




CHAPTER 2

WORKING WITH FUNCTIONS

Class XII
Session – 2020-21



LEARNING OBJECTIVES

This presentation will help you to analyse and comprehend the following topics followed with a link to attempt a quiz:

- The concept of functions and its advantage.
- Types of function
- Scope and Lifetime of a variable
- Types of Parameters used in functions

DEFINING A FUNCTION

- ❖ **Keyword def:** This is the keyword used to say that a function will be defined now, and the next word that is there, is the function name.
- ❖ **Function name:** This is the name that is used to identify the function. The function name comes after the def keyword.
- ❖ **Parameter list:** Parameter or Argument list are place holders that define the parameters that go into the function. The parameters help to generalise the transformation/computation/task that is needed to be done.
- ❖ **Function docstrings:** These are optional constructs that provide a convenient way for associated documentation to the corresponding function. Docstrings are enclosed by triple quotes `'''you will write the docstring here'''`
- ❖ **Function returns:** Python functions returns a value. You can define what to return by the return keyword. In case you do not define a return value, the function will return None.

TYPES OF FUNCTIONS IN PYTHON

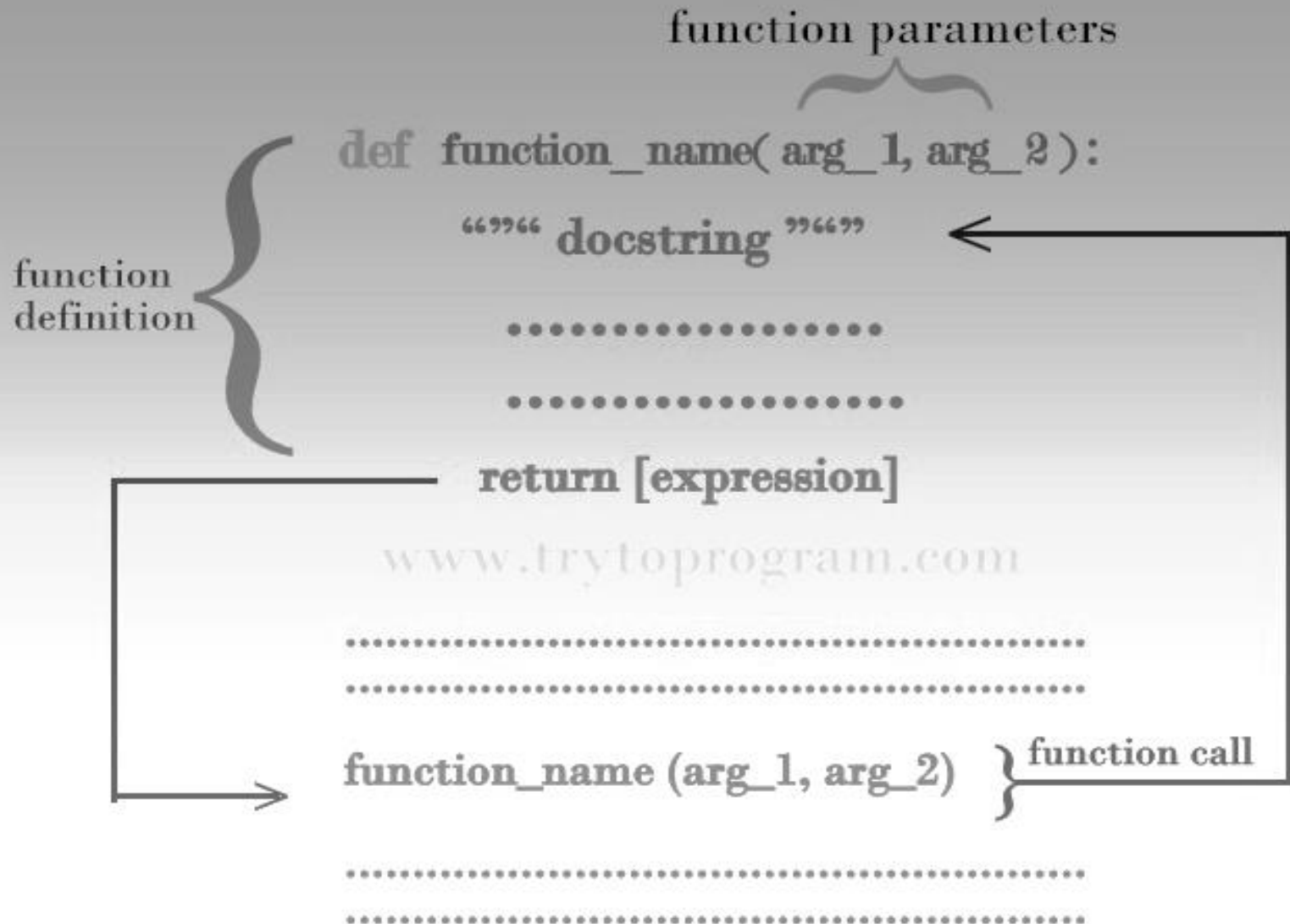
There are two types of functions in Python.

