python

UNIT – II

Contents

Function

A *function* is a block of code that takes in some data and either performs some kind of transformation and returns the transformed data or performs some task on that data..

Types of Functions

Built in functions- Functions that are built into Python.

Example:

print() :function prints the given object to the standard output device (screen) or to the text stream file.

id() :returns identify of an Object

range():return sequence of integers between start and stop

User defined functions: functions defined by the users themselves.

Example :

sum()

fact()

Defining a Function

Syntax:

```
def name_of_the_function(parameters):
        '" doc string of the function"'
        ------------------
        ------------------    // body of the function
        ------------------
        return the return value or expression
```

Ex:

```
def sum(num1,num2):
        result=num1+num2
        return result
c=sum(10,20)
print(c)
print(sum(10.3,14.7))
print(sum(2+3j,3+3J))
print(sum('hello','hi'))
```

- ➢ The first line is known as the *function header*. It marks the beginning of the function definition.
- ➢ The function header begins with the key word def, followed by the name of the function, followed by a set of parentheses, followed by a colon.
- ➢ Function name should be in lower case and should be separated with an underscore if it has multiple words

- ➢ Function doc strings are optional describes what the function does. Function doc strings can be displayed using the syntax ***functionname._ _doc_ _***

- ➢ Beginning at the next line is a set of statements known as a block. A *block* is simply a set of statements that belong together as a group.
- ➢The body of the function is identified by 4 spaces
- ➢ An optional return statement to return a value from the function.

Calling a Function

➢ A function definition specifies what a function does, but it does not cause the function to execute. To execute a function, you must *call* it.

➢ When a function is called, the interpreter jumps to that function and executes the statements in its block.

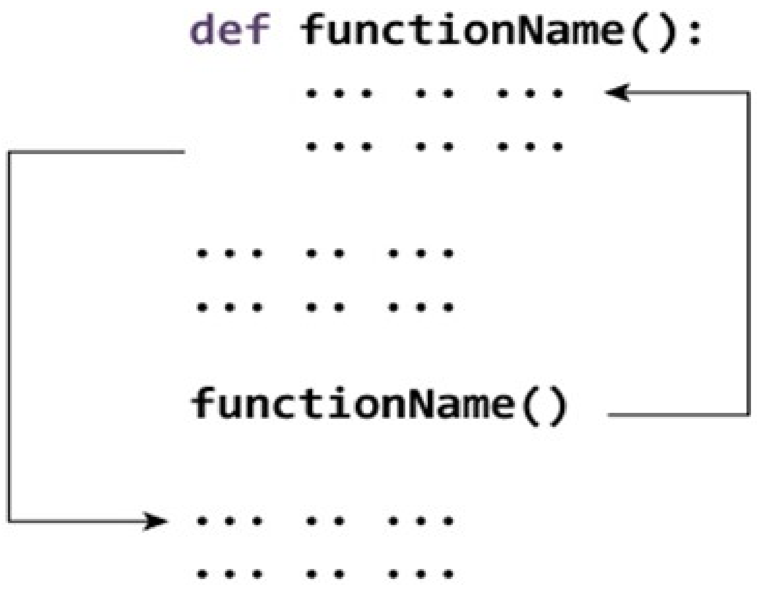
➢ When the end of the block is reached, the interpreter jumps back to the part of the program that called the function

➢Once we have defined a function, we can call it from another function, program or even the Python prompt.

Example :

```
def display(name):
        print("hello"+name)
>>> display("snist")

def display(name):
        print("hello"+name)
display("ITF4")
```



Function with return

- The return statement is used to exit a function and go back to the place from where it was called.
- If the function has a return value it returns a value else the function would return **None**

Example: **return a** **return 100**

return a, b, c **return lst**

Function with no return value

- Function performs some task but it does not return anything

Ex

```
def greet():
        print("welcome to python programming")
greet()
```

Output

```
welcome to python programming
```

Function with return value

Ex

```
def add(a , b):
        r = a+b
        return r
c=add(10,20)
print c
```

Output

```
30
```

return multiple values from function

A function returns a single value in the programming languages like C or Java. But in python, a function can return multiple values

return a, b, c

Here ,three values which are in a , b and c are returned .These values are returned by function as a tuple .If you want to grab these values ,we can use three variables at the time of calling function as **x , y, z=fun()**

Ex

```
def fun(a,b):
        x=a+b
        y=a-b
        return(x,y)
m, n=fun(10,20)
print(m,n)
```

Output

```
30,-10
```

return list from function

```
def fun():
        a=[1,2,3,4]
        return a
x=fun()
print(x)
```

Output

```
1,2,3,4
```

Function

In python ,functions are considered as first class objects. When we create a function, the python interpreter internally creates an object. Since functions are objects, we can pass a function to another function just like we pass an object to a function. Also it is possible to return a function from another function

- **Assign a function to a variable**

Ex:

```
def  greet(msg):
        return "HELLO"+msg
x=greet
print(x("snist"))
```

Output

```
HELLO SNIST
```

- **Define one function inside another function**

```
def  add(a):
        def  fun2( ):
                b=int(input("enter b"))
                return b
        return(a+fun2())
print(add(5))
```

➢ Pass a function as parameter to another function

```
def   sub(x , y):
            return (x-y))
def  readb():
            b=int (input("enter b"))
            return b
a=int(input("enter a"))
c=sub( a, readb())
Print(c)
```

```
def   sub(fun,y):
            return (fun()-y)
def  geta():
            a=int (input("enter a"))
            return a
b=int(input("enter b"))
c=sub( geta, b)
print(c)
```

➢ A function can return another function

```
def  fun():
        def mul():
                a=int(input("enter a"))
                b=int(input("enter b"))
                return(a*b)
        return(mul())
f=fun()
print(f())
```

Advantages of Functions

- ➢ Reducing duplication of code
- ➢ Decomposing complex problems into simpler pieces
- ➢ Improving clarity of the code.
- ➢ Reuse of code

Function arguments

In Python, user-defined functions can take four different types of arguments.

- Positional arguments
- Keyword arguments
- Default arguments
- Variable-length arguments

Positional arguments

- positional arguments are the arguments passed to a function in correct positional order. Here, the number and position of arguments in the function call should match exactly with the function definition.

