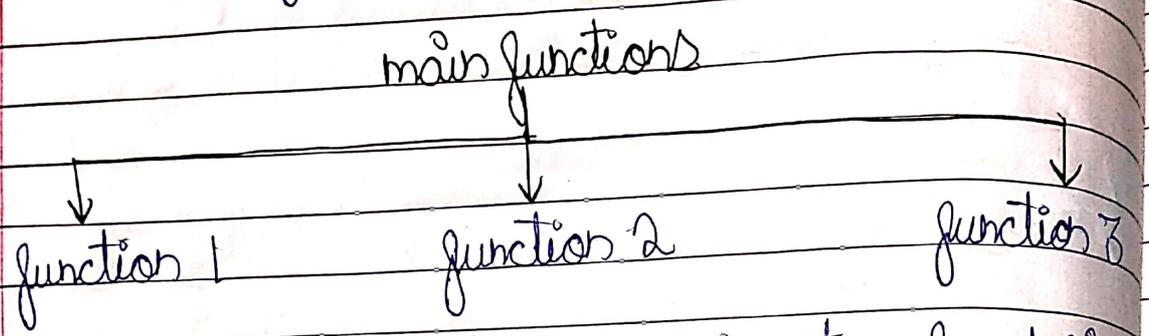


07.10.19

Unit - III Functions and Modules

Need for Functions -

- 1) In Python function helps to simplify the complex program by splitting it into smaller parts.
- 2) Python provides library functions and allows user to define their own functions.



Dividing the program into separate when defined functions facilitates each functions to be returns and tested separately. This simplifies the process of program development.

Advantages of Functions -

- 1) It is easy to combine code into small functions according to their functionalities.
- 2) As the huge code divided into functions it is to read, debug, test, maintain & modify the function individually.
- 3) Code reusability is achieved by defining the function. Any number of times the block of code with different values can be executed.
- 4) Length of program gets reduced as function is defined only once & letter whenever required can be called any number of times.

Defining Functions -

- 1) Functions are basically of two types, built-in and user defined.
- 2) Python provides built-in function, such as print etc.
- 3) But it is possible to create our own functions. These functions are called user defined functions.

Elements of Function -

- 1) Function header - It is made up of def keyword, function name and list of formal parameters enclosed in parenthesis.
- 2) Function body - It consist of local variables, declaration statements, function statements & a return statement.

- 1) Function Name - It is compulsory & it indicates the name of the functions. Function name is any valid Python identifier. Function name starts with a letter or underscore & then followed by any combination of letters, digits or underscores. Function name doesn't allow space, bar character.

- 2) List of Formal Parameter - The Formal parameter list is optional, it may be present or absent depending on the need of the function. Formal parameter list declares the variables which hold the values sent by calling function. The formal parameter in the list are separated by comma and the list is enclosed in parenthesis.

Function Body -

The function body includes a set of statements that define operation of the function. The code block within all of the respective functions start with a colon and is indented.

- 1) The first statement of a function body can be an optional statement - the documentation string of the function or docstring.
 - 2) Declaration of local variables are needed by the function.
 - 3) Executable statements that performs tasks of function.
 - 4) Returned value as the output of function.
- Local variables are variables that are defined within function and their life is only till the function and cannot be used outside the function.

Syntax -

```
def function_name (parameters):
    """ docstring """
    statement(s)
```

NOTE: def is compulsory.

09.10.
eg.

```
def my_function():  
    print("Hello friends")  
    print("This is a user defined function")
```

O/P → ~~Hello friends~~
~~This is a user defined function~~

This is definition of function and only defining a function is not sufficient, we have to call this function to execute the code.

Call to function ⇒

```
def my_function():  
    print("Hello friends")  
    print("This is a user defined function")  
my_function()
```

O/P ⇒ Hello friends
This is a user defined function

Parameters (Arguments) in function call -

1) While defining a function we can declare variables in its parenthesis. These variables are known as formal parameters. When such function is called we have to pass values for the parameters. Those values are known as actual parameters or arguments.

