.
- Triple quotes are used to extend the docstring to multiple lines.he docstring specified can be accessed through the _doc_ attribute of the function.

- In case of multiple lines in the documentation string, the second line should be blank so as to separate the summary from the rest of the description. The other lines should be more of a paragraph describing the code.
- Unlike comments docstrings are retained throughout the runtime of the program. So user an inspect them during program execution



```
def func():
    '''The program just prints a message

         It will display Hello World'''
    print("hello world")


print(func.__doc__)
func()
```

```
The program just prints a message

         It will display Hello World
hello world
```

BUILT IN FUNCTIONS

Python provides a set of built-in functions. Find it here:
https://docs.python.org/3/library/functions.html

USER DEFINED FUNCTIONS

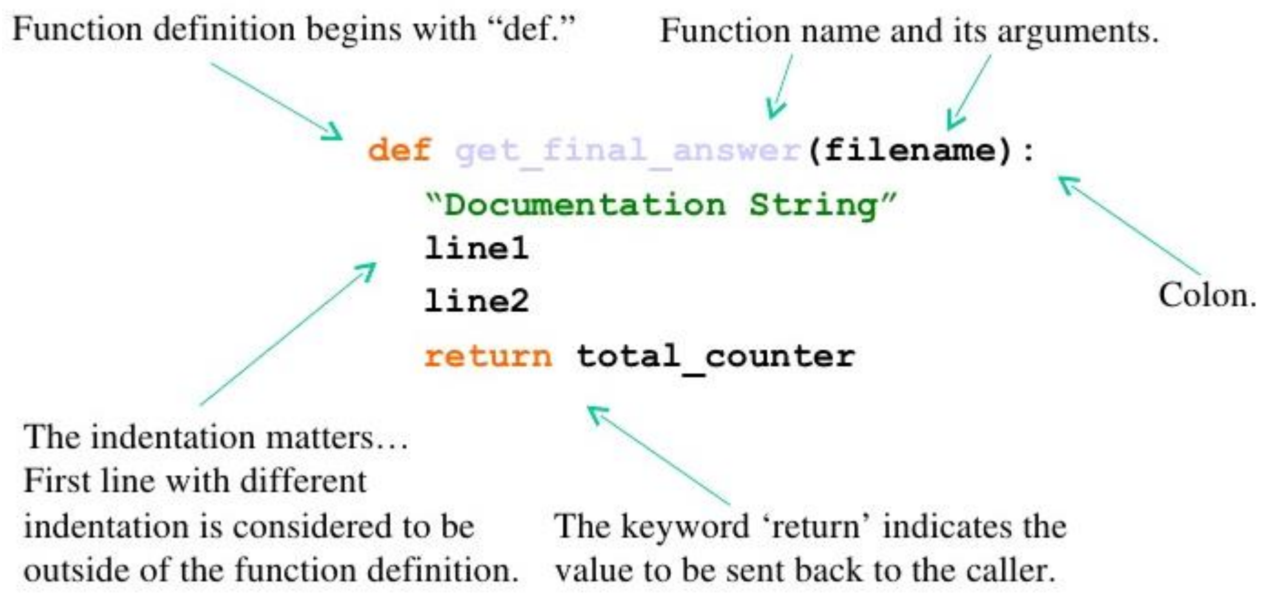
To define a function, the following points should be kept in mind:

- Function blocks starts with the keyword **def**
- The keyword is followed by the function name and parentheses ( ) . The function name is used to uniquely identify the function
- After the parenthesis a colon (:) is placed.
- Parameters or arguments that the function accepts are placed within parenthesis. Though these parameter values are passed to the function.

They are optional. In case no values are passed, nothing is places within the parenthesis.

- The code block within the function is properly indented to form the block code.
- A function may have a return[expression] statement. That is the return statement is optional. If it exists it passes back an expression to the caller. A return statement with no argument is the same as return None.
- You can assign the function name to a variable. Doing this will allow you to call the same function using the name of that variable.

- The first statement if the function can be an optional statement- the documentation string of the fuction or ***docstring*** describe what the function does.


Function definition begins with "def."
Function name and its arguments.
def get_final_answer(filename):
"Documentation String"
line1
line2
return total_counter
Colon.
The indentation matters… First line with different indentation is considered to be outside of the function definition.
The keyword 'return' indicates the value to be sent back to the caller.