Ex

```
def  display(s1,s2):
          print(s1+s2)
display(‘New’ , ’Delhi’)
```

- To call the function display*()*, you definitely need to pass two arguments, otherwise it would give a syntax error

Keyword arguments

- Keyword arguments are arguments that identify the parameters by their names.
- When we call a function with some values, these values get assigned to the arguments according to their position
- This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters.

EX

```
def   div(a, b):
            print(a/b)
      div(4,5)
      div(a=6,b=2)
      div(b=4,a=10)        div(a=10,b=4)
```

- we can mix positional arguments with keyword arguments during a function call. But we must keep in mind that keyword arguments must follow positional argument

```
div(12,b=10)
div(a=10,2)
```

Default arguments

- Function arguments can have default values. We can provide a default value to an argument by using the assignment operator (=).
- the parameters name and msg has default values. If a value is provided, it will overwrite the default value.

Ex1

```
def  display(name = “snist”, msg=“Good Morning”):
        print(“Hello,”+name+’ ’+msg)
    display()
    display(name=“ITF4”, msg=“Good morning”)
```

Ex2

```
def  mul(a,b=10)
        print(a*b)
mul(4)
    mul(4,5)
```

- Any number of arguments in a function can have a default value. But once we have a default argument, all the arguments to its right must also have default values.

```
mul(a=12,8)                    #gives syntax error
```

Variable-length arguments

- sometimes, we do not know in advance the number of arguments that will be passed into a function. Python allows us to handle this kind of situation through function calls with variable length arguments.
- To denote this kind of argument, we use an asterisk (*) before the parameter name in the **function definition**
- You may need to process a function for more arguments than you specified while defining the function.
- These arguments are called *variable-length* arguments and are not named in the function definition, unlike required and default arguments.

EX

```
 def greet(*names):
    for name in names:
                          print("Hello",name)
greet("Mounika","Luke","steve","John")
```

```
def  calsum( *arr):
                        sum=0
                        for   i  in
    arr:

    sum+=i
```

Recursion

A function called by itself is called recursion

Ex:

```
def fact(x):
            if x == 1:
                  return 1
            else:
             return (x * fact(x-1))
num = int(input(“enter number”))
print("The factorial of", num, "is", fact(num))
```

Anonymous Function

- A function which does not contain any name is known as a lambda function or Anonymous function.
- we can assign Lambda function to the variable and we can call the function through that variable.
- Lambda functions can have any number of arguments but only one expression. The expression is evaluated and returned. Lambda functions can be used wherever function objects are required.

Syntax: lambda arguments: expression

Ex

```
myfun=lambda  x: x*x
p=myfun(10)
print(p)
```

lambda functions can be used with 3 built in functions i.e. map, filter and reduce

filter

The filter() function in Python takes in a function and a list as arguments. It filter out all the elements of a sequence “sequence”, for which the function returns True.

Ex
```
my_list=[1,5,4,6,8,11,3,12]
new_list=list(filter(lambda x:(x%2==0),my_list))
print(new_list)
```

Output
```
[4,6,8,12]
```

map

The map() function in Python takes in a function and a list as argument. The function is called with a lambda function and a list and a new list is returned which contains all the lambda modified items returned by that function for each item.

Ex
```
my_list=[1,5,4,6,8,11,3,12]
new_list=list(map(lambda x:( x*x),my_list))
print(new_list)
```

Output [1,25,16,36,64,121,9,144]

reduce()

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

EX

```
from  functools  import  reduce
li = [5, 8, 10, 20, 50, 100]
sum = reduce((lambda x, y: x + y), li)
print (sum)
```

Global and Local Variables

Local variable

- A local variable is created inside a function and cannot be accessed by statements that are outside the function.
- Different functions can have local variables with the same names because the functions cannot see each other's local variables.

- **EX**

```
def  main():
        a=10
        a/=2
        print(a)
main()
print(a)     # shows error
```

Global variable

➢ When a variable is created by an assignment statement that is written outside all the functions in a program file, the variable is *global*.

➢ A global variable is accessible to all the functions in a program file.

EX1

```
value = 10
def   show():
          print(“inside show  ”,value)
def  display():
          print(inside display", value)
show()
display()
print(“outside “,value)
```

Output

```
inside show  10
inside display 10
outside 10
```

Ex2:

```
a=10
def fun():
          a=10
          print(“ inside fun “,a)
fun()
print(“value of a =“, a)
```

global keyword

In order to modify the global variables data with in the function we should use global declaration with in the function.

EX1:

```
a=10
def  fun():
        a+=2      # shows error
        print(a)
fun()
```

EX2:

```
a=10
def  fun1():
        global a
        print(a)
        a+=30
def  fun2():
        print(a)
fun1()
fun2()
```

Output:

```
10
40
```

Mutable V/s Immutable Data Types

Mutable Data Type

- objects can be modified after creation
  Ex: Lists, Dictionary, Sets
- Operations like add, delete and update can be performed

Immutable Data Type

- objects cannot be modified after creation
  Ex: Strings, Tuples
- Operations like add, delete and update cannot be performed

Strings

- In Python, string is a sequence of Unicode character. Unicode was introduced to include every character in all languages and bring uniformity in encoding.

Create a string

- Strings can be created by enclosing characters inside a single quote (‘), double quotes (“) and triple code (“ “ “)
- It must start and end with same type of quote
- Tripe quotes are used to span string across multiple lines
- Index starts from zero
- Can be accessed using negative indices. Last character will start with -1 and traverses from right to left

Syntax:

```
word = ‘Python Programming’
sentence = “ Object Oriented Programming”
paragraph =“ “ “ Python is a Object Oriented Programming Language. It
is a Biginners Language “””
```

Ex :

```
my_string = ‘Hello‘
print(my_string)
my_string1 = input("Enter a string")
print(my_string1)
my_string2 = """Hello, welcome to the world of Python"""
print(my_string2)
```

Access characters in a string

- ➢ We can access individual characters using **indexing** and a range of characters using **slicing.** Index starts from 0. Trying to access a character out of index range will raise an Index Error.
- ➢ The index must be an integer. We can't use float or other types, this will result into Type Error.
- ➢ Python also allows **negative** indexing for its sequences.

