

Function and Module

UNIT: III

Problem Solving using Python
(ACSE0101)

B.Tech 1st semester



Sachin Kumar
(Asst. Professor)
IT Department



- Introduction of Function
- Calling a function
- Function arguments
- Mutability and Immutability
- Built in function
- Scope rules
- Namespaces
- Garbage Collection
- Passing function to a function
- Recursion
- Lambda functions
- Map
- Filter
- Reduce
- Introduction of Modules and Packages
- Importing Modules
- writing own modules
- Standard library modules
- dir() Function
- Packages in Python

Course Objective

- To impart adequate knowledge on the need of programming languages
- To develop programming skills using the fundamentals and basics of python language.
- **To enable effective usage of functions, module, lambda function and the packages.**
- To develop modules and import the packages .

Course Outcome

Course Outcome (CO)	At the end of course , the student will be able to:	Bloom's Knowledge Level (KL)
CO1	Analyse and implement simple python programs.	K3, K4
CO2	Develop Python programs using decision control statements.	K3, K6
CO3	Implement user defined functions and modules in python.	K2
CO4	Implement python data structures –string, lists, tuples, set, dictionaries.	K3
CO5	Perform input/output operations with files in python, apply exception handling for uninterrupted execution.	K3, K4

Engineering Graduates will be able to:

- **PO1 : Engineering Knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals and an engineering specialization to the solution of complex engineering problems.
- **PO2 : Problem Analysis:** Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences and engineering sciences.

- **PO3 : Design/Development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate considerations for the public health and safety, and the cultural, societal and environmental considerations.
- **PO4 : Conduct Investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data and synthesis of the information to provide valid conclusions.

- **PO5 : Modern tool usage:** Create, select and apply appropriate techniques, resources and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations
- **PO6 : The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and consequent responsibilities relevant to the professional engineering practice.

- **PO7 : Environment and sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development
- **PO8 : Ethics:** Apply the ethical principles and commit to professional ethics, responsibilities, and norms of engineering practice.
- **PO9 : Individual and teamwork:** Function effectively as an individual, and as a member or leader in diverse teams and multidisciplinary settings.

Program Outcomes (PO's)

- **PO10 : Communication:** Communicates effectively on complex engineering activities with the engineering community and with society such as being able to comprehend and write effective reports and design documentation, make effective presentations and give and receive clear instructions.
- **PO11 : Project management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in team, to manage projects and in multidisciplinary environments.
- **PO12 : Life-long learning:** Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadcast context of technological change.

CO-PO Mapping

CO.K	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
CO1	3	3	3	3	2	2	1	-	1	-	2	2
CO2	3	3	3	3	2	2	1	-	1	1	2	2
CO3	3	3	3	3	3	2	2	-	2	1	2	3
CO4	3	3	3	3	3	2	2	1	2	1	2	3
CO5	3	3	3	3	3	2	2	1	2	1	2	2
AVG	3.0	3.0	3.0	3.0	2.6	2.0	1.6	0.4	1.6	0.8	2.0	2.4

Unit Objective

- To develop and use of functions in the python.
- To implement lambda functions and its advantage in the python
- To import the functions from module.
- To understand and installation of packages.

Prerequisite and Recap

- Conditional statement
- Nested if else
- loops

*Let's
Recap*

- Conditional statement:

Syntax:

- if condition :
 statement
else:
 statement

- Loops

Syntax:

- for n in range():

*Let's
Recap*

Function and Modules(CO3)

Introduction to
functions

Arguments and
calling functions

Built in functions

Scope rules

Passing function
to the functions

Recursion

Lambda
Functions

Modules and
importing
modules

Writing own
modules

Standard Library
Modules

Packages in python

Objective:

- To develop and use of functions in the python.
- To implement lambda functions and its advantage in the python.

Introduction to Functions (CO3)

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.

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

Introduction to Functions (CO3)

Type of Functions(CO3)

Basically, we can divide functions into the following two types:

- **Built-in functions** - Functions that are built into Python.

Example:

`print(),input(),sum(),abs(),pow(), sqrt()`etc.

- **User-defined functions** - Functions defined by the users themselves.

Example:

```
def function():  
    print('hello')
```

- **Rules to define a function in Python:**

- Function blocks begin with the keyword **def** followed by the function name and parentheses ().
- Any input parameters or arguments should be placed within these parentheses.
- The first statement of a function can be an optional statement .
- The code block within every function starts with a colon (:) and is indented
- The statement **return [expression]** exits a function, optionally passing back an expression to the caller
- `return[expression]` can return value or can be empty.
- If `return` statement itself is not present inside a function, then the function will return the `None` object.