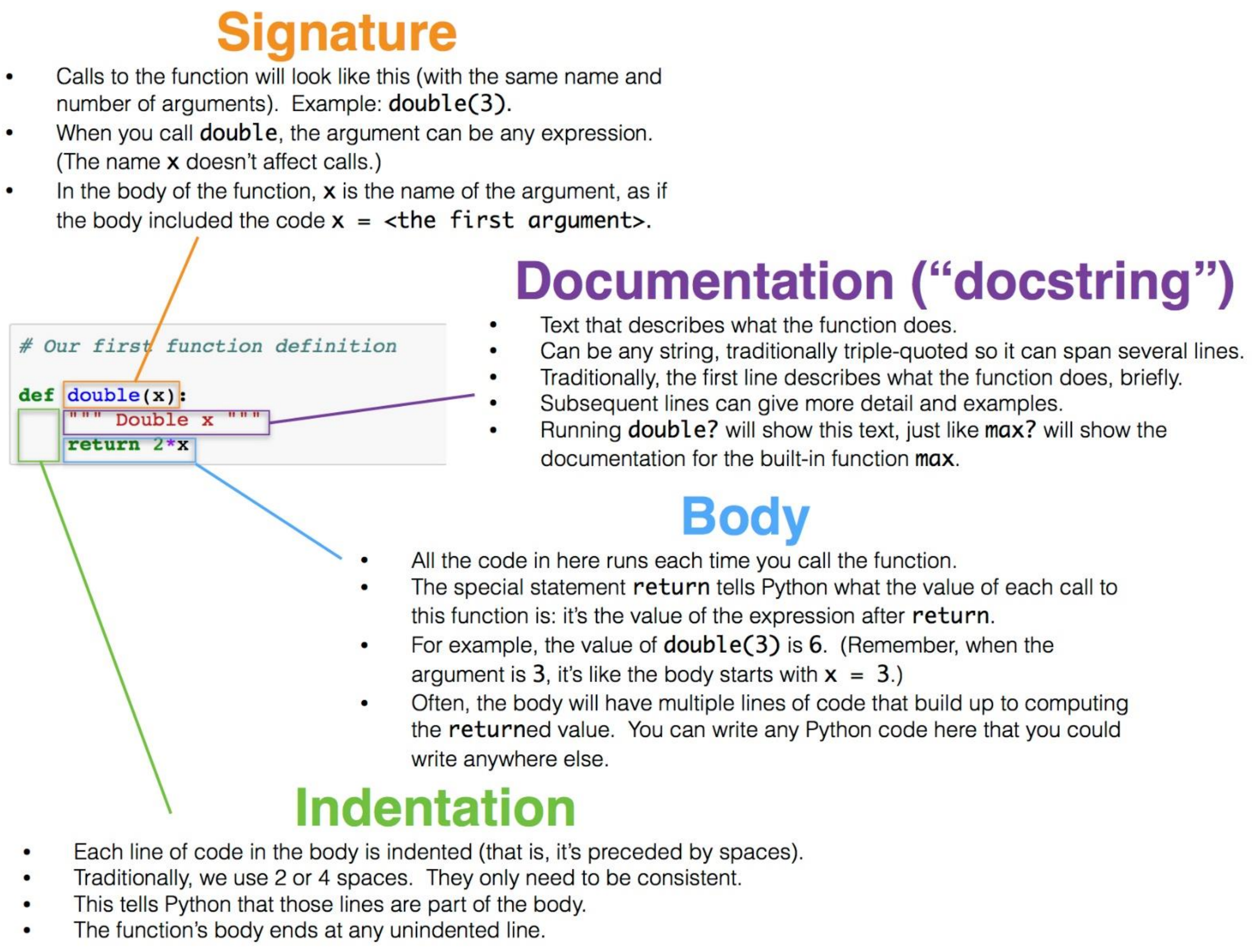
No header file or declaration of types of function or arguments.

To call the function use the function name followed by parenthesis.

```
def my_function():
  print("Hello from a function")
my_function()
```

PASS STATEMENT IN FUNCTIONS

`function` definitions cannot be empty, but if you for some reason have a `function` definition with no content, put in the `pass` statement to avoid getting an error.



Signature

- Calls to the function will look like this (with the same name and number of arguments). Example: `double(3)`.
- When you call `double`, the argument can be any expression. (The name `x` doesn't affect calls.)
- In the body of the function, `x` is the name of the argument, as if the body included the code `x = <the first argument>`.

```
# Our first function definition

def double(x):
    """ Double x """
    return 2*x
```

Documentation ("docstring")

- Text that describes what the function does.
- Can be any string, traditionally triple-quoted so it can span several lines.
- Traditionally, the first line describes what the function does, briefly.
- Subsequent lines can give more detail and examples.
- Running `double?` will show this text, just like `max?` will show the documentation for the built-in function `max`.

