Python Functions

- A *function* is a group of statements that execute upon request.
- Functions are the most important aspect of an application. A function can be defined as the organized block of reusable code which can be called whenever required.
- A function is similar to a program that consists of a group of statements that are intended to perform a specific task.
- In other words, we can say that the collection of functions creates a program. The function is also known as procedure or subroutine in other programming languages.

- Python provide us various inbuilt functions like range() or print() or sqrt() or power(). Similar to these, Python allows programmers to create his/her own functions called as user-defined functions.
- Functions defined within class statements are also known as *methods*.
- In Python, functions are objects (values) that are handled like other objects. Thus, you can pass a function as an argument in a call to another function. Similarly, a function can return another function as the result of a call. A function, just like any other object, can be bound to a variable, an item in a container, or an attribute of an object. Functions can also be keys into a dictionary.
- For example, if you need to quickly find a function's inverse given the function, you could define a dictionary whose keys and values are functions and then make the dictionary bidirectional. Here's a small example of this idea, using some functions from module math.

```
inverse = {'sin':'asin', 'cos':'acos', 'tan':'atan', 'log':'exp'}
for f in inverse.keys( ):
    inverse[inverse[f]] = f
```

- The fact that functions are ordinary objects in Python is often expressed by saying that functions are *first-class* objects.

Monolithic code:

As the number of statements within our block of code increases, the code can become unwieldy. A single block of code (like in all our programs to this point) that does all the work itself is called **monolithic code.**

Monolithic code that is long and complex is undesirable for several reasons:

- **It is difficult to write correctly**. All the details in the entire piece of code must be considered when writing any statement within that code.
- **It is difficult to debug**. If the sequence of code does not work correctly, it is often difficult to find the source of the error. The effects of an erroneous statement that appears earlier in the code may not become apparent until a correct statement later uses the erroneous statement's incorrect result.

- **It is difficult to extend**. All the details in the entire sequence of code must be well understood before it can be modified. If the code is complex, this may be a formidable task.

Advantage of Functions in Python

There are the following advantages of Python functions.

- By using functions, we can avoid rewriting same logic/code again and again in a program. i.e code redundancy can be avoided.
- We can call python functions any number of times in a program and from any place in a program.
- We can track a large python program easily when it is divided into multiple functions. Code maintenance will become easy.
- Reusability is the main achievement of python functions.
- Functions provide modularity for programming.

Function Basics

There are two aspects to every Python function:

- **Function definition**. The definition of a function contains the code that determines the function's behavior.
- **Function invocation**. A function is used within a program via a function invocation. we invoked standard functions that we did not have to define ourselves. Every function has exactly one definition but may have many invocations.

An ordinary function definition consists of three parts:

- **Name**—Most Python functions have a name. The name is an identifier. As with variable names, the name chosen for a function should accurately portray its intended purpose or describe its functionality. (Python allows specialized anonymous function called lambda functions)
- **Parameters**—every function definition specifies the parameters that it accepts from callers. The parameters appear in a parenthesized comma-separated list. The list of parameters is empty if the function requires no information from code that calls the function.
- **Body**—every function definition has a block of indented statements that constitute the function's body. The body contains the code to execute when clients invoke the function. The code within the body is responsible for producing the result, if any, to return to the client.

Creating a function

In python, we can use **def** keyword to define the function. The syntax to define a function in python is given below.

```
def function_name([parameters]):
      "optional documentation string"
      function_suite/function_body
```

The function block is started with the colon (:) and all the same level block statements remain at the same indentation.

function-name is an identifier. It is a variable that gets bound (or rebound) to the function object when def executes.

parameters is an optional list of identifiers, known as *formal parameters* or just parameters, that get bound to the values supplied as arguments when the function is called.

In the simplest case, a function doesn't have any formal parameters, which means the function doesn't take any arguments when it is called. In this case, the function definition has empty parentheses after *function-name*.

When a function does take arguments, *parameters* contain one or more identifiers, separated by commas (,). In this case, each call to the function supplies values, known as *arguments*, corresponding to the parameters listed in the function definition. The parameters are local variables of the function.

A function can accept any number of parameters that must be the same in the definition and function calling and each call to the function binds these local variables to the corresponding values that the caller supplies as arguments.

In the comma-separated list of parameters, zero or more mandatory parameters may be followed by zero or more ***optional parameters***, where each optional parameter has the syntax:

```
Identifier = expression
```

The def statement evaluates each such *expression* and saves a reference to the expression's value, known as the *default value* for the parameter, among the attributes of the function object. When a function call does not supply an argument corresponding to an optional parameter, the call binds the parameter's identifier to its default value for that execution of the function. Note that each default value gets computed when the def statement evaluates, *not* when the resulting function gets called. In particular, this means that the *same* object, the default value, gets bound to the optional parameter whenever the caller does not supply a corresponding argument. This can be tricky when the default value is a mutable object and the function body alters the parameter. For example:

```
def f(x, y=[]):
y.append(x)
return y
print f(23) # prints: [23]
prinf f(42) # prints: [23, 42]
The second print statement prints [23, 42] because the first call to f altered the
default value of y, originally an empty list [], by appending 23 to it.
```

If you want y to be bound to a new empty list object each time f is called with a single argument, use the following style instead:

```
def f(x, y=None):
if y is None: y = []
y.append(x)
return y
print f(23) # prints: [23]
prinf f(42) # prints: [42]
```

At the end of the parameters, you may optionally use either or both of the special forms ***identifier1*** and ****identifier2***. If both forms are present, the form with two asterisks must be last.

- **identifier1* specifies that any call to the function may supply any number of extra positional arguments
- ***identifier2* specifies that any call to the function may supply any number of extra named arguments (positional and named arguments).

Every call to the function binds *identifier1* to a tuple whose items are the extra positional arguments (or the empty tuple, if there are none). Similarly, *identifier2* gets bound to a dictionary whose items are the names and values of the extra named arguments(or the empty dictionary, if there are none).

Here's a function that accepts any number of positional arguments and returns their sum:

```
def sum_args(*numbers):
    return sum(numbers)
print sum_args(23, 42) # prints: 65
```

The number of parameters of a function, together with the parameters' names, the number of mandatory parameters, and the information on whether (at the end of the parameters) either or both of the single- and double-asterisk special forms are present, collectively form a specification known as the **function's *signature***. A function's signature defines the ways in which you can call the function.

The nonempty sequence of statements, known as the ***function body***, does not execute when the def statement executes. Rather, the function body executes later, each time the function is called. Generally, the first statement of function body is an optional ***'docstring'*** that gives information about the function.

Docstrings are - unlike regular comments - stored as an attribute of the function they document, meaning that you can access them programmatically.

An example function

```
def func():
    """This is a function that does nothing at all"""
    return
```

The docstring can be accessed using the__doc__attribute:

```
print(func.__doc__)
This is a function that does nothing at all

help(func)
Help on function func in module__main_:
func()
This is a function that does nothing at all
```

function._doc is just the actual docstring as a string, while the help function provides general information about a function, including the docstring.

Followed by docstring, function body constitutes the logic of the function that reflects how to do a task apart from zero or more occurrences of the ***return*** statement.

'def' Executes at Runtime

The Python **def** is a true executable statement: when it runs, it creates a new function object and assigns it to a name along with a list of zero or more *arguments* (sometimes called *parameters*) in parentheses. The argument names in the header are assigned to the objects passed in parentheses at the point of call. (Remember, all we have in Python is *runtime*; there is no such thing as a separate compile time.)

One way to understand this code is to realize that the def is much like an = statement: it simply assigns a name at runtime. Unlike in compiled languages such as C, Python functions do not need to be fully defined before the program runs.

More generally, defs are not evaluated until they are reached and run, and the code *inside* defs is not evaluated until the functions are later called.