Syntax to Define function:(CO3)

- Syntax:

```
def functionname( parameters ):
```

```
    """function_docstring"""
```

```
    function_suite
```

```
    return [expression]
```

- Example:

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

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

```
    print (str)
```

```
    return
```

Docstring:

- Python docstrings are the string literals that appear right after the definition of a function, method, class, or module.
- They are used to document our code.
- We can access these docstrings using the `__doc__` attribute.
- For example:

```
def square(n):
```

```
    '''Takes in a number n, returns the square of n'''
```

```
    return n**2
```

```
print(square.__doc__)
```

Calling to the function:(CO3)

- Once the basic structure of function is finalized function can be executed by calling it from the function or directly from python command prompt:

Example:

function definition

```
def printme( str):  
    "This print in the function"  
    print( str)  
    return
```

#Now call of function

```
printme(" Welcome to the world of python")
```

Calling to the function:(CO3)

```
def absolute_value(num):
```

```
    """This function returns the absolute value of the entered  
    number"""
```

```
    if num >= 0:
```

```
        return num
```

```
    else:
```

```
        return -num
```

```
print(absolute_value(2))
```

```
print(absolute_value(-4))
```

Output:

2

4

Pass by reference vs value (CO3)

- All parameters (arguments) in the Python language are passed by reference. It means ,the change also reflects in the calling function .

Function definition is here

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

```
    mylist[1]='123'
```

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

```
    return
```

Now you can call changeme function

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

```
print("before calling values are", mylist)
```

```
changeme( mylist )
```

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


Pass by reference vs value(CO3)

In case of immutable data types changes will not be reflected or raise error

Function definition is here

```
def changeme( mylist ):  
    mylist[1]='123'  
    print ("Values inside the function: ", mylist)  
    return
```

Now you can call changeme function

```
mylist = (10,20,30)  
print("before calling values are", mylist)  
changeme( mylist )  
print ("Values outside the function: ", mylist)
```

will raise error

Note: Pass by reference(CO3)

Passing objects by reference has two important conclusions:

- 1) The process is faster than is copies of objects were passed.
- 2) Mutable objects that are modified in functions are permanently changed.

Arguments/parameters:

- It is the information that are passed into a function.

Actual argument/parameter

- Arguments are passed to the functions at the time of calling.

Formal arguments/parameter

- Arguments are passed to the functions at the time of definition.

We can call a function by using the following types of formal arguments –

- Required arguments/Positional arguments
- Keyword arguments
- Default arguments
- Variable-length arguments

Required Arguments(CO3)

- Required arguments are the arguments passed to a function in correct positional order
- The number of arguments in the function call should match exactly with the function definition.

Example:

Function definition is here

```
def printme( str ):  
    "This prints a passed string into this function"  
    print (str)  
    return
```

Now you can call printme function

```
printme()
```

Note: at call of printme() will give syntax error

Keyword Arguments(CO3)

- Keyword arguments are related to the function calls
- The caller identifies the arguments by the parameter name.
- This allows to skip arguments or place them out of order because the Python interpreter can use the keywords provided to match the values with parameters.

Note:

A non keyword/positional argument can be followed by keyword argument, but keyword argument can not be followed by non keyword/positional argument

Example Keyword Arguments(CO3)

- # Function definition is here
def printinfo(name, age):
 "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="Ram")

Output

Name: Ram
Age 50

Default Arguments(CO3)

- A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument.

- 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="Ram" )
```

```
printinfo( name ="Ram")
```

Output

Name: Ram

Age 50

Name: Ram

Age 35

Variable-length Arguments(CO3)

- To process a function for more arguments than 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
- `*(var_name)` creates tuple.

Syntax:

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

Note: An asterisk (*) is placed before the variable name that holds the values of all non keyword variable arguments.

Example Variable-length Arguments(CO3)

- # 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

Output is:
10
Output is:
70
60
50

Scope of Variables(CO3)

- The scope of a variable determines the portion of the program where you can access a particular identifier.
- There are two basic scopes of variables in Python:
 - **Global variables**
 - **Local variables**
- Variables that are defined inside a function body have a local scope, and those defined outside have a global scope.
- local variables can be accessed only inside the function in which they are declared, whereas global variables can be accessed throughout the program body by all functions.

Global vs local(CO3)

EXAMPLE:

- `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
```

`# 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

- It is combination of name (refers to the name of object) and space (location from where object is accessed).
- A namespace is a collection of currently defined symbolic names along with information about the object that each name references.
- A namespace can be as a dictionary in which the keys are the object names, and the values are the objects themselves.
- Each key-value pair maps a name to its corresponding object.

