4...

Python Functions, Modules and Packages

Chapter Outcomes...

- Use the Python standard functions for the giving problem.
- Develop relevant user defined functions for the given problem using the Python code.
- Write Python module for the given problem.
- Write Payton package for the given problem.

Learning Objectives...

- To learn Basic Concepts of Functions
- To study use of Python Built-in Functions
- To understand User Defined Functions with its Definition, Calling, Arguments Passing etc.
- To study Scope of Variables like Global and Local
- To learn Module Concept with Writing and Importing Modules
- To study Python Built-in Modules like Numeric, Mathematical, Functional Programming Module
- To learn Python Packages with its Basic Concepts and User Defined Packages

4.0 INTRODUCTION

- Functions, modules and packages are all constructs in Python programming that promote code modularization. The modularization (modular programming) refers to the process of breaking a large programming task into separate, smaller, more manageable subtasks or modules.
- A function is a block of organized, reusable code that is used to perform a single, related action/operation. Python has excellent support for functions.
- A function can be defined as the organized block of reusable code which can be called whenever required. A function is a piece of code that performs a particular task.
- A function is a block of code which only runs when it is called. Python gives us many built-in functions like print() but we can also create our own functions called as user-defined functions.
- A module in Python programming allows us to logically organize the python code. A module is a single source code file. The module in Python have the .py file extension. The name of the module will be the name of the file.
- A python module can be defined as a python program file which contains a python code including python functions, class, or variables. In other words, we can say that our Python code file saved with the extension (.py) is treated as the module.
- In Python, packages allow us to create a hierarchical file directory structure of modules. For example, mymodule.mod1 stands for a module mod1, in the package mymodule.
- A Python package is a collection of modules which have a common purpose. In short, modules are grouped together to forms packages.

4.1 PYTHON BUILT–IN FUNCTIONS

- Function are the self contained blocked statements that act like a program that performs specific task.
- The Python interpreter has a number of functions that are always available for use. These functions are called built-in functions. For example, print() function prints the given object to the standard output device (screen) or to the text stream file.
- Python built-in (library) functions can be used to perform specific tasks. Some of these functions comes in category of mathematical functions and some are called type conversion function and so on.

4.1.1 Type Data Conversion Functions

- Sometimes it's necessary to perform conversions between the built-in types. To convert between types we simply use the type name as a function.
- In addition, several built-in functions are supplied to perform special kinds of conversions. All of these functions return a new object representing the converted value.
- Python defines type conversion functions to directly convert one data type to another. Data conversion in Python can happen in following two ways:
  1. Either we tell the compiler to convert a data type to some other type explicitly, and/or
  2. The compiler understands this by itself and does it for us.

1. Python Implicit Data Type Conversion:

- Implicit conversion is when data type conversion takes place either during compilation or during run time and is handled directly by Python for us.

Example: For implicit data/time conversion.

```
>>> a=10
>>> b=25.34
>>> sum=a+b
>>> print sum
35.34
>>>
```

- In the above example, an int value a is added to float value b, and the result is automatically converted to a float value sum without having to tell the compiler. This is the implicit data conversion.
- In implicit data conversion the lowest priority data type always get converted to the highest priority data type that is available in the source code.

2. Python Explicit Data Type Conversion:

- Explicit conversion also known as type casting is when data type conversion takes place because we clearly defined it in our program. We basically force an expression to be of a specific type.
- While developing a program, sometimes it is desirable to convert one data type into another. In Python, this can be accomplished very easily by making use of built-in type conversion functions.
- The type conversion functions result into a new object representing the converted value. A list of data type conversion functions with their respective description is given in following table.

Sr. No.	Function	Description	Example
1.	`int(x [,base])`	Converts x to an integer. base specifies the base if x is a string.	`x=int('1100',base=2)=12` `x=int('1234',base=8)=668`
2.	`long(x [,base] )`	Converts x to a long integer. base specifies the base if x is a string.	`x=long('123'base=8)=83L` `x=long('11',base=16)=17L`
3.	`float(x)`	Converts x to a floating-point number.	`x=float('123.45')=123.45`

4.	complex(real[,imag])	Creates a complex number.	x=complex(1,2) = (1+2j)
5.	str(x)	Converts object x to a string representation.	x=str(10) = '10'
6.	repr(x)	Converts object x to an expression string.	x=repr(3) = 3
7.	eval(str)	Evaluates a string and returns an object.	x=eval('1+2') = 3
8.	tuple(s)	Converts s to a tuple.	x=tuple('123') = ('1', '2', '3') x=tuple([123]) = (123,)
9.	list(s)	Converts s to a list.	x=list('123') = ['1', '2', '3'] x=list(['12'] = ['12']
10.	set(s)	Converts s to a set.	x=set('Python') = {'y', 't', 'o', 'P', 'n', 'h'}
11.	dict(d)	Creates a dictionary. d must be a sequence of (key, value) tuples.	dict={'id':'11','name':'vijay'} print(dict) ={'id': '11', 'name': 'vijay'}
12.	chr(x)	Converts an integer to a character.	x=chr(65) = 'A'
13.	unichr(x)	Converts an integer to a Unicode character.	x=unichr(65) =u'A'
14.	ord(x)	Converts a single character to its integer value.	x=ord('A')= 65
15.	hex(x)	Converts an integer to a hexadecimal string.	x=hex(12) = 0xc
16.	oct(x)	Converts an integer to an octal string.	x=oct(8) = 0o10

Formatting numbers and strings

- The format() function formats a specified value into a specified format.

  Syntax: format(value, format)

Example: For string and number formation

```
>>> x=12.345
>>> format(x,".2f")
'12.35'
>>>
```

Parameter values:

Sr. No.	Format	The Format we want to Format the Value Into.	Example

1.	<	Left aligns the result (within the available space).	>>> x=10.23456
			>>> format(x,"<10.2f")
			'10.23 '

2.	>	Right aligns the result (within the available space).	>>>x=10.23456 >>> format(x,">10.2f") ' 10.23'
3.	^	Center aligns the result (within the	>>>x=10.23456

		available space).	>>> format(x,"^10.2f") ' 10.23 '
4.	=	Places the sign to the left most position.	

5.	+	Use a sign to indicate if the result is positive or negative.	>>> x=123
			>>> format(x,"+")
			'+123'
			>>>x=-123
			>>> format(x,"+")
			'-123'

6.	–	Use a sign for negative values only.	>>> x=-123
			>>> format(x,"+")
			'-123'

7.	‘ ‘	Use a leading space for positive numbers.	

8.	,	Use a comma as a thousand separator.	>>> x=10000000
			>>> format(x,",")
			'10,000,000'

9.	_	Use a underscore as a thousand separator.	>>> x=100000
			>>> format(x,"_")
			'100_000'

10.	b	Binary format.	>>> x=10
			>>> format(x,"b")
			'1010'

11.	c	Converts the value into the corresponding unicode character.	>>> x=10
			>>> format(x,"c")
			'\n'

12.	d	Decimal format.	>>> x=10
			>>> format(x,"d")
			'10'

13.	e	Scientific format, with a lower case e.	>>> x=10
			>>> format(x,"e")
			'1.000000e+01'

14.	E	Scientific format, with an upper case E.	>>> x=10
			>>> format(x,"E")
			'1.000000E+01'

15.	f	Fix point number format.	>>> x=10
			>>> format(x,"f")
			'10.000000'

16.	F	Fix point number format, upper case.	

17.	g	General format.	>>> x=10
			>>> format(x,"g")
			'10'

18.	G	General format (using a upper case E for scientific notations).	
19.	o	Octal format.	>>> x=10

			>>> format(x,"o") '12'

20.	x	Hex format, lower case.	>>> x=10
			>>> format(x,"x")
			'a'

21.	X	Hex format, upper case.	>>> x=10
			>>> format(x,"X")
			'A'

22.	n	Number format.	

23.	%	Percentage format.	>>> x=100
			>>> format(x,"%")
			'10000.000000%'

24.	>10s	String with width 10 in left justification.	>>> x="hello"
			>>> format(x,">10s")
			' hello'

25.	<10s	String with width 10 in right justification.	>>> x="hello"
			>>> format(x,"<10s")
			'hello '

4.1.2 In-Built Mathematical Functions

- Function can be described as a piece of code that may or may not take some value(s) as input, process it, and then finally may or may not return any value as output.
- Python's math module is used to solve problems related to mathematical calculations. Some functions are directly executed for maths functions we need to import math module first.