-6	**-5**	**-4**	**-3**	**-2**	**-1**
P	**Y**	**T**	**H**	**O**	**N**
0	**1**	**2**	**3**	**4**	**5**

Ex:

```
str = 'python’
print('str = ', str)
print('str[0] = ', str[0])            #first character
print('str[-1] = ', str[-1])          #last character
print (‘str[-5]=‘,str[-5])            # second character
```

Access characters in a string(using slice:)

stringname[start:stop:stepsize]

str=‘Hello Python’

Example

```
print(str[1:5])
print(str[1:7:1])
print(str[1:6:2])
print(str[1:5])
print(str[3:])
print(str[::2])
print(str[4::])
print(str[:3:])
print(str[-4:-1])
print(str[3:-2])
print(str[::]
```

When step size is negative ,then the elements are displayed from right to left

```
Print(str[::-1])
```

Concatenation of Two or More Strings

➢ Joining of two or more strings into a single one is called concatenation. The + operator does this in Python.

Ex:
```
s1 = 'Hello’
s2 ='World!‘
s3=s1+s2
print(s3)                    #  HelloWorld
```

Repeating strings

The * operator can be used to repeat the string for a given number of times.

Ex:
```
s1 = 'Hello’
print(s 1* 3)                #  HelloHelloHello
print(s1[1:3]*4)
```

String Membership Test

We can test if a sub string exists within a string or not using the keyword **in**

Ex ‘a’ in ‘program’ # True

change or delete a string

- Strings are immutable. This means that elements of a string can not be changed once it has been assigned. We can simply reassign different strings to the same name.
- We can not delete or remove characters from a string. But deleting the string entirely is possible using the keyword del.

Ex:

```
str = 'perl'
str = 'Python‘
print (str)          # ‘Python’
str[3]=‘l’                     #Error
del str
print (str)         # Name Error
```

Escape Sequence

➢ If we want to print a text like –He said, "What‘s there?"-we can neither use single quote or double quotes. This will result into Syntax Error as the text itself contains both single and double quotes.

➢ One way to get a round this problem is to use triple quotes. Alternatively, we can use escape sequences.

➢ An escape sequence starts with a backslash and is interpreted differently. If we use single quote to represent a string, all the single quotes inside the string must be escaped. Similar is the case with double quotes

Example

```
# using triple quotes             print('''He said, "What's there?"''')
# escaping single quotes          print('He said, "What\'s there?"')
# escaping double quotes          print("He said, \"What's there?\"")
```

Escape character	**Meaning**
\n	newline
\\	backslash(keeps a \ )
\’	single quote(keeps ‘)
\\”	double quote(keeps “)
\a	Bell or Alert
\b	Backspace
\f	form feed
\r	carriage return
\t	Horizontal tab space
\v	Vertical tab space

String Formatting Operator

- Python defines % binary operator to work on strings. The % operator provides a simple way to format values as strings .
- on the left of the % operator provide a format string containing one or more embedded conversion targets.
- on the right of the % operator provide the object that you want python to insert into the format string.

Ex

```
print ("My name is %s and average is %f!" % ('student', 85) )
```

Output

```
My name is student and average is 85 !
```

Code	Meaning
%s	string (or any object)
%r	s, but uses repr, not str
%c	character
%d	decimal (integer)
%i	integer
%u	unsigned (integer)
%o	octal integer
%x	Hex integer
%X	x, but prints uppercase
%e	floating point exponent
%E	e, but prints in uppercase
%f	Floating point decimal
%g	Floating point e or f
%a	floating point E or f
%%	literal %

format() method

the format() method is available with the string object. It contains curly braces {} as place holders.

Default order

r = "{} {} {}".format(‘this’,‘ is', ‘ITF4’) # this is ITF4

Positional order

r = "{1},{0} and {2}".format(‘java‘ ,‘c',‘python') # c, java and python

Keyword order

r = "{p},{c} and {j}".format(j='Java‘ , p=‘python', c=‘c') # python ,c and java

Functions and Methods

- Python has a number of string functions which are in the string library
- These functions are already built into every string - we invoke them by appending the function to the string variable
- These functions do not modify the original string instead they return a new string that has been altered
- len() function returns the number of bytes in a string

capitalize
casefold
center
count
endswith
expandtabs
find
format
index
isalnum
isalpha

isdecimal
isdigit
isidentifier
islower
isnumeric
isprintable
isspace
istitle
isupper
join
ljust
rjust

lower
lstrip
rstrip
Strip
split
partition
replace
rfind
rindex
rpartition
rsplit

startswith
swapcase
title
translate
Upper
Encode

String methods

capitalize()

returns a copy of s with the first character converted to uppercase and all other characters converted to lowercase.

center(width)

returns a string which is padded with the specified character. It doesn’t modify the original string.

count()

returns the number of occurrences of the substring in the given string

encode()

returns encoded version of the given string

endswith()

returns true if a string ends with the specified suffix , if not returns false.

String methods

find()

returns the index of first occurrence of the substring , if not found it returns -1

index()

returns the index of a substring inside the string , if the substring is not found it raises an exception.

isalnum()

returns true if all characters in the string are alphanumeric, if not returns false.

isalpha()

returns true if all characters in the string are alphabets, if not returns false.

isdigit()

returns true if all characters in a string are digits, if not returns false.

String methods

expandtabs()

returns a copy of string with all tab characters '\t' replaced with whitespace characters until the next multiple of tab size parameter.

isspace()

returns true if there are only whitespace characters in the string.

istitle()

returns true if the string is a title string, if not returns false.

isuppercase()

returns whether or not all charcters in a sting are uppercase or not.

join()

Returns a string concatenated with the elements of an iterable

ljust()

Returns a left justified string of a given minimum width

rjust()

Returns a right justified string of a given minimum width

lower()

Converts all uppercase characters in a string into lowercase characters

istring()

returns a copy of the string with leading characters removed

split()

breaks up a string at the specified separator

List

- ➢ Creating a list is as simple as putting different comma-separated values between square brackets
- ➢ Important thing about a list is that items in a list need not be of the same type
- ➢ An ordered group of sequences enclosed inside square brackets and separated by symbol ,
- ➢ List are mutable type,Python will not create a new list if we modify an element in the list.