Because it's a statement, a **def** can appear anywhere a statement can—even nested in other statements. For instance, although **defs** normally are run when the module enclosing them is imported, it's also completely legal to nest a function **def** inside an if statement to select between alternative definitions:

```
if test:
        def func(): # Define func this way
        ...
else:
        def func(): # Or else this way
        ...
        ...
func() # Call the version selected and built
```

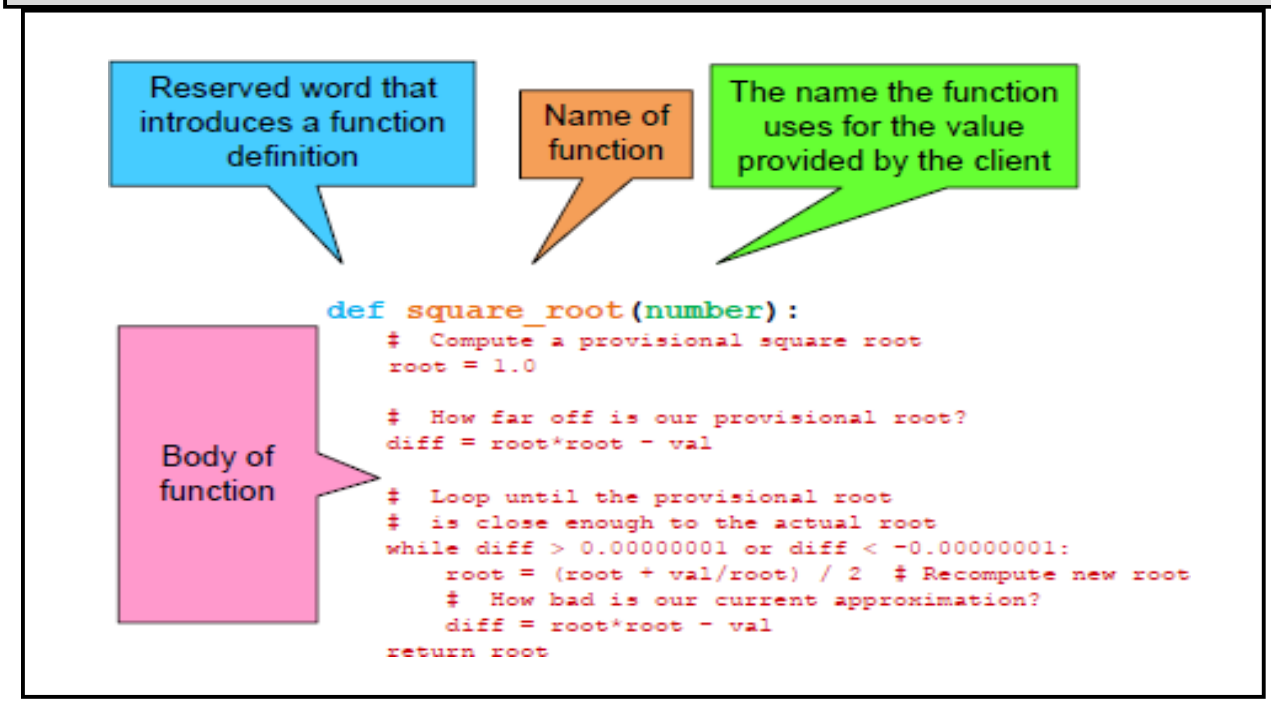
Function calling

A request to execute a function is known as a *function call*.

In python, a function must be defined before the function calling otherwise the python interpreter gives an error. Once the function is defined, we can call it from another function or the python prompt.

To call the function, use the function name followed by the parentheses.

```
# Statement(s)
function_name([arguments])
# Statement(s)
```




```
# Example1: Print a message to prompt the user for input
# Function Definition
def prompt():
        print("Please enter an integer value: ", end="")

# Start of program
print("This program adds together two integers.")
prompt()        # Call the function
value1 = int(input())
prompt()        # Call the function again
value2 = int(input())
sum = value1 + value2;
print(value1, "+", value2, "=", sum)
```

The two lines

```
def prompt():
        print("Please enter an integer value: ", end="")
```

make up the prompt function definition. When called, the function simply prints the message "Please enter an integer value:" and leaves the cursor on the same line.

The program runs as follows:

1. The program's execution begins with the first line in the "naked" block; that is, the block that is not part of the function definition.
2. The first executable statement prints the message of the program's intent.
3. The next statement is a call of the prompt function. At this point the program's execution transfers to the body of the prompt function. The code within prompt is executed until the end of its body or until a return statement is encountered. Since prompt contains no return statement, all of prompt's body (the one print statement) will be executed.
4. When prompt is finished, control is passed back to the point in the code immediately after the call of prompt.
5. The next action after prompt call reads the value of value1 from the keyboard.
6. A second call to prompt transfers control back to the code within the prompt function. It again prints its message.
7. When the second call to prompt is finished, control passes back to the point of the second input statement that assigns value2 from the keyboard.
8. The remaining two statements in the code, the arithmetic and printing statements, are executed, and then the program's execution terminates.

```
#output of above program
This program adds together two integers.
Please enter an integer value: 10
Please enter an integer value: 20
10 + 20 = 30
```

```
  #A simple function that prints the message "Hello Word" is given below.
def hello_world():
    print("hello world")

hello_world()
Output:
hello world
```

The return Statement

- The return statement is used to return values from a function.
- The return statement takes zero or more values, separated by commas. Using commas actually returns a single tuple.
- To return multiple values, use a tuple or list. Don't forget that (assignment) unpacking can be used to capture multiple values. Returning multiple items separated by commas is equivalent to returning a tuple.

Example:

```
In [8]: def test(x, y):

...: return x * 3, y * 4

...:

In [9]: a, b = test(3, 4)

In [10]: print a

9

In [11]: print b
16
```

- The return statement in Python is allowed only inside a function body and can optionally be followed by an expression.
- When return executes, the function terminates, and the value of the expression is the function's result.
- A function returns None if it terminates by reaching the end of its body or by executing a return statement that has no expression (or, of course, by executing 'return None'). ***i.e*** The default value is None.
- If some return statements in a function have an expression, all return statements should have an expression.

The following code defines a function that that computes the greatest common divisor (also called greatest common factor) of two integers. It determines largest factor (divisor) common to its parameters:

```
#A Python function to print GCD of two integers
def gcd(num1, num2):
    # Determine the smaller of num1 and num2
    min = num1 if num1 < num2 else num2
    # 1 is definitely a common factor to all ints largest Factor = 1
    for i in range(1, min + 1):
        if (num1 % i == 0 and num2 % i == 0):
            largestFactor = i    # Found larger factor
    return largestFactor


# Start of program
Print(gcd(4,2)
```

Functions are First Class Objects

In Python functions are considered as first class objects. It means we can use functions as perfect objects. In fact when we create a function, the python interpreter internally creates an object.

The following things are noteworthy:

- It is possible to assign a function to a variable.
- It is possible to define one function inside another function.
- It is possible to pass a function as parameter to another function
- It is possible that a function can return another function.

```
# A Program to assign a function to a variable
def display(s):
    '''A Program to see how a function can be assigned to a variable'''
    return 'hello '+s

# Assign function to a variable x
x = display('Kalam')
print(x)
```

Output:

```
hello Kalam
```

```
# A Python Program to know how to define one function inside another function
def display(s):
    '''A Program illustrating how to define one function inside another function'''
    def message():
        return ' How are you?'
    result = message() + ' '+s
```

```
    return 'hello '+s + result

# call display() function
print(display('kalam.'))
```

Output:

```
hello kalam. How are you? kalam.
```

```
# A Python Program to know how to pass a function as parameter to another function
def display(fun):
    '''A Program illustrating how to pass a function as parameter to another function'''

    return ' How are you?'+fun
def message():
    return ' kalam'

# call display() function
print(display(message()))
```

Output:

```
How are you? kalam
```

Pass by Object Reference

General means of passing arguments to functions are:

- Pass by value
- Pass by reference

Neither of the above two concepts is applicable in python.