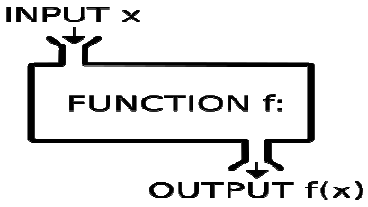


Fig. 4.1

- Various mathematical functions in Python programming are explained below:

1. **abs():** Return the absolute value of a number.
   Syntax: `abs(number)`

   Example: For abs() function.
   ```
   >>> abs(10)
   10
   >>> abs(-10)
   10
   ```

2. **all():** The all() function returns True if all items in an iterable are true, otherwise it returns False. If the iterable object is empty, the all() function also returns True.
   Syntax: `all(iterable)`

   Example: For all() function.
   ```
   >>> x=[True, True, True]
   >>> all(x)
   True
   >>> x=[0,1,1]
   >>> all(x)
   ```

```
False
>>>
```

3. **any():** The any() function returns True if any item in an iterable are true, otherwise it returns False. If the iterable object is empty, the any() function will return False.
   Syntax: `any(iterable)`

   Example: For any() function.

```
>>> x=[True, False, True]
>>> any(x)
True
>>> x=[False, True, False]
>>> any(x)
True
>>> x=[0,1,False]
>>> any(x)
True
>>> x=(0,1,0)
>>> any(x)
True
>>> x=(0,0,0)
>>> any(x)
False
>>> x=()
>>> any(x)
False
```

4. **bin():** The bin() function returns the binary version of a specified integer. The result will always start with the prefix 0b.
   Syntax: `bin(n)`

   Example: For bin().

```
>>> bin(10)
'0b1010'
>>> bin(36)
'0b100100'
```

5. **bool():** The bool() function returns the boolean value of a specified object.
   Syntax: `bool(object)`

   Example: For bool() math function.

```
>>> bool(1)
True
>>> bool(0)
False
```

6. **ceil()**: This function returns the smallest integral value greater than the number. If number is already integer, same number is returned.
   Syntax: `cell(number)`

   Example: For ceil() function.

```
>>> math.ceil(2.3)
3
```

7. **cos():** This function returns the cosine of value passed as argument. The value passed in this function should be in radians.
   Syntax: `cos(number)`

Example: For cos() function.

```
>>> math.cos(3)
-0.9899924966004454
>>>math.cos(-3)
```

```
-0.9899924966004454
>>>math.cos(0)
1.0
```

8. **cosh():** Returns the hyperbolic cosine of x.
   Syntax: `cosh(x)`

   Example: For cosh() function.
   ```
   >>> print(math.cosh(3))
   10.067661995777765
   ```

9. **copysign():** Return x with the sign of y. On a platform that supports signed zeros, copysign(1.0, -0.0) returns -1.0.
   Syntax: `copysign(x,y)`

   Example:
   ```
   >>> math.copysign(10,-12)
   -10.0
   ```

10. **degrees():** This function is used to convert argument value from radians to degrees.
    Syntax: `degrees(rad)`

    Where, rad: The radians value that one needs to convert into degrees.

    Returns: This function returns the floating point degrees equivalent of argument.

    Computational Equivalent: 1 Degrees = pi/180 Radians.

    Example: For degree() function.
    ```
    >>>math.degrees(1.57)
    89.95437383553924
    ```

    e: mathematical constant e (2.71828...).

    Syntax: `(e)`

    Example: For e constant.
    ```
    >>> print(math.e)
    2.718281828459045
    ```

11. **exp():** The method exp() returns returns exponential of x: ex.
    Syntax: `import math math.exp(x)`
    Note: This function is not accessible directly, so we need to import math module and then we need to call this function using math static object.
    x: This is a numeric expression.

    Example:
    ```
    >>> math.exp(1)
    2.718281828459045
    >>>
    ```

12. **floor():** This function returns the greatest integral value smaller than the number. If number is already integer, same number is returned.
    Syntax: `floor(number)`

    Example:
    ```
    >>> math.floor(2.3)
    2
    ```

13. **fabs():** This function will return an absolute or positive value.
    Syntax: `fabs(x)`

    Example:
    ```
    >>> math.fabs(10)
    10.0
    >>> math.fabs(-20)
    20.0
    ```

14. **factorial():** Returns the factorial of x.

Syntax: `factorial(x)`

Example:

```
>>> math.factorial(5)
120
```

15. **fmod():** This function returns x % y.

Syntax: `fmod(x,y)`

Example:

```
>>> math.fmod(50,10)
0.0
>>>math.fmod(50,20)
10.0
```

16. **frexp(x):** Returns the mantissa and exponent of x as the pair (m, e).

Syntax: `frexp(x)`

Example:

```
>>> print(math.frexp(2))
(0.5, 2)
>>> print(math.frexp(3))
(0.75, 2)
```

17. **fsum():** Returns an accurate floating point sum of values in the iterable.

Syntax: `fsum(iterable)`

Example:

```
>>> print(sum([.1, .1, .1, .1, .1, .1, .1, .1, .1, .1]))
0.9999999999999999
>>> print(math.fsum([.1, .1, .1, .1, .1, .1, .1, .1, .1, .1]))
1.0
```

18. **hypot():** Returns the Euclidean norm, sqrt(x*x + y*y).

Syntax: `hypot(x, y)`

Example:

```
>>> print(math.hypot(5,3))
5.830951894845301
>>> print(math.hypot(4,3))
5.0
```

19. **isfinite():** Returns True if x is neither an infinity nor a NaN (Not a Number).

Syntax: `isfinite(x)`

Example:

```
>>> print(math.isfinite(2))
True
```

20. **isinf():** Returns True if x is a positive or negative infinity.

Syntax: `isinf(x)`

Example:

```
>>> print(math.isinf(2))
False
```

21. **Log():** Python offers many in-build logarithmic functions under the module “math” which allows us to compute logs using a single line. There are four variants of logarithmic functions.

    (i) log(a,(Base)): This function is used to compute the natural logarithm (Base e) of a. If 2 arguments are passed, it computes the logarithm of desired base of argument a, numerically value of log(a)/log(Base).

    Syntax: `math.log(a, base)`

    Parameters:

    a: The numeric value.

Base: Base to which the logarithm has to be computed.

Return Value: Returns natural log if 1 argument is passed and log with specified base if 2 arguments are passed.

Example:

```
>>> print(math.log(14))
2.6390573296152584
>>> print(math.log(14,5))
1.6397385131955606
```

(ii) log2(a): This function is used to compute the logarithm base 2 of a. Displays more accurate result than log(a, 2).

Syntax: `math.log2(a)`

Parameters:

a: The numeric value.

Return Value: Returns logarithm base 2 of a.

Example:

```
>>> print(math.log2(14))
3.807354922057604
>>>
```

(iii) log10(a): This function is used to compute the logarithm base 10 of a. Displays more accurate result than log(a, 10).

Syntax: `math.log10(a)`

Parameters:

A: The numeric value.

Return Value: Returns logarithm base 10 of a.

Example:

```
>>> print(math.log10(14))
1.146128035678238
```

(iv) log1p(a): This function is used to compute logarithm(1+a) .

Syntax: `math.log1p(a)`

Parameters:

a: The numeric value

Return Value: Returns log(1+a)

Example:

```
>>> print(math.log1p(14))
2.70805020110221
>>>
```

22. min(): Returns smallest value among supplied arguments.

Syntax: `min(x1, x2, …, xn)`

Example:

```
>>> min(20, 10, 30)
10
```

23. max(): Returns largest value among supplied arguments

Syntax: `max(x1, x2, …, xn)`

Example:

```
>>> max(20, 10, 30)
30
```

24. modf(): Returns the fractional and integer parts of x

Syntax: `modf(x)`

Example:

```
>>> print(math.modf(4))
(0.0, 4.0)
>>> print(math.modf(4.3))
(0.2999999999999998, 4.0)
```

25. **pow():** The pow() function returns the value of x to the power of y (xy). If a third parameter is present, it returns x to the power of y, modulus z.
Syntax: `pow(x, y, z)`

Example:

```
>>> pow(2,3)
8
>>> pow(2,3,2)
0
```

pi: Mathematical constant, the ratio of circumference of a circle to it's diameter (3.14159...).
Syntax: `(pi)`

Example:

```
>>> print(math.pi)
3.141592653589793
```

26. **round():** The round() function returns a floating point number that is a rounded version of the specified number, with the specified number of decimals. The default number of decimals is 0, meaning that the function will return the nearest integer.
Syntax: `round(number[, ndigits])`

Example:

```
>>> round(10.2345)
10
>>> round(5.76543)
6
>>> round(5.76543,2)
5.77
```

27. **radians(deg):** This function accepts the "degrees" as input and converts it into its radians equivalent.
Syntax: `radians(deg)`
Parameters:
deg: The degrees value that one needs to convert into radians.
Returns: This function returns the floating point radians equivalent of argument.
Computational Equivalent: 1 Radians = 180/pi Degrees.

Example:

```
>>> math.radians(89.95)
1.5699236621688994
```

28. **sin()**: This function returns the sine of value passed as argument. The value passed in this function should be in radians.
Syntax: `sin(number)`

Example:

```
>>> math.sin(3)
0.1411200080598672
>>> math.sin(-3)
-0.1411200080598672
>>> math.sin(0)
0.0
```

29. **sinh():** Returns the hyperbolic sine of x.
Syntax: `sinh(x)`

Example:

```
>>> print(math.sinh(3))
10.017874927409903
```

30. sqrt(): The method sqrt() returns the square root of x for x > 0.

Syntax: `import math math.sqrt( x )`

Example:

```
>>> math.sqrt(100)
10.0
>>> math.sqrt(5)
2.23606797749979
>>>
```

31. tan(): This function returns the tangent of value passed as argument. The value passed in this function should be in radians.

Syntax: `tan(number)`

Example:

```
>>> math.tan(3)
-0.1425465430742778
>>> math.tan(-3)
0.1425465430742778
>>> math.tan(0)
0.0
```

32. tanh(): Returns the hyperbolic tangent of x.

Syntax: `tanh(x)`

Example:

```
>>> print(math.tanh(3))
0.9950547536867305
```

33. trunc(): This function returns the truncated integer of x.

Syntax: `trunc(x)`

Example:

```
>>>math.trunc(3.354)
3
```

4.2 USER DEFINED FUNCTIONS

- Functions in Python programming are self-contained programs that perform some particular tasks. Once, a function is created by the programmer for a specific task, this function can be called anytime to perform that task.
- Python gives us many built-in functions like print(), len() etc. but we can also create our own functions. These functions are called user-defined functions.
- User defined function are the sells contained block of statements created by users according to their requirements.
- A user-defined function is a block of related code statements that are organized to achieve a single related action. A key objective of the concept of the user-defined function is to encourage modularity and enable reusability of code.