- ❖ **Built-in Functions** - These are the built-in functions of Python with pre-defined functionalities.
- ❖ **User-defined Functions** - As the name goes, these are the functions defined by the users as per the requirement at different stages. We define our own functionalities and give a name to these functions. User-defined function with return type has return statements to return values after calculation whereas, function with non-return type has no return statement.

Advantages of user-defined functions

1. User-defined functions help to decompose a large program into small segments which makes program easy to understand, maintain and debug.
2. If repeated code occurs in a program. Function can be used to include those codes and execute when needed by calling that function.
3. Programmers working on large project can divide the workload by making different functions.

WORKING OF A FUNCTION



WORKING OF A FUNCTION

```
Python10.1.py x
1  #define a function
2  def func1():
3      print("I am learning Python Function")
4
5  func1()
6  #print func1()
7  #print func1
8
9
```

Function definition

Function Call

Run Python10.1

```
"C:\Users\DK\Desktop\Python code\Python Test\Python 10\Python10
10\Python10 Code\Python10.1.py"
I am learning Python Function
```

Function output

SCOPE & LIFETIME

Scope of a variable is the portion of a program where the variable is recognized. Parameters and variables defined inside a function is not visible from outside. *Hence, they have a local scope.*

Lifetime of a variable is the period throughout which the variable exists in the memory. The lifetime of variables inside a function is as long as the function executes.

They are destroyed once we return from the function. Hence, a function does not remember the value of a variable from its previous calls.

An example to illustrate the scope of a variable inside a function

script.py

IPython Shell

```
1 def my_func():
2     x = 10
3     print("Value inside function:",x)
4
5 x = 20
6 my_func()
7 print("Value outside function:",x)
```

Output

Value inside function: 10

Value outside function: 20

This is because the variable x inside the function is different (local to the function) from the one outside. Although they have same names, they are two different variables with different scope.

Local Variables vs Global Variables

- Variables or parameters defined inside a function are called local variables as their scope is limited to the function only. On the contrary, Global variables are defined outside of the function.
- Local variables can't be used outside the function whereas a global variable can be used throughout the program anywhere as per requirement.
- The lifetime of a local variable ends with the termination or the execution of a function, whereas the lifetime of a global variable ends with the termination of the entire program.
- The variable defined inside a function can also be made global by using the global statement.

Function with Parameters

It is possible to define a function to receive one or more parameters (**also called arguments**) and use them for processing inside the function block. Parameters/arguments should be given suitable formal names. The SayHello() function is now defined to receive a string parameter called name. Inside the function, print() statement is modified to display the greeting message addressed to the received parameter.