In python everything is an object. An object can be imagined as memory block where we can store some value.

Objects are created on heap memory which is available during runtime of a program.

The size of heap depends on RAM of the computer.

When we pass values like numbers, strings, tuples or list to a function, the references of these objects are passed to the function.

```
# A python program to illustrate 'pass by object reference (immutable object)'
# passing an integer to a function
def modify(x):
    '''reassign a value to the variable '''
```

```
    x = 15
    print('within the function:      ',x,'\t',id(x))

# call modify() and pass x
x = 10
print('                        x \t id(x)')
print('                        -- \t ------ ')
print('before to function call: ',x,'\t',id(x))
modify(x)
print('After the function call: ',x,'\t',id(x))
```

Output:

```
                              x      id(x)
                            -----   ----------------
before to function call:    10     2007751040
within the function:        15     2007751120
After the function call:    10     2007751040
```

```
# A python program to illustrate 'pass by object reference (immutable object)'

# passing an integer to a function

def modify(x):
   '''reassign a value to the variable '''
   x = 10
   print('within the function:     ',x,'\t',id(x))

# call modify() and pass x
x = 10
print('                    x \t id(x)')
print('                    -- \t------- ')
print('before to function call: ',x,'\t',id(x))
modify(x)
print('After the function call: ',x,'\t',id(x))
```

Output:

```
                                x         id(x)
                               --         -------
```

```
before to function call:  10          2007751040
within the function:      10          2007751040
After the function call:  10          2007751040

# A python program to illustrate 'pass by object reference (mutable object)'
# passing an list to a function
def modify(list):
   '''to add new element to list '''
   list.append(15)
   print('within the function:    ',list,'\t',id(list))


# call modify() and pass x
list = [1,2,3,4]
print('                    list \t\t id(list)')
print('                    ----\t\t\t ------- ')
print('before to function call: ',list,'\t\t',id(list))
modify(list)
print('After the function call: ',list,'\t',id(list))
```

Output:

```
                                list                    id(list)
                                ----                    -------
before to function call:  [1, 2, 3, 4]                  80555064
within the function:      [1, 2, 3, 4, 15]              80555064
After the function call:  [1, 2, 3, 4, 15]              80555064
```

```
# A pyhton program to illustsrate 'pass by object reference (immutable object)'
# passing an tuple to a function
def modify(tuple):
   '''to add new element to tuple '''
   tuple = (1,2,3,4)
   print('within the function:    ',tuple,'\t\t',id(tuple))

# call modify() and pass x
tuple = (5,6,7,8)
print('                    tuple \t\t id(tuple)')
print('                    ----\t\t\t ------- ')
print('before to function call: ',tuple,'\t\t',id(tuple))
modify(tuple)
print('After the function call: ',tuple,'\t\t',id(tuple))
```

Output:

```
                                tuple                   id(tuple)
                                ------                  -----------
before to function call:  (5, 6, 7, 8)                  100651632
within the function:      (1, 2, 3, 4)                  77992272
After the function call:  (5, 6, 7, 8)                  100651632

# A pyhton program to create a new object inside a function does not modify outside object (mutable object)
# passing an list to a function
def modify(list):
   '''to create a new list '''
   list = [5,6,7,8]
   print('within the function:     ',list,'\t\t',id(list))

# call modify() and pass x
list = [1,2,3,4]
print('                    list \t\t id(list)')
print('                    ----\t\t\t ------- ')
print('before to function call: ',list,'\t\t',id(list))
modify(list)
print('After the function call: ',list,'\t\t',id(list))
```

Output:

```
                                list                    id(list)
                                ----                    -------
before to function call:  [1, 2, 3, 4]                  80554544
within the function:      [5, 6, 7, 8]                  80569448
After the function call:  [1, 2, 3, 4]                  80554544

# A pyhton program to illustsrate 'pass by object reference (immutable object)'
# passing an string to a function

#defining the function
def change_string (str):
   str = str + " How are you";
   print("printing the string inside function :          ",str, '\t',id(str));

string1 = "Hi I am there"
print("printing the string before calling function :   ",string1,'\t\t',id(string1))

#calling the function
change_string(string1)
```

```
print("printing the string after calling function :     ",string1,'\t\t',id(string1))
```

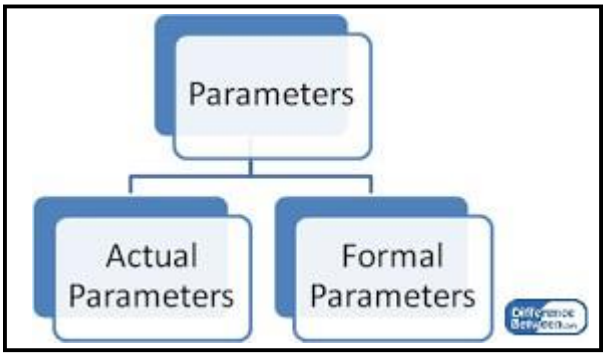
Output:

```
printing the string before calling function :    Hi I am there                80718608
printing the string inside function :          Hi I am there How are you      96954536
printing the string after calling function :     Hi I am there                80718608
```

Note:

1. If the object is immutable, the modified value is not available outside the function and if the object is mutable, the modified value is available outside the function.
2. If we create a new object inside a function, it will not be available outside the function.

Types of arguments



Parameters are inputs to the function. If a function contains parameters, then at the time of calling, compulsory we should provide values otherwise, otherwise we will get error.

Testing a function

- function must be defined before you use it
- To use it:

formal parameters

```
def addit(parm1, parm2):
    ans=parm1 + parm2
    return ans
```

actual parameters

```
mysum = addit (5,2)
print ("result of function call is = ", mysum)
```

There may be several types of arguments (actual parameters) which can be passed at the time of function calling.

1. Positional arguments
2. Keyword arguments
3. Default arguments
4. Variable-length arguments

Positional Arguments

we can provide the arguments at the time of function calling. As far as the positional arguments are concerned, these are the arguments which are required to be passed at the time of function calling with the ***exact match of their positions and number of arguments in the function call and function definition***.

If we change the order then result may be changed.

If we change the number of arguments then python interpreter will give error message.

Consider the following example.

Example 1

```
#the argument name is the required argument to the function func

def func(name):
    message = "Hi "+name
    return message
name = input("Enter the name? ")
print(func(name))
```

Output:

```
Enter the name? John
Hi John
```

Example 2

```
#the function simple_interest accepts three arguments and returns the simple interest accordingly

def simple_interest(p,t,r):
    return (p*t*r)/100
p = float(input("Enter the principle amount? "))
r = float(input("Enter the rate of interest? "))
t = float(input("Enter the time in years? "))
print("Simple Interest: ",simple_interest(p,r,t))
```

Output:

```
Enter the principle amount? 10000
Enter the rate of interest? 5
Enter the time in years? 2
Simple Interest:  1000.0
```

Example 3

```
#the function calculate returns the sum of two arguments a and b

def calculate(a,b):
    return a+b
calculate(10) # this causes an error as we are missing a required arguments b.
```

Output:

```
TypeError: calculate() missing 1 required positional argument: 'b'
```

Keyword arguments

Python allows us to call the function with the keyword arguments. This kind of function call will enable us to pass the arguments in the random order.

The name of the arguments is treated as the keywords and matched in the function calling and definition. If the same match is found irrespective of their position, the values of the arguments are copied in the function definition.

Consider the following example.