Creation of list

Syntax:

```
list1= [ ]                          # Creation of empty List
list2= [ Sequence1, ]       # In this case symbol , is not mandatory
list3= [ Sequence1, Sequence2]
```

Ex:

```
lan=[]
language=[“Python”]
languages =[“Python”, “C”,”C++”,”Java”, ”Perl”, ”R”]
```

Accessing Values in Lists

➢A list can be created by putting the value inside the square bracket and separated by comma.

➢The elements are stored in the index basis with starting index as 0.

Syntax:

```
<list_name>=[value1,value2,value3,...,value n]
```

For accessing list :

```
<list_name>[index]
```

Ex:

```
lst1 = ['physics', 'chemistry', 1997, 2000]
lst2 = [1, 2, 3, 4, 5, 6, 7 ]
print ("lst1[0]: ", lst1[0])
print ("lst2[1:5]: ", lst2[1:5])
print (“lst1[0:] : “, lst1[0:])
```

➢Python allows negative indexing for its sequences. The index of -1 refers to the last item, -2 to the second last item and so on.

```
print(lst2[-4])
print(lst[-4:-1])
```

Note: If the index provided in the list slice is outside the list, then it raises an IndexErrorexception

Nested list:

A list can even have another list as an item. This is called nested list.

Ex:

```
my_lst=['ITF4',[2,4,6,8],[' a',' e',' i', 'o' ,'u']]
matrix=[[1,2,3],[2,4,6],[3,5,7]]
```

Accessing nested list

```
nlst=['SNIST',[1,2,3,4,5]]
print(nlst[0][3])
 print(nlst[0][-1])
print(nlst[0][1:2])
print(nlst[1][1])
print(nlst[1][2:5])
print(nlst[1][-3])
```

Note

List do not store the elements directly at the index. In fact a reference is stored at each index which subsequently refers to the object stored somewhere in the memory. This is due to the fact that some objects may be large enough than other objects and hence they are stored at some other memory location.

List operations

Adding Lists **:** The + operator concatenates lists:

```
a = [1, 2, 3]        # list1
b = [4, 5, 6]        # list2
c = a + b            # list3
print c
```

Output : [1,2, 3, 4, 5, 6]

Replicating lists: Similarly, the * operator repeats a list a given number of times:

```
a= [7, 8, 9]
b=a*3
print b
```

Output [7, 8, 9, 7, 8, 9, 7, 8, 9]

Note: '+'operator implies that both the operands passed must be list else error

```
list1=[10,20]
list1+30
print list1          # error
```

List membership: in and not in are Boolean operators that test membership in a sequence.

```
list1=[1,2,3,4,5,6]
print 5 in list1                    #True
```

Other Operations:

Apart from above operations various other functions can also be performed on List such as Updating, Appending and Deleting elements from a List:

Updating elements in a List:

To update or change the value of particular index of a list, assign the value to that particular index of the List.

Syntax: <list_name>[index]=<value>

Ex:

```
ls=[2,4,6,8]
ls[0]=1
print(ls)
ls[1:4]=[3,5,7]
print(ls)
```

Appending elements to a List:

append() method is used to append i.e., add an element at the end of the existing elements.

Syntax: <list_name>.append(argument)

```
ls.append(8)
```

Note: extend() method is used to add several items at the end of list

```
ls.extend(10,12,14)
```

Delete List Elements

del statement can be used to an element from the list. It can also be used to delete items from start Index to end Index. It can even delete the list entirely.

```
lst1 = [‘ITF4', ‘python', 89, 97.3,67,46.8];
print (lst1)
del (lst1[1])
print ("After deleting value at index 1 : “)
print( lst1)
del (lst1[3:5]
print(lst1)
del lst1
print lst1
```

Functions and Methods of Lists:

There are many Built-in functions and methods for Lists. They are as follows:

There are following List functions:

Function	**Description**
min(list)	Returns the minimum value from the list given.
max(list)	Returns the largest value from the given list.
len(list)	Returns number of elements in a list.
cmp(list1,list2)	Compares the two list.
list(sequence)	Takes sequence types and converts them to lists.
Any	return true any element of list is true
All	return true when all elements in list is true
Sorted	return sorted list from given iterable

Example program for min , max and len

```
f3, f4 = [‘c1’,’d3’,’f5’], [‘j1’,’k2’,’l3’,’m4’,’n6’]
print ("First list length : ", len(list1))
print ("Second list length : ", len(list2))
m1=[77,87,95,100,67]
print (“min from m1 : ", min(m1))
lst_string=['a', 'zx', 'y']
print ("min['a', 'zx', 'y'] : ", min(lst_string))
print(“max from m1:”,max(m1)
print(“ max from lst_string:”,max(lst_string))
```

Example

```
aTuple= (123, 'xyz', 'zara', 'abc')
aList= list(aTuple)
print ("List elements : ", aList)
```

Output

```
List elements : [123, 'xyz', 'zara', 'abc']
```

cmp(list1,list2)(in python2):

Explanation: If elements are of the same type, perform the comparison and return the result. If elements are different types, check whether they are numbers.

If numbers, perform comparison.

If either element is a number, then the other element is returned.

Otherwise, types are sorted alphabetically .

If we reached the end of one of the lists, the longer list is "larger." If both list are same it returns 0.

Eg:

```
list1=[101,981,'abcd','xyz','m']
list2=['aman','shekhar',100.45,98.2]
list3=[101,981,'abcd','xyz','m']
print(cmp(list1,list2))
printcmp((list2,list1))
Printcmp((list3,list1))
```

There are following built-in methods of List:

Methods	**Description**
index(object)	Returnstheindexvalueoftheobject.
count(object)	returns the number of times an object is repeated in list
pop()/pop(index)	Returns the last object or the specified indexed object. It removes the popped object.
insert(index,object)	Insert an object at the given index.
append(object)	it add an object to the end of the list
extend(sequence)	It adds the sequence to existing list.
remove(object)	It removes the object from the given List.
reverse()	Reverse the position of all the elements of a list.
sort()	It is used to sort the elements of the List.
clear()	Remove all items from list
Copy()	returns a shallow copy of

Built-in List Methods

index(): The method index() returns the lowest index in list that obj appears.

