UNIT-III

Functions

Topics covered

FUNCTIONS

- Defining a Function,
- calling a function,
- Formal and Actual Arguments,
- Positional Arguments,
- keyword Arguments,
- Default Arguments,
- variable length arguments,
- local and global variables,
- Anonymous Functions or Lambda
- Command Line arguments

Functions

- A function is a block of organized, reusable code that is used to perform a single, related action.

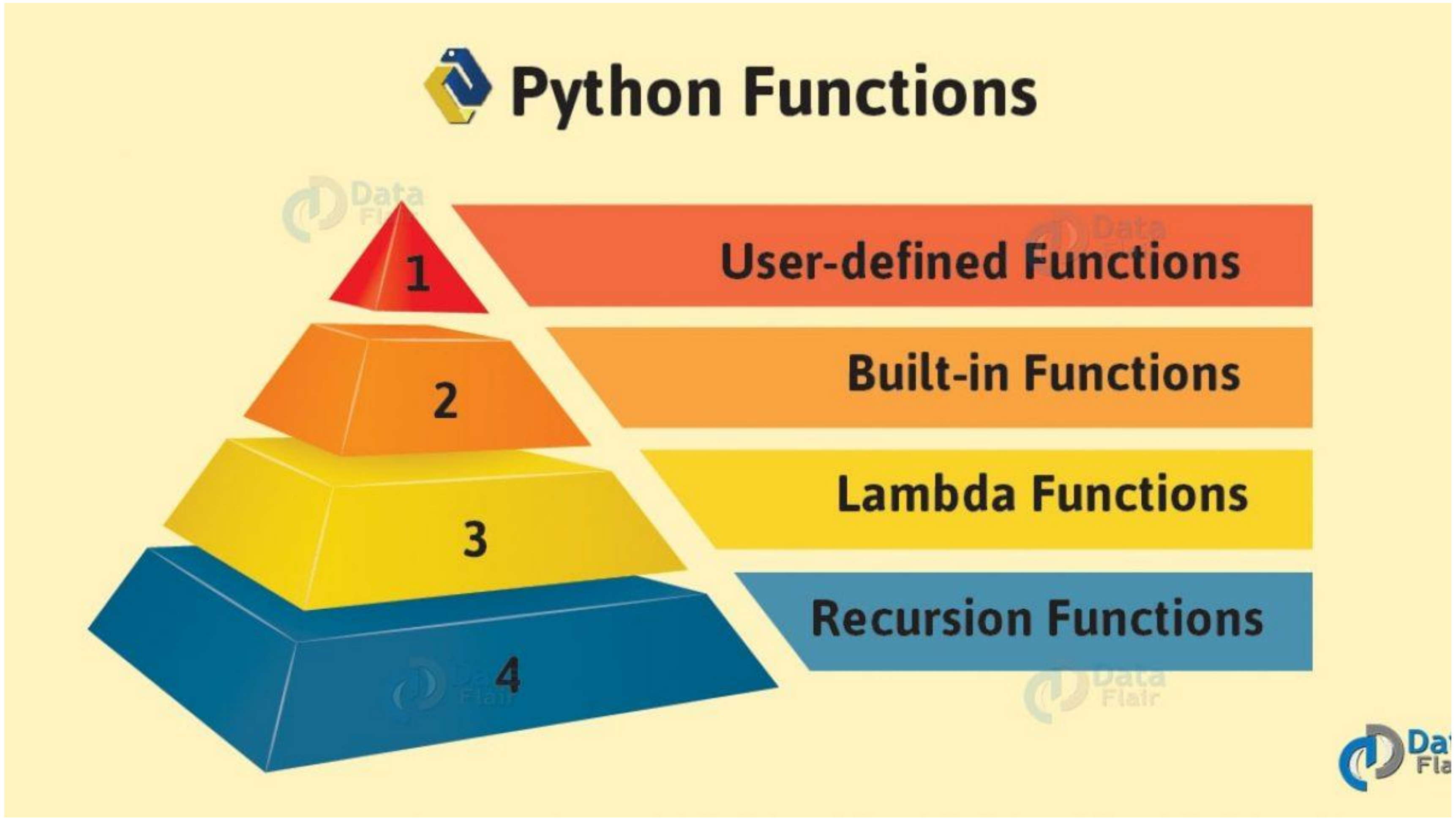
- Functions provide better modularity for your application and a high degree of code reusing.

- A function is a block of code which only runs when it is called.
- You can pass data, known as parameters, into a function.
- A function can return data as a result.