Example 1

```
#function func is called with the name and message as the keyword arguments

def func(name,message):

    print("printing the message with: ",name, "and ",message)

func(name = "John",message="hello") #name and message is copied with the values John and hello respectively

func(message="hello",name = "John") #name and message is copied with the values John and hello respectively
```

Output:

```
printing the message with John and hello
printing the message with John and  hello
```

Example 2 providing the values in different order at the calling

```
#The function simple_interest(p, t, r) is called with the keyword arguments the order of arguments doesn't matter in this case

def simple_interest(p,t,r):

    return (p*t*r)/100

print("Simple Interest: ",simple_interest(t=10,r=10,p=1900))
```

Output:

```
Simple Interest:  1900.0
```

Note:

1. If we provide the different name of arguments at the time of function call, error will be thrown.

Consider the following example.

Example 3

```
#The function simple_interest(p, t, r) is called with the keyword arguments.
def simple_interest(p,t,r):
    return (p*t*r)/100

print("Simple Interest: ",simple_interest(time=10,rate=10,principle=1900)) # doesn't find the exact match of the name
he arguments (keywords)
```

Output:

```
TypeError: simple_interest() got an unexpected keyword argument 'time'
```

Note:

The python allows us to provide the mix of the positional arguments and keyword arguments at the time of function call. However, the positional argument must not be given after the keyword argument, i.e., once the keyword argument is encountered in the function call, the following arguments must also be the keyword arguments.

Consider the following example.

Example 4

```
def func(name1,message,name2):
    print("printing the message with",name1,",",message,",and",name2)
func("John",message="hello",name2="David") #the first argument is not the keyword argument
```

Output:

```
printing the message with John , hello ,and David
```

The following example will cause an error due to an in-proper mix of keyword and positional arguments being passed in the function call.

Example 5

```
def func(name1,message,name2):
    print("printing the message with",name1,",",message,",and",name2)
func("John",message="hello","David")
```

Output:

```
SyntaxError: positional argument follows keyword argument
```

Default Arguments

Python allows us to initialize the arguments in the function definition. If the value of any of the argument is not provided at the time of function call, then that argument can be initialized with the value given in the definition even if the argument is not specified at the function call.

Example 1

```
def printme(name, age=22):
    print("My name is" ,name, "and age is", age)

#the variable age is not passed into the function however the default value of age is consid
 ered in the function
printme(name = "john")
```

Output:

My name is john and age is 22

Example 2

```
def printme(name,age=22):
    print("My name is",name,"and age is",age)

#the variable age is not passed into the function however the default value of age is consid
  ered in the function
printme(name = "john")
#the value of age is overwritten here, 10 will be printed as age
printme(age = 10,name="David")
```

Output:

My name is john and age is 22

My name is David and age is 10

Note:
After default arguments we should not take non default arguments.

Variable length Arguments

In the large projects, sometimes programmer may not know the number of arguments to be passed in advance. In such cases, Python provides us the flexibility to provide the comma separated values which are internally treated as tuples at the function call.

We can declare it with '*' before the argument as

```
def function_name(pos_arg,*args):
        -----
        -----
```

Consider the following example.

Example

```
def printme(*names):  #'*names' parameter takes 0 or more values
    print("type of passed argument is ",type(names))
    print("printing the passed arguments...")
    for name in names:
        print(name)
printme("john","David","smith","nick")
```

Output:

```
type of passed argument is <class 'tuple'>
printing the passed arguments...
```

```
john
David
smith
nick
```

A keyword variable-length argument is an argument that can accept any number of values in the format of keys and values.
We can declare it with '**' before the argument as

```
def function_name(pos_arg,**kwargs):
        -----
        -----
```

Here '**kwargs' is called as keyword variable argument. This argument internally represents a dictionary object.

```
# A python Program to illustrate Keyword variable length parameters'
def display(n,**kwargs):
    '''display given values'''
    print('type(n) = ',type(n),' type(kwargs) = ', type(kwargs))
    print('pos_arg = ',n)
    for x,y in kwargs.items():
        print('key = {},values = {}'.format(x,y))
# call display() with 1pos_arg and 2kwargs
display(5,rno = 525)
print()
# call display() with 1pos_arg and 4kwargs
display(5,rno = 525,name = 'Laxman')
```

Output:

```
type(n) = <class 'int'> type(kwargs) = <class 'dict'>
pos_arg =  5
key = rno,values = 525

type(n) = <class 'int'> type(kwargs) = <class 'dict'>
pos_arg =  5
key = rno,values = 525
key = name,values = Laxman
```

Note:

- We can have combination of variable length arguments and positional arguments but positional argument should appear before variable length argument.
- After variable length argument, if we are taking any other arguments then we should provide values as keyword variable length arguments or keyword arguments.

Case Study:

```
def f(arg1,arg2,arg3=4,arg4=8):
        print(arg1,arg2,arg3,arg4)
```

1. f(3,2) ==> 3 2 4 8
2. f(10,20,30,40) ===>10 20 30 40
3. f(25,50,arg4=100) ==>25 50 4 100
4. f(arg4=2,arg1=3,arg2=4)===>3 4 4 2
5. f()===>Invalid

   TypeError: f() missing 2 required positional arguments: 'arg1' and 'arg2'
6. f(arg3=10,arg4=20,30,40) ==>Invalid

   SyntaxError: positional argument follows keyword argument

   [After keyword arguments we should not take positional arguments]
7. f(4,5,arg2=6)==>Invalid

   TypeError: f() got multiple values for argument 'arg2'
8. f(4,5,arg3=5,arg5=6)==>Invalid

   TypeError: f() got an unexpected keyword argument 'arg5'

Note:

Function vs Module vs Library:

1. A group of lines with some name is called a function

2. A group of functions saved to a file , is called Module

3. A group of Modules is nothing but Library

Scope of variables

The scopes of the variables depend upon the location where the variable is being declared. The variable declared in one part of the program may not be accessible to the other parts.

In python, the variables are defined with the two types of scopes.

1. Global variables
2. Local variables

The variable defined outside any function is known to have a global scope i.e can be accessed through out the program and it becomes a global variable whereas the variable

defined inside a function is known to have a local scope, i.e can be only in the function where it is defined and it becomes a local variable.

Consider the following example.

Example 1

```
def print_message():
    message = "hello !! I am going to print a message." # the variable message is local to t
he function itself
    print(message)
print_message()
print(message) # this will cause an error since a local variable cannot be accessible here.
```

Output:

```
hello !! I am going to print a message.
  File "/root/PycharmProjects/PythonTest/Test1.py", line 5, in
    print(message)
NameError: name 'message' is not defined
```

Example 2

```
def calculate(*args):
    sum=0
    for arg in args:
        sum = sum +arg
    print("The sum is",sum)
sum=0
calculate(10,20,30) #60 will be printed as the sum
print("Value of sum outside the function:",sum) # 0 will be printed
```

Output:

```
The sum is 60
Value of sum outside the function: 0
```

The Global Keyword:

Keyword 'global' is used to refer to global variable inside a function.
When local and global variables have same name, the function, by default refers to local variable and ignores global variable.

```
# A program to illustrate scope of variable
```

```
# global variable example
b = 1    # b is global variable
def myfunction():
    a = 1       # a is local variable
    a += 1
    print('a local variable = ',a)
    b = 2
    print('b local variable within function = ',b)

print('value of global variable "b" before function call = ',b)
myfunction()
print('value of global variable "b" after function call = ',b)
print('a global variable = ',a)      # NameError: name 'a' is not defined
```

Output:

```
value of global variable "b" before function call =  1
a local variable =  2
b local variable within function =  2
value of global variable "b" after function call =  1
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-98-d3f71be65f00> in <module>
     12 myfunction()
     13 print('value of global variable "b" after function call = ',b)
---> 14 print('a global variable = ',a)       # NameError: name 'a' is not
defined

NameError: name 'a' is not defined
```

```
# A program to illustrate Global Keyword

b = 1    # b is global variable
def myfunction():
    global b
    a = 1       # a is local variable
    a += 1
    print('a local variable = ',a)
    b = 2
    print('b global variable inside function = ',b)

print('value of global variable "b" before function call = ',b)
myfunction()
print('value of global variable "b" after function call = ',b)
#print('a global variable = ',a)      # NameError: name 'a' is not defined
```

Output:

```
value of global variable "b" before function call =  1
a local variable =  2
b global variable inside function =  2
value of global variable "b" after function call =  2
```

When we want to access global variable inside a function where we have another local variable defined inside it, such that the local variable essence is retained, then ***globals() function*** can be used.
This is a built-in function which returns a table of current global variables in the form of dictionary.