Syntax **list.index(obj)**

Ex

```
aList= [123, 'xyz', 'zara', 'abc']
print ("Index for xyz : ", aList.index( 'xyz' ) )
print ("Index for zara: ", aList.index( 'zara' ))
```

Output: 1 2

count(): The method count() returns count of how many times obj occurs in list.

Syntax: **list.count(obj)**

Ex

```
aList= [123, 'xyz', 'zara', 'abc', 123]
print( "Count for 123 : ", aList.count(123))
print( "Count for zara: ", aList.count('zara'))
```

Output

```
Count for 123 : 2
Count for  zara: 1
```

Built-in List Methods

append(): The method append() appends a passed object into the existing list.

Syntax: **list.append(obj)**

Ex

```
aList, b= [123, 'xyz', 'zara', 'abc'],[20,40]
aList.append( 2009 );
print ("Updated List : ", aList)
alist.append(b)
 print ("Updated List : ", aList)
```

Output:

```
Updated List : [123, 'xyz', 'zara', 'abc', 2009]
Updated List : [123, 'xyz', 'zara', 'abc', 2009,[20,40]]
```

extend():The method extend() appends the contents of sequence to list.

Syntax: **list.extend(seq)**

Ex

```
aList= [123, 'xyz', 'zara', 'abc', 123]
bList= [2009, 'manni']
aList.extend(bList)
print( "Extended List : ", aList)
```

Output

```
Extended List : [123, 'xyz', 'zara', 'abc', 123, 2009, 'manni']
```

Built-in List Methods

pop(): The method pop() removes and returns last object or object from the list.

Syntax: **list.pop(index)**

Ex

```
aList= [123, 'xyz', 'zara', 'abc']
print ("A List : ", aList.pop())
print ("B List : ", aList.pop(2) )
```

Output

```
Alist: abc
Blist: zara
```

insert(): The method insert() inserts object obj ectinto list at offset index.

Syntax **list.insert(index, obj)**

Ex:

```
aList= [123, 'xyz', 'zara', 'abc']
 aList.insert( 3, 2009)]
 print ("Final List : ", aList)
```

Output

```
Sinal List : [123, 'xyz', 'zara', 2009, 'abc‘]
```

remove(): The remove() method searches for the given element in the list and removes the first matching element. This method does not return any value but removes the given object from the list.

Syntax **list.remove(obj)**

Ex
```
  a= [123, 'xyz', 'zara', 'abc', 'xyz']
a.remove('xyz')
print ("List : ", aList)
a.remove('abc')
print ("List : ", aList)
```

Output
```
List : [123, 'zara', 'abc', 'xyz']
List : [123, 'zara', 'xyz']
```

copy(): The copy() method returns a shallow copy of the list.

Syntax **new_lst=list.copy()**

Ex
```
a=['hi ','this','is','itF4']
b=a
c=a.copy()
print(a,b,c)
b.append('itF1')
c.append('itF2')
print(a,b,c)
```

reverse(): The method reverse() reverses objects of list in place. This method does not return any value but reverse the given object from the list.

Syntax **list.reverse()**

Ex

```
aList= [123, 'xyz', 'zara', 'abc', 'xyz']
aList.reverse()
print ("List : ", aList)
```

Output

```
List : ['xyz', 'abc', 'zara', 'xyz', 123]
```

sort() : The method sort() sorts objects of list

Syntax **list.sort()**

Ex

```
aList= [123, 'xyz', 'zara', 'abc', 'xyz']
aList.sort()
print ("List : ", aList)
```

Output

```
List : [123, 'abc', 'xyz', 'xyz', 'zara']
```

Clear(): The clear() method removes all items from the list.

Syntax **listclear()**

Ex

```
aList= [2, ‘x', ‘z', ‘a', ‘x']
aList.clear()
print ("List : ", aList)
```

Output

```
List : []
```

Aliasing

Since variables refer to objects, if we assign one variable to another, both variables refer to the same object

Example

```
a=[1,2,3]
b=a
if a is b:
    print( "True“)
else:
    print ("False“)
```

Output

```
True
```

cloning

If we want to modify a list and also keep a copy of the original, we need to be able to make a copy of the list itself, not just the reference. This process is sometimes called cloning, to avoid the ambiguity of the word copy. The easiest way to clone a list is to use the slice operator:

```
a=[1,2,3]
b=a[:]
print (a,b,sep=‘\t’)
 b[0]=5
print (a,b,sep=‘\t’)
```

Output

```
[1, 2, 3]        [1, 2, 3]
[1, 2, 3]        [5, 2, 3]
```

for loop Syntax Or VARIABLE in LIST: BODY

Note: Each time through the loop, the variable I is used as an index into the list, printing the i’th element. This pattern of computation is called a list traversal.

Ex

```
rl = [“ram", "laxman", "Bharat", "janaki"]
for f in rl:
        print (f)
```

Output

```
ram
laxman
Bharat
janaki
```

Enumerate() method adds a counter to an iterable and returns it in a form of enumerate object. This enumerate object can then be used directly in for loops or be converted into a list of tuples using list() method.

Ex

```
it=[‘F1’,’F2’,’F3’,’F4’)
list(enumerate(it))        # [(0, ‘F1'), (1, ‘F2'), (2, ‘F3'),(3,’f4’]
for  val in enumerate(it):
        print(val)
for cn,val in enumerate(it,6):
        print(cn,val)
```

Functions and lists

Passing a list as an argument actually passes a reference to the list, not a copy or clone of the list. So parameter passing creates an alias for you:the caller has one variable referencing the list, and the called function has an alias, but there is only one underlying list object.

EX
```
def   fun(y):
        y.append(24)
marks= [ 20, 13, 23, 15]
print(marks)                                Output
fun(marks)                                  [20, 13, 23, 15]
print (marks)                               [20, 13, 23, 15, 24]
```

Strings and lists

An optional argument called a delimiter can be used to specify which string to use as the boundary marker between substrings. The following example uses the string **in as the delimiter:**

Example
```
s="The rain in india“
w=list(s.partition("in"))
print (w)
```

Output
```
['the ra', 'in', ' in india']
```

list comprehension:

List comprehension is an elegant and concise way to create new list from an existing list in Python. A list comprehension generally consist of these parts : Output expression,
input sequence, a variable representing member of input sequence
an optional predicate part.