4.2.1 Function Definition

- Function definition is a block where the statements inside the function body are written. Functions allow us to define a reusable block of code that can be used repeatedly in a program.

Syntax:

```
def function-name(parametes):
"function_docstring"
function_statements
return [expression]
```

Defining Function:

- 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. We can also define parameters inside these parentheses.
- The first statement of a function can be an optional statement - the documentation string of the function or docstring.
- 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. A return statement with no arguments is the same as return None
- The basic syntax for a Python function definition is explained in Fig. 4.2.

Fig. 4.2

4.2.2 Function Calling

- The def statement merely creates a function but does not call it. After the def has run, we can can call (run) the function by adding parentheses after the function's name.

Example: For calling a function.

```
>>>def square(x):   # function definition
return x*x
>>>square(4)       # function call
16
>>>
```

Advantages of user defined functions:

1. User-defined functions help to decompose a large program into small segments which makes program easy to understand, maintain and debug.
2. By using functions, we can avoid rewriting same logic/code again and again in a program.
3. We can call python functions any number of times in a program and from any place in a program.
4. We can track a large python program easily when it is divided into multiple functions.
5. Reusability is the main achievement of Python functions.
6. Functions in Python reduces the overall size as the program.

4.2.3 Function Arguments

- Many build in functions need arguments to be passed with them. Many build in functions require two or more arguments. The value of the argument is always assigned to a variable known as parameter.
- There are four types of arguments using which can be called are Required arguments, Keyword arguments, Default arguments, and Variable-length arguments.

4.2.3.1 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.

Example: For required argument.

```
>>> def display(str):
```

```
    "This print a string passed as argument"
    print(str)
    return
>>> display()                          #required argument
Traceback (most recent call last):
    File "<pyshell#17>", line 1, in <module>
    display()
TypeError: display() missing 1 required positional argument: 'str'
>>>
```

- There is an error because we did not pass any argument to the function display. We have to pass argument like display ("hello").

4.2.3.2 Keywords Arguments

- Keyword arguments are related to the function calls. When we use keyword arguments in a function call, the caller identifies the arguments by the parameter name.
- This allows us 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.

Example 1: For keywords arguments.

```
>>> def display(str):
    "This print a string passed as argument"
    print(str)
    return
>>> display()
Traceback (most recent call last):
  File "<pyshell#17>", line 1, in <module>
    display()
TypeError: display() missing 1 required positional argument: 'str'
>>> display(str="Hello Python")
Hello Python
```

Example 2:

```
>>> def display(name,age):
    print("Name:",name)
    print("Age:",age)
    return
>>> display(name="meenakshi",age=35)
Name: meenakshi
Age: 35
>>> display(age=35,name="meenakshi")
Name: meenakshi
Age: 35
>>>
```

4.2.3.3 Default Arguments

- 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: For default arguments.

```
>>> def display(name,age=20):
    print("Name:",name)
    print("Age:",age)
    return
>>> display(name="meenakshi")
Name: meenakshi
Age: 20
```

```
>>> display(name="meenakshi",age=35)
Name: meenakshi
Age: 35
>>>
```

4.2.3.4 Variable Length Arguments

- In many cases where we are required to process a function with more number of arguments than we specified in the function definition. These types of arguments are known as variable length arguments.

Syntax:

```
def function-name([formal_args,] *var_args_tuple ):
   "function_docstring"
function_statements
   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.

Example: For variable length argument.

```
>>> def display(arg1,*list):
   "This prints a variable passed arguments"
   print(arg1)
   for i in list:
      print(i)
   return
>>> display(10,20,30)
10
20
30
>>>
```

4.2.4 return Statement

- The return statement is used to exit a function. The return statement is used to return a value from the function. A function may or may not return a value.
- If a function returns a value, it is passed back by the return statement are argument to the caller. If it does not return a value, we simply write return with no arguments.

Syntax: `return(expression)`

Example: For return statement.

```
>>> def sum( arg1, arg2 ):
   "Add both the parameters and return them."
   total = arg1 + arg2
   print ("Sum: ", total)
   return total
>>> result=sum(30,35)
Sum:  65
>>>
```

Fruitful Function:

- Fruitful functions are those functions which return values. The built-in functions we have used, such as abs, pow, and max, have produced results.
- Calling each of these functions generates a value, which we usually assign to a variable or use as part of an expression.
- Following function calculates area of circle and return the value:

```
>>> def area(radius):
   temp = 3.14159 * radius**2
```

```
    return temp
>>> area(5)
78.53975
```

- **void Functions:** void functions are those functions which do not return any value.

Example: For void function.

```
>>> def show():
    str="hello"
    print(str)
>>> show()
hello
>>>
```

Function Parameters:

- Information can be passed to functions as parameter. Parameters are used to pass input into a function, and Python has several kinds of parameters.

1. Pass By Reference:

- Python passes parameters to a function using a technique known as pass by reference. This means that when we pass parameters, the function refers to the original passed values using new names.
- For example, consider the following simple program:

```
# reference.py
def add(a, b):
return a + b
```

- Run IDLE's interactive command line and type following:

```
>>> x, y = 3, 4
>>> add(x, y)
```

2. Pass By Value:

- In pass by value, the function receives a copy of the argument objects passed to it by the caller, stored in a new location in memory.
- When a parameter is passed by value, a copy of it is made and passed to the function. If the value being passed is large, the copying can take up a lot of time and memory. Python, however, does not support pass by value.

Example: For pass by value.

```
>>> def try_to_modify(x, y, z):
        x = 23
        y.append(42)
        z = [99]            # new reference
        print(x)
        print(y)
        print(z)

>>> a = 77                  # immutable variable
>>> b = [99]                # mutable variable
>>> c = [28]
>>> try_to_modify(a, b, c)
23
[99, 42]
[99]
>>> print(a)
77
>>> print(b)
[99, 42]
>>> print(c)
[28]
```

4.2.5 Scope of Variable

- All variables in a program may not be accessible at all locations in that program. This depends on where we have declared a variable.
- The scope of a variable determines the portion of the program where you can access a particular variable/identifier.
- The availability/accessibility of a variable in different parts of a program is referred to as its scope. There are two basic scopes of variables in Python:

1. Global Variables: Global variables can be accessed throughout (outside) the program body by all functions.

2. Local Variables: Local variables can be accessed only inside the function in which they are declared.

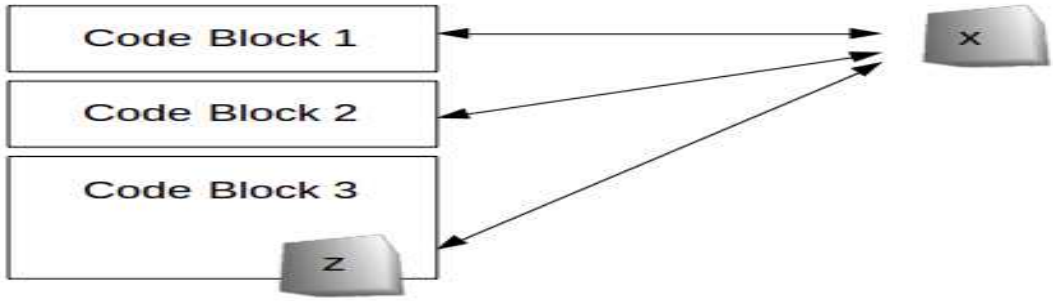


Fig. 4.3

- Fig. 4.3 shows, global variable (x) can be reached and modified anywhere in the code, local variable (z) exists only in block 3.

Example: For scope of variables.

```
>>> g=10                          # global variable g
>>> def show():
    l=20                          # local variable l
    print("local variable=",l)
    print("Global variable=",g)
>>> show()
local variable= 20
Global variable= 10
>>>
```

Difference between Local Variables and Global Variables:

Sr. No.	Local Variables	Global Variables
1.	Local variables are declared inside a function.	Global variables are declared outside any function.
2.	Accessed only by the statements, inside a function in which they are declared.	Accessed by any statement in the entire program.
3.	Local variables are alive only for a function	Global variables are alive till the end of the program
4.	A local variable is destroyed when the control of the program exit out of the block in which local variable is declared.	Global variable is destroyed when the entire program is terminated.

Concept of Actual and Formal Parameters:

1. Actual Parameters:

- The parameters used in the function call are called actual parameters. These are the actual values that are passed to the function. The actual parameters may be in the form of constant values or variables.

- The data types of actual parameters must match with the corresponding data types of formal parameters (variables) in the function definition.
  (i) They are used in the function call
  (ii) They are actual values that are passed to the function definition through the function call.
  (iii) They can be constant values or variable names (such as local or global).

2. Formal Parameters:

- The parameters used in the header of function definition are called formal parameters of the function. These parameters are used to receive values from the calling function.
  (i) They are used in the function header.
  (ii) They are used to receive the values that are passed to the function through function call.
  (iii) They are treated as local variables of a function in which they are used in the function header.

Example: For actual and formal parameters.