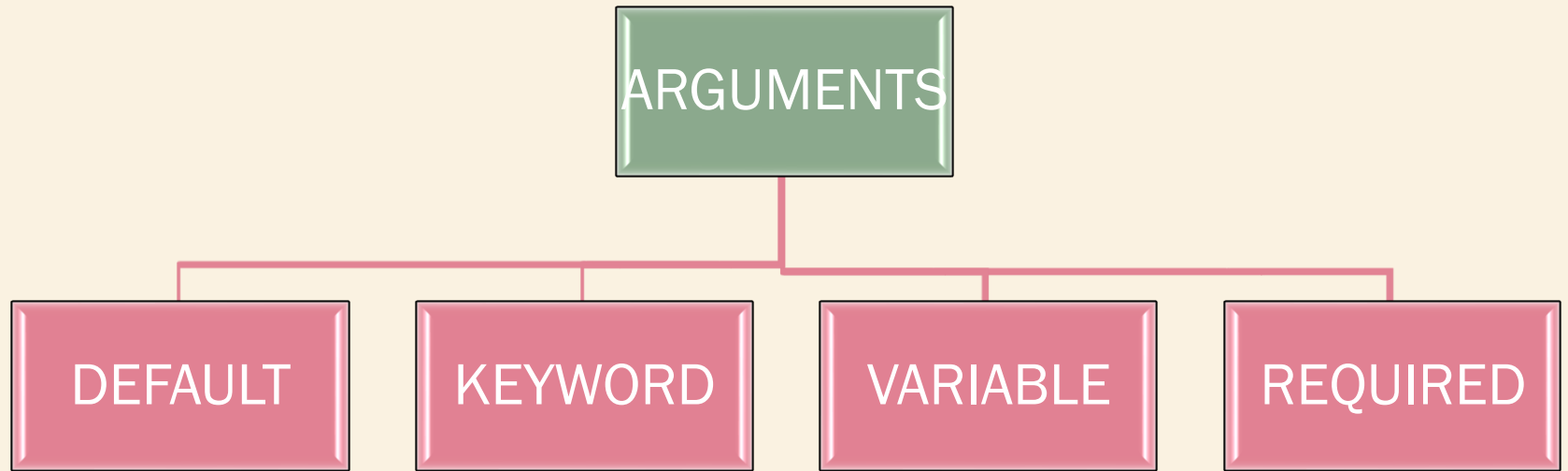
Example: Parameterized Function

```
def SayHello(name):  
    print ("Hello {}!".format(name))  
    return
```

You can call the above function as shown below.

```
>>> SayHello("Gandhi")  
Hello Gandhi!
```

TYPES OF ARGUMENTS



DEFAULT ARGUMENTS

- Default arguments are those that take a default value if no argument value is passed during the function call. You can assign this default value by with the assignment operator =, just like in the following example:

```
# Function definition is here
def printinfo( name, age = 35 ):
    "This prints a passed info into this function"
    print ("Name: ", name)
    print ("Age ", age)
    return;
```

```
# Now you can call printinfo function
printinfo( age=50, name="miki" )
printinfo( name="miki" )
```

Common Programming Errors

Using a non-default argument after default arguments raise a **Syntax Error**. In Python functions, a default argument can only be followed by a default argument. Non-default arguments must be placed before default arguments.

keyword arguments

- If you have some functions with many parameters and you want to specify only some parameters, then you can give values for such parameters by naming them this is called keyword arguments.
- We use the name instead of the position which we have been using all along.
- This has two advantages - One, using the function is easier since we do not need to worry about the order of the arguments.
- Two, we can give values to only those parameters which we want, provided that the other parameters have default argument values.

```
#!/usr/bin/python
# Filename : func_key.py

def func(a, b=5, c=10):
    print 'a is', a, 'and b is', b, 'and c is', c

func(3, 7)
func(25, c=24)
func(c=50, a=100)
```

Output

```
a is 3 and b is 7 and c is 10
a is 25 and b is 5 and c is 24
a is 100 and b is 5 and c is 50
```