- The idea is to put some commonly or repeatedly done task together and make a function so that instead of writing the same code again and again for different inputs, we can call the function to reuse code contained in it over and over again.

- Python gives you many **built-in functions** like print(), etc. but you can also create your own functions. These functions are called ***user-defined functions***.

Functions


Python Functions
1
2
3
4
User-defined Functions
Built-in Functions
Lambda Functions
Recursion Functions

Functions

Advantage of Functions in Python

- Using functions, we can avoid rewriting the same logic/code again and again in a program.
- We can call Python functions multiple times in a program and anywhere in a program.
- We can track a large Python program easily when it is divided into multiple functions.
- Reusability is the main achievement of Python functions.
- However, Function calling is always overhead in a Python program.

Built in Functions

Function	Description
abs()	Returns the absolute value of a number
ascii()	Returns a readable version of an object. Replaces none-ascii characters with escape character
bin()	Returns the binary version of a number
bool()	Returns the boolean value of the specified object
bytearray()	Returns an array of bytes
bytes()	Returns a bytes object
chr()	Returns a character from the specified Unicode code.
compile()	Returns the specified source as an object, ready to be executed
complex()	Returns a complex number
dict()	Returns a dictionary (Array)
dir()	Returns a list of the specified object's properties and methods
divmod()	Returns the quotient and the remainder when argument1 is divided by argument2
enumerate()	Takes a collection (e.g. a tuple) and returns it as an enumerate object
eval()	Evaluates and executes an expression

Built in Functions

Function	Description
exec()	Executes the specified code (or object)
float()	Returns a floating point number
format()	Formats a specified value
globals()	Returns the current global symbol table as a dictionary
hex()	Converts a number into a hexadecimal value
id()	Returns the id of an object
input()	Allowing user input
int()	Returns an integer number
isinstance()	Returns True if a specified object is an instance of a specified object
len()	Returns the length of an object
list()	Returns a list
locals()	Returns an updated dictionary of the current local symbol table
object()	Returns a new object

Built in Functions

Function	Description
oct()	Converts a number into an octal
open()	Opens a file and returns a file object
ord()	Convert an integer representing the Unicode of the specified character
pow()	Returns the value of x to the power of y
print()	Prints to the standard output device
range()	Returns a sequence of numbers, starting from 0 and increments by 1 (by default)
reversed()	Returns a reversed iterator
round()	Rounds a numbers
set()	Returns a new set object
setattr()	Sets an attribute (property/method) of an object
slice()	Returns a slice object
sorted()	Returns a sorted list
str()	Returns a string object
sum()	Sums the items of an iterator
tuple()	Returns a tuple
type()	Returns the type of an object

User defined Functions

Syntax: **def** function_name(parameters):

```
    """docstring"""         # optional
    statement(s)
    return <expression>     #optional
```

Creating a Function

- In Python a function is defined using the **def** keyword:

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

Calling a Function

- To call a function, use the function name followed by parenthesis:

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

User defined Functions

Docstring

- The first string after the function is called the Document string or Docstring in short.
- This is used to describe the functionality of the function.
- The use of docstring in functions is optional but it is considered a good practice.

- The below syntax can be used to print out the docstring of a function:
- Syntax:

```
        print(function_name.__doc__)
```

```
def say_Hi():
    “’’Hello!  ‘“

print(say_Hi.__doc__)
```

Output: Hello!

User defined Functions

Parameters

- 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.

- You can call a function by using the following types of formal arguments-
  - Required arguments
  - Keyword arguments
  - Default arguments
  - Variable-length arguments

User defined Functions

- Information can be passed to functions as parameter.
- Parameters are specified after the function name, inside the parentheses.
- You can add as many parameters as you want, just separate them with a comma.
- The following example has a function with one parameter (fname).
- When the function is called, we pass along a first name, which is used inside the function to print the full name:

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

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

User defined Functions

Default Parameter Value

- The following example shows how to use a default parameter value.
- If we call the function ***without parameter***, 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")
```

Output:

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

User defined Functions

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.