```
>>> def cube(x):      # formal parameters
    return x*x*x
>>> result = cube(7)  # actual parameters
>>> print(result)
```

Recursive Functions:

- Recursion is a way of programming or coding a problem, in which a function calls itself one or more times in its body.
- A function is said to be a recursive if it calls itself. For example, lets say we have a function abc() and in the body of abc() there is a call to the abc().
- The factorial of 6 (denoted as 4!) is 1*2*3*4 = 24.

Example: For recursive function.

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

print(fact(0))
print(fact(4))
print(fact(6))
```

Output:

```
1
24
720
```

- Each function call multiples the number with the factorial of number 1 until the number is equal to one.

```
calc_factorial(4                    # 1st call with 4
4 * calc_factorial(3)               # 2nd call with 3
4 * 3 * calc_factorial(2)           # 3rd call with 2
4 * 3 * 2 * calc_factorial(1)       # 4th call with 1
4 * 3 * 2 * 1                       # return from 4th call as number=1
4 * 3 * 2                           # return from 3rd call
4 * 6                               # return from 2nd call
24                                  # return from 1st call
```

- Our recursion ends when the number reduces to 1. This is called the base condition. Every recursive function must have a base condition that stops the recursion or else the function calls itself infinitely.

Advantages of Recursion:

1. Recursive functions make the code look clean and elegant.
2. A complex task can be broken down into simpler sub-problems using recursion.

3. Sequence generation is easier with recursion than using some nested iteration.

Disadvantages of Recursion:

1. Sometimes the logic behind recursion is hard to follow through.
2. Recursive calls are expensive (inefficient) as they take up a lot of memory and time.
3. Recursive functions are hard to debug.
4. It consumes more storage space because the recursive calls along with variables are stored on the stack.
5. It is not more efficient in terms of speed and execution time.

Example: Programs to convert U.S. dollars to Indian rupees.

```
def dol_rup():
    dollars = float(input("Please enter dollars:"))
    rupees = dollars * 70
    print("Dollars: ",dollars)
    print("Rupees: ",rupees)
def euro_rup():
    euro= float(input("Please enter euro:"))
    rupees = euro * 79.30
    print("Euro: ",euro)
    print("Rupees: ",rupees)
def menu():
    print("1: Doller to Rupees")
    print("2: Euro to Rupees")
    print("3: Exit")
    choice=int(input("Enter your choice: "))
    if choice==1:
    dol_rup()
    if choice==2:
        euro_rup()
    if choice==3:
        print("Good bye!")
menu()
```

Output:

```
1: Doller to Rupees
2: Euro to Rupees
3: Exit
Enter your choice: 1
Please enter dollars:75
Dollars:  75.0
Rupees:  5250.0
```

4.3 MODULES

- Modules are primarily the (.py) files which contain Python programming code defining functions, class, variables, etc. with a suffix .py appended in its file name.
- A file containing .py python code is called a module.
- If we want to write a longer program, we can use file where we can do editing, correction. This is known as creating a script. As the program gets longer, we may want to split it into several files for easier maintenance.
- We may also want to use a function that you've written in several programs without copying its definition into each program.
- In Python we can put definitions in a file and use them in a script or in an interactive instance of the interpreter. Such a file is called a module.

4.3.1 Writing Module

- Writing a module means simply creating a file which can contains python definitions and statements. The file name is the module name with the extension .py. To include module in a file, use import statement.
- Follow the following steps to create modules:
  1. Create a first file as a python program with extension as .py. This is your module file where we can write a function which perform some task.
  2. Create a second file in the same directory called main file where we can import the module to the top of the file and call the function.
  3. Second file needs to be in the same directory so that Python knows where to find the module since it's not a built-in module.

Example : For creating a module. Type the following code and save it as p1.py.

```
def add(a, b):
   "This function adds two numbers and return the result"
   result = a + b
   return result
def sub(a, b):
   "This function subtract two numbers and return the result"
   result = a - b
return result
```

4.3.2 Importing Modules

- Import statement is used to imports a specific module by using its name. Import statement creates a reference to that module in the current namespace. After using import statement we can use refer to things defined in that module.
- We can import the definitions inside a module to another module or the interactive interpreter in Python. We use the import keyword to do this.
- **Import the definitions inside a module:**

```
import p1
print(p1.add(10,20))
print(p1.sub(20,10))
```

Output:

```
30
10
```

- **Import the definitions using the interactive interpreter:**

```
>>> import p1
>>> p1.add(10,20)
30
>>> p1.sub(20,10)
10
>>>
```

Importing Objects From Module:

- Import in python is similar to #include header_file in C/C++. Python modules can get access to code from another module by importing the file/function using import. Python provides three different ways to import modules.

1. From x import a:

- Imports the module x, and creates references in the current namespace to all public objects defined by that module. If we run this statement, we can simply use a plain name to refer to things defined in module x.
- We can access pi directly without dot notation.

Example: For importing module.

```
>>> from math import pi
>>> pi
3.141592653589793
>>> from math import sqrt
>>> sqrt(144)
12.0
```

2. From x import a, b, c:

- Imports the module x and creates references in the current namespace to the given objects. Or we can use a, b and c function in we program.

Example:

```
>>> from math import sqrt, ceil, floor
>>> sqrt(144)
12.0
>>> ceil(2.6)
3
>>> floor(2.6)
2
```

3. From x import *:

- We can use * (asterisk) operator to import everything from the module.

Example:

```
>>> from math import *
>>> cos(60)
-0.9524129804151563
>>> sin(60)
-0.3048106211022167
>>> tan(60)
0.320040389379563
```

4.3.3 Aliasing Modules

- It is possible to modify the names of modules and their functions within Python by using the 'as' keyword.
- We can make alias because we have already used the same name for something else in the program or we may want to shorten a longer name.

Syntax: `import module as another_name`

Example: Create a module to define two functions. One to print Fibonacci series and other for finding whether the given number is palindrome or not.

Step 1: Create a new file p1.py and write the following code in it and save it.

```
# Fibonacci numbers module
def fib(n):        # write Fibonacci series up to n
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()

# To check whether given number is palindrome or not
def palin(x):
    rev=0
    x1=x
    while x>0:
        r=x%10
        x=x//10
        rev=rev*10+r
```

```
    if x1==rev:
       print("Number ", x1 ," is a palindrome")
    else:
    print("Number ", x1 ," is not a palindrome")
```

Step 2: Create new file p2.py to include the module. Add the following code and save it.

```
import p1 as m     # aliasing module
def main():
    n=int(input("Enter number to print fibonacci series upto numbers= "))
    m.fib(n)
    x=int(input("Enter number to check palindrome or not?= "))
    m.palin(x)
main()
```

Step 3: Execute p2.py file.

```
Enter number to print fibonacci series upto numbers= 4
0 1 1 2 3
Enter number to check palindrome or not?= 122
Number 122 is not a palindrome
```

4.3.4 Python Built in Modules

- A module is a collection of Python objects such as functions, classes, and so on. Python interpreter is bundled with a standard library consisting of large number of built-in modules,
- Built-in modules are generally written in C and bundled with Python interpreter in precompiled form. A built-in module may be a Python script (with .py extension) containing useful utilities.
- A module may contain one or more functions, classes, variables, constants, or any other Python resources.

(I) Numeric and Mathematical Modules:

- This module provides numeric and math-related functions and data types. Following are the modules which are classified as numeric and mathematical modules
  - (i) numbers (Numeric abstract base classes).
  - (ii) math (Mathematical functions).
  - (iii) cmath (Mathematical functions for complex numbers).
  - (iv) decimal (Decimal fixed point and floating point arithmetic).
  - (v) fractions (Rational numbers).
  - (vi) random (Generate pseudo-random numbers).
  - (vii) statistics (Mathematical statistics functions).
- The numbers module defines an abstract hierarchy of numeric types. The math and cmath modules contain various mathematical functions for floating-point and complex numbers. The decimal module supports exact representations of decimal numbers, using arbitrary precision arithmetic.

1. math and cmath Modules:

- Python provides two mathematical modules named as math and cmath. The math module gives us access to hyperbolic, trigonometric, and logarithmic functions for real numbers and cmath module allows us to work with.
- The mathematical functions for complex numbers.

Example 1: For math module.

```
>>> import math
>>> math.ceil(1.001)
2
>>> from math import *
>>> ceil(1.001)
2
```

```
>>> floor(1.001)
1
>>> factorial(5)
120
>>> trunc(1.115)
1
>>> sin(90)
0.8939966636005579
>>> cos(60)
-0.9524129804151563
>>> exp(5)
148.4131591025766
>>> log(16)
2.772588722239781
>>> log(16,2)
4.0
>>> log(16,10)
1.2041199826559246
>>> pow(144,0.5)
12.0
>>> sqrt(144)
12.0
>>>
```

Example 2: For cmath module.

```
>>> from cmath import *
>>> c=2+2j
>>> exp(c)
(-3.074932320639359+6.71884969742825j)
>>> log(c,2)
(1.5000000000000002+1.1330900354567985j)
>>> sqrt(c)
(1.5537739740300374+0.6435942529055826j)
```

2. Decimal Module:

- Decimal numbers are just the floating-point numbers with fixed decimal points. We can create decimals from integers, strings, floats, or tuples. A Decimal instance can represent any number exactly, round up or down, and apply a limit to the number of significant digits.

Example : For decimal module.

```
>>> from decimal import Decimal
>>> Decimal(121)
Decimal('121')
>>> Decimal(0.05)
Decimal('0.05000000000000000277555756156289135105907917022705078125')
>>> Decimal('0.15')
Decimal('0.15')
>>> Decimal('0.012')+Decimal('0.2')
Decimal('0.212')
>>> Decimal(72)/Decimal(7)
Decimal('10.285714285714285714285714285')
>>> Decimal(2).sqrt()
Decimal('1.4142135623730950488016887242')
```

3. Fractions Module:

- A fraction is a number which represents a whole number being divided into multiple parts. Python fractions module allows us to manage fractions in our Python programs.