Syntax: **lst=[output expression   input sequence predicate part]**

Ex1:
```
lst  =  [x ** 2     for x in range (1, 11)      if  x % 2 == 1]
print(lst)                                   # [1, 9, 25, 49, 81]
```
Ex2:
```
p= [2**x    for x in range(1,6)   ]
print(p)                                     # [2, 4, 8, 16, 32]
```
Ex3:
```
str = "my phone number is :9646780456!!"
phoneno = [x   for x in string   if x.isdigit()]
print (phoneno)                                   #9646780456
```
Ex4:
```
s="SnisT"
c=[x      for x in s      if x.isupper()]
print(c)                                          #['S','T"]
```
Ex5:
```
x, y=[10,20,30],[15,25,35]
ls=[i+j    for i in x for j in y]
```

Tuples

- An ordered group of sequence seperated by symbol, and enclosed inside the parenthesis
- A tuple is a sequence of immutable objects, therefore tuple can not be changed
- In Python, tuple is similar to a list.Only the difference is that list is enclosed between square bracket,tuple between parenthesis and List have mutable objects where asTuple have immutable objects.

NOTE :If Parenthesis is not given with a sequence, it is by default treated asTuple.

Creation of Tuple: A tuple is created by placing all the items (elements) **inside a parentheses**(), separated by comma. A tuple can have any number of items and they may be of different types(integer, float, list,stringetc.).

Syntax:

```
tuple1=()           #creationofemptytuple
Tuple2=(Sequence1,)#,symbol is mandatory with out which it becomes
                           just a string assignment operator
Tuple3=(Sequence1,Sequence2)
```

Example:

```
my_tuple=(1,3.2,"mouse",[8,4,6],(1,2,3))
```

Nested tuple: A tuple can have sequence which can be tuple ,list ,string or dictionary

Ex
```
tup1='a','ITF4',10.56
tup2=tup1,(10,20,30)
print (tup1,tup2)
```
Output:
```
('a','mahesh',10.56)  (('a','mahesh',10.56),(10,20,30))
```

Accessing Elements in a Tuple

- We can use the index operator[] to access an item in a tuple where the index starts from 0. So, a tuple having 6 elements will have index from 0 to 5.Trying to access an element other that(6,7,...)will raise an Index Error.
- The index must be an integer, so we can not use float or other types. This will result into Type Error. Likewise ,nested tuple are accessed using nested indexing as shown in the example below.
- Python allows negative indexing for its sequences.

Ex
```
t = (1,"mouse", [8, 4, 6], (1, 2, 3))
print(t[1][3])          #s
print(t[-2][1])         #4
print(t[1])             #mouse
print(t[3][2])          #3
```

Tuple Operations : Slicing

➢ We can access arange of items in a tuple by using the slicing operator-colon":".

➢ A sub part of a tuple can be retrieved on the basis of index. This subpart is known as tuple slice.

➢ If the index provided in the Tuple slice is outside the list, then it raises an Index Error exception

Ex1

```
my_tuple = ('p','r','o','g','r','a','m')
printnt(my_tuple[1:4])
print(my_tuple[:-5])
print(my_tuple[5:])
print(my_tuple[:])
print(my_tuple[-1])
```

Ex2

```
t = (1,"mouse", [8, 4, 6], (1, 2, 3))
print(t[-1][0:]
print(t[-2][::-1])
print(t[1][2:4])
```

Packing and unpacking a tuple

- In Python there is a very powerful tuple assignment feature that assigns right hand side of values into left hand side.
- A tuple can also be created without using parentheses. This is known as tuple packing.
- In packing, we put values into a new tuple while in unpacking we extract those values into a single variable.

Ex

```
    t = (“hello ", “ITF4“,99)       # Packs  values  into  variable  a
   (h, b, a) = t                    # unpacks values of  variable a
    print(h)                        #  hello
    print(b)                        #  ITF4
    print(a)                        # 99
```

Tuple operations

Adding tuples : The + operator concatenates lists:

```
a = (1, 3, 5)       # tuple1
b = (2, 4, 6)       # tuple2
c = a + b           # tuple3
print c
```

Output : (1,3, 5, 2,4,6)

Replicating tuple: Similarly, the * operator repeats tuple a given number of times:

```
a= (7, 8, 9)
b=a*3
print b
```

Output (7, 8, 9, 7, 8, 9, 7, 8, 9)

tuple membership: in and not in are Boolean operators that test membership in a sequence.

```
t=(1,2,3,4,5,6)
print 5 in t                    #True
```

Iterating Through a Tuple: Using a for loop we can iterate through each item in a tuple.

```
t=(1,”snist”,98.4)
for i in t:
        print(i)
```

Changing a Tuple

- Unlike lists, tuples are immutable. This means that elements of a tuple can not be changed once it has been assigned. But, if the element is itself a mutable datatype like list,its nested items can be changed. We can also assign a tuple to different values(reassignment).

Example

```
t1= (4, 2, 3, [6, 5])
t1[3][0] = 9
print(t1)                    #(4, 2, 3, [9, 5])
t1[1]=40          #TypeError: 'tuple' object does not support item assignment
print(t1)                    #(4, 2, 3, [9, 5])
```

Deleting a Tuple

- we cannot delete or remove items or elements from a tuple.
- But deleting a tuple entirely is possible using the keyword del.

Example

```
t = ('p','r','o','g','r','a','m')
del t[1]                              #Error
del  t                       # will delete the tuple data
print(t)          # will show an error since tuple data is already deleted
```

Tuple functions

- all (T) - Return True if all elements of the tuple are true (or if the tuple is empty).
- any(T) - Return True if any element of the tuple is true. If the tuple is empty, return False.
- enumerate(T) -Return an enumerate object. It contains the index and value of all the items of tuple as pairs.
- len(T) -Return the length (the number of items) in the tuple.
- max(T) -Return the largest item in the tuple.
- min(T) -Return the smallest item in the tuple
- sorted(T) -Take elements in the tuple and return a new sorted list (does not sort the tuple itself).
- sum(T) -Retrun the sum of all elements in the tuple.
- tuple(T) -Convert an inerrable (list, string, set, dictionary) to a tuple.
- cmp(t1,t2) –To compare the two given tuples
- tuple(sequence) –Converts the sequence into tuple

cmp(tuple1,tuple2)

- ➢ If elements are of the same type, perform the comparison and return the result. If elements are different types, check whether they are numbers
- ➢ If numbers, perform comparison.
- ➢ If either element is a number, then the other element is returned.
- ➢ Otherwise, types are sorted alphabetically.
- ➢ If we reached the end of one of the lists, the longer list is "larger.“ If both list are same it returns 0.