- 2) The method of passing data between functions is called as parameter passing.
- 3) It is possible to declare as many parameters as we want, just we have to separate them with a comma (,).

eg.

```
def test_fun(name):
    print("Welcome to Python" + name)
test_fun("Kunal")
test_fun("Shhita")
test_fun("Shrey")
test_fun("Pari")
```

↓
, also can be used

O/P ⇒
Welcome to Python Kunal
Welcome to Python Shhita
Welcome to Python Shrey
Welcome to Python Pari

Here name is formal parameter and Kunal, Shhita, Shrey, Pari are actual parameters/arguments.

Types of function arguments -

1) Default arguments

While defining a function we can set value to parameter. This value is known as default parameter value.

When such function is called if we do not pass any value for the parameter then default argument is considered.

But if argument is passed then default argument is ignored and given value is considered.

```

eg. def fun (cntry = "India"):
    print ("I am from", cntry, "country")
fun ("Japan")
fun ()
fun ("USA")
fun ("UK")

```

O/P=> I am from Japan country
 I am from India country
 I am from USA country
 I am from UK country

In the above eg while calling the function 2nd time, we do not pass argument hence default value i.e. India is considered for the cntry parameter.

2) Keyword Arguments -

- 1) In function the values which have been passed through arguments are assigned to the respective parameters in a sequence, by their position.
- 2) With keyword arguments it is possible to use the name of the parameter irrespective of its position while calling the function to provide the values.

10.10.19

3) Every keyword argument must match one of arguments excepted by the function

```
eg. def name (name 1, name 2) :  
    print (name 1, "and", name 2, "are friends")  
name (name 2 = "Kunal", name 1 = "Shrey")
```

O/P => Shrey and Kunal are friends.

By using keyword arguments, there is no need to worry regarding the position of the arguments. This makes using function easier for processing as per our need as we don't need to worry about the order of arguments.

3) Variable length arguments -

1) If we don't know in advance regarding the total no. of arguments required in function, we can use concept of variable length arguments which is also called arbitrary arguments.

2) To use this concept we have to place an asterisk (*) before a parameter in function definition which can hold non-keyworded variable length arguments and a double asterisk (***) can be placed before a parameter in function which can hold keyworded variable length arguments.

10.10.19

```
eg. def display (* name, ** address):
    for items in name:
        print (items)
    for items in address.items():
        print (items)
```

```
display ('Kunal', 'Ashita', 'Shrey', Kunal = 'Pune',
        Ashita = 'London', Shrey = 'Nasik')
```

O/P =>

```
Kunal
Ashita
Shrey
('Kunal', 'Pune')
('Ashita', 'London')
('Shrey', 'Nasik')
```

• Passing List as Argument -

It is possible to send list as a parameter to function.

```
def friends (frnd):
    for x in frnd:
        print (x)
```

```
** frnd = ["Kunal", "Shourya", "Raj"]
friends (frnd)
```

O/P =>

```
Kunal
Shourya
Raj
```

10.10.19

● Variable Scope and Lifetime -

- 1) Scope of a variable is the part of a program in which the variable is accessible. Parameters and variables defined inside a function are not accessible from outside. Hence they have a local scope.
- 2) 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 doesn't remember the value of a variable from its previous call.
- 3)

eg.

```
def var = scope():  
    x = 5 # local variable  
    print("Value inside function:", x)  
  
x = 10 # global variable  
var = scope()  
print("Value outside function:", x)
```

O/P ⇒
Value inside function : 5
Value outside function : 10

11.10.19

● Comparison between local and global variables -

Global variable	Local variable
1) They are defined in the main body of the program file.	They are defined within a function and are local to that function.
2) They can be accessed throughout the program file.	They can be accessed from the point of its definition until the end of the block in which it is defined.
3) They are accessible to all functions in the program.	They are not related in any way to other variables with the same names used outside the function.

```
num1 = 10
```

```
print("Global variable =", num1)
```

```
def func(num2):
```

```
    print("In function-Local variable num2 =", num2)
```

```
    num3 = 30
```

```
    print("In function-Local variable num3 =", num3)
```

```
func(20)
```

```
print("num1 again =", num1)
```

```
O/P => print("num3 outside function =", num3)
```