In a Python program, there are four types of namespaces:

- Built-In
- Global
- Enclosing
- Local

1. The Built-in namespace:

- The **built-in namespace** contains the names of all of Python's built-in objects.
- These are always available when Python is running.
- we can list the objects in the built-in namespace with the following command:

```
>>> dir(__builtins__)
```

Note:

- 1. The Python interpreter creates the built-in namespace when it starts up.**
- 2. This namespace remains in existence until the interpreter terminates.**

2. The Global namespace:

- The **global namespace** contains any names defined at the level of the main program.
- Python creates the global namespace when the main program body starts, and it remains in existence until the interpreter terminates.

```
x = 10                # x has global namespace  
def myfun():  
    print(x)  
myfun()
```

- Any variable defined inside a function using **global** keyword has also global namespace.

#Example

```
x = 10
```

```
def myfun():
```

```
    global x    # defined x using global keyword
```

```
    x = 12      # access the global variable, does not creates the local
```

```
variable
```

```
    print("Inside function definition",x)
```

```
myfun()
```

```
print("Outside function definition",x)
```

Output

```
Inside function definition 12
Outside function definition 12
```

3. Local namespace

- The **local namespace** contains any names defined at the block or function level of the main program.
- Python creates the local namespace when the execution of function body starts and terminates after the execution of body of function.

#Example

```
x = 10
```

```
def myfun():
```

```
    x = 12      # Creates the local variable x
```

```
    print("Inside function definition",x)
```

```
myfun()
```

```
print("Outside function definition",x)
```

Output

```
Inside function definition
12
Outside function definition
10
```


4. Enclosing namespace

- It refers to the definition of inner() function and other statement nested inside the definition of outer() function.
- It occurs only inside the nested function.
- inner() function can access the enclosing namespace variable (**also known as the non-local variable with respect to the inner() function**) but can't modify it.
- If inner() function try to modify the enclosing variable, then a local variable is declared inside the inner() function namespace.

Namespaces in Python (CO1)

Example of enclosing namespace

x = 10

def outer():

 x = 12 # Enclosing namespace starts here, x is defined inside the enclosing namespace

 def inner():

 print("Inside the inner function", x) # It refers to the enclosing namespace variable

 inner()

 print("Inside outer definition", x) # Enclosing namespace ends here
outer()

print("Outside function definition", x)

Inside function definition 12
Inside outer definition 12
Outside function definition 10

Namespaces in Python (CO3)

Example of inner() trying to modify the enclosing variable

x = 10

def outer():

 x = 12 # Enclosing namespace starts here, x is defined inside the enclosing namespace

 def inner():

 x = 7 # x now declared as local variable of inner

 print("Inside the inner function", x) # It refers to the enclosing namespace variable

 inner()

 print("Inside outer definition", x) # Enclosing namespace ends here

outer()

print("Outside function definition", x)

Output

Inside function definition 7
Inside outer definition 12
Outside function definition 10

Namespaces in Python (CO3)

Example of inner() function modifying the enclosing variable using non-local keyword.

```
x = 10          # x is declared in global namespace

def outer():
    x = 12      # Enclosing namespace starts here, x is defined inside the enclosing namespace
    def inner():
        nonlocal x  # x is defined using global keyword
        x = 7       # x now refers to enclosing variable of outer() function
        print("Inside the inner function", x)  # It refers to the enclosing namespace variable
    inner()
    print("Inside outer definition", x)  # Enclosing namespace ends here
outer()
print("Outside function definition", x)
```

Output

```
Inside function definition 7
Inside outer definition 7
Outside function definition 10
```

Namespace resolution rules.

1. First the environment looks for the variable (object) in local namespace.
2. If variable is not declared inside the local namespace, then it looks for the variable defined inside the enclosing namespace.
3. If variable is not declared inside the enclosing namespace, then it looks for the variable defined inside the global namespace.
4. If variable is not defined inside the global namespace, then it looks for the variable (object) in builtins namespace.
5. Finally, If variable is not present in builtins namespace, then “NameError” exception is raised by the environment.

Example: Namespaces (CO3)

```
a = 'global a'  
y = 'global y'
```

Global

```
def test_namespace():  
    a = 'enclosing a'
```

Enclosing

```
    def inner_namespace():  
        a = 'local a'  
        print(a)  
        print(y)
```

Local

```
    inner_namespace()
```

```
    print(a)
```

```
test_namespace()
```

```
print(a)
```

```
local a  
global y  
enclosing a  
global a
```

Garbage Collection (CO3)

- It is the process by Python periodically reclaims unwanted memory.
- Garbage collection in Python is automatic i.e. it deletes all the objects that are not needed or have gone out of scope to free the memory space.
- It runs in the background during the program execution.
- It immediately reclaims its memory as soon as the object's reference count reaches to zero.

- Object's reference count increases when it is created, and its aliases are created.
- That is, when object is assigned a new name, or it is referenced within in a list, tuple, or other data structure.
- The object's reference count decreases when it is being assigned to some other reference or its reference goes out of scope.
- The object' reference count can be decreased manually by deleting it with the **del** command.