We can refer to global variable 'a' as:
globals()['a']

```
# A program to get a copy of global variable into a function and work with it

a = 1   # 'a' is global variable

def myfunction():
    a = 2     # 'a is local variable'
    x = globals()['a'] # get global variable value into x
    #print('global variable "a" within function: ', x)
    x = x + 4
    #print('global variable "a" within function: ', x)
    print('local variable "a" within function: ', a)

print('global variable "a" before function call: ', a)
myfunction()
print('global variable "a" after function call: ', a)
```

Output:

```
global variable "a" before function call: 1
local variable "a" within function:  2
global variable "a" after function call:  1
```

Passing a group of elements to a function

To pass a group of elements like numbers or strings, we can accept them into a list and then pass the list to the function where the required processing can be done.

```
# A program to accept group of integer values and find their sum and average

def sum_avg(list):
    '''find sum and average'''
    n = len(list)
    sum = 0
    for i in list:
        sum = sum + i
    avg = sum/n
    return sum,avg

list = [int(x) for x in input('enter group of elements seperated by spaces: ').split()]
x,y = sum_avg(list)
print('sum = ',x)
print('Avg = ',y)
```

Output:

```
enter group of elements seperated by spaces: 10 20 30 40 50
sum =  150
Avg =  30.0
```

Recursive Functions

A function that calls itself is known as recursive function.

```
# A program to find factorial using recursion
def fact_recur(n):
   if n==0:
      result = 1
   else:
      result = n*fact_recur(n-1)
   return result

for i in range(1,11):
   print('factorial of {} is {}'.format(i,fact_recur(i)))
```

Output:

```
factorial of 1 is 1
factorial of 2 is 2
factorial of 3 is 6
factorial of 4 is 24
```

```
factorial of 5 is 120
factorial of 6 is 720
factorial of 7 is 5040
factorial of 8 is 40320
factorial of 9 is 362880
factorial of 10 is 3628800
```

Python Lambda Functions

A function without a name is called 'anonymous function'. Python allows us to not declare the function in the standard manner, i.e., by using the def keyword. Rather, the anonymous functions are declared by using lambda keyword. However, Lambda functions can accept any number of arguments, but they can return only one value in the form of expression.

The anonymous function contains a small piece of code. It simulates inline functions of C and C++, but it is not exactly an inline function.

The syntax to define an Anonymous function is given below.

```
lambda arguments : expression
```

Let's take a normal function,

```
def square(x):
    return x*x
```

The same function can be written as anonymous function as:

```
Lambda  x: x * x
```

```
f = lambda x: x*x
```

Lambda functions return a function and hence they should be assigned to a function as:

call to anonymous function can be made as follows,

```
value = f(5)
```

Example 1

```
x = lambda a:a+10 # a is an argument and a+10 is an expression which got evaluated and returned.

print("sum = ",x(20))
```

Output:

```
sum =  30
```

Example 2

Multiple arguments to Lambda function

```
x = lambda a,b:a+b # a and b are the arguments and a+b is the expression which gets evaluated and returned
int("sum = ",x(20,10))
```

Output:

```
sum =  30
```

Why use lambda functions?

The main role of the lambda function is better described in the scenarios when we use them anonymously inside another function. In python, the lambda function can be used as an argument to the higher order functions as arguments. Lambda functions are generally used with functions like filter(), map() or reduce().

Example 1

```
#the function table(n) prints the table of n
def table(n):
    # a will contain the iteration variable i and a multiple of n is returned at each function c
      all
       return lambda a:a*n;
n = int(input("Enter the number?"))
#the entered number is passed into the function table. b will contain a lambda function wh
ich is called again and again with the iteration variable i
b = table(n)
for i in range(1,11):
    print(n,"X",i,"=",b(i)); #the lambda function b is called with the iteration variable i,
```

Output:

```
Enter the number?10
10 X 1 = 10
10 X 2 = 20
10 X 3 = 30
10 X 4 = 40
10 X 5 = 50
10 X 6 = 60
10 X 7 = 70
10 X 8 = 80
10 X 9 = 90
10 X 10 = 100
```

Example 2

Use of lambda function with filter

```
#program to filter out the list which contains odd numbers
List = {1,2,3,4,10,123,22}
# the list contains all the items of the list for which the lambda function evaluates to true
Oddlist = list(filter(lambda x:(x%3 == 0),List))
print(Oddlist)
```

Output:

```
[3, 123]
```

Example 3

Use of lambda function with map

```
#program to triple each number of the list using map
List = {1,2,3,4,10,123,22}
# this will return the triple of each item of the list and add it to new_list
new_list = list(map(lambda x:x*3,List))
print(new_list)
```

Output:

```
[3, 6, 9, 12, 30, 66, 369]
```

String Data Type

The most commonly used object in any project and in any programming language is String only.

What is String?

Any sequence of characters within either single quotes or double quotes is considered as a String.

Syntax:

```
s='prasanna'
s="prasanna"
```

Note: In most of other languges like C, C++,Java, a single character with in single quotes is treated as char data type value. But in Python we are not having char data type. Hence it is treated as String only.

Eg:

```
>>> ch='a'
>>> type(ch)
<class 'str'>
```

How to define multi-line String literals:

We can define multi-line String literals by using triple single or double quotes.

Eg:

```
>>> s='''hello
        prasanna
          how are you'''
```

We can also use triple quotes to use single quotes or double quotes as symbol inside String literal.

Eg:

```
s='This is ' single quote symbol' ==>invalid
s='This is \' single quote symbol' ==>valid
s="This is ' single quote symbol"====>valid
s='This is " double quotes symbol' ==>valid
s='The "Python Notes" by 'prasanna' is very helpful' ==>invalid s="The "Python Notes" by 'durga' is very helpful"==>invalid s='The \"Python Notes\" by \'durga\' is very helpful' ==>valid s='''The "Python Notes" by 'durga' is very helpful''' ==>valid
```

How to access characters of a String:

We can access characters of a string by using the following ways.

1. By using index
2. By using slice operator

1. By using index:

- Python supports both +ve and -ve index.
- +ve index means left to right(Forward direction)
- -ve index means right to left(Backward direction)

Eg:

```
>>> s='prasanna'
>>> s[0]----- 'p'
>>> s[4]------ 'a'
>>> s[-1]----- 'a'
>>> s[10] -----IndexError: string index out of range
```

Note: If we are trying to access characters of a string with out of range index then we will get error saying : IndexError

- **Write a program to accept some string from the keyboard and display its characters by**

index wise(both positive and nEgative index)

```
s=input("Enter Some String:")
i=0
for x in s:
    print("The character present at +ve index {} and at -ve index {} is {}".format(i,i-len(s),x))
    i=i+1
```

Output:

```
Enter Some String: naresh
        The character present at +ve index 0 and at -ve index -6 is n
        The character present at +ve index 1 and at -ve index -5 is a
        The character present at +ve index 2 and at -ve index -4 is r
        The character present at +ve index 3 and at -ve index -3 is e
        The character present at +ve index 4 and at -ve index -2 is s
        The character present at +ve index 5 and at -ve index -1 is h
```

2. Accessing characters by using slice operator:

Syntax: s[beginindex:endindex:step]

beginindex: From where we have to consider slice(substring)
endindex: We have to terminate the slice(substring) at **endindex-1**
step: incremented value

Note: If we are not specifying begin index then it will consider from beginning of the string.
If we are not specifying end index then it will consider up to end of the string
The default value for step is 1

Eg:

```
s="Learning Python is very very easy!!!"
s[1:7:1]------------------ 'earnin'
s[1:7] --------- 'earnin'
 s[1:7:2] ----'eri'
s[:7]--------- 'Learnin'
```

s[7:] --------'g Python is very very easy!!!'
s[::]--------- 'Learning Python is very very easy!!!'
s[:] ---------- 'Learning Python is very very easy!!!'
s[::-1] -------'!!!ysae yrev yrev si nohtyP gninraeL'

Behaviour of slice operator:

s[begin:end:step]

step value can be either +ve or –ve

- **if +ve** then it should be **forward direction**(left to right) and we have to consider begin to **end-1**
- **if -ve** then it should be **backward direction**(right to left) and we have to consider begin to **end+1**

Note:

- In the backward direction **if end value is -1** then result is always empty.
- In the forward direction **if end value is 0** then result is always empty.