O/P => Global variable = 10

In function - Local variable num 2 = 20

In function - Local variable num 3 = 30
num 1 again = 10

Name Error: name 'num 3' is not defined

Using the 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 variables of the function to have global scope.

eg. var = "Good"

```
def show():
```

```
    global var
```

```
    var1 = "Morning"
```

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

```
show()
```

```
print("Outside function, var1 is", var1)
```

~~variable~~

```
print("var is", var)
```

O/P =>

In function var is Good

Outside function, var1 is Morning

var is Good

11.10.19

classmate

Date
Page 93

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

eg.

```
var = "Good"
def show():
    var = "Morning"
    print("In function var is -", var)
show()
print("Outside function, var is -", var)
```

O/P => In function var is - Morning
Outside function, var is - Good

* If we have a global variable and then create another global variable using the global statement then changes made in the variable will be reflected everywhere in the program.

eg.

```
var = "Good"
def show():
    global var
    var = "Morning"
    print("In function var is -", var)
show()
print("Outside function, var is -", var)
var = "Fantastic"
print("Outside function, after modification, var is -", var)
```

11.10.19

O/P \Rightarrow In function var is - Morning
Outside function var is - Morning
Outside function, after modification, var is - Fantastic

* In case of nested functions, the inner function can access variables defined in both outer as well as inner functions, but the outer function can access variables defined only in the outer function.

eg.

```
def outer_func():  
    outer_var = 10  
    def inner_func():  
        inner_var = 20  
        print("Outer variable =", outer_var)  
        print("Inner variable =", inner_var)  
    inner_func()  
    print("Outer variable =", outer_var)  
    print("Inner variable =", inner_var)  
outer_func()
```

O/P \Rightarrow Outer variable = 10
Inner variable = 20
Outer variable = 10
Name Error : name 'inner_var' is not defined.

* If a variable in inner function is defined with the same name as that of a variable defined in the outer function then a new variable is created in the inner function.

14/10/19

```

def outer_func():
    var = 10
    def inner_func():
        var = 20
        print("Inner variable =", var)
    inner_func()
    print("Outer variable =", var)
outer_func()

```

O/P => Inner variable = 20
Outer variable = 10

The return statement -

Every function has an implicit return statement as the last instruction in the function body. This implicit return statement returns nothing to its caller, so it is said to return None, where None means nothing.

But you can change this default behaviour, by explicitly using the return statement to return some value back to the caller.

Syntax

```
return [expression]
```

14.10.19

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. If no expression is specified then the function returns None.

```
eg. def ret(num):
    sq = num * num
    return sq
square = ret(10)
print("Square is :", square)
```

O/P Square is 100

```
eg. def cube(x):
    return (x * x * x)
num = 10
result = cube(num)
print("Cube of", num, "is", result)
```

O/P Cube of 10 = 1000

● 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.

14.10.19

classmate

Date

Page 97

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 demand from LISP programmers.

Syntax -

lambda arguments : expression

lambda functions contain only a single line. The arguments contain a comma separated list of arguments and the expression is an arithmetic expression that uses these arguments. The function can be assigned to a variable to give it a name.

eg. $x = \text{lambda } a : a + 10$
 $\text{print}(x(5))$

O/P 15

eg. $x = \text{lambda } x, y : x * y$
 $\text{print}(x(10, 5))$

O/P 50

eg. $x = \text{lambda } x, y, z : x + y + z$
 $\text{print}(x(3, 5, 8))$

O/P 16

14.10.19

Documentation String -

docstrings serve the same purpose as that comments as they are designed to explain the code. However, they are more specific and have a proper syntax. They are created by putting a multiline string to explain the function.

Syntax -

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

As per the syntax the first statement of function body can optionally be a string literal which is also known as docstring.