Example

A = 100	# Creates object A
B = A	# Object's reference count increases by 1-object assigned
C = [1,2,B]	# Object's reference count increases by 1-object used in list
B = 200	# Object's reference count decreases by 1-Reassignment of B
C[2] = 3	# Object's reference count decreases by 1- not used in list
del A	# Object's reference count is zero – object removed from
memory	

Function as object

- Function variable (name of the function) is the object of function class , that can be reassigned to another function variable.

Function as object:::

```
def fun(text):  
    return text + ' students'
```

Output

```
hello students  
hi students
```

```
print(fun('hello'))
```

```
fun_obj = fun # function treated as object. fun_obj refers to fun() function.
```

```
print(fun_obj('hi'))
```

Defining Functions Inside other Functions(CO3)

- Function defined inside the function is also called inner function.

defining function inside function

```
def plus_one(number):  
    def add_one(num):  
        return num + 1  
    result = add_one(number)  
    return result  
  
# call of function  
print(plus_one(4))
```

Output

5

Passing Functions as Arguments to other Functions

- Functions can also be passed as parameters to other functions. Functions that can accept other function as argument are also called **Higher-order functions**.

Passing Functions as Arguments to other Functions

```
def one(msg):  
    return msg + '1'  
  
def two(msg):  
    return msg + '2'  
  
def three(fun):  
    x=fun("Message inside")  
    print(x)  
  
three(one)  
three(two)
```

Output

```
Message inside 1  
Message inside 2
```

Functions Returning other Functions(CO3)

Defining Functions returning other Functions

```
def hello_function():  
    def say_hi():  
        return "Hi"  
    return say_hi  
hello = hello_function()  
  
print(hello())
```

Output

Hi

Recursion(CO3)

- A recursive function is a function defined in terms of itself via self-referential expressions.
- It means that the function will continue to call itself and repeat its behaviour until some condition is met to return a result.
- All recursive functions share a common structure made up of two parts: base case and recursive case

Demonstration of Recursion(CO3)

- **Recursive function for calculating n!**
 1. Decompose the original problem into simpler instances of the same problem:
$$n! = n \times (n-1) \times (n-2) \times (n-3) \cdots \times 3 \times 2 \times 1$$
$$n! = n \times (n-1)! \text{ (Recursive case)}$$
 2. Recursive case repeated till than it reaches base case:
$$n! = n \times (n-1)!$$
$$n! = n \times (n-1) \times (n-2)!$$
$$n! = n \times (n-1) \times (n-2) \times (n-3)! \cdots$$
$$n! = n \times (n-1) \times (n-2) \times (n-3) \cdots \times 3 \times 2!$$
$$n! = n \times (n-1) \times (n-2) \times (n-3) \cdots \times 3 \times 2 \times 1! \text{ (1! Is base case)}$$

Demonstration of Recursion(CO3)

- def factorial_recursive(n):
 # Base case: $1! = 1$
 if n == 1: return 1
 # Recursive case: $n! = n * (n-1)!$
 else:
 return n * factorial_recursive(n-1)

#Now call of function
factorial_recursive(5)

Output

120

The *Anonymous* Functions (CO3)

- These functions are called anonymous because they are not declared in the standard manner by using the *def* keyword
- **lambda** keyword is used.
- Lambda is inline function that do not have **name**.
- Lambda forms can take any number of arguments but return just one value in the form of an expression.
- An anonymous function cannot be a direct call to print because lambda requires an expression.
- Lambda functions have their own local namespace.
- It can access variables in their parameter list and those in the global namespace.

The *Anonymous* Functions(CO3)

- **Syntax:**

lambda [arg1 [,arg2,.....argn]]:expression

- **Example:**

Function definition is here

sum = lambda arg1, arg2: arg1 + arg2;

Now you can call sum as a function **Output**

print ("Value of total : ", sum(10, 20))

print ("Value of total : ", sum(20, 20)),

Value of total : 30

Value of total : 40

The *Anonymous* Functions(CO3)

Using lambda square the input value:

```
def square(x):
```

```
    return x*x
```

```
lambda_sq=lambda x: x * x
```

```
print("using regular function:",square(3))
```

```
print("using lambda function ",lambda_sq(4))
```

Output

```
using regular function : 9  
using lambda function : 16
```

map() Function (CO3)

- It is a built in function that process all the items in an iterable without using an explicit for loop, also called mapping.
- It is useful when it is required to apply a transformation into a new iterable.
- It supports functional programming style in python.
- Syntax:
`map(function, iterable1[, iterable2, iterable3, ,iterableN])`
- map() applies **function** on each item in iterable and returns a new iterator that yields transformed items on demand.