Mathematical Operators for String:

We can apply the following mathematical operators for Strings.

1. + operator for concatenation
2. * operator for repetition

print("prasanna"+"p").....prasannap
print("prasanna"*2).....prasannaprasanna

Note:

1. To use + operator for Strings, compulsory both arguments should be **str type**
2. To use * operator for Strings, compulsory one argument should be str and other argument should be int

len() function:

We can use len() function to find the number of characters present in the string.

```
s='prasanna'
print(len(s)) -----8
```

Checking Membership:

We can check whether the character or string is the member of another string or not by using in and not in operators

```
s=’prasanna’
print('a' in s) #True
print('z' in s) #False
```

Comparison of Strings:

We can use comparison operators (<,<=,>,>=) and equality operators(==,!=) for strings.

Comparison will be performed **based on alphabetical order**

Eg:

```
s1=input("Enter first string:")
s2=input("Enter Second string:")
if s1==s2:
   print("Both strings are equal")
elif s1<s2:
   print("First String is less than Second String")
else:
   print("First String is greater than Second String")
```

o/p :

Enter first string: **prasanna**
Enter Second string: **prasanna**
Both strings are equal

Enter first string: **prasanna**
Enter Second string: **vamsi**
First String is less than Second String

Enter first string: **prasanna**
Enter Second string: **laxmi**
First String is greater than Second String

Removing spaces from the string:

We can use the following 3 methods

1. rstrip()===>To remove spaces at right hand side
2. lstrip()===>To remove spaces at left hand side
3. strip() ==>To remove spaces both sides

Finding Substrings:

We can use the following 4 methods

For forward direction:

- find()
- index()

For backward direction:

- rfind()
- rindex()

1. find():

Syntax:s.find(substring)

Returns index of first occurrence of the given substring. If it is not available then we will get -1

Eg:

```
s="Learning Python is very easy"
 print(s.find("Python")).....#9
 print(s.find("Java")) ……# -1
```

```
print(s.find("r")) ..... #3
print(s.rfind("r")) ..... #21
```

By default find() method can search total string. We can also specify the boundaries to search.

s.find(substring,bEgin,end)

It will always search from bEgin index to end-1 index

Eg:
```
s="durgaravipavanshiva"
print(s.find('a')).....#4
print(s.find('a',7,15)) .....#10
print(s.find('z',7,15))……#-1
```

2. index() method:

index() method is exactly same as find() method except that if the specified substring is not available then we will get ValueError.

```
s="Learning Python is very easy"
print(s.index("Python")) ..... #9
print(s.index("r")) ..... #3
print(s.rindex("r")) ..... #21
print(s.index("Java")).....# substring not found
```

Counting substring in the given String:

We can find the number of occurrences of substring present in the given string by using count() method.

1. s.count(substring) ==> It will search through out the string

2. s.count(substring, bEgin, end) ===> It will search from bEgin index to end-1 index

Eg:
```
s="abcabcabcabcadda"
print(s.count('a')) ..... 6
print(s.count('ab')).....4
print(s.count('a',3,7)).....2
```

Replacing a string with another string:

Syntax: s.replace(oldstring,newstring)

inside s, every occurrence of oldstring will be replaced with newstring.

Eg1:
```
s="Learning Python is very difficult"
s1=s.replace("difficult","easy")
print(s1)
```

Output: Learning Python is very easy

Eg2: All occurrences will be replaced
```
s="abababababab"
s1=s.replace("a","b")
print(s1)
```

Output: bbbbbbbbbbbb

- **String objects are immutable then how we can change the content by using replace() method.**

Once we creates string object, we cannot change the content.This non changeable behaviour is nothing but immutability. If we are trying to change the content by using any method, then with those changes a new object will be created and changes won't be happend in existing object.

Hence with replace() method also a new object got created but existing object won't be changed.

Eg:

```
s="abab"
s1=s.replace("a","b")
print(s,"is available at :",id(s))
print(s1,"is available at :",id(s1))
```

Output:

```
abab is available at : 4568672
bbbb is available at : 4568704
```

In the above example, original object is available and we can see new object which was created because of replace() method.

Splitting of Strings:

- We can split the given string according to specified seperator by using split() method.

```
l=s.split(seperator)
```

- The default seperator is space. The return type of split() method is List

Eg1:

```
s="welcome to tkr college"
l=s.split()
for i in l:
   print(i)
```

output:

```
welcome
to
tkr
college
```

Eg2:

```
s="11-02-2020"
l=s.split('-')
for i in l:
        print(i)
```

output:

```
11
02
2020
```

Joining of Strings:

We can join a group of strings(list or tuple or set) wrt the given seperator.

```
s=seperator.join(group of strings)
```

Eg:

```
s=("welcome"," to"," tkr"," college")
l="$".join(s)
print(l)
```

output: welcome$ to$ tkr$ college

Eg2:

```
s=["welcome"," to"," tkr"," college"]
l=":".join(s)
print(l)
```

output: welcome: to: tkr: college

Changing case of a String:

We can change case of a string by using the following 4 methods.

1. upper()===>To convert all characters to upper case

2. lower() ===>To convert all characters to lower case

3. swapcase()===>converts all lower case characters to upper case and all upper case characters to lower case

4. title() ===>To convert all character to title case. i.e first character in every word should be upper case and all remaining characters should be in lower case.

5. capitalize() ==>Only first character will be converted to upper case and all remaining characters can be converted to lower case

Eg:

```
s='learning Python is very Easy'
print(s.upper()) ..... LEARNING PYTHON IS VERY EASY
print(s.lower()) ..... learning python is very easy
print(s.swapcase())..... LEARNING pYTHON IS VERY eASY
print(s.title()) ..... Learning Python Is Very Easy
print(s.capitalize()) ..... Learning python is very easy
```

Checking starting and ending part of the string:

Python contains the following methods for this purpose

1. **s.startswith(substring)**: returns true if main string starts with given substring otherwise returns false

2. **s.endswith(substring):** returns true if main string ends with given substring otherwise returns false

Eg:

```
s='learning Python is very easy'
print(s.startswith('learning'))…… True
print(s.endswith('learning')) ..... False
print(s.endswith('easy')) ..... True
```

To check type of characters present in a string:

Python contains the following methods for this purpose.

1) isalnum(): Returns True if all characters are alphanumeric(a to z , A to Z ,0 to9)

2) isalpha(): Returns True if all characters are only alphabet symbols(a to z, A to Z)

3) isdigit(): Returns True if all characters are digits only(0 to 9)

4) islower(): Returns True if all characters are lower case alphabet symbols

5) isupper(): Returns True if all characters are upper case aplhabet symbols

6) istitle(): Returns True if string is in title case

7) isspace(): Returns True if string contains only spaces

Eg:

```
print('Prasanna786'.isalnum()) ..... #True
print('prasanna786'.isalpha())..... #False
print('prasanna'.isalpha())..... #True
print('prasanna'.isdigit())..... #False
print('786786'.isdigit())..... #True
print('abc'.islower())..... #True
print('Abc'.islower()) ..... #False
print('abc123'.islower())..... #True
print('ABC'.isupper())..... #True
print('Learning python is Easy'.istitle())…… #False
print('Learning Python Is Easy'.istitle())……#True
print(' '.isspace()) ..... #True
```

Formatting the Strings:

We can format the strings with variable values by using replacement operator {} and format() method.