Program to swap two numbers.

```
def swap(a, b):  
    c = a  
    a = b  
    b = c  
    print("After swapping", "a = :", a)  
    print("After swapping", "b = :", b)  
swap(a, b):  
print("Enter value of a")  
print("Enter value of b")
```

14.10.19

Program to swap two numbers -

```
def swap(a, b):  
    c = a  
    a = b  
    b = c  
    print("After swapping a is", a)  
    print("After swapping b is", b)  
a = int(input("Enter value of a: "))  
b = int(input("Enter value of b: "))  
swap(a, b)
```

Program to find smaller of two nos. using lambda function.

```
def small(a, b):  
    if (a < b):  
        return a  
    else:  
        return b  
sum = lambda x, y : x + y  
diff = lambda x, y : x - y  
print("Smaller of two numbers =",  
      small(sum(-3, -2), diff(-1, 2)))
```

Program to find sum of first 10 natural numbers

```
sum = lambda a  
for a in range(1, 11):  
x = lambda : sum(range(1, 11))  
print(x())
```

15.10.19

O/P => 27

```

Imp add = lambda x, y : x + y
Multiply - and - add = lambda x, y, z : x * add(y, z)
print (Multiply - and - add (3, 4, 5))

```

- (i) As the first line it should be short & concise highlighting the summary of the object purpose.
- (ii) It should not specify information like the objects name or type.
- (iii) It should begin with a capital letter and end with a period.
- (iv) Triple quotes are used to extend the docstring to multiple lines. This docstring specified can be accessed through the `__doc__` attribute of the function.

Doc Attribute of the Function -

- (v) In case of multiple lines in the docstring the second line should be blank to separate the summary from the rest of the description.
- (vi) Unlike comments docstrings are retained throughout the runtime of the program so users can inspect them during program execution.

```

eg. def power (a, b) :
    """ Returns arg 1 raised to power arg 2 """
    return a ** b
print (power.__doc__)
print (power (5, 3))

```

O/P => Returns arg 1 raised to power arg 2.
125

Multiline docstring =>

```
def my_function (arg1):
```

```
    """
```

```
    — X — X —
```

```
    Summary line.
```

```
    Extended description of function.
```

```
    Parameters:
```

```
    arg1 (int): Description of arg1
```

```
    Returns:
```

```
    int: Description of return value
```

```
    """
```

```
    return arg1
```

```
print (my_function! — doc —)
```

O/P =>

```
    Summary line.
```

```
    Extended description of function.
```

```
    Parameters:
```

```
    arg1 (int): Description of arg1
```

```
    Returns:
```

```
    int: Description of return value
```

● Recursive Functions -

- (i) The process of calling a function inside itself is called as recursion.
The function which calls itself is referred as recursive function.
- (ii) Recursion is the process of defining something in terms of itself. It is little bit difficult to understand the recursion but in some cases it provides better solution than looping.

Advantages of Recursive Function -

- 1) It helps to reduce the time complexity of program.
- 2) It helps to reduce the length of program.
- 3) It helps code reusability.

eg.

```
def exp_rec(x, y):  
    if (y == 0):  
        return 1  
    else:
```

```
        return (x * exp_rec(x, y-1))  
n = int(input("Enter 1st no. "))  
m = int(input("Enter 2nd no. "))  
print("Result =", exp_rec(n, m))
```

O/P =>

no. of calls

$$\begin{aligned} (1) & 2 * 2 = 4 \\ (2) & 4 * 2 = 8 \\ (3) & 8 * 2 = 16 \end{aligned} \quad (2^4 = 16)$$

Explain

$$\begin{aligned} \text{exch_rec}(2, 4) &= 2 * \text{exch_rec}(2, 3) \\ \text{exch_rec}(2, 3) &= 2 * \text{exch_rec}(2, 2) \\ \text{exch_rec}(2, 2) &= 2 * \text{exch_rec}(2, 1) \\ \text{exch_rec}(2, 1) &= 2 * \text{exch_rec}(2, 0) \\ \text{exch_rec}(2, 0) &= 1 \\ \text{exch_rec}(2, 1) &= 2 * 1 = 2 \\ \text{exch_rec}(2, 2) &= 2 * 2 = 4 \\ \text{exch_rec}(2, 3) &= 2 * 4 = 8 \\ \text{exch_rec}(2, 4) &= 2 * 8 = 16 \end{aligned}$$