map() Function (CO3)

```
# Example: WAP to square the each element of the list
# using map() function, two things 1. function 2. iterable
def mysquare(x):
    return x**2
mylist = [10,11,12,13, 14]
mob=map(mysquare, mylist) # mob is the map object
mylist1 = list(mob)      # convert the map object into a list
print("Original List :", mylist)
print("New list :", mylist1)
```

Output

```
Original List : [10, 11, 12, 13, 14]
New list : [100, 121, 144, 169, 196]
```

map() Function (CO3)

Example: WAP to square the each element of the list using map() function
Also use the lambda function to square each number of list.

```
mylist = [10,11,12,13,14]  
mylist1 = list(map(lambda x:x**2,mylist))  
print("Original List :", mylist)  
print("New List :", mylist1)
```

Output

```
Original List : [10, 11, 12, 13, 14]  
New list : [100, 121, 144, 169, 196]
```

reduce() Function (CO3)

- The reduce() function in Python takes in a function and a list as an argument.
- The function is called with a lambda function and an iterable and a new reduced result is returned.
- This performs a repetitive operation over the pairs of the iterable. The reduce() function belongs to the *functools* module.

reduce() Function (CO3)

Example : WAP to compute the sum of all numbers in the list

```
from functools import reduce
```

```
li = [1, 2, 3, 4, 5]
```

```
print("Sum :",reduce(lambda a, b : a+b, li))
```

Output

Sum : 15

reduce() Function (CO3)

Example : WAP to find the maximum element in the list.

```
from functools import reduce
```

```
li = [11,21,53,42,51]
```

```
print("Maximum :",reduce(lambda a, b : a if a>b else b, li))
```

Output

Maximum : 53

Quiz-Function(CO3)

1. What is the output of the following function call?

```
def fun1(name, age=20):  
    print(name, age)
```

```
fun1('NIET', 25)
```

- a) NIET 25
- b) NIET 20

Ans
Option a

2. Select which is true for Python function

- a) A Python function can return only a single value
- b) A function can take an unlimited number of arguments.
- c) A Python function can return multiple values
- d) Python function doesn't return anything unless and until you add a return statement

Ans:
Option a ,b
and c

3. Select which true for Python function

- a) A function is a code block that only executes when it is called.
- b) Python function always returns a value.
- c) A function only executes when it is called and we can reuse it in a program
- d) Python doesn't support nested function

Quiz-Function(CO3)

4. What is the output of the following displayPerson() function call

```
def displayPerson(*args):
```

```
    for i in args:
```

```
        print(i)
```

```
displayPerson(name="Emma", age="25")
```

a) TypeError

b) Emma

25

c) name

age

d) None

5. What is the output of the following display() function call?

```
def display(**kwargs):
```

```
    for i in kwargs:
```

```
        print(i)
```

```
display(emp="Kelly", salary=9000)
```

a) TypeError

b) Kelly

9000

c) ('emp', 'Kelly')

('salary', 9000)

d) emp

salary

Ans

Option a

Ans:

Option d

Weekly Assignment(CO3)

- Explain functions in python with example.
- Write a python function to check number passed as argument is perfect number or not.
- Python program to check string is palindrome or not using functions.
- Explain keyword and default argument in function with example.
- Write a function in python to develop lambda function to swap two numbers.
- Explain lambda function.
- Python program to demonstrate the variable length arguments in functions.
- Python program to count the even, odd numbers in a given array of integers using lambda function.

Prerequisite and Recap(CO3)

- **Functions**
- **Lambda functions**
- **Passing function to the functions**

*Let's
Recap*

Prerequisite and Recap(CO3)

- **Functions**
 - Arguments and its use
 - Scope rules
- **Lambda functions**

Syntax:

`lambda [arg1 [,arg2,.....argn]]:expression`

*Let's
Recap*

Topic objective(CO3)

Objective:

- To import the functions from module.
- To understand and installation of packages.

- A module allows to logically organize Python code.
- A module makes the code easier to understand and use.
- A module is a Python object with arbitrarily named attributes that can bind and reference.
- A module is a file consisting of Python code.
- A module can define functions, classes and variables.
- A module can also include runnable code.

import Module(CO3)

- Python source file as a module by executing an import statement in some other Python source file.
- The *import* has the following syntax –
 - `import module1[, module2[,... moduleN]`

Hello.py

```
def SayHello(name):  
    print("Hello", name,"! How are you?")  
    return
```

Output

Hello Students! How are you?