Body

- All the code in here runs each time you call the function.
- The special statement `return` tells Python what the value of each call to this function is: it's the value of the expression after `return`.
- For example, the value of `double(3)` is `6`. (Remember, when the argument is `3`, it's like the body starts with `x = 3`.)
- Often, the body will have multiple lines of code that build up to computing the `return`ed value. You can write any Python code here that you could write anywhere else.

Indentation

- Each line of code in the body is indented (that is, it's preceded by spaces).
- Traditionally, we use 2 or 4 spaces. They only need to be consistent.
- This tells Python that those lines are part of the body.
- The function's body ends at any unindented line.

PARAMETERS / ARGUMENTS

The terms *parameter* and *argument* can be used for the same thing: information that are passed into a function.

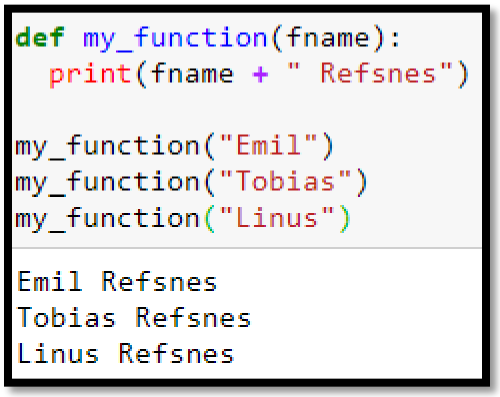
From a function's perspective:

- A parameter is the variable listed inside the parentheses in the function definition.

- An argument (args) is the value that is sent to the function when it is called.

Information can be passed into functions as arguments. Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.

The following example has a function with one argument (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name:

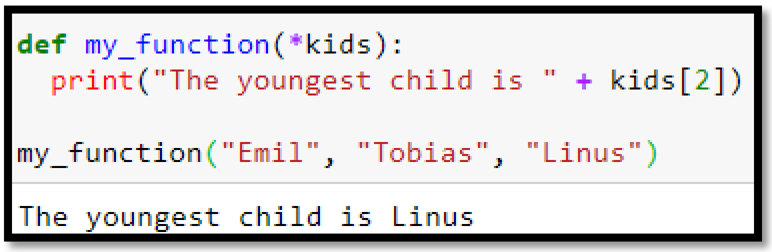


```python
def my_function(fname):
  print(fname + " Refsnes")

my_function("Emil")
my_function("Tobias")
my_function("Linus")
```

```
Emil Refsnes
Tobias Refsnes
Linus Refsnes
```

ARBITRARY ARGUMENTS

If you do not know how many arguments that will be passed into your function, add a * before the parameter name in the function definition. This way the function will receive a ***tuple* of arguments**, and can access the items accordingly. Arbitrary arguments are often shortened to *args in Python documentation.

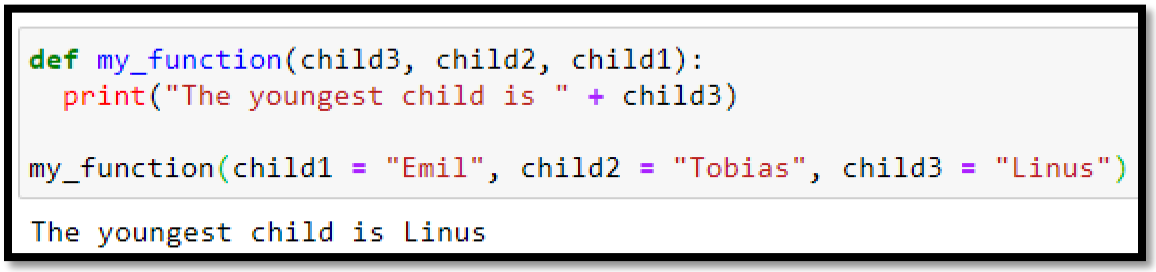


```python
def my_function(*kids):
  print("The youngest child is " + kids[2])

my_function("Emil", "Tobias", "Linus")
```

```
The youngest child is Linus
```

KEYWORD ARGUMENTS

When we call a function with some values, the values are assigned to the arguments based on their position. (Positional Arguments). Python also allows functions to be called using keyword arguments in which the order (position) of the arguments can be changed with the ***key = value*** syntax. This way the order of the arguments does not matter. Keywords arguments are often shortened to kwargs in Python documentation.



