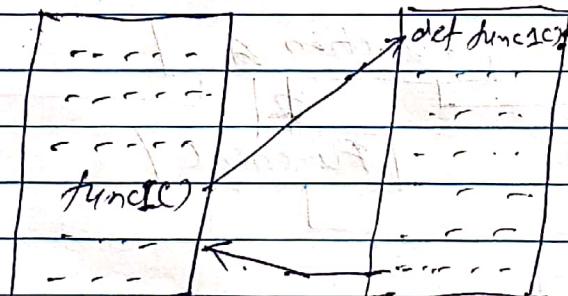


## UNIT - III.

### functions and modules

#### \* functions

python enables its programmers to break up a program into segments commonly known as functions, each of which can be written more or less independently of the others. Every function in the program is supposed to perform a well defined task.



In other words, we can say that the collection of functions creates a program. The function is also known as procedure or subroutine in other programming languages.

python provides us various inbuilt functions like range(), print(), etc.

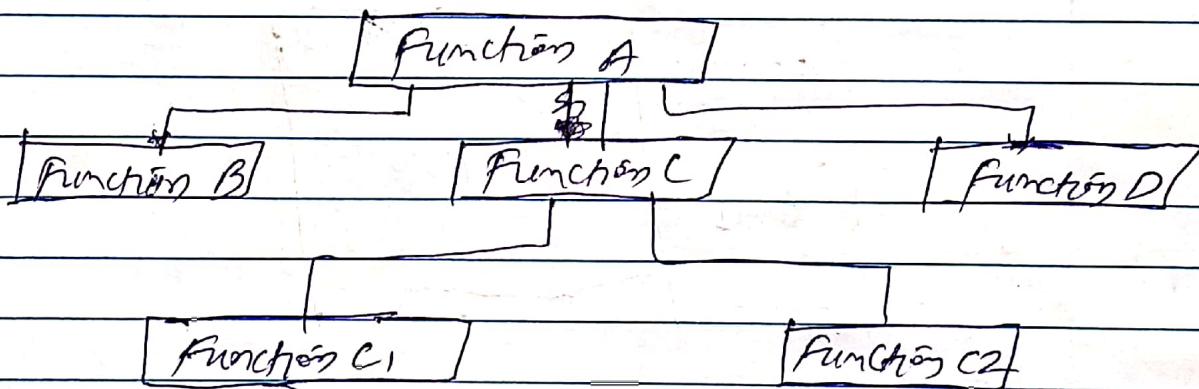
Although the user can create its functions which can be called user-defined functions.

#### \* Need for functions

Each function to be written and tested separately. Understanding, coding and testing multiple separate functions is far easier. Without the use of any function, then there will be countless lines in the code and maintaining it will be a big mess.

programmers use functions without worrying about their code details. This speeds up program development, by allowing the programmer to concentrate only on the code that he has to write.

Different python libraries, programmers can also make their functions and use them from different point in the main program or any other program that needs its functionalities.



## A function declaration and definition

- A function  $f$  that uses another function  $g$ , is known as the calling function and  $g$  is known as the called function. The inputs that the function takes are known as arguments/parameters.
- When a called function returns some result back to the calling function, it is said to return that result.
- The calling function may or may not pass parameters to the called function. If the called function accepts arguments, the calling function will pass parameters, else not.
- Function declaration is a declaration statement that identifies a function with its name, a list of arguments that it accepts and the type of data it returns.

- Function definition consists of a function header that identifies the function, followed by the body of the function containing the executable code for that function.

## \* Function Definition

Function block starts with the keyword def.

- The keyword is followed by the function name and ~~parameters~~ parentheses ( ) .
- After the parentheses a colon (:) is placed.
- Parameters or arguments that the function accept are placed within parentheses.
- The first statement of a function can be an optional ~~statement~~ statement - the ~~defining~~ docstring describe what the function does.
- The code block within the function is properly indented to form the block code.
- A function may ~~be~~ have a return [expression] statement. That is, the return statement is optional.
- You can assign the function name to a variable. Doing this will allow you to call same function using the name of that variable.

Ex

```
def diff(x,y):    # function to subtract two numbers
    return x-y
```

a = 20

b = 10

```
operation = diff    # function name assigned to a variable
print(operation(a,b)) # function called using variable name
```

Output

10

## \* function call

The function call statement invokes the function. When a function is invoked the program control jumps to the called function to execute the statements that are a part of that function. Once the called function is executed, the program control passes back to the calling function.