Ex:

```
data1=(10,20,'rahul',40.6,'z')
data2=(20,30,'sachin',50.2)
print cmp(data1,data2)
print cmp(data2,data1)
data3=(20,30,'sachin',50.2)
printcmp(data2,data3)
```

Output:

```
-1
 1
 0
```

Example

```
pyTuple =(20,55,43,22,67,90,0) # data=(10,20,'Ravi',40.6,'z')
print  (all(pyTuple))
print (any(pyTuple))
print (len(pyTuple))
print (max(pyTuple))
print (min(pyTuple))
print (sum(pyTuple))
print (sorted(pyTuple))
a=enumerate(pyTuple)
print (tuple(a))
for item in enumerate(pyTuple):
        print((item))
```

Tuple Methods

- Methods that add items or remove items are not available with tuple. Only the following two methods are available.
- count(x) Return the number of items that is equal to x
- index(x) Return index of first item that is equal to x

Example

```
my_tuple = ('a','p','p','l','e',)
print(my_tuple.count('p'))#Output:2
print(my_tuple.index('l'))#Output:3
```

Advantages of Tuple over List

- Since tuple are immutable, iterating through tuple is faster than with list. So there is a slight performance boost.
- It makes the data safe as tuples are immutable and hence can not be changed.
- Tuples that contain immutable elements can be used as key for a dictionary. With list, this is not possible.
- Tuples are used for String formatting.

Dictionaries

- ➢ Python dictionary is an unordered collection of items
- ➢ Dictionaries are mutable i.e., it is possible to add, modify and delete key-value pairs
- ➢ Keys are used instead of indexes
- ➢ Keys are used to access elements in dictionary and keys can be of type- strings, number, list etc
- ➢ A list of elements with key and value pairs (seperated by symbol:) inside curly braces
- ➢ The key must be unique [Immutable], separated by colon(:) and enclosed with curly braces
- ➢ While other compound data types have only value as an element, a dictionary has a set of key & value pair known as item.
- ➢ Dictionary is known as Associative Array

creation of dictionary

- Creating a dictionary is as simple as placing items inside curly braces{} separated by comma.
- We can also create a dictionary using the built-in function dict()

Ex:

```
p={}                         #empty dictionary
p[1]="Rose"
p[2]="Lotus"
p["name"]="Jasmin"
p["color"]="Green"
print(p)           #{1: 'rose', 2: 'lotus', 'name': 'jasmin', 'color': 'green'}

d1= {1: 'apple', 2: 'ball'}           # dictionary with integer keys
d2= {'name': 'John', 1: [2, 4, 3]}          #dictionary with mixed keys
d3=dict([(1,'a'),(2,'b‘)])                  # using dict( )
d4=dict(n="ITF4",r=12,c='snist')            # using dict( )
```

Accessing Elements in a Dictionary

- ➢ While indexing is used with other container types to access values, dictionary uses keys.
- ➢ Key can be used either inside square brackets or with the get() method.
- ➢ The difference while using get() is that it returns None instead of Key Error, if the key is not found.

Example

```
my_dict= {'name':'student', 'age': 26}
print (my_dict['name'])
print (my_dict.get('age'))
print (my_dict.get('address'))
print (my_dict['address'])
```

Output

```
student
26
None
KeyError: 'address'
```

Changing or Adding Elements in a Dictionary

- Dictionaries are mutable. We can add new items or change the value of existing items using assignment operator.
- If the key is already present, value gets updated, else a new key: value pair is added to the dictionary.

EX:
```
d={'age': 26, 'name': ‘siri'}
d['age'] = 27          # update value
print(d)
d['address'] = 'Downtown'          # add item
print(d)
                                        output
                              {'age': 27, 'name': ‘siri'}
                    {'age': 27, 'name': ‘siri', 'address': 'Downtown'}
```

Dictionary Membership Test

- We can test if a key is in a dictionary or not using the keyword in. Notice that membership test is for keys only, not for values.

Exe
```
squares={1:1,3:9,5:25,7:49,9:81}
        print(1 in squares)
        print(2 not in squares)
        print(49 in squares)     #membership tests for key only not value
```
Output True True False

Loop through Dictionary: Using a for loop we can iterate though each key in a dictionary.

Ex: d1={1:'snist', 'IT':'F4','a':96.5}

1. **Print all key names in the dictionary, one by one:**

```
        for i in d1:
                print(i, end=' ')                                        #1
IT   a
```

2. **Print all *values* in the dictionary, one by one:**

```
        for i in d1:
                print(d[i],end=' ')                         # snist  f4    96.5
```

3. **use the values() function to return values of a dictionary**

```
        for x in d1.values():
                print(x)                               # snist  f4    96.5
```

4. **Loop through both *keys* and *values*, by using the items() function**

```
        for i ,x in d1.items()
                print(i,x)
                                                        output
                                                        1   snist
                                                        IT    F4
```

Deleting or Removing Elements from a Dictionary

- We can remove a particular item in a dictionary by using the method **pop()**. This method removes as item with the provided key and returns the value.
- The method, **popitem()** can be used to remove and return an arbitrary item(key,value) form the dictionary.
- All the items can be removed at once using the **clear()** method.
- We can also use the **del** keyword to remove individual items or the entire dictionary itself.