```
#Python function to calculate the sum of two variables
#defining the function
def sum (a,b):
    return a+b;

#taking values from the user
a = int(input("Enter a: "))
b = int(input("Enter b: "))

#printing the sum of a and b
print("Sum = ",sum(a,b))
```

Output:

```
Enter a: 10
Enter b: 20
Sum = 30
```

User defined Functions

Keyword Arguments

- Keyword arguments are **related to the function calls**. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.
- This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters

```
# Python program to demonstrate Keyword Arguments
def student(firstname, lastname):
        print(firstname, lastname)


# Keyword arguments
student(firstname=‘Mani ', lastname='Pal')
student(lastname='Pal', firstname=‘Mani')
```

Output:

```
Mani Pal
Mani Pal
```

User defined Functions

Variable-length Arguments

- You may need to process a function for more arguments than you specified while defining the function. These arguments are called *variable-length arguments and are not named in* the function definition, unlike required and default arguments.

  Syntax for a function with non-keyword variable arguments is this –

```
def functionname([formal_args,] *var_args_tuple ):
    "function_docstring"
    function_suite
    return [expression]
```

- An asterisk (*) is placed before the variable name that holds the values of all **nonkeyword** variable arguments.
- This tuple remains empty if no additional arguments are specified during the function call.

User defined Functions

Variable-length Arguments

```
# Function definition is here
def printinfo( arg1, *vartuple ):
    "This prints a variable passed arguments"
    print ("Output is: " )
    print (arg1)
    for var in vartuple:
        print (var)
    return

# Now you can call printinfo function
printinfo( 10 )
printinfo( 70, 60, 50 )
```

```
Output is:
10

Output is:
70 60 50
```

User defined Functions

Variable-length Arguments

```
# Python program to illustrate
# *args for variable number of arguments
def myFun(*argv):
    for arg in argv:
        print(arg)
myFun('Hello', 'Welcome', 'to', ‘AIML')
```

Output:
```
Hello
Welcome
To
AIML
```

```
# Python program to illustrate
# *kargs for variable number of keyword arguments
def myFun(**kwargs):
    for key, value in kwargs.items():
        print("%s == %s" % (key, value))
# Driver code
myFun(first=‘Malla', mid=‘Reddy', last=‘University')
```

Output:
```
first == Malla
 mid == Reddy
 last == University
```

User defined Functions

Return Values

- To let a function return a value, use the return statement:
- Example

```
def my_function(x):
        return 5 * x

print(my_function(3))
print(my_function(5))
print(my_function(9))
```

- The return statement is used at the end of the function and returns the result of the function.
- It terminates the function execution and transfers the result where the function is called.
- The return statement cannot be used outside of the function.
- If there is no expression in the statement or the return statement itself is not present inside a function, then the function will return the **None object.**

User defined Functions

MALLA REDDY UNIVERSITY

Return Values

```
# Defining function
def sum():
    a = 10
    b = 20
    c = a+b
    return c
# calling sum() function in print statement
print("The sum is:",sum())
```

- **Output:**

```
The sum is: 30
```

```
def func(name):
    message = "Hi "+name
    return message
name = input("Enter the name:")
print(func(name))
```

```
#The function simple_interest(p, t, r) is called with the keywor
d arguments the order of arguments doesn't matter in this cas
e
def simple_interest(p,t,r):
    return (p*t*r)/100

print("Simple Interest: ",simple_interest(t=10,r=10,p=1900))
```

User defined Functions

Scope of variables

- Scope of a variable is the portion of a program where the variable is recognized
- The scopes of the variables **depend upon the location** where the variable is being declared.
- The variable declared in one part of the program may not be accessible to the other parts.

- In python, the variables are defined with the two types of scopes.
  1. **Global variables**
  2. **Local variables**
- The variable defined **outside** any function is known to have a **global scope**, whereas the variable defined **inside a function** is known to have a **local scope**.

User defined Functions

Lifetime of variables

- The lifetime of a variable is the period throughout which the **variable exits 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.

User defined Functions

Local Variable

```
def print_message():
    message = "hello !! I am going to print a message.“   # the variable message is local to the function itself
    print(message)

print_message()
print(message)                    # this will cause an error since a local variable cannot be accessible here.
```

User defined Functions

Global Variable

```
total = 0; # This is global variable.
# Function definition is here
def sum( arg1, arg2 ):  # Add both the parameters and return them."
        total = arg1 + arg2; # Here total is local variable.
        print ("Inside the function local total : ", total )
        return total;

# Now you can call sum function
sum( 10, 20 )
 print ("Outside the function global total : ", total )
```

Output:

```
Inside the function local total : 30
Outside the function global total : 0
```

User defined Functions

Global Variable

- variables outside of the function are visible from inside. They have a global scope.
- We can read these values from inside the function but cannot change (write) them. In order to modify the value of variables outside the function, they must be declared as global variables using the keyword **global**.