## Function parameters

A function can take parameters which are nothing but some values that are passed to it so that the function can manipulate them to produce the desired result. These parameters are normal variables with a small difference that the values of these variables are defined (initially initialized) when we call the function and are then passed to the function.

Function name and the number and type of arguments in the function call must be same as that given in the function definition. If the data type of the argument passed does not match with that expected in function then an error is generated.

Ex-

Syntax

function header

def function\_name(variables, variables, ...):

documentation string

Statement block { function body }

return [expression]

Ex- (1)

def func():

for i in range(4):

print("Hello world")

func() # Function call

Zeal Education Institutes

O/p

Hello world

Hello world

Hello world

Hello world

(2)

```
def func(i):
    print("Hello world", i)
func(5, 2, 3)
```

O/P

Hello world 11.

(3)

```
def total(a, b):      # Function accepting parameters
    result = a + b
    print("sum of ", a, "and", b, "=", result)
a = int(input("Enter the first number: "))
b = int(input("Enter the second number: "))
total(a, b)      # Function call with two arguments.
```

O/P

Enter the first number: 10

Enter the second number: 20

sum of 10 and 20 = 30.

(4)

```
def func(i):      # Function definition header accepts a
    print("Hello world", i)      variable with name i
i = 10
func(i)      # Function is called using variable i.
```

O/P

Hello world 10

## \* Local and Global variables.

A variable which is defined within a function is local to that function. Also, a local variable can be accessed from the point of its definition until the end of the function in which it is defined. It exists as long as the function is executing. Function parameters behave like local variables in the function. Moreover, whenever we use the assignment operator (=) inside a function, a new local variable is created.

Global variables: are those variables which are defined in the main body of the program file. They are visible throughout the program file. As a good programming habit, you must try to avoid the use of global variables because they may get altered by mistake and then result in erroneous output.

Ex

```
num1 = 10      # Global variable
print("Global variable num1 = ", num1)
def func(num2):      # num2 is function parameter
    print("In function - local variable num2 = ", num2)
    num3 = 30      # num3 is a local variable
    print("In function - local variable num3 = ", num3)
func()      # 20 is passed as an argument to the function
print("num1 again = ", num1)      # global variable is being accessed
print("num3 outside function = ", num3)
```

O/P

Global variable num1 = 10

In function - local variable num2 = 20

In function - local variable num3 = 30

num1 again = 10. Zeal Education Institutes

num3 outside function = NameError: name 'num3' is not defined

## # Using Global statement

To define a variable defined inside a function as global, you must use the global statement. This declares the local or the inner variable of the function to have module scope.

ex

var = "Good"

def show():

global var1

var1 = "morning"

print("In function var is - ", var)

show()

print("outside function, var1 is - ", var1) # accessible as it is global variable

print("var is - ", var)

O/P

In function var is - Good

In function var

outside function, var1 is - morning

var is - Good

## # Resolution of names

scope defines the visibility of a name within a block. If a local variable is defined in a block, its scope is that particular block. If it is defined in a function, then its scope is all blocks within that function. When a variable name is used in a code block,

It is resolved using the nearest enclosing scope. ~~If~~ If no variable of that name is found, then a NameError is raised. In the code given ~~the~~ below, str is a global string because it has been defined before calling the function.

Ex

def func():

```
    print(str)
    str = "Hello world!!!"
func()
```

Output (Ans)

Hello world!!!

## The return statement

The syntax of return statement is,

return [expression]

The expression is written in brackets because it is optional. If the expression is present, it is evaluated and the resultant value is returned to the calling function. However, if no expression is specified then the function will return None.

The return statement is used for two things

- Return a value to the caller.
- To end and exit a function and go back to its caller.

Ex

```
def cube(x):
    return(x * x * x)
num = 10
result = cube(num)
print('Cube of ', num, ' = ', result)
```

O/p

Cube of 10 = 1000.

## \* Types of arguments

### 1) Required Arguments

In the required arguments, the arguments are passed to a function in correct positional order. Also, the number of arguments in the function call should exactly match with the number of arguments specified in the function definition.

Ex 1

```
def display():
    print("Hello")
display("Hi")
```

O/p

Type error: display() takes  
no arguments (1 given)

Ex 2

```
def display(str):
    print(str)
display()
```

O/p

Type error: display() takes  
exactly 1 argument (0 given)

Ex (3)

```
def display(str):
    print(str)
```

str = "Hello"

display(str)

O/P

Hello.

## 2) Keyword Arguments.

When we call a function with some values, the values are assigned to the arguments based on their position. Python also allow functions to be called using keyword arguments in which the order (or position) of the arguments can be changed. The values are not assigned to arguments according to their position but based on their name (or keyword).