Required arguments

- Required arguments are the arguments passed to a function in correct positional order. Here the number of arguments in the function call should match exactly with the function definition.
- To call the function `printme()` you definitely need to pass one argument otherwise it would give a syntax error

```
def printme( str ):
```

```
    "This prints a passed string into this function"
```

```
    print (str)
```

```
    return;
```

```
# Now you can call printme function
```

```
printme()
```

When the above code is executed, it produces the following result –

Traceback (most recent call last):

File "test.py", line 11, in <module>

printme();

TypeError: printme() takes exactly 1 argument (0 given)



```
def result(m1,m2,m3):
```

```
    ttl=m1+m2+m3
```

```
    percent=ttl/3
```

```
    if percent>=50:
```

```
        print ("Result: Pass")
```

```
    else:
```

```
        print ("Result: Fail")
```

```
    return
```

```
p=int(input("Enter your marks in physics: "))
```

```
c=int(input("Enter your marks in chemistry: "))
```

```
m=int(input("Enter your marks in maths: "))
```

```
result(p,c,m)
```


VARIABLE Arguments/ Arbitrary Arguments

- You may not always know how many arguments you'll get. In that case, you use an asterisk(*) before an argument name.

- `>>> def sayhello(*names):`

`for name in names:`

```
    print("Hello, {name}")
```

- And then when you call the function with a number of arguments, they get wrapped into a **Python tuple**. We iterate over them using the **for loop in python**.

```
>>> sayhello('Ayushi', 'Aryan', 'Megha')
```

Output:

```
Hello, Ayushi
```

```
Hello, Aryan
```

```
Hello, Megha
```

return statement

- The return statement is used to return from a function i.e. break out of the function. We can optionally return a value from the function as well.
- Every function implicitly contains a *return None* statement. You can see this by running `print someFunction()` where the function *someFunction* does not use the return statement

```
#!/usr/bin/python  
# Filename : return.py
```

```
def max(x, y):  
    if x > y:  
        return x  
    else:  
        return y
```

```
print(max(2, 3))
```

Output

3

```
def someFunction():  
    pass
```

The `pass` statement is used in Python to indicate an empty block of statements.

Pass by reference vs value

- All parameters (arguments) in the Python language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects back in the calling function.

```
def changeme( mylist ):
```

```
    "This changes a passed list into this function"
```

```
    mylist.append([1,2,3,4]);
```

```
    print ("Values inside the function: ", mylist)
```

```
    return
```

```
# Now you can call changeme function
```

```
mylist = [10,20,30];
```

```
changeme( mylist );
```

```
print ("Values outside the function: ", mylist)
```

```
Values inside the function: [10, 20, 30, [1, 2, 3, 4]]
```

```
Values outside the function: [10, 20, 30, [1, 2, 3, 4]]
```

Function example

- There is one more example where argument is being passed by reference but inside the function, but the reference is being over-written.

```
#!/usr/bin/python

# Function definition is here
def changeme( mylist ):
    "This changes a passed list into this function"
    mylist = [1,2,3,4]; # This would assign new reference in mylist
    print "Values inside the function: ", mylist
    return

# Now you can call changeme function
mylist = [10,20,30];
changeme( mylist );
print ("Values outside the function: ", mylist)
```

Output

```
Values inside the function: [1, 2, 3, 4]
Values outside the function: [10, 20, 30]
```

The Anonymous Functions

- You can use the lambda keyword to create small anonymous functions. These functions are called anonymous because they are not declared in the standard manner by using the def keyword.
- Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.
- An anonymous function cannot be a direct call to print because lambda requires an expression.
- Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.

```
>>>sum = lambda x, y, z : x + y + z
```

```
>>>sum(5, 10, 15)
```

```
30
```

The expression does not need to always return a value. It can be void.

```
>>>disp = lambda str: print('Output: ' + str)
```

```
>>>disp("Hello World!")
```

```
Output: Hello World!
```

ASSESSMENT

Click on the link given below and attempt the quiz.

<https://forms.gle/nRbf2B2C8CKRLJM37>