Example: For fractions module.

```
>>> import fractions
>>> for num, decimal in [(3, 2), (2, 5), (30, 4)]:
    fract = fractions.Fraction(num, decimal)
    print(fract)
3/2
2/5
15/2
```

- It is also possible to convert a decimal into a Fractional number. Let's look at a code snippet:

```
>>> import fractions
>>> for deci in ['0.6', '2.5', '2.3', '4e-1']:
   fract = fractions.Fraction(deci)
   print(fract)
```

Output:

```
3/5
5/2
23/10
2/5
>>>
```

4. Random Module:

- Sometimes, we want the computer to pick a random number in a given range, pick a random element from a list etc.
- The random module provides functions to perform these types of operations. This function is not accessible directly, so we need to import random module and then we need to call this function using random static object.

Example: For random module.

```
>>> import random
>>> print(random.random())       # It generate a random number in the range (0.0, 1.0)
0.27958089234907935
>>> print(random.randint(10,20)) # It generate a random integer between x and y inclusive
13
```

5. Statistics Module:

- Statistics module provides access to different statistics functions. Example includes mean (average value), median (middle value), mode (most often value), standard deviation (spread of values).

Example: For statistics module.

```
>>> import statistics
>>> statistics.mean([2,5,6,9])
5.5
>>> import statistics
>>> statistics.median([1,2,3,8,9])
3
>>> statistics.median([1,2,3,7,8,9])
5.0
>>> import statistics
>>> statistics.mode([2,5,3,2,8,3,9,4,2,5,6])
2
>>> import statistics
>>> statistics.stdev([1,1.5,2,2.5,3,3.5,4,4.5,5])
1.3693063937629153
```

(II) Functional Programming Modules:

- These modules provide functions and classes that support a functional programming style and general operations on callables.
- Following are modules which comes under functional programming modules.
  (i) itertools (functions creating iterators for efficient looping).
  (ii) functools (higher-order functions and operations on callable objects).
  (iii) operator (standard operators as functions).

1. itertools Module:

- Python programming itertools module provide us various ways to manipulate the sequence while we are traversing it.
- Python itertools chain() function just accepts multiple iterable and return a single sequence as if all items belongs to that sequence.

Example: For itertools module with chain().

```
>>> from itertools import *
>>> for value in chain([1.3, 2.51, 3.3], ['C++', 'Python', 'Java']):
    print(value)
```

Output:

```
1.3
2.51
3.3
C++
Python
Java
```

- Python itertools cycle() function iterate through a sequence upto infinite. This works just like a circular Linked List.

Example: For intools module with cycles.

```
>>> from itertools import *
>>> for item in cycle(['C++','Python','Java']):
index=index+1
if index==10:
   break
print(item)
```

Output:

```
C++
Python
Java
C++
Python
Java
C++
Python
Java
>>>
```

2. functools Module:

- Python functools module provides us various tools which allows and encourages us to write reusable code.
- Python functools partial() functions are used to replicate existing functions with some arguments already passed in. It also creats new version of the function in a well-documented manner.
- Suppose we have a function called multiplier which just multiplies two numbers. Its definition looks like:

```
def multiplier(x, y):
```

```
        return x * y
```

- Now, what if we want to make some dedicated functions to double or triple a number? We will have to define new functions as:

```
def multiplier(x, y):
    return x * y
def doubleIt(x):
    return multiplier(x, 2)
def tripleIt(x):
    return multiplier(x, 3)
```

- But what happens when we need 1000 such functions? Here, we can use partial functions:

```
from functools import partial
def multiplier(x, y):
    return x * y
double = partial(multiplier, y=2)
triple = partial(multiplier, y=3)
print('Double of 2 is {}'.format(double(5)))
```

3. Operator Module:

- The operator module supplies functions that are equivalent to Python's operators. These functions are handy in cases where callables must be stored, passed as arguments, or returned as function results.
- Functions supplied by the operator module are listed in following table:

Sr. No.	Function	Signature/Syntax	Behaves Like
1.	abs	abs(a)	abs(a)
2.	add	add(a,b)	a+b
3.	and_	and_(a,b)	a&b
4.	div	div(a,b)	a/b
5.	eq	eq(a,b)	a==b
6.	gt	gt(a,b)	a>b
7.	invert, inv	invert(a), inv(a)	~a
8.	le	**le(**a,b**)**	a<=b
9.	lshift	lshift(a,b)	a<<b
10.	lt	lt(a,b)	a<b
11.	mod	mod(a,b)	a%b
12.	mul	mul(a,b)	a*b
13.	ne	ne(a,b)	a!=b
14.	neg	neg(a)	-a
15.	not_	not_(a)	not a
16.	or_	or_(a,b)	a|b
17.	pos	pos(a)	+a
18.	repeat	repeat(a,b)	a*b
19.	rshift	rshift(a,b)	a>>b
20.	xor_	xor(a,b)	a^b

4.3.5 NameSspace and Scoping

- A namespace is a system to have a unique name for each and every object in Python. An object might be a variable or a method. Python itself maintains a namespace in the form of a Python dictionary.

- Python interpreter understands what exact method or variable one is trying to point to in the code, depending upon the namespace. So, the division of the word itself gives little more information: Name (which means name, an unique identifier) + Space(which talks something related to scope). Here, a name might be of any Python method or variable and space depends upon the location from where is trying to access a variable or a method.
- A namespace in python is a collection of names. So, a namespace is essentially a mapping of names to corresponding objects.
- At any instant, different python namespaces can coexist completely isolated- the isolation ensures that there are no name collisions/problem.
- A scope refers to a region of a program where a namespace can be directly accessed, i.e. without using a namespace prefix.
- Scoping in Python revolves around the concept of namespaces. Namespaces are basically dictionaries containing the names and values of the objects within a given scope.

Types of Namespaces:

- When a user creates a module, a global namespace gets created, later creation of local functions creates the local namespace. The built-in namespace encompasses global namespace and global namespace encompasses local namespace.
  1. **Local Namespace:** This namespace covers the local names inside a function. Python creates this namespace for every function called in a program. It remains active until the function returns.
  2. **Global Namespace:** This namespace covers the names from various imported modules used in a project. Python creates this namespace for every module included in your program. It will last until the program ends.
  3. **Built-in Namespace: This** namespace covers the built-in functions and built-in exception names. Python creates it as the interpreter starts and keeps it until we exit.

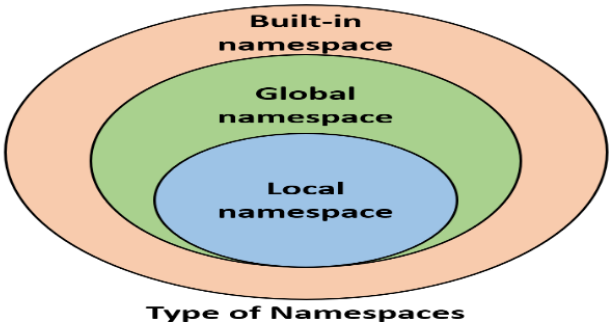


Fig. 4.4

- Namespaces help us uniquely identify all the names inside a program. According to Python's documentation “A scope is a textual region of a Python program, where a namespace is directly accessible.” Directly accessible means that when we are looking for an unqualified reference to a name Python tries to find it in the namespace.
- Scopes are determined statically, but actually, during runtime, they are used dynamically. This means that by inspecting the source code, we can tell what the scope of an object is, but this doesn't prevent the software from altering that during runtime.
- In Python programming for different scopes makes accessible as given below:
  1. The **local scope,** which is the innermost one and contains the local names.
  2. The **enclosing scope,** i.e., the scope of any enclosing function. It contains non-local names and also non-global names.
  3. The **global scope** contains the global names.
  4. The **built-in scope** contains the built-in names. Python comes with a set of functions that we can use in an off-the-shelf fashion, such as print, all, abs, and so on. They line in the built-in scope.

Python Variable Scoping:

- Scope is the portion of the program from where a namespace can be accessed directly without any prefix.

- Namespaces are a logical way to organize variable names when a variable inside a function (a local variable) shares the same name as a variable outside of the function (a global variable).
- Local variables contained within a function (either in the script or within an imported module) and global variables can share a name as long as they do not share a namespace.
- At any given moment, there are at least following three nested scopes:
  1. Scope of the current function which has local names.
  2. Scope of the module which has global names.
  3. Outermost scope which has built-in names.
- When a reference is made inside a function, the name is searched in the local namespace, then in the global namespace and finally in the built-in namespace.
- If there is a function inside another function, a new scope is nested inside the local scope. Python has two scopes.
  1. **Local Scope Variable:** All those variables which are assigned inside a function known as local scope Variable
  2. **Global Scope Variable:** All those variables which are outside the function termed as global variable.

Example: For global and local scope.

```
global_var = 30      # global scope
def scope():
local_var = 40       # local scope
print(global_var)
print(local_var)
scope()
print(global_var)
```

Output:

```
30
40
3
```

4.4 PYTHON PACKAGES

- Suppose we have developed a very large application that includes many modules. As the number of modules grows, it becomes difficult to keep track of them all.
- This is particularly so if they have similar names or functionality. It is necessary to group and organize them by some mean which can be achieved by packages.

4.4.1 Introduction

- A package is a hierarchical file directory structure that defines a single Python application environment that consists of modules and subpackages and sub-subpackages and so on.
- Packages allow for a hierarchical structuring of the module namespace using dot notation. Packages are a way of structuring many packages and modules which help in a well-organized hierarchy of data set, making the directories and modules easy to access.
- A package is a collection of Python modules, i.e., a package is a directory of Python modules containing an additional _ _init_ _.py file (For example: Phone/_ _init_ _.py).