Import module hello

```
import Hello  
Hello.SayHello("Students")
```

The *from...import* Statement(CO3)

- Python's *from* statement lets you import specific attributes from a module into the current namespace.
- The *from...import* has the following syntax –
 - **from modname import name1[, name2[, ... nameN]**

For example,

To import the function Fibonacci from the module fib, use the following statement

from fib import fibonacci

- When you import a module, the Python interpreter searches for the module in the following sequences –
 - The current directory.
 - If the module isn't found, Python then searches each directory in the shell variable PYTHONPATH.
 - If all else fails, Python checks the default path.

Note: The module search path is stored in the system module `sys` as the **`sys.path`** variable. The `sys.path` variable contains the current directory, PYTHONPATH, and the installation-dependent default.

Renaming the Imported Module(CO3)

- use the as keyword to rename the imported module as shown below:

Example:

```
import math as cal  
cal.log(4)
```

Output

1.3862943611198906

dir() function(CO3)

- The dir() function returns all properties and methods of the specified object, without the values.
- This function will return all the properties and methods, even built-in properties which are default for all object.

Syntax:

`dir(object)`

Note: Object that want to see the attributes

Example:

`dir(math)`

`dir(io)`

Standard library module's function(CO3)

- The Python interpreter has several built-in functions.
- They are loaded automatically as the interpreter starts and are always available.
- Built-in modules are written in C and integrated with the Python interpreter
- Each built-in module contains resources for certain system-specific functionalities such as OS management, disk IO, etc
- Some of listing is as follows:
 - os module
 - math module
 - statistics module
 - random module

OS Module(CO3)

- The OS module in Python provides functions for creating and removing a directory (folder), fetching its contents, changing and identifying the current directory etc.
- Creating Directory
 - mkdir()** function create directory
- Removing a Directory
 - rmdir()** function removes directory
- List Files and Sub-directories
 - listdir()** function returns the list of all files and directory.
- Changing the Current Working Directory
 - chdir()** function to change current directory.

sys Module(CO3)

- The sys module provides functions and variables used to manipulate different parts of the Python runtime environment.
- **sys.argv**
 - returns a list of command line arguments passed to a Python script
- **sys.exit**
 - This causes the script to exit back to either the Python console or the command prompt
- **sys.path**
 - This is an environment variable that is a search path for all Python modules.

- Some of the most popular mathematical functions are defined in the math module.
- These include trigonometric functions, representation functions, logarithmic functions, angle conversion functions, etc.

- Example are:

`math.pow()`

`math.log()`

`math.log10()`

`math.exp()`

- The statistics module provides functions to mathematical statistics of numeric data.
- The following popular statistical functions are defined in this module.
 - mean()
 - median()
 - mode()
 - stdev()

random module(CO3)

- Functions in the **random** module depend on a pseudo-random number generator function `random()`, which generates a random float number between 0.0 and 1.0.
- The following popular random functions are defined in this module.
 - `random.random()`
 - `random.randint ()`
 - `random.randrange()`
 - `random.choice()`
 - `random.shuffle()`

Creating and Installing Packages in Python(CO3)

- A package in Python takes the concept of the modular approach to next logical level..
- A package can contain one or more relevant modules. Physically, a package is actually a folder containing one or more module files.
- **Steps to create packages:**
 - Create a new folder named D:\MyApp.
 - Inside MyApp, create a subfolder with the name 'mypackage'.
 - Create an empty `__init__.py` file in the mypackage folder.
 - Using a Python-aware editor like IDLE, create modules `greet.py` and `functions.py` with following code:

More in Creating and Installing Packages in Python(CO3)