Eg:

```
name='prasanna'
sub='python'
age=30
print("{} 's sub is {} and his age is {}".format(name,sub,age))
print("{0} 's sub is {1} and his age is {2}".format(name,sub,age))
print("{x} 's sub is {y} and his age is {z}".format(z=age,y=sub,x=name))
```

output:

```
prasanna 's sub is python and his age is
30prasanna 's sub is python and his age
is 30prasanna 's sub is python and his
age is 30
```

Programs Regarding Strings:

Q1. **Write a program to reverse the given String**

input: prasanna

output: annasarp

1st Way:

```
s=input("Enter Some String:")
print(s[::-1])
```

2nd Way:

```
s=input("Enter Some String:")
print(''.join(reversed(s)))
```

3rd Way:

```
s=input("Enter Some String:")
i=len(s)-1
target=''
while i>=0:
        target=target+s[i]
        i=i-1
print(target)
```

Q2. Program to reverse order of words.

```
s=input("Enter Some String:")
l=s.split()
l1=[]
i=len(l)-1
while i>=0:
   l1.append(l[i])
   i=i-1
   output=' '.join(l1)
print(output)
```

output: Enter Some String: haii how are you

you are how haii

Q3. Program to reverse internal content of each word.

input: haii how are you

output: iiah woh era uoy

```
s=input("Enter Some String:")
l=s.split()
l1=[]
i=0
while i<len(l):
        l1.append(l[i][::-1])
        i=i+1
 rev=' '.join(l1)
 print(rev)
```

Q4. Write a program to find the number of occurrences of each character present in the given String?

input: aaaabbababcbabcacc
output: a = 8 Times
b = 6 Times
c = 4 Times
program:

```
s=input("Enter the Some String:")
d={}
for x in s:
    if x in d.keys():
        d[x]=d[x]+1
    else:
        d[x]=1
for k,v in d.items():
    print("{} = {} Times".format(k,v))
```

List Data Structure

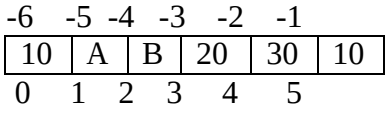
If we want to represent a group of individual objects as a single entity where insertion order preserved and duplicates are allowed, then we should go for List.

- insertion order preserved.
- duplicate objects are allowed
- heterogeneous objects are allowed.
- List is dynamic because based on our requirement we can increase the size and decrease the size.
- In List the elements will be placed within square brackets and with comma seperator.

We can differentiate duplicate elements by using index and we can preserve insertion order by using index. Hence index will play very important role.

Python supports both positive and negative indexes. +ve index means from left to right where as negative index means right to left

[10,"A","B",20, 30, 10]



List objects are mutable. i.e. we can change the content.

Creation of List Objects:

1. We can create empty list object as follows

```
list=[]
print(list) ..... []
print(type(list)) ..... <class 'list'>
```

2. If we know elements already then we can create list as follows

```
list=[10,20,30,40]
```

3. With dynamic input:

```
list=eval(input("Enter List:"))
print(list)
print(type(list))
```

output:

```
Enter List:[10,20,30,40]
[10, 20, 30, 40]
 <class 'list'>
```

4. With list() function:

```
l=list(range(0,10,2))
print(l) ..... [0,2,4,6,8]
print(type(l)) ..... <class 'list'>
```

Eg:

```
s="naresh"
l=list(s)
 print(l) ..... ['n', 'a', 'r', 'e', 's','h']
```

5. With split() function:

```
s="Learning Python is very very easy !!!"
l=s.split()
print(l)
print(type(l))
```

Output:

```
['Learning', 'Python', 'is', 'very', 'very', 'easy', '!!!']
<class 'list'>
```

Sometimes we can take list inside another list, such type of lists are called nested lists.

Eg:[10,20,[30,40]]

Accessing elements of List:

We can access elements of the list either by using index or by using slice operator(:)

1.By using index:

- List follows zero based index. ie index of first element is zero.
- List supports both +ve and -ve indexes.
- +ve index meant for Left to Right
- -ve index meant for Right to Left

list=[10,20,30,40,50,60]

-6	-5	-4	-3	-2	-1
10	20	30	40	50	60
0	1	2	3	4	5

print(list[0]) ==>10

print(list[-1]) ==>40

print(list[10]) ==>IndexError: list index out of range

2. By using slice operator:

Syntax:

list2= list1[start:stop:step]

start ==>it indicates the index where slice has to start default value is 0

stop ===>It indicates the index where slice has to end default value is max allowed index of list ie length of the list

step ==>increment value default value is 1

```
n=[1,2,3,4,5,6,7,8,9,10]
print(n[2:7:2])…… [3, 5, 7]
print(n[4::2])..... [5, 7, 9]
print(n[3:7])..... [4, 5, 6, 7]
print(n[8:2:-2]) ..... [9, 7, 5]
print(n[4:100])..... [5, 6, 7, 8, 9, 10]
```

Traversing the elements of List:

The sequential access of each element in the list is called traversal.

1. By using while loop:

```
n=[0,1,2,3,4,5,6,7,8,9,10]
i=0
while i<len(n):
    print(n[i],end=" ")
    i=i+1
```

output: 0 1 2 3 4 5 6 7 8 9 10

2. By using for loop:

```
n=[0,1,2,3,4,5,6,7,8,9,10]
for n1 in n:
    print(n1,end=" " )
```

output: 0 1 2 3 4 5 6 7 8 9 10

3. To display elements by index wise:

```
l=["A","B","C"]
x=len(l)
for i in range(x):
        print(l[i],"is available at positive index: ",i,"and at negative index: ",i-x)
```

Output

```
A is available at positive index: 0 and at negative index: -3
B is available at positive index: 1 and at negative index: -2
C is available at positive index: 2 and at negative index: -1
```

To get information about list:

1. len(): returns the number of elements present in the list

Eg:
```
n=[10,20,30,40]
print(len(n))==>4
```

2. count(): It returns the number of occurrences of specified item in the list

```
n=[1,2,2,2,2,3,3]
print(n.count(1))……1
print(n.count(2))……4
print(n.count(3)) ..... 2
print(n.count(4)) ..... 0
```

3. index(): returns the index of first occurrence of the specified item.

Eg:
```
n=[1,2,2,2,2,3,3]
print(n.index(1)) ==>0
print(n.index(2)) ==>1
print(n.index(3)) ==>5
print(n.index(4)) ==>ValueError: 4 is not in list
```

Note: If the specified element not present in the list then we will get ValueError. Hence before index() method we have to check whether item present in the list or not by using in operator.

```
print( 4 in n)==>False
```

Manipulating elements of List:

1. append(): We can use append() function to add item at the end of the list.

Eg:
```
list=[]
list.append("A")
list.append("B")
list.append("C")
print(list)
```

output

```
['A', 'B', 'C']
```

Eg: To add all elements to list upto 100 which are divisible by 10

```
list=[]
for i in range(101):
        if i%10==0:
                list.append(i)
print(list)
```

output: [0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100]

2. insert(): To insert item at specified index position

```
n=[1,2,3,4,5]
n.insert(1,888)
print(n)
```

output: [1, 888, 2, 3, 4, 5]

Eg:

```
n=[1,2,3,4,5]
n.insert(10,777)
n.insert(-10,999)
print(n)
```

Output: [999, 1, 2, 3, 4, 5, 777]

Note: If the specified index is greater than max index then element will be inserted at last position. If the specified index is smaller than min index then element will be inserted at first position.