4.4.2 Writing Python Packages

- Creating a package is quite easy, since it makes use of the operating system's inherent hierarchical file structure as shown in Fig. 4.5

Fig. 4.5

- Here, there is a directory named mypkg that contains two modules, p1.py and p2.py. The contents of the modules are:

p1.py

```
def m1():
print("first module")
```

p2.py

```
def m2():
print("second module")
```

- Here mypkg a folder /directory which consist of p1.py and p2.py.we can refer these two modules with dot notation(mypkg.p1, mypkg.p2) and import them with the one of the following syntaxes:

Syntax 1:

```
import <module_name>[, <module_name> ...]
```

Example:

```
>>> import mypkg.p1,mypkg.p2
>>> mypkg.p1.m1()
first module
>>> p1.m1()
```

Syntax 2:

```
from <module_name> import <name(s)>
```

Example:

```
>>> from mypkg.p1 import m1
>>> m1()
first module
>>>
```

Syntax 3:

```
from <module_name> import <name> as <alt_name>
```

Example:

```
>>> from mypkg.p1 import m1 as function
>>> function()
first module
>>>
```

Syntax 4:

```
from <package_name> import <modules_name>[, <module_name> ...]
```

Example:

```
>>> from mypkg import p1,p2
>>> p1.m1()
first module
>>> p2.m2()
second module
>>>
```

4.4.3 Standard Packages

- NumPy and SciPy are the standards packages used by Python programming.
- NumPy enriches the programming language Python with powerful data structures, implementing multi-dimensional arrays and matrices.
- SciPy (Scientific Python) is often mentioned in the same breath with NumPy. SciPy needs Numpy, as it is based on the data structures of Numpy and furthermore its basic creation and manipulation functions.

- It extends the capabilities of NumPy with further useful functions for minimization, regression, Fourier-transformation and many others.
- Both NumPy and SciPy are not part of a basic Python installation. They have to be installed after the Python installation. NumPy has to be installed before installing SciPy.

4.4.3.1 Math

- Some of the most popular mathematical functions are defined in the math module. These include trigonometric functions, representation functions, logarithmic functions and angle conversion functions.
- Two mathematical constants are also defined in this module.
- Pie (π) is a well-known mathematical constant, which is defined as the ratio of the circumference to the diameter of a circle and its value is 3.141592653589793.

```
>>> import math
>>> math.pi
3.141592653589793
>>>
```

- Another well-known mathematical constant defined in the math module is e. It is called Euler's number and it is a base of the natural logarithm. Its value is 2.718281828459045.

```
>>> import math
>>> math.e
2.718281828459045
>>>
```

- Different Mathematical functions of Math module already explained in 4.1.2 (refer details).

4.4.3.2 NumPy

- NumPy is the fundamental package for scientific computing with Python. NumPy stands for "Numerical Python". It provides a high-performance multidimensional array object, and tools for working with these arrays.
- An array is a table of elements (usually numbers), all of the same type, indexed by a tuple of positive integers and represented by a single variable. NumPy's array class is called ndarray. It is also known by the alias array.
- In NumPy arrays, the individual data items are called elements. All elements of an array should be of the same type. Arrays can be made up of any number of dimensions.
- In NumPy, dimensions are called axes. Each dimension of an array has a length which is the total number of elements in that direction.
- The size of an array is the total number of elements contained in an array in all the dimension. The size of NumPy arrays are fixed; once created it cannot be changed again.
- Numpy arrays are great alternatives to Python Lists. Some of the key advantages of Numpy arrays are that they are fast, easy to work with, and give users the opportunity to perform calculations across entire arrays.
- Fig. 4.6 shows the axes (or dimensions) and lengths of two example arrays; (a) is a one-dimensional array and (b) is a two-dimensional array.

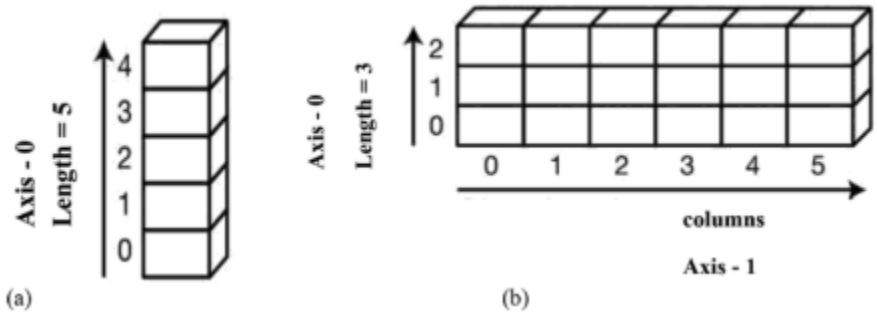


Fig. 4.6: Dimensions of NumPy Array

- A one dimensional array has one axis indicated by Axis-0. That axis has five elements in it, so we say it has length of five.
- A two dimensional array is made up of rows and columns. All rows are indicated by Axis-0 and all columns are indicated by Axis-1. If Axis-0 in two dimensional array has three elements, so its length it three and Axis-1 has six elements, so its length is six.
- Execute Following command to install numpy in window, Linux and MAC OS:

```
python -m pip install numpy
```

- To use NumPy you need to import Numpy:

```
import numpy as np        # alias np
```

- Using NumPy, a developer can perform the following operations:
  1. Mathematical and logical operations on arrays.
  2. Fourier transforms and routines for shape manipulation.
  3. Operations related to linear algebra.
  4. NumPy has in-built functions for linear algebra and random number generation.

Array Object:

- NumPy's main object is the homogeneous multidimensional array. It is a table of elements (usually numbers), all of the same type, indexed by a tuple of positive integers.
- In NumPy dimensions are called axes. The number of axes is rank. NumPy's array class is called ndarray. It is also known by the alias array.
- Basic attributes of the ndarray class as follow:

Sr. No.	Attributes	Description
1.	Shape	A tuple that specifies the number of elements for each dimension of the array.
2.	Size	The total number elements in the array.
3.	Ndim	Determines the dimension an array.
4.	nbytes	Number of bytes used to store the data.
5.	dtype	Determines the datatype of elements stored in array.

Example: For NumPy with array object.

```
>>> import numpy as np
>>> a=np.array([1,2,3])             # one dimensional array
>>> print(a)
[1 2 3]
>>> arr=np.array([[1,2,3],[4,5,6]]) # two dimensional array
>>> print(arr)
[[1 2 3]
 [4 5 6]]
>>> type(arr)
<class 'numpy.ndarray'>
>>> print("No. of dimension: ", arr.ndim)
No. of dimension:  2
>>> print("Shape of array: ", arr.shape)
Shape of array:  (2, 3)
>> >print("size of array: ", arr.size)
size of array:  6
>>> print("Type of elements in array: ", arr.dtype)
Type of elements in array:  int32
>>> print("No of bytes:", arr.nbytes)
No of bytes: 24
```

Basic Array Operations:

- In NumPy, arrays allow a wide range of operations which can be performed on a particular array or a combination of Arrays. These operation include some basic Mathematical operation as well as Unary and Binary operations. In case of +=, -=, *= operators, the existing array is modified.
  1. **Unary Operators:** Many unary operations are provided as a method of ndarray class. This includes sum, min, max, etc. These functions can also be applied row-wise or column-wise by setting an axis parameter.
  2. **Binary Operators:** These operations apply on array elementwise and a new array is created. You can use all basic arithmetic operators like +, –, /, , etc. In case of +=, –=, = operators, the existing array is modified.

Example: For basic array operators.

```
>>> arr1=np.array([1,2,3,4,5])
>>> arr2=np.array([2,3,4,5,6])
>>> print(arr1)
[1 2 3 4 5]
>>>print("add 1 in each element:",arr1+1)
add 1 in each element: [2 3 4 5 6]
>>>print("subtract 1 from each element: ", arr1-1)
subtract 1 from each element:  [0 1 2 3 4]
>>>print("multiply 10 with each element in array: ",arr1*10)
multiply 10 with each element in array:  [10 20 30 40 50]
>>>print("sum of all array elements: ",arr1.sum())
sum of all array elements:  15
>>>print("array sum=:", arr1+arr2)
array sum=: [ 3  5  7  9 11]
>>>print("Largest element in array: ",arr1.max())
Largest element in array:  5
```

Reshaping of array:

- We can also perform reshape operation using python numpy operation. Reshape is when you change the number of rows and columns which gives a new view to an object.

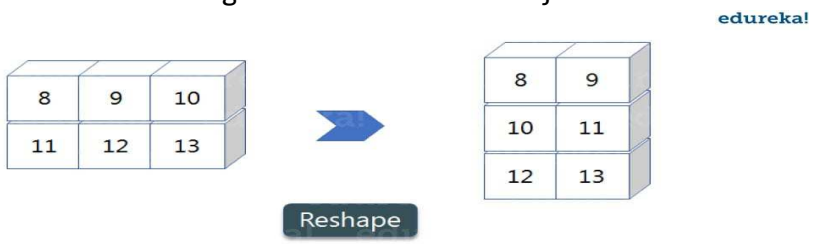


Fig. 4.7

Example:

```
>>> arr=np.array([[1,2,3],[4,5,6]])
>>> a=arr.reshape(3,2)
>>> a
array([[1, 2],
       [3, 4],
       [5, 6]])
```

Slicing of Array:

- Slicing is basically extracting particular set of elements from an array. Consider an array([(1,2,3,4),(5,6,7,8)]).
- Here, the array(1,2,3,4) is your index 0 and (3,4,5,6) is index 1 of the python numpy array. We need a particular element (say 3) out of a given array.
- Let us consider the below example:

```
>>> import numpy as np
>>> a=np.array([(1,2,3,4),(5,6,7,8)])
>>> print(a[0,2])
3
```

- Now we need the 2nd element from the zeroth and first index of the array. The code will be as follows:

```
import numpy as np
a=np.array([(1,2,3,4),(5,6,7,8)])
print(a[0:,2])
[3 7]
```

- Here, colon represents all the rows, including zero.