greet.py

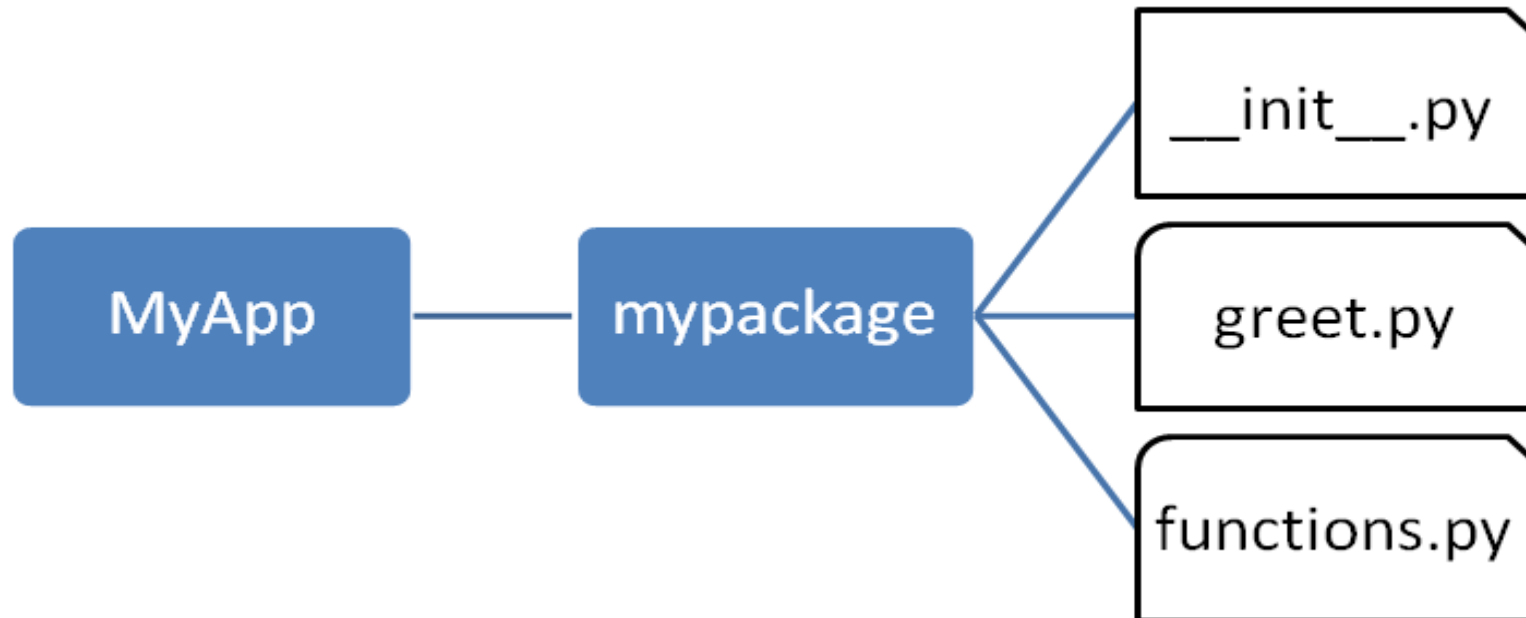
```
def SayHello(name):  
    print("Hello " + name)  
    return
```

functions.py

```
def sum(x,y):  
    return x+y  
  
def average(x,y):  
    return (x+y)/2  
  
def power(x,y):  
    return x**y
```

More in Creating and Installing Packages in Python (CO3)

We have created our package called mypackage. The following is a folder structure:



Importing a Module from a Package(CO3)

- Invoke the Python prompt from the MyApp folder.

```
D:\MyApp>python
```

- Import the functions module from the mypackage package and call its power() function

```
from mypackage import functions
```

```
>>> functions.power(3,2)
```

```
9
```

- It is also possible to import specific functions from a module in the package

```
from mypackage.functions import sum
```

```
>>> sum(10,20)
```

```
30
```

`__init__.py`(CO3)

- The package folder contains a special file called `__init__.py`, which stores the package's content.
- It serves two purposes:
 - The Python interpreter recognizes a folder as the package if it contains `__init__.py` file.
 - `__init__.py` exposes specified resources from its modules to be imported.
- Note that `__init__.py` is essential for the folder to be recognized by Python as a package. You can optionally define functions from individual modules to be made available.

__init__.py(CO3)

- The __init__.py file is normally kept empty.
- However, it can also be used to choose specific functions from modules in the package folder and make them available for import.
- Modify __init__.py as below:
 - from .functions import average, power
 - from .greet import SayHello
- The specified functions can now be imported in the interpreter session or another executable script.

- Create test.py in the MyApp folder to test mypackage.

test.py

```
from mypackage import power, average
```

```
SayHello SayHello()
```

```
x=power(3,2)
```

```
print("power(3,2) : ", x)
```

Output of above script:

```
D:\MyApp>python test.py
```

```
Hello world
```

```
power(3,2) : 9
```

Installing packages(CO3)

- Once a package is created, it can be installed for system wide use by running the setup script. The script calls `setup()` function from `setup tools` module.

Steps are as follows:

- Save the following code as `setup.py` in the parent folder 'MyApp'.
- The script calls the `setup()` function from the `setup tools` module.
- The `setup()` function takes various arguments such as name, version, author, list of dependencies etc.
- The `zip_safe` argument defines whether the package is installed in compressed mode or regular mode.

Installing packages(CO3)

- Example: setup.py:

```
from setuptools import setup
setup(name='mypackage',
      version='0.1',
      description='Testing installation of Package',
      url='#', author='malhar', author_email='mlathkar@gmail.com',
      license='MIT', packages=['mypackage'],
      zip_safe=False)
```

Installing packages(CO3)

- Now execute the following command to install mypack using the pip utility. Ensure that the command prompt is in the parent folder, in this case D:\MyApp

D:\MyApp>pip install .

Processing d:\MyApp

Installing collected packages: mypack

Running setup.py install for mypack ... done

Successfully installed mypackage-0.1

- Now mypackage is available for system-wide use and can be imported in any script or interpreter.