Keyword arguments are beneficial in two cases

- First, if you skip arguments.
- Second, if in the function call you change the order of parameters.

Ex

```
def display(str, int_x, float_y):
```

print("The string is: ", str)

print("The integer value is: ", int\_x)

print("The Floating point value is: ", float\_y)

display(float\_y = 56789.045, str = "Hello", int\_x = 1234)

## Output (Op)

The string is: Hello

The integer value is: 1234

The Floating point value is: 56789.045

### 3) Variable-length arguments

In some situations, it is not known in advance how many arguments will be passed to a function. In such cases, Python allows programmers to make function calls with arbitrary (or any) number of arguments.

When we use arbitrary arguments or variable length arguments, then the function definition use an asterisk (\*) before the parameter name. The syntax for a function using variable arguments can be given as,

```
def functionname([arg1, arg2, ...], *var args tuple):
    function statements
    return [expression]
```

Ex

```
def func(name, *var subjects):
```

```
    print("\n", name, "likes to read")
```

```
    for subject in var subjects:
```

```
        print(subject)
```

```
func("Gourav", "mathematics", "Android programming")
```

```
func("Richa", "c", "data structures", "Design and analysis of algorithms")
```

```
func("Kanish")
```

O/P

Goransh likes to read Mathematics Android programming.  
Richa likes to read C Data structures design and analysis of Algorithms.  
Kish likes to read.

#### 4) Default arguments

Python allows users to specify function arguments that can have default values. This means that a function can be called with fewer arguments than it is defined to have. That is, if the function accepts three parameters, but function call provides only two arguments, then the third parameter will be assigned the default (already specified) value. The default value to an argument is provided by using the assignment operator (=). Users can specify a default value for one or more arguments.

Ex:

```
def display(name, course = "BTech"):  
    print("Name: " + name)  
    print("course: ", course)  
display(course = "BCA", name = "Azar")  
display(name = "Reyansh")
```

O/P

Name: Azar

Course: BCA

Name: Reyansh

Course: BTech.

## \* Lambda Functions or Anonymous Functions

Lambda or anonymous functions are so called because they are not declared as other functions using the def keyword. Rather, they are created using the lambda keyword. Lambda functions are throw-away functions, i.e. they are just needed where they have been created and can be used anywhere a function is required. The Lambda feature was added to Python due to the demand from LISP programmers.

Lambda functions contain only a single line. Its syntax can be given as,

Lambda arguments : expression.

Ex

```
sum = lambda x, y : x+y
print ("Sum = ", sum(3,5))
```

O/P

Sum = 8.

## \* Documentation Strings

Docstrings (documentation strings) serve the same purpose as that of comments, as they are designed to explain code. However, they are more specific and have a proper syntax.

```
def function_name(parameters):
    "function-docstring"
    function statements
    return [expression]
```

proper syntax

Ex

`def func():`

`"""` The program just prints a message.

It will display Hello World!!! `"""`

`print("Hello World!!!")`

`print(func.__doc__)`

Output

The program just prints a message.

It will display Hello World!!!

\*

## Recursive functions

A recursive function is defined as a function that calls itself to solve a smaller version of its task until until a final call is made which does not require a call to itself. Every recursive solution has two major cases, which are as follows:

- base case, in which the problem is simple enough to be solved directly without making any further calls to the same function.
- Recursive case, in which first the problem at hand is divided into simpler sub parts.

Recursion often utilized divide and conquer technique of problem solving.

Ex

```
def fact(n):
    if (n == 1 or n == 0):
        return 1
    else:
        return n * fact(n-1)
```

```
n = int(input("Enter the value of n: "))
print("The factorial of", n, "is", fact(n))
```

### Output

Enter the value of n: 6  
The factorial of 6 is 720.

### \* Introduction to modules.

### # The from ... import statement.

A module may ~~be~~ contain definition for many variables and functions. When you import a module, you can ~~see~~ use any variable or function defined in that module. But if you want to use only selected variables or functions, then you can use the from ... import statement. For example, in the aforementioned program you are using only the path variable in the ~~sys~~ sys module, so you could have better written ~~from sys~~ import path.

#### Ex

```
from math import pi
print("PI = ", +pi)
```

### Output

PI = 3.141592653589793.

To import more than one item from a module, use a comma separated list. For example, to import the value of pi and sqrt() from the math module you can write,

from math import pi, sqrt

## # Making your own modules

Every python program is a module, that is, every file that you save as .py extension is a module.