Array Manipulation Functions:

- Several routines are available in NumPy package for manipulation of elements in ndarray object. They can be classified into the following types

Changing Shape	Shape	Description
	reshape	Gives a new shape to an array without changing its data.
	flat	A 1-D iterator over the array.
	flatten	Returns a copy of the array collapsed into one dimension.
	ravel	Returns a contiguous flattened array.
Transpose Operations	**Operation**	**Description**
	transpose	Permutes the dimensions of an array.
	ndarray.T	Same as self.transpose().
	rollaxis	Rolls the specified axis backwards.
	swapaxes	Interchanges the two axes of an array.
Changing Dimensions	**Dimension**	**Description**
	broadcast	Produces an object that mimics broadcasting.
	broadcast_to	Broadcasts an array to a new shape.
	expand_dims	Expands the shape of an array.
	squeeze	Removes single dimensional entries from the shape of an array.
Joining Arrays	**Array**	**Description**
	concatenate	Joins a sequence of arrays along an existing axis.
	stack	Joins a sequence of arrays along a new axis.
	hstack	Stacks arrays in sequence horizontally (column wise).
	vstack	Stacks arrays in sequence vertically (row wise).
Splitting Arrays	**Array**	**Description**
	split	Splits an array into multiple sub-arrays.
	hsplit	Splits an array into multiple sub-arrays horizontally (column wise).
	vsplit	Splits an array into multiple sub-arrays vertically (row wise).
Addint/Removing Elements	**Elements**	**Description**

	resize	Returns a new array with the specified shape.
	append	Appends the values to the end of the array.
	insert	Inserts the value along the given axis before the given indices.
	delete	Returns a new array with sub array along an axis deleted.
	unique	Finds the unique elements in the array.
Bitwise Operations	**Operation**	**Description**
	bitwise_and	Compute bitwise AND operation of array elements.
	bitwise_or	Compute bitwise OR operation of array elements.
	invert	Compute bitwise NOT.
	left_shift	Shifts bits of a binary representation to the left.
	right_shift	Shifts bits of a binary representation to the right.

- Using numpy, we can deal with 1D polynomials by using the class poly1d. This class takes coefficients or roots for initialization and forms a polynomial object. When we print this object we will see it prints like a polynomial.
- Let us have a look at example code:

```
from numpy import poly1d
poly1 = poly1d([1,2,3])
print(poly1)
print("\nSquaring the polynomial: \n")
print(poly1* poly1)
print("\nIntegrating the polynomial: \n")
print(poly1.integ(k=3))
print("\nFinding derivative of the polynomial: \n")
print(poly1.deriv())
print("\nSolving the polynomial for 2: \n")
print(poly1(2))
```

- The program when run gives output like below image, we hope comments made it clear what each piece of code is trying to do:

```
    2
1 x + 2 x + 3
Squaring the polynomial:
   4     3     2
1 x + 4 x + 10 x + 12 x + 9
Integrating the polynomial:
          3     2
0.3333 x + 1 x + 3 x + 3
Finding derivative of the polynomial:
2 x + 2
Solving the polynomial for 2:
11
```

4.4.3.3 SciPy

- SciPy is a library that uses NumPy for more mathematical functions. SciPy uses NumPy arrays as the basic data structure, and comes with modules for various commonly used tasks in scientific programming, including linear algebra, integration (calculus), ordinary differential equation solving, and signal processing.
- Following command execute to install SciPy in Window, Linux and MAC OS:

```
python -m pip install scipy
```

- SciPy is organized into subpackages covering different scientific computing domains. These are summarized in the following table:

Sr. No.	Subpackage	Description
1.	cluster	Clustering algorithms.
2.	constants	Physical and mathematical constants.
3.	fftpack	Fast Fourier Transform routines.
4.	integrate	Integration and ordinary differential equation solvers.
5.	interpolate	Interpolation and smoothing splines.
6.	io	SciPy has many modules, classes, and functions available to read data from and write data to a variety of file formats.
7.	linalg	Linear algebra.
8.	ndimage	N-dimensional image processing.
9.	odr	Orthogonal distance regression.
10.	optimize	Optimization and root-finding routines.
11.	signal	Signal processing.
12.	sparse	Sparse matrices and associated routines.
13.	spatial	Spatial data structures and algorithms.
14.	special	Special functions.
15.	stats	Statistical distributions and functions.

Example 1: Using linalg sub package of SciPy.

```
>>> import numpy as np
>>> from scipy import linalg
>>> a = np.array([[1., 2.], [3., 4.]])
>>> linalg.inv(a)
array([[-2. ,  1. ],
       [ 1.5, -0.5]])
>>>
```

Example 2: Using linalg sub package of SciPy.

```
>>> import numpy as np
>>> from scipy import linalg
>>> a = np.array([[1,2,3], [4,5,6], [7,8,9]])
>>> linalg.det(a)
0.0
>>>
```

Example 3: Using special sub package of SciPy.

```
>>> from scipy.special import cbrt
>>> cb=cbrt([81,64])
>>> cb
array([4.32674871, 4.           ])
```

4.4.3.4 Matplotlib

- matplotlib.pyplot is a plotting library used for 2D graphics in python programming language. It can be used in python scripts, shell, web application servers and other graphical user interface toolkits.

- There are various plots which can be created using python matplotlib like bar graph, histogram, scatter plot, area plot, pie plot.
- Following command execute to install matplotlib in Window, Linux and MAC OS:
  `python –m pip install matplotlib`

Importing matplotlib:

```
from matplotlib import pyplot as plt
```

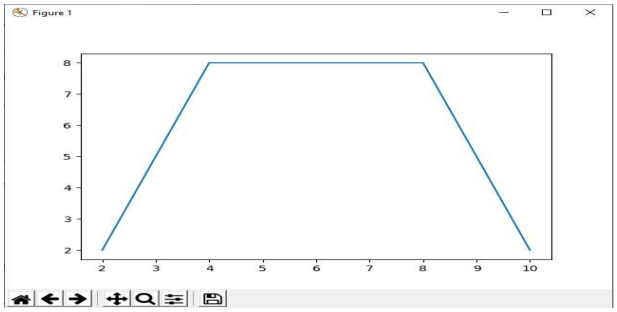
OR

```
import matplotlib.pyplot as plt
```

Example: Line Plot

```
>>> from matplotlib import pyplot as plt
>>> x=[2,4,8,10]
>>> y=[2,8,8,2]
>>> plt.plot(x,y)
[<matplotlib.lines.Line2D object at 0x02E69B70>]
>>> plt.show()
```

Output:



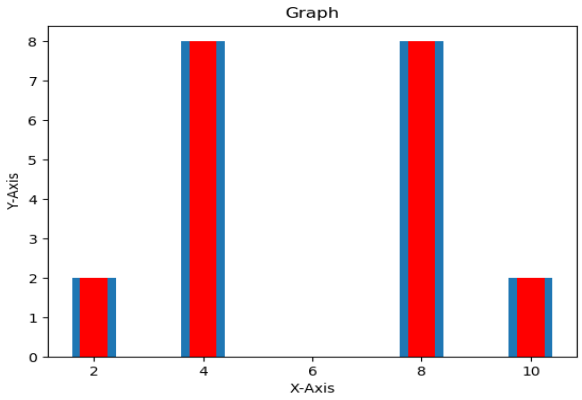
Bar Graph:

- A bar graph uses bars to compare data among different categories. It is well suited when you want to measure the changes over a period of time. It can be represented horizontally or vertically.

Example: For bar graph.

```
>>> from matplotlib import pyplot as plt
>>> x=[2,4,8,10]
>>> y=[2,8,8,2]
>>> plt.xlabel('X-Axis')
Text(0.5, 0, 'X-Axis')
>>> plt.ylabel('Y-Axis')
Text(0, 0.5, 'Y-Axis')
>>> plt.title('Graph')
Text(0.5, 1.0, 'Graph')
>>> plt.bar(x,y,label="Graph",color='r',width=.5)
<BarContainer object of 4 artists>
>>> plt.show()
```

Output:



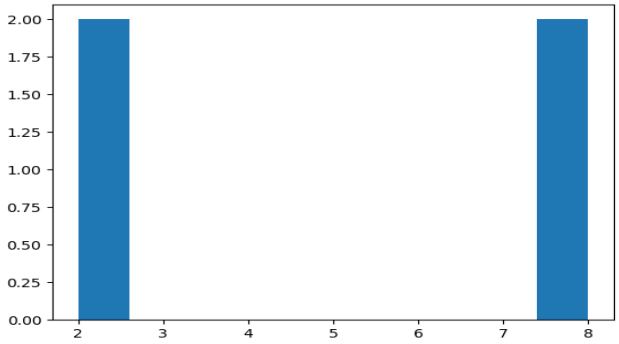
Histogram:

- Histograms are used to show a distribution whereas a bar chart is used to compare different entities. Histograms are useful when you have arrays or a very long list.