```
y=7
def func4():
        print(y)

func4()
```

Output:

```
7
```

```
Y=7
def func4():
                global y
                y+=1
                print(y)

func4()
```

Output:

```
8
```

Recursion Functions

- Python also accepts function recursion, which means a defined function can call itself.
- Recursion is a common mathematical and programming concept.
- It means that a function calls itself.
- This has the benefit of meaning that you can loop through data to reach a result.

In this example, tri_recursion() is a function that we have defined to call itself ("recurse"). We use the k variable as the data, which decrements (-1) every time we recurse. The recursion ends when the condition is not greater than 0 (i.e. when it is 0).

```
def tri_recursion(k):
    if(k>0):
        result = k+tri_recursion(k-1)
        print(result)
    else:
        result = 0
    return result
print("\n\nRecursion Example Results")
tri_recursion(6)
```

Output:

```
1
3
6
10
15
21
```

Python Lambda / anonymous function

- A lambda function is a small anonymous function.
- An anonymous function means that a **function is without a name**
- A lambda function can take any number of arguments, but can only have one expression.

  As we already know the **def** keyword is used to define the normal functions and the **lambda** keyword is used to create anonymous functions

- The anonymous function contains a small piece of code.
- It simulates inline functions of C and C++, but it is not exactly an **inline function**.
- The syntax to define an anonymous function is given below.

lambda arguments: expression

```
# a is an argument and a+10 is an expression which got evaluated and returned.
x = lambda a:a+10
# Here we are printing the function object
print(x)
print("sum = ",x(20))
```

output: sum = 30

Python Lambda / anonymous function

- x= lambda a: a+10
- **print**(sum = x(10))

Equivalent to

```
def x(a):
        return a+10
print(sum = x(10))
```

```
# Python code to illustrate the cube of a number
# using lambda function


def cube(x): return x*x*x

cube_v2 = lambda x : x*x*x

print(cube(7))
print(cube_v2(7))
```

Python Lambda / anonymous function

- Lambda functions **have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace**.
- We use lambda functions when we require a nameless function for a short period of time.
- In Python, we generally use it as an argument to a **higher-order function** (a function that takes in other functions as arguments). Lambda functions are used along with built-in functions like filter(), map() etc.
- **Example use with filter()**
- The filter() function in Python takes in a function and a list as arguments.
- The function is called with all the items in the list and a new list is returned which contains items for which the function evaluates to True.
- Here is an example use of filter() function to filter out only even numbers from a list.

```
# Program to filter out only the even items from a list
my_list = [1, 5, 4, 6, 8, 11, 3, 12]
new_list = list(filter(lambda x: (x%2 == 0) , my_list))
print(new_list)
```

Python Lambda / anonymous function

- **Example use with map()**
- The map() function in Python takes in a function and a list.
- The function is called with all the items in the list and a new list is returned which contains items returned by that function for each item.
- Here is an example use of map() function to double all the items in a list.

```
# Program to double each item in a list using map()
 my_list = [1, 5, 4, 6, 8, 11, 3, 12]
new_list = list(map(lambda x: x * 2 , my_list))
print(new_list)
```

Python Lambda / anonymous function

- # Function definition is here
- sum = lambda arg1, arg2: arg1 + arg2; # Now you can call sum as a function
- print ("Value of total : ", sum(10, 20))
- print ("Value of total : ", sum(20, 20))

Command Line Arguments

- The arguments that are given after the name of the program in the command line shell of the operating system are known as **Command Line Arguments**.

- Python provides various ways of dealing with these types of arguments.
- The three most common are:
- Using sys.argv
- Using getopt module
- Using argparse module

Command Line Arguments

- **Using sys.argv**
- The sys module provides functions and variables used to manipulate different parts of the Python runtime environment.
- This module provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter.

- One such variable is **sys.argv** which is a simple list structure.
- It's main purpose are:
- It is a list of command line arguments.
- len(sys.argv) provides the number of command line arguments.
- sys.argv[0] is the name of the current Python script.

Command Line Arguments

- **Using sys.argv**
- Python script for adding two numbers and the numbers are passed as command-line arguments.

```
# Python program to demonstrate  command line arguments
import sys
n = len(sys.argv)                                    # total arguments
print("Total arguments passed:", n)
print("\nName of Python script:", sys.argv[0])       # Arguments passed
print("\nArguments passed:", end = " ")
for i in range(1, n):
        print(sys.argv[i], end = " ")
Sum = 0                                              #  finding the sum
for i in range(1, n):
        Sum += int(sys.argv[i])
print("\n\nResult:", Sum)
```

MALLA REDDY
UNIVERSITY

www.mallareddyuniversity.ac.in