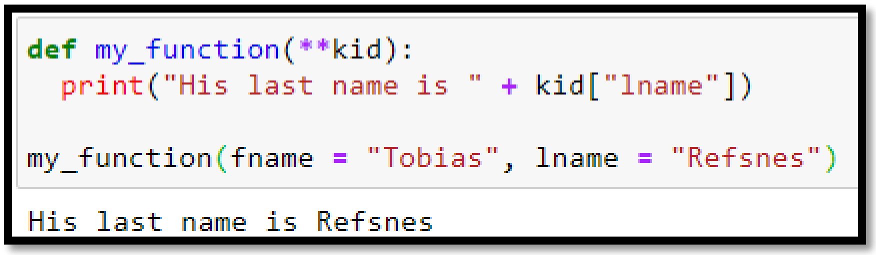
```
def my_function(child3, child2, child1):
  print("The youngest child is " + child3)

my_function(child1 = "Emil", child2 = "Tobias", child3 = "Linus")
```

```
The youngest child is Linus
```

ARBITRARY KEYWORD ARGUMENTS, **kwargs

If you do not know how many keyword arguments that will be passed into your function, add two asterisk: ** before the parameter name in the function definition.

This way the function will receive a ***dictionary* of arguments**, and can access the items accordingly:

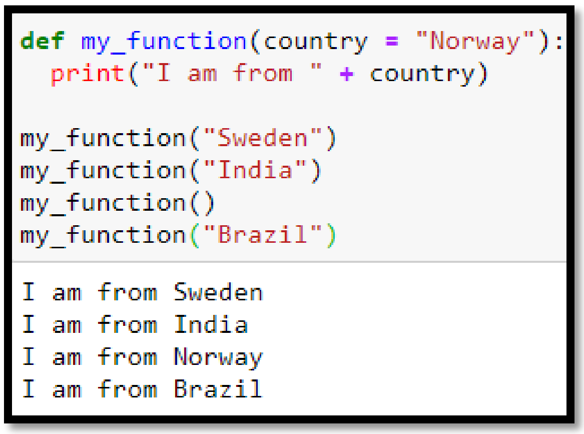


```
def my_function(**kid):
  print("His last name is " + kid["lname"])

my_function(fname = "Tobias", lname = "Refsnes")
```

```
His last name is Refsnes
```

DEFAULT PARAMETER VALUE

We can specify any number of default arguments in the function. In case of default arguments then they must be written after the non-default arguments. The following example shows how to use a default parameter value.
If we call the function without argument, it uses the default value:



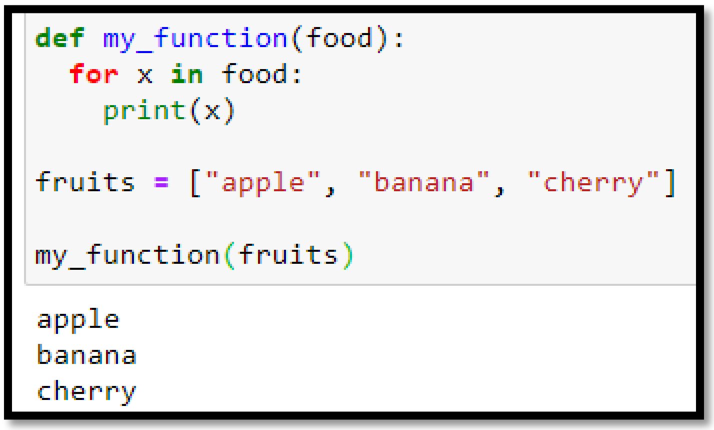
```
def my_function(country = "Norway"):
  print("I am from " + country)

my_function("Sweden")
my_function("India")
my_function()
my_function("Brazil")
```

```
I am from Sweden
I am from India
I am from Norway
I am from Brazil
```

PASSING A LIST AS AN ARGUMENT

You can send any data types of argument to a function (string, number, list, dictionary etc.), and it will be treated as the same data type inside the function.

E.g. if you send a List as an argument, it will still be a List when it reaches the function:



```
def my_function(food):
  for x in food:
    print(x)

fruits = ["apple", "banana", "cherry"]

my_function(fruits)
```

```
apple
banana
cherry
```

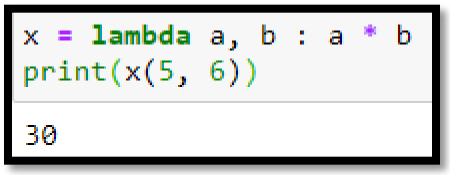
LAMDA FUNCTIONS

A lambda function is a small anonymous function. A lambda function can take any number of arguments, but can only have one expression.

SYNTAX: **lambda *arguments* : *expression***

The expression is executed and the result is returned. It is better to use lambda functions when an anonymous function is required for a short period of time or for a small purpose.