Example: For histogram.

```
>>> from matplotlib import pyplot as plt
>>> y=[2,8,8,2]
>>> plt.hist(y)
(array([2., 0., 0., 0., 0., 0., 0., 0., 0., 2.]), array([2. , 2.6, 3.2, 3.8, 4.4, 5.
, 5.6, 6.2, 6.8, 7.4, 8. ]), <a list of 10 Patch objects>)
>>>plt.show()
```

Output:



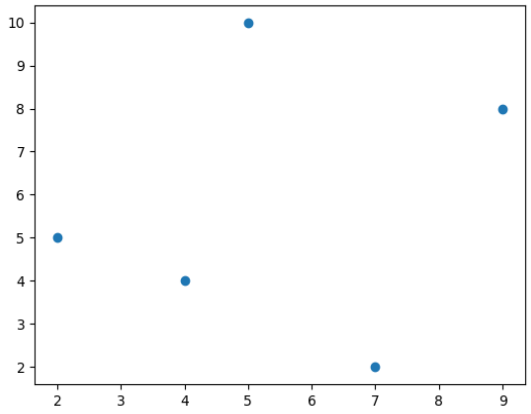
Scatter Plot:

- Usually we need scatter plots in order to compare variables, for example, how much one variable is affected by another variable to build a relation out of it.
- The data is displayed as a collection of points, each having the value of one variable which determines the position on the horizontal axis and the value of other variable determines the position on the vertical axis.

Example: For scatter plot.

```
>>> from matplotlib import pyplot as plt
>>> x = [5, 2, 9, 4, 7]
>>> y = [10, 5, 8, 4, 2]
>>> plt.scatter(x,y)
<matplotlib.collections.PathCollection object at 0x05792770>
>>> plt.show()
```

Output:



4.4.3.5 Pandas

- Pandas is an open-source Python Library providing high-performance data manipulation and analysis tool using its powerful data structures.
- It is built on the Numpy package and its key data structure is called the DataFrame. DataFrames allow you to store and manipulate tabular data in rows of observations and columns of variables.

Installing Pandas:

```
pip install pandas
```

Data structures supported by Pandas:

- Pandas deals with the following three data structures:

Sr. No.	Data Structure	Dimensions	Description
1.	Series	1	1D labeled homogeneous array, size immutable.
2.	Data Frames	2	General 2D labeled, size-mutable tabular structure with potentially heterogeneously typed columns.
3.	Panel	3	General 3D labeled, size-mutable array.

Series:

- Series is a one-dimensional array like structure with homogeneous data. The Series is a one Dimensional array which is Labelled and it is capable of holding array of any type like Integer, Float, String and Python Objects.
- For example, the following series is a collection of integers 10, 22, 30, 40,...

```
pandas.Series(data, index, dtype, copy)
```

- It takes four arguments:
  1. **data:** It is the array that needs to be passed so as to convert it into a series. This can be Python lists, NumPy Array or a Python Dictionary or Constants.
  2. **index:** This holds the index values for each element passed in data. If it is not specified, default is numpy.arange(length_of_data).
  3. **dtype:** It is the datatype of the data passed in the method.
  4. **copy:** It takes a Boolean value specifying whether or not to copy the data. If not specified, default is false.

Example 1: Using Series data structure of Panda.

```
>>> imort pandas as pd
>>> import numpy as np
>>> numpy_arr = array([2, 4, 6, 8, 10, 20])
>>> si = pd.Series(arr)
>>> print(si)
0     2
1     4
2     6
```

```
3     8
4     10
5     20
dtype: int32
```

Example 2: Using Series data structure of Panda.

```
>>> import pandas as pd
>>> Data=[10,20,30,40,50]
>>> Index=['a','b','c','d','e']
>>> si=pd.Series(Index,Data)
>>> si
10    a
20    b
30    c
40    d
50    e
dtype: object
>>>
```

Data Frames:

- Data Frame is a two-dimensional array with heterogeneous data. We can convert a Python's list, dictionary or Numpy array to a Pandas data frame.
- For example:

Roll No	Name	City
11	Vijay	Thane
12	Amar	Pune
13	Santosh	Mumbai

```
pandas.DataFrame(data, index, columns, dtype, copy)
```

- It takes following arguments:
  1. **data:** The data that is needed to be passed to the DataFrame() Method can be of any form line ndarray, series, map, dictionary, lists, constants and another DataFrame.
  2. **index:** This argument holds the index value of each element in the DataFrame. The default index is np.arange(n).
  3. **columns:** The default values for columns is np.arange(n).
  4. **dtype:** This is the datatype of the data passed in the method.
  5. **copy:** It takes a Boolean value to specify whether or not to copy the data. The default value is false.

Example:

```
>>> import pandas as pd
>>> li = [1, 2, 3, 4, 5, 6]
>>> df = pd.DataFrame(li)
>>> print(df)
0  1
1  2
2  3
3  4
4  5
5  6
```

Example: Using DataFrame data structure of Panda.

```
>>> import pandas as pd
>>> dict={"Name":["Meenakshi","Anurag","Khwahish","Aarul"],"Age":[35,39,7,3]}
>>> df=pd.DataFrame(dict)
```

```
>>> print(df)
      Name         Age
0     Meenakshi    35
1     Anurag       39
2     Khwahish     7
3     Aarul        3
>>>
```

Panel:

- Panel is a three-dimensional data structure with heterogeneous data. It is hard to represent the panel in graphical representation. But a panel can be illustrated as a container of DataFrame.

```
pandas.Panel(data, item, major_axis, minor_axis, dtype, copy)
```

- It takes following arguments:
  1. **data:** The data can be of any form like ndarray, list, dict, map, DataFrame.
  2. **item:** axis 0
  3. **major_axis:** axis 1
  4. **minor_axis:** axis 2
  5. **dtype:** The data type of each column
  6. **copy:** It takes a Boolean value to specify whether or not to copy the data. The default value is false.

Example:

```
import pandas as pd
import numpy as np
information = np.random.rand(1, 2, 3)
pandas_panel = pd.Panel(information)
print(pandas_panel)
```

Output:

```
<class 'pandas.core.panel.Panel'>
Dimensions: 1 (items) x 2 (major_axis) x 3 (minor_axis)
Items axis: 0 to 0
Major_axis axis: 0 to 1
Minor_axis axis: 0 to 2
```

4.4.3.6 User Defined Packages

- We organize a large number of files in different folders and subfolders based on some criteria, so that we can find and manage them easily. In the same way, a package in Python takes the concept of the modular approach to next logical level.
- As we know, a module can contain multiple objects, such as classes, functions, etc. A package can contain one or more relevant modules.
- Physically, a package is actually a folder containing one or more module files. Let's create a package named MyPkg, using the following steps:

Step 1: Create a folder MyPkgon "C:\Users\Meenakshi\AppData\Local\Programs\Python\Python37\".

Create modules Message.py and Mathematics.py with following code:

Message.py

```
defSayHello(name):
print("Hello " + name)
return
```

Mathematics.py

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

```
def power(x,y):
return x**y
```

Step 2: Create an empty _ _init_ _.py file in the MyPkg folder. The package folder contains a special file called _ _init_ _.py, which stores the package's content. It serves two purposes:

1. The Python interpreter recognizes a folder as the package if it contains _ _init_ _.py file.
2. _ _init_ _.py exposes specified resources from its modules to be imported.

- An empty _ _init_ _.py file makes all functions from above modules available when this package is imported. Note that _ _init_ _.py is essential for the folder to be recognized by Python as a package. We can optionally define functions from individual modules to be made available.

Step 3: Create P1.py file in MyPkg folder and write following code:

```
fromMyPkg import Mathematics
fromMyPkg import Message
greet.SayHello("Meenakshi")
x=functions.power(3,2)
print("power(3,2) : ", x)
```

Output:

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

Using _ _init_ _.py File:

- 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:

_ _init_ _.py

```
from .Mathematics import average, power
from .Message import SayHello
```

- The specified functions can now be imported in the interpreter session or another executable script. Create test.py in the MyPkgfolder.nd write following code:

test.py

```
fromMyPkg import power, average, SayHello
SayHello()
x=power(3,2)
print("power(3,2) : ", x)
```

- Note that functions power() and SayHello() are imported from the package and not from their respective modules, as done earlier. The output of above script is:

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

Practice Questions

1. What is function?
2. What is module?
3. What is package?
4. Define function. Write syntax to define function. Give example of function definition.
5. Can a Python function return multiple values? If yes, how it works?
6. How function is defined and called in Python.
7. Explain about void functions with suitable examples.

8. What is actual and formal parameter? Explain the difference along with example.
9. Explain about fruitful functions with suitable examples.
10. Discuss the difference between local and global variable.
11. Explain any five basic operations performed on string.
12. Explain math module with its any five functions.
13. Differentiate between match() and search() function. Explain with example.
14. Explain type conversion of variable in Python.
15. Write a function that takes single character and prints 'character is vowel' if it is vowel, 'character is not vowel' otherwise.
16. Explain various string operations that can be performed using operators in Python.
17. Explain with an example, how + and * operators work with strings.
18. Explain str.find() function with suitable example.
19. Define is module? What are the advantages of using module?
20. How to create a module and use it in a python program explain with an example.
21. Explain various functions of math module.
22. List and explain any four built in string manipulation functions supported by Python.
23. Explain string slicing in Pyhton. Show with example.
24. Explain the concept of namespaces with an example.
25. Write about the concept of scope of a variable in a function.
26. Write about different types of arguments in a function.
27. What type of parameter passing is used in Python? Justify your answer with sample programs.
28. Write in brief about anonymous functions.
29. What is the use of islower() and isupper() method?
30. Give the syntax and significance of string functions: title() and capitalize().
31. What is recursive function? Write a Python program to calculate factorial of a number using recursive function?
32. Write a python program to calculate factorial of given number using recursive function.
33. Write a python program to find reverse of a given number using user defined function.
34. Write a Python program that interchanges the first and last characters of a given string.

■■■