D:\>python

>>> import mypackage

>>> mypackage.average(10,20)

>>> mypackage.power(10,2)

Output:

15.0

10

Quiz-Module(CO3)

1. Which of these definitions correctly describes a module?

Ans: b

- a) Denoted by triple quotes for providing the specification of certain program elements
- b) Design and implementation of specific functionality to be incorporated into a program
- c) Defines the specification of how it is to be used
- d) Any program that reuses code

2. Which of the following is not an advantage of using modules?

- a) Provides a means of reuse of program code
- b) Provides a means of dividing up tasks
- c) Provides a means of reducing the size of the program
- d) Provides a means of testing individual parts of the program

Ans: c

3. Program code making use of a given module is called a _____ of the module.

- a) Client
- b) Docstring
- c) Interface
- d) Modularit

Ans: a

4. Which of the following is not a valid namespace?

- a) Global namespace
- b) Public namespace
- c) Built-in namespace
- d) Local namespac

Ans: b

Weekly Assignment(CO3)

- Explain Modules in python with example.
- Create a packages and install in the system with example.
- Explain the standard library modules with example.
- What is dir() function? Explain in detail.
- What is `__init__`?
- Explain Regular expression.
- Define recursion.
- Python program to find sum of n numbers using recursive functions.
- Write a module in python to implement arithmetic calculator that has following user-defined functions: `add()`, `sub()`, `mul()`, `div()`. Write a python program to import this module and perform any operation.
- Differentiate between global and non-local variables.
- Python program to demonstrate the variable length arguments in functions.

Faculty Video Links, You tube & NPTEL Video Links and Online Courses Details(CO3)

- Youtube/other Video Links
- Functions:
 - <https://youtu.be/oSPMmeaiQ68>
- Module:
 - <https://youtu.be/7GXaobCrBb4>

1. What is the output of the following function call

```
def outerFun(a, b):
```

```
    def innerFun(c, d):
```

```
        return c + d
```

```
    return innerFun(a, b)
```

```
    return a
```

```
result = outer
```

```
Fun(5, 10)
```

```
print(result)
```

a) 5

b) 15

c) (15,5)



d) syntax error

2. What is the output of the following function call?

```
def fun1(num):
```

```
    return num + 25
```

```
fun1(5)
```

```
print(num)
```

a) 25

b) 5

c) name error


d) none



3. Choose the correct function declaration of `fun1()` so that we can execute the following function call successfully

`fun1(25, 75, 55)`

`fun1(10, 20)`

- a) `def fun1(**kwargs)`
- b) No, it is not possible in Python
- c) `def fun1(args*)`
- d) `def fun1(*data)` 

4 . What is the output of the add() function call?

```
def add(a, b):  
    return a+5, b+5
```

```
result = add(3, 2)  
print(result) ←
```

a) (8,7)

b) 15

c) 8

d) none

Old Question Papers(CO3)

- Define recursion. [NIET Autonomous 2020-21 (Odd) Marks-1]
- Differentiate between global and non-local variables.
[NIET Autonomous 2020-21 (Odd) Marks-2]
- Explain lambda function.
[NIET Autonomous 2020-21 (Odd) Marks-2]
- Write a module in Python to implement arithmetic calculator that has following user-defined functions: add(), sub(), mul(), div(). Write a python program to import this module and perform any operation.
[NIET Autonomous 2020-21 (Odd) Marks-6]

Old Question Papers(CO3)

- Discuss functions in python with its part and scope. Explain with example(Take simple calculator with add , subtract , multiplication and division.) [AKTU 2019-2020(odd), 10 marks]
- Explain higher order functions with respect to lambda expressions. Write a python code to count occurrences of an element in the list. [AKTU 2019-2020(odd), 10 marks]
- Write a program to sort list of dictionaries by value in python-using lambda functions. [AKTU 2019-2020(odd), 2 marks]
- Differentiate iterators and recursion. Write a program for recursive Fibonacci series. [AKTU 2019-2020(odd), 2 marks]

Expected Questions for University Exam(CO3)

- Explain keyword and default argument in function with example
- Write a program to implement simple calculator with add , subtract , multiplication and division
- Explain higher order functions with respect to lambda expressions. Write a python code to count occurrences of an element in the list.
- Explain packages in the python with installation steps.
- Differentiate iterators and recursion. Write a program for recursive factorial of a number.

Summary(CO3)

- A function is a block of organized, reusable code that is used to perform a single, related action.
- A module is a Python object with arbitrarily named attributes that can bind and reference.
- Package can be installed for system wide use by running the setup script.

References(CO3)

- Python Programming using problem solving Approach by Reema Thareja
- Think Python: An Introduction to Software Design by Allen B. Downey
- www.python.org
- www.w3schools.com

Thank You