EXAMPLE: Add 10 to argument a, and return the result:



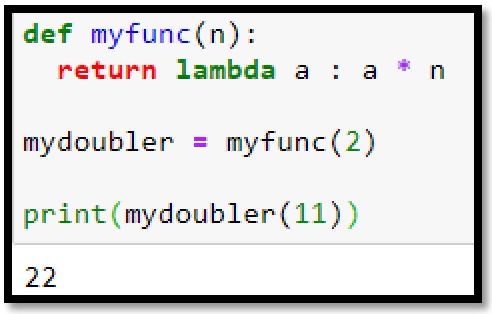
```
x = lambda a, b : a * b
print(x(5, 6))
```

```
30
```

The power of lambda is better shown when you use them as an anonymous function inside another function. Say you have a function definition that takes one argument, and that argument will be multiplied with an unknown number:

EXAMPLE:

Use the **myfunc** function definition to make a function that always doubles the number you send in:



```
def myfunc(n):
  return lambda a : a * n

mydoubler = myfunc(2)

print(mydoubler(11))
```

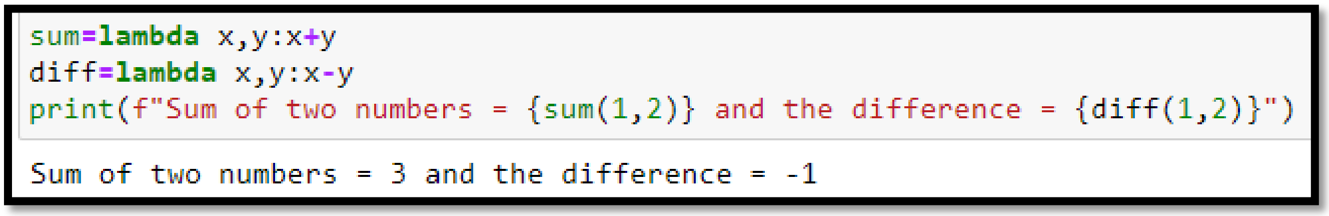
```
22
```

Lambda functions are used along with built-in functions like filter( ), map( ), reduce( ) etc.

POINTS TO REMEMBER

- Lambda functions have no name
- Lambda functions can take any number of arguments
- Lambda functions can return only one value in the form of an expression

- Lambda functions do not have an explicit return statement but it always contains an expression which is returned
- They are a one-line version of a function and hence cannot contain multiple expressions
- Lambda functions cannot access variables other than those not in the parameter list and not even the global ones.
- You can pass lambda functions as arguments in other functions.



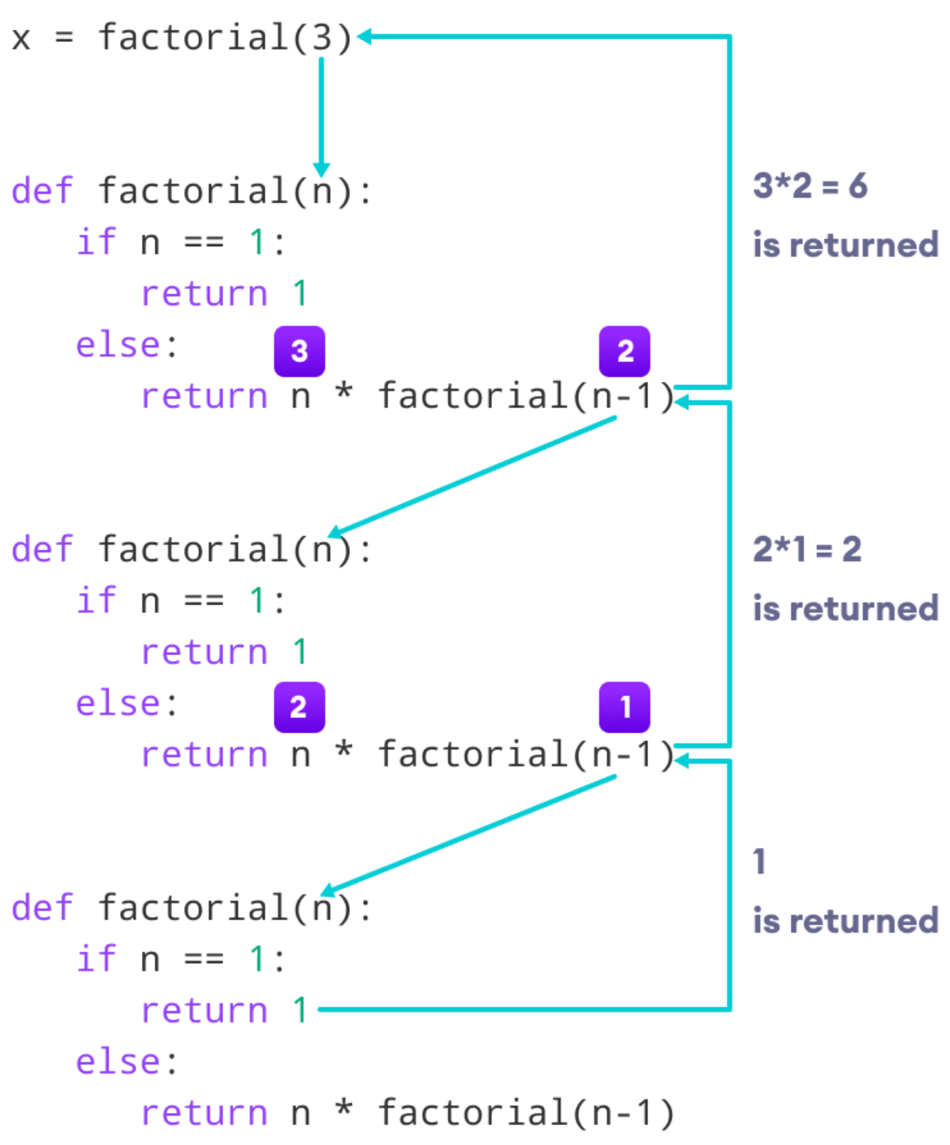
```
sum=lambda x,y:x+y
diff=lambda x,y:x-y
print(f"Sum of two numbers = {sum(1,2)} and the difference = {diff(1,2)}")
```

```
Sum of two numbers = 3 and the difference = -1
```