$2 * 2^{4-1}$
$2 * 2^{4-1}$
$2 * 2^3$
$2 * 8$
16

eg. def fact(num):
 if num == 1:
 return (1)
 else:
 f = num * fact(num-1)
 return (f)

n = int(input("Enter a number"))
 f = fact(n)
 print("Factorial value =", f)

$$5 * (5-1) * 4 * 3 * 2 * 1$$

O/P => Enter a number 5
 Factorial value = 120

● Good Programming Practice -

- 1) Instead of tabs use four-spaces for indentation.
- 2) Insert blank lines to separate functions and classes and statement blocks inside functions.
- 3) Whenever required use comments to explain the code.
- 4) Use docstrings that explains the purpose of the function.
- 5) Use spaces around operators and after commas.
- 6) Name of the classes should be written as `ClassName`.
- 7) Name of the functions should be in lowercase with underscores to separate words.
eg. `get_data()` and `display_info()`
- 8) Do not use non-ASCII characters in function names or any other identifier.

● Modules

- 1) Python provides concept of modules which helps to logically organize the Python code. In module we can group related code which makes the code easier to understand as well as use.
- 2) A module is considered as a Python object with arbitrarily named attributes which we can bind and reference.
- 3) A module is a file consisting of Python code. A module has ability to define functions, classes and variables and executable code.

4) Module is a file with .py extension.
 5) The basic way to use a module is to add import module name as the first line of your program and then writing modulename.var to access functions and values with the name var in the module.

Standard library modules.

```
import sys # sys is lib
print("\n PYTHONPATH = \n", sys.path)
```

We import the sys (system) module using the import statement to use its functionality ~~is~~ related to the Python interpreter and its environment.

• The from import statement -

If you want to use only selected variables or functions then we can use the from import statement.

```
eg. from math import pi # math is lib
print("PI = ", pi)
```

O/P => PI = 3.141592653589793

You can also import a module with different name using the as keyword. This is particularly more important when a module either has a long or confusing name.

eg. `from math import sqrt as square_root`
`print(square_root(81))`

O/P => 9.0

Name of module -

Every module has a name. You can find the name of the module by using the `__name__` attribute of the module.

eg. `print("Hello")`
`print("Name of this module is :", __name__)`

O/P => Hello

Name of this module is : `__main__`
Statements are written globally

NOTE: Always remember that for any stand alone program written by the user the name of the module is `__main__`

Making your own modules -

- 1 > You can easily create as many modules as you want.
- 2 > In fact you have already been doing that because every Python program is a module that is every file that you save as .py extension is a module.

18.10.19

MyModule.py

def display():

print("Hello")

print("Name of module is:", __name__)

str = "Welcome to the world of Python!!!"

start → # main.py

import MyModule

① print("MyModule.str =", MyModule.str)

③ MyModule.display()

⑥ print("Name of calling module is:", __name__)

OP ⇒ MyModule.str = Welcome to the world of Python!!!

Hello

Name of module is : MyModule

Name of calling module is __main__

Code in my module

MyModule

def large(a, b):

if a > b:

return a

else:

return b

Find.py

import Module

print("Large(50, 100) =", Module.large(50, 100))

print("Large('B', 'C') =", Module.large('B', 'C'))

print("Large('HI', 'BI') =", Module.large('HI', 'BI'))

O/P => Large(50, 100) = 100
 Large('B', 'C') = C
 Large('HI', 'BI') = HI

dir() function -
 dir() is a built-in-function list, list the identifiers defined in a module. These identifiers may include functions, classes and variables.

```
eg. # dir()
def print_var(x):
    print(x)
x = 10
print_var(x)
print(dir())
```

O/P => 10
 ['_annotations_', '_builtins_', '_doc_', '_file_', '_loader_', '_name_', '_package_', '_spec_', 'print_var', 'x']

If no name is specified the dir() will return the list of names defined in the current module. Import the sys package and try to dir its contents. You will see a big list of identifiers. However the dir(sys) does not list the name of built-in-functions and variables. To see the list of these write dir(builtins) as they are defined in the standard built in module.

```
import builtins
print (dir(builtins))
```