Example

```
s = {1:1, 2:4, 3:9, 4:16, 5:25}        # create a dictionary
print(s.pop(4))                        # remove a particular item (16)
print(s)                               #{1: 1, 2: 4, 3: 9, 5: 25}
print(s.popitem())                     # remove an arbitrary item(5,25)
print(s)                               #{1: 1, 2: 4, 3: 9}
del(s[2])                              # delete a particular item(4)
print(s)                               #{1: 1, 3: 9}
s.clear()                              #removes all items
print(s)                               #{}
del s                                  #delete entire dictionary
print(s)                               #Error
```

Dictionary methods

clear():	Removes all the elements from the dictionary
copy():	Return a shallow copy of the dictionary.
fromkeys(seq , v):	Return a new dictionary with keys from seq and value equal to v (defaults to None).
get(key , v):	Return the value of key. If key doesnot exit, return v (defaults to None).
Items():	Returns a list containing the a tuple for each key value pair
keys():	Returns a list containing the dictionary's keys
pop(key , v):	Remove the item with key and return its value. if key is not found,return d. If d is not provided and key is not found, raises KeyError.
popitem():	Remove and return an arbitary item (key, value). raises KeyError if the dictionary is empty.
setdefault(key,v):	Returns the value of the specified key. If the key does not exist: insert the key, with the specified value(defaults to None)
update(dict):	Update the dictionary with the key/value pairs from other
values():	Returns a list of all the values in the dictionary

Dictionary Methods

copy() : The method copy() returns a shallow copy of the dictionary.

Syntax **dict.copy()**

Ex

```
dict1 = {'Name': ‘ITF4', ‘code‘:12}
dict2 = dict1.copy()
print ("New Dictinary: “ ,dict2)
```

Output New Dictinary: { 'Name': ‘ITF4‘,’code’=12}

fromkeys( ) : The method fromkeys( ) creates a new dictionary with keys from sequence and values set to value. If we don’t specify a value in Dict, it assumes as keyword None

Syntax: **dict.fromkeys(seq[, value]))**

Ex

```
seq= ('name', 'age', ‘g')
d1= dict.fromkeys(seq)
print ("New Dictionary : “,d1))
d1= dict.fromkeys(seq, 10)
print ("New Dictionary : “ ,d1)
```

Output

```
New Dictionary : {'name': None, 'age': None, 'g': None}
New Dictionary : {'name': 10, 'age': 10, 'g': 10}
```

get(key , v): It return the value of key. If key doesnot exit, return v . If we don’t specify a value in Dict, it assumes as keyword None.

Syntax **d.get(key,v)**

Ex

```
d={'name':'ITF4',1:245,'a':5}
print(d.get('name'))                    #  ITF4
print(d.get(4))                         #   None
print(d.get('x',10))                    #   10
```

setdefault(key,v): It returns the value of the specified key. If the key does not exist: insert the key, with the specified value. If we don’t specify a value in dict, it assumes as keyword None.

Syntax **d.setdefault(key,v)**

Ex:

```
 d={1:1,2:4,3:'A','branch':"ITF4"}
 d.setdefault(3))          #A
d.setdefault(‘a’)          #{1: 1, 2: 4, 3: 'A', 'branch': 'ITF4‘, 'a': None}
d.setdefault(‘z’,126)
                #{1: 1, 2: 4, 3: 'A', 'branch': 'ITF4', ‘a': None, ‘z':126}
```

pop(key , v): It remove the item with key and return its value. if key is not found , return d. If d is not provided and key is not found, raises KeyError.

Ex:

```
d1={1:1,2:4,3:9,4:16}
d1.pop(2)                #4
print(d1)                # {1: 1, 3: 9, 4: 16}
d1.pop(5,25)             #25
d1.pop(6)                #keyError
```

popitem(): **It r**emove and return an arbitary item (key, value). if the dictionary is empty, raises KeyError

Ex:

```
 d1={1: 1, 4: 16}
 d1.popitem()            #((4, 16)
 d1.popitem()            # (1, 1)
d1.popitem()             #KeyError
```

update(dict): It update the dictionary with the key/value pairs from other

Ex:

```
d1={1:1,3:9}
d2={2:4,4:16}
d1.update(d2)
print(d1)                #{1: 1, 3: 9, 2: 4, 4: 16}
print(d2)                #{2: 4, 4: 16}
```

keys(): **It r**eturns a list containing the dictionary's keys

Syntax **d.keys()**

Ex:
```
d={1:1,2:4,3:9,4:16,5:25}
d.keys()
```
Output:
```
dict_keys([1, 2, 3, 4, 5])
```

values(): It returns a list of all the values in the dictionary

Syntax **d.values()**

Ex:
```
d={1:1,2:4,3:9,4:16,5:25}
d.values()
```
Output:
```
dict_values([1, 4, 9, 16, 25])
```

Items(): **It r**eturns a list containing the a tuple for each key value pair

Syntax **d.items()**

Ex:
```
d={1:1,2:4,3:9,4:16,5:25}
d.items()
```
Output:
```
dict_items([(1, 1), (2, 4), (3, 9), (4, 16), (5, 25)])
```

has_key( ) : The method has_key() returns true if a given key is available in the dictionary, otherwise it returns a false.

Syntax: **dict.has_key(key)**

```
d={1:1,2:4,3:9,4:16,5:25}
print(d.has_key(3))          #true
```

Built-in Functions with Dictionary

- all() Return True if all keys of the dictionary are true(or if the dictionary is empty).
- any() Return True if any key of the dictionary is true. If the dictionary is empty, return False.
- len() Return the length (the number of items) in the dictionary.
- cmp(d1,d2) Compares items of two dictionaries.
- sorted() Return a new sorted list of keys in the dictionary.

Ex1:

```
squares={1: 1, 3: 9, 5: 25, 7: 49}
print len(squares)
print sorted(squares)
```

Output: 4

```
[1, 3, 5, 7]
```

Ex2:

```
d={}
print all(d)
print any(d)
```

Output:

```
True
False
```

Dictionary Comprehension

- Dictionary comprehension is an elegant and concise way to create new dictionary from an iterable in Python.
- Dictionary comprehension consists of an expression pair(key:value) followed by for statement inside curly braces{}.
- A dictionary comprehension can optionally contain more **for** or **if** statements.
- Here is an example to make a dictionary with each item being a pair of a number and its square.

Ex:

```
squares = {x: x*x for x in range(6)}
print (squares)                                   Output
                                 {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
#This code is equivalent to
squares = {}
for x in range(6):
        squares[x] = x*x
print( squares)                                   Output
                                 {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

- An optional if statement can filter out items to form the new dictionary.

Ex

```
odd_squares={x:x*x for x in range(11) if x%2==1}
     print(odd_squares)          {1:1,3:9,5:25,7:49,9:81}
```