RECURSION

When function is called within the same function, it is known as recursion in Python. The function which calls the same function, is known as recursive function. A function that calls itself, and doesn't perform any task after function call, is known as tail recursion. In tail recursion, we generally call the same function with return statement.

Understanding the simple factorial problem in Recursion:



```
x = factorial(3)

def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)

def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)

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

VARIABLE SCOPE AND LIFETIME

SCOPE OF A VARIABLE: Part of the program in which a variable is accessible is called its scope

LIFETIME OF A VARIABLE: Duration for which the variable exists is called its lifetime.

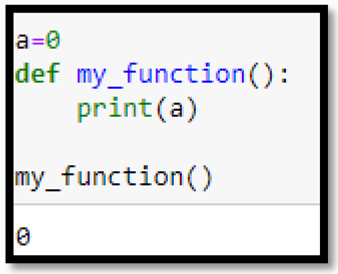
GLOBAL VARIABLE

A variable which is defined in the main body of a program file is called a *global* variable. It will be visible throughout the file, and also inside any file which imports that file. Only objects which are intended to be used globally, like functions and classes, should be put in the global namespace.

LOCAL VARIABLE

A variable which is defined inside a function is *local* to that function. It is accessible from the point at which it is defined until the end of the function, and exists for as long as the function is executing. The parameter names in the function definition behave like local variables, but they contain the values that we pass into the function when we call it. When we use the assignment operator (=) inside a function, its default behaviour is to create a new local variable – unless a variable with the same name is already defined in the local scope.

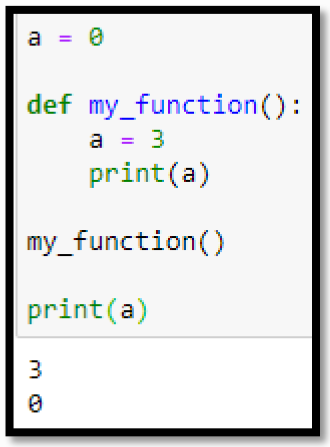
What if we want to access a global variable from inside a function? It is possible, but doing so comes with a few caveats:



```
a=0
def my_function():
    print(a)

my_function()
```

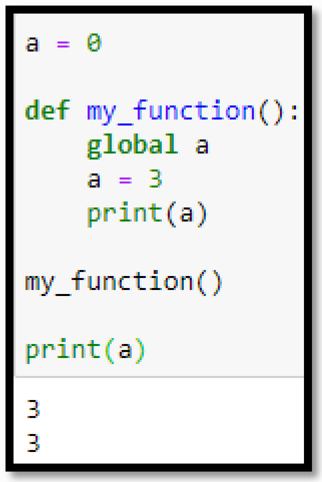
```
0
```

But what about this program?