Correct syntax
of builtins

Modules and namespaces

- 1) A namespace is a container that provides a named context for identifiers. To
- 2) ~~To~~ identifiers with the same ~~or~~ name in the same scope will lead to name clash.
- 3) Python does not allow programmer to have two different identifiers with the same name.
- 4) In some situations we need to have same name identifiers.
- 5) To cater to such situations namespaces is the keyword. Namespaces enable programs to avoid potential name clashes by associating each identifier with the same name from which it originates.

```
# namespaces :
```

```
# module 1
```

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

```
# module 2
```

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

```
# module 3
```

```
import module1
import module2
```

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

Correct code / Unambiguous

module 3

```
import module 1
```

```
import module 2
```

```
result = module 1.repeat_x(10)
```

```
result = module 2.repeat_x(10)
```

O/P
10
100

~~O/P~~ In Python each module has its own namespace. This namespace includes the names of all functions and variables defined in the module.

2) There are two instances of `repeat_x()` each defined in their own module are distinguished by being fully qualified with the name of the module in which each is defined as `module 1.repeat_x` and `module 2.repeat_x`.

● Local, Global and built in namespaces -

1) The built in namespace as the name suggests contains names of all the built in functions constants etc that are already defined in Python.

2) The global namespace contains identifiers of the currently executing module and the local namespace has identifiers defined in the current executing function if any.

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

```
def max(numbers):
    print("USER DEFINED FUNCTION MAX...")
    large = -1
    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
```

NOTE: Syntax of dir

```
dir("module name")
```

Builtins

```
dir(__builtins__)
```

● Packages in Python -

- (i). A package is a hierarchical file directory structure that has modules and other packages within it. Like modules you can easily create packages in Python.
- (ii) Every package in Python is a directory which must have a special file called `__init__`. This file may not even have a single line of code.
- (iii) It is simply added to indicate that this directory is not an ordinary directory and contains a Python package.
- (iv) In your Programs you can import a package in the same way as you import any module.

For eg. `import mypackage.mymodule`

or
`from mypackage import mymodule`

```
eg. 1 # main
from apurva import MyModule
print("MyModule str = ", MyModule.str)
MyModule.display()
print("Name of calling module is: ", name)

# MyModule
def display():
    print("Hello")
    print("Name of module is: ", name)
str = "Welcome to the world of Python!!!"
```

O/P MyModule Str = Welcome to the world of Python
Hello

Name of module is : apurva.MyModule
Name of calling module is : main

Here apurva is the package which contains files
namingly main, MyModule, and (empty) init.

eg.2 import math

```
def cube(x):
    return x ** 3
```

```
a = -100
```

```
print("a=", a)
```

```
a = abs(a)
```

modified a = 100

```
print("abs(a) =", a)
```

```
print("Square root of", a, "=", math.sqrt(a))
```

```
print("Cube of", a, "=", cube(a))
```

O/P a = -100

abs(a) = 100

Square root of 100 = 10.0

Cube of 100 = 1000000

eg.3 import random

```
for i in range(10):
```

```
    value = random.randint(1, 100)
```

```
    print(value)
```

O/P

64
37
81
14
13
21
11
62
15
49

● Standard Library Modules -

- (i) Python supports three types of modules: those written by the Programmer, those that are installed from external sources and those that are preinstalled by Python.
- (ii) Modules that are preinstalled are known as the Standard library.
- (iii) Some useful modules in the Standard library are string, datetime, math, random, os, multiprocessing, subprocess, socket, email, json, doctest, writetest, path, argparse and sys, calendar etc.
- (iv) You can use these modules for performing tasks like string parsing, data serialization, testing, debugging and manipulating, dates, command line arguments etc.

```
# Time module
```

```
import time
```

```
localtime = time.localtime(time.time())  
print("Local current time :", localtime)
```

O/P Local current time : Mon Nov 04 09:52:08 2019

```
# Calendar
```

```
import calendar
```

```
print(calendar.month(2019, 11))
```

● Function redefinition -

```
import datetime
```

```
def showMessage(msg):  
    print(msg)
```

```
showMessage("Hello")
```

```
def showMessage(msg):  
    now = datetime.datetime.now()
```

```
    print(msg)
```

```
    print(str(now))
```

```
showMessage("Current Date and Time is :")
```

O/P => Hello

Current Date and Time is :

2019-11-05 11:28:29.743475