Example:

def First write these lines in a file and save the file as mymodule.py

```
def display():
    # function definition
    print("Hello")
    print("Name of called module is: ", __name__)
str = "welcome to the world of python!!! # variable definition.
```

Then, open another file (main.py) and write the lines of code given below

```
import mymodule
print("mymodule str = ", mymodule.str) # Using variable defined in mymodule.
mymodule.display() # Using function defined in mymodule.
print("Name of calling module is: ", __name__)
```

When you run this code, you will get the following output

mymodule str = welcome to the world of python !!!  
Hello

Name of called module is: mymodule  
Name of calling module is: main.

### \* The dir() function

dir() is a built-in function that lists the identifiers defined in a module. These identifiers may include functions, classes and variables. If no name is specified, the dir() will return the list of names defined in the current module.

Ex

```
def print_var(x):  
    print(x)
```

x = 10

```
print_var(x)  
print(dir())
```

Output

10

```
['builtins', 'doc', 'file', 'name', 'package', 'print_var', 'x']
```

### \* Modules and Namespaces

A namespace is a container that provides a named context for identifiers. Two identifiers with the same name in the same scope will lead to a name clash. In simple terms, Python does not allow programmers

to have two different identifiers with the same name. However, in some situations we need to have same name identifiers. To cater to such situations, namespaces is the keyword. Name spaces Namespaces enable programs to avoid potential name clashes by associating each identifier with the namespace from which it originates.

Ex

# module 1

```
def repeat_x(x):  
    return x*2
```

# module 2

```
def repeat_x(x):  
    return x**2
```

```
import module1
```

```
import module2
```

```
result = repeat_x(10) # ambiguous reference for identifier  
repeat_x
```

## \* Local, Global, and Built-in Namespaces

During a program's execution, there are three main namespaces that are referenced - the built-in namespace, the global namespace, and the local namespace.

The built-in namespace, as the name suggests contains names of all the built-in built-in functions, constants, etc. that are already defined in python. The global namespace contains identifiers of the currently executing module and

the local namespace has identifiers defined in the currently executing function (if any).

When the python interpreter sees an identifier, it first searches the local namespace, then the global namespace, and finally the built-in namespace. Therefore, if two identifiers with same name are defined in more than one of these namespaces, it becomes masked.

Ex

```
def max(numbers):      # global namespace
    print("User Defined function max....")
    large = -1          # local namespace
    for i in numbers:
        if i > large:
            large = i
    return large
numbers = [9, -1, 4, 2, 7]
print(max(numbers))
print("Sum of these numbers = ", sum(numbers)) # built-in namespace
```

Output

USER DEFINED FUNCTION MAX....

User defined function max....

9

Sum of these numbers = 21.

## \* Module private variables

In python, all identifiers defined in a module are public by default. This means that all identifiers are accessible by any other module that imports it. But, if you want some variables or functions in a module to be privately used within the module, but not to be accessed from outside it, then you need to declare those identifiers as private.

By python identifiers whose name starts with two underscores (\_\_) are known as private identifiers. These identifiers can be used only within the module. In no way, that they can be accessed from outside the module. Therefore, when the module is imported using the import from modulename, all the identifiers of a module's namespace is imported except the private ones (ones beginning with double underscores). Thus, private identifiers becomes inaccessible from within the importing module.

## \* Packages in python

A package is a hierarchical file directory structure that has modules and other packages within it. Like modules, you can very easily create packages in python.

Every package in python is directly directory which must have a special file called init.py. This file may not even have a single line of code. It is simply added to indicate that this directory is not an ordinary directory and contains a python package.

In your programs, you can import a package in the same way as you import any module.

For example, to create a package called Mypackage, create a directory called mypackage having the module Mymodule and the init.py file. Now, to use Mymodule in a program, you must first import it. This can be done in two ways -

import Mypackage.Mymodule

or

from Mypackage import Mymodule.

\* Introduction to standard library modules

\*\* globals(), locals(), And Reload().

The globals() and locals() functions are used to return the names in the global and local namespaces (In python, each function, module, class, package, etc. owns a "namespace" in which variable names are identified and resolved). The result of these functions is of course, dependent on the location from where they are called. For example,

If locals() is called from within a function, names that can be accessed locally from that function will be returned

If globals() is called from within a function, all the names that can be accessed globally from that function is returned.

Reload() — When a module is imported into a program, the code in the module is executed only once. If you want to re-execute the top-level code in a module, you must use the `reload()` function. This function again imports a module that was previously imported.