```
a = 0

def my_function():
    a = 3
    print(a)

my_function()

print(a)
```
```
3
0
```

By default, the assignment statement creates variables in the local scope. So the assignment inside the function does not modify the global variable `a` – it creates a new local variable called `a`, and assigns the value `3` to that variable. The first print statement outputs the value of the new local variable – because if a local variable has the same name as a global variable **the local variable will always take precedence**. The last print statement prints out the global variable, which has remained unchanged.

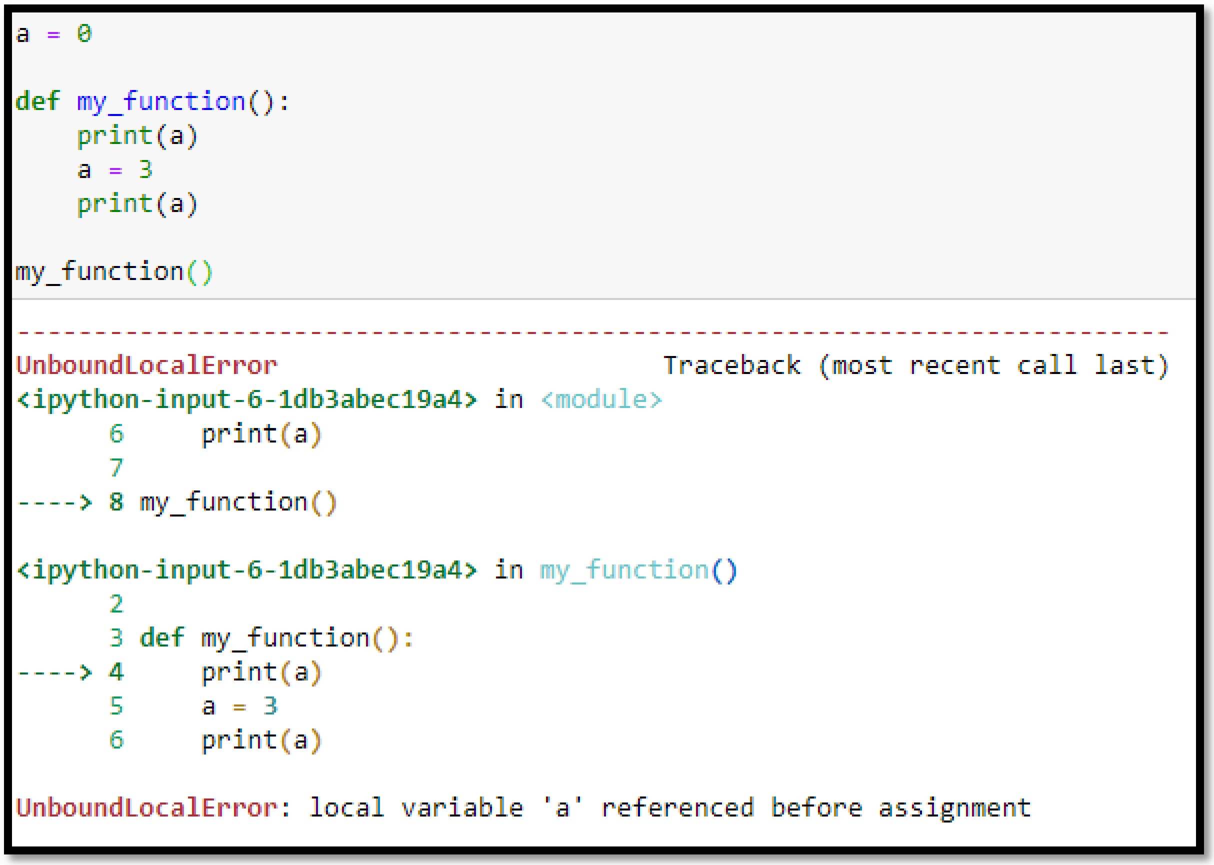
What if we really want to modify a global variable from inside a function? We can use the `global` keyword:


```
a = 0

def my_function():
    global a
    a = 3
    print(a)

my_function()

print(a)
```
```
3
3
```

If we have a global variable and create another global variable with the same name using the global keyword, the changes made in the new variable will be reflected everywhere in the program.

We can have a variable with the same name as that of a global variable in the program with **global** keyword. In such case a local variable of that name is created which is different from the global variable.

We sometime cannot **refer** to both a global variable and a local variable by the same name inside the same function. This program will give us an error:

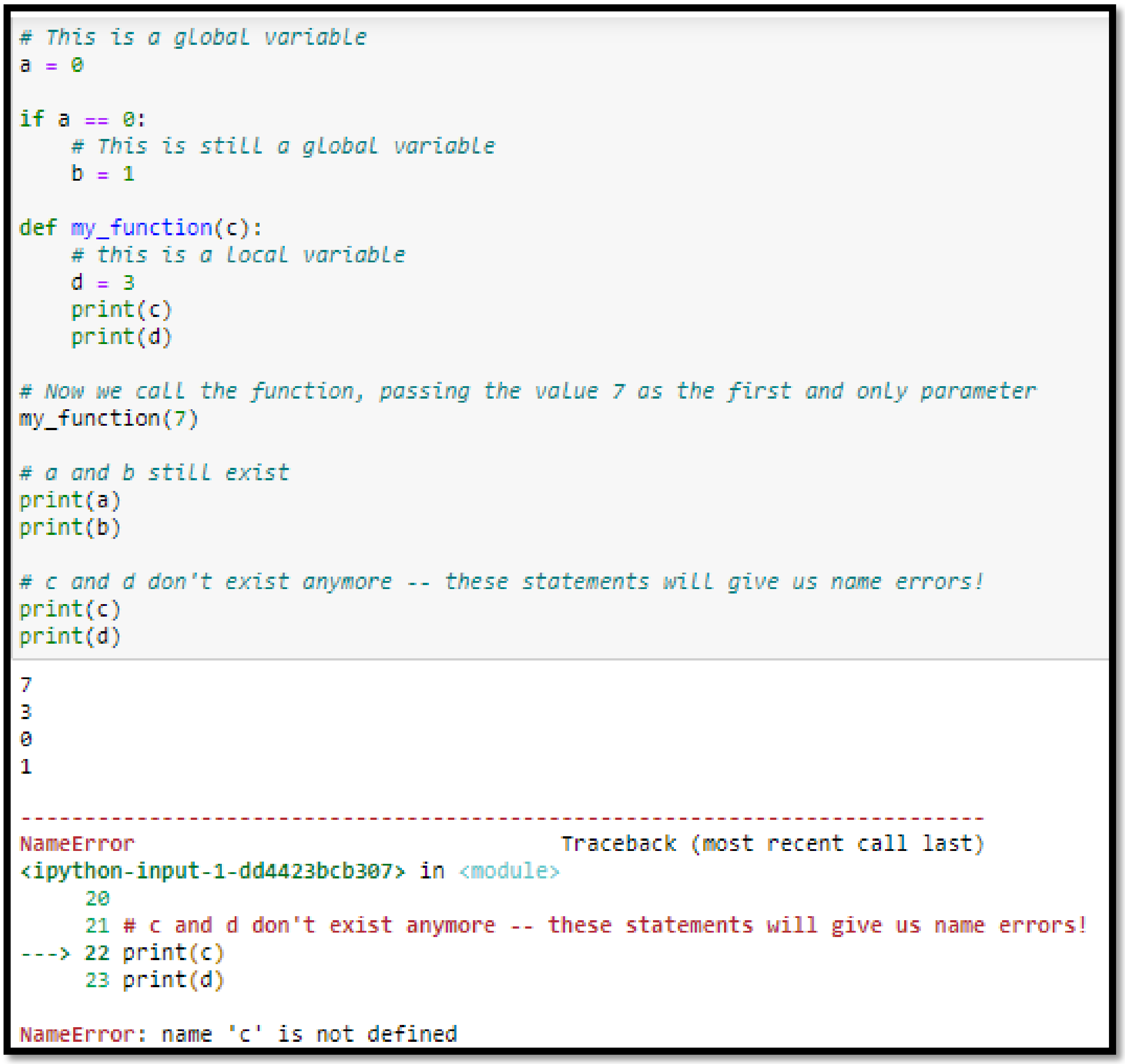


```
a = 0

def my_function():
    print(a)
    a = 3
    print(a)

my_function()
```

```
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
<ipython-input-6-1db3abec19a4> in <module>
      6     print(a)
      7
----> 8 my_function()

<ipython-input-6-1db3abec19a4> in my_function()
      2
      3 def my_function():
----> 4     print(a)
      5     a = 3
      6     print(a)

UnboundLocalError: local variable 'a' referenced before assignment
```

Because we haven't declared `a` to be global, the assignment in the second line of the function will create a local variable `a`. This means that we can't refer to the global variable `a` elsewhere in the function, even before this line! The first print statement now refers to the local variable `a` – but this variable doesn't have a value in the first line, because we haven't assigned it yet!

However, if we define a=3 before print statement then it would work appropriately.



```python
# This is a global variable
a = 0

if a == 0:
    # This is still a global variable
    b = 1

def my_function(c):
    # this is a local variable
    d = 3
    print(c)
    print(d)

# Now we call the function, passing the value 7 as the first and only parameter
my_function(7)

# a and b still exist
print(a)
print(b)

# c and d don't exist anymore -- these statements will give us name errors!
print(c)
print(d)
```

```
7
3
0
1

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-1-dd4423bcb307> in <module>
     20 
     21 # c and d don't exist anymore -- these statements will give us name errors!
---> 22 print(c)
     23 print(d)

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