Differences between append() and insert()

append(): In List when we add any element it will come in last i.e. it will be last element.
insert(): In List we can insert any element in particular index number

3. extend(): To add all items of one list to another list

```
l1.extend(l2)
```

all items present in l2 will be added to l1

Eg:

```
order1=["Chicken","Mutton","Fish"]
order2=["pepsi","sprite","appy"]
order1.extend(order2)
print(order1)
```

output: ['Chicken', 'Mutton', 'Fish', 'pepsi', 'sprite', 'appy']

Eg:

```
order=["Chicken","Mutton","Fish"]
order.extend("Mushroom")
print(order)
```

output: ['Chicken', 'Mutton', 'Fish', 'M', 'u', 's', 'h', 'r', 'o', 'o', 'm']

4. remove():

We can use this function to remove specified item from the list. If the item present multiple times then only first occurrence will be removed.

```
n=[10,20,10,30]
n.remove(10)
print(n)
```

output: [20, 10, 30]

If the specified item not present in list then we will get ValueError

```
n=[10,20,10,30]
n.remove(40)
print(n)
```

output: ValueError: list.remove(x): x not in list

Note: Hence before using remove() method first we have to check specified element present in the list or not by using in operator.

pop() function:

- It removes and returns the last element of the list.
- This is only function which manipulates list and returns some element.

Eg:

```
n=[10,20,30,40]
print(n.pop())
print(n.pop())
print(n)
```

output:40
30
[10, 20]

If the list is empty then pop() function raises IndexError

Eg:

```
n=[]
print(n.pop()) ==> IndexError: pop from empty list
```

Note:

1. pop() is the only function which manipulates the list and returns some value
2. In general we can use append() and pop() functions to implement stack datastructure by using list,which follows LIFO(Last In First Out) order.

In general we can use pop() function to remove last element of the list. But we can use to remove elements based on index.

n.pop(index)===>To remove and return element present at specified index.
n.pop()==>To remove and return last element of the list

```
n=[10,20,30,40,50,60]
print(n.pop()) ……#60
print(n.pop(1)) …..#20
print(n.pop(10)) ==>IndexError: pop index out of range
```

Differences between remove() and pop()

remove()

1) We can use to remove special element from the List.
2) It can't return any value.
3) If special element not available then we get VALUE ERROR**.**

pop()

1) We can use to remove last element from the List.
2) It returned removed element.
3) If List is empty then we get Error.

List objects are dynamic. i.e based on our requirement we can increase and decrease the size.

append(),insert() ,extend() ===>for increasing the size/growable nature
remove(), pop() ======>for decreasing the size /shrinking nature

reverse(): We can use to reverse() order of elements of list.

```
n=[10,20,30,40]
n.reverse()
print(n)
```

output: [40, 30, 20, 10]

sort() function:

In list by default insertion order is preserved. If want to sort the elements of list according to default natural sorting order then we should go for sort() method.

For numbers ==>default natural sorting order is Ascending Order
For Strings ==> default natural sorting order is Alphabetical Order

```
Eg1:  n=[20,5,15,10,0]
      n.sort()
      print(n) …….[0,5,10,15,20]

Eg2 : s=["Dog","Banana","Cat","Apple"]
      s.sort()
      print(s)..... ['Apple','Banana','Cat','Dog']
```

Note: To use sort() function, compulsory list should contain only homogeneous elements. otherwise we will get TypeError

Eg:

```
n=[20,10,"A","B"]
n.sort()
print(n)
```

TypeError: '<' not supported between instances of 'str' and 'int'

Note: In Python 2 if List contains both numbers and Strings then sort() function first sort numbers followed by strings

```
n=[20,"B",10,"A"]
n.sort()
print(n) ..... [10,20,'A','B']
```

To sort in reverse of default natural sorting order:

We can sort according to reverse of default natural sorting order by using reverse=True argument.

Eg:

```
n=[40,10,30,20]
n.sort()
print(n) ==>[10,20,30,40]
n.sort(reverse=True)
print(n) ===>[40,30,20,10]
n.sort(reverse=False)
print(n) ==>[10,20,30,40]
```

clear(): We can use clear() function to remove all elements of List.

Eg:

```
n=[10,20,30,40]
print(n)
n.clear()
print(n)
```

Output

```
[10, 20, 30, 40]
[]
```

Nested Lists:

Sometimes we can take one list inside another list. Such type of lists are called nested lists.

Eg:
```
n=[10,20,[30,40]]
print(n)
print(n[0])
print(n[2])
print(n[2][0])
print(n[2][1])
```

Output
```
[10, 20, [30, 40]]
10
[30, 40]
30
40
```

Note: We can access nested list elements by using index just like accessing multi dimensional array elements.

List Comprehensions:

It is very easy and compact way of creating list objects from any iterable objects(like list,tuple,dictionary,range etc) based on some condition.

Syntax:
```
list=[expression for item in list if condition]
```

Eg1:
```
s=[ x*x for x in range(1,11)]
print(s)
v=[2**x for x in range(1,6)]
print(v)
m=[x for x in s if x%2==0]
print(m)
```

Output:
```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
[2, 4, 8, 16, 32]
[4, 16, 36, 64, 100]
```

Eg2:
```
words=["Balaiah","Nag","Venkatesh","Chiranjeevi"]
l=[w[0] for w in words]
print(l)
```

output:
```
['B', 'N', 'V', 'C']
```

Tuple Data Structure

1. Tuple is exactly same as List except that it is immutable. i.e once we creates Tuple object, we cannot perform any changes in that object. Hence Tuple is Read Only version of List.
2. If our data is fixed and never changes then we should go for Tuple.
3. Insertion Order is preserved
4. Duplicates are allowed
5. Heterogeneous objects are allowed.
6. We can preserve insertion order and we can differentiate duplicate objects by using index. Hence index will play very important role in Tuple also.

   Tuple support both +ve and -ve index.
   +ve index means forward direction(from left to right) and
   -ve index means backward direction(from right to left)

7. We can represent Tuple elements within Parenthesis and with comma separator. Parenthesis are optional but recommended to use.

Eg:

```
t=10,20,30,40
print(t)..... (10, 20, 30, 40)
print(type(t)).....<class 'tuple'>
```

Note: We have to take special care about single valued tuple. compulsory the value should ends with comma, otherwise it is not treated as tuple.

```
t=(10)
t1=10
t2=10,
print(type(t))..... <class 'int'>
print(type(t1))..... <class 'int'>
print(type(t2))..... <class 'tuple'>
```

Tuple creation:

1. **creation of empty tuple**
   ```
   t=()
   ```
2. **creation of single valued tuple ,parenthesis are optional,should ends with comma**
   ```
   t=(10,)
   t=10,
   ```
3. **creation of multi values tuples & parenthesis are optional**
   ```
   t=10,20,30
   t=(10,20,30)
   ```
4. **By using tuple() function:**
   ```
   list=[10,20,30]
   t=tuple(list)
   print(t) .....(10,20,30)
   t1=tuple(range(10,20,2))
   print(t1).....(10,12,14,16,18)
   ```

Accessing elements of tuple:

We can access either by index or by slice operator

1. By using index:

```
t=(10,20,30,40,50,60)
print(t[0])……10
print(t[-1])……60
print(t[100])…… IndexError: tuple index out of range
```

2. By using slice operator:

```
t=(10,20,30,40,50,60)
print(t[2:5]) ..... (30, 40, 50)
print(t[2:100]) ..... (30, 40, 50, 60)
print(t[::2]) ..... (10, 30, 50)
```

Tuple vs immutability:

- Once we creates tuple, we cannot change its content.
- Hence tuple objects are immutable.

Eg:

```
t=(10,20,30,40)
t[1]=70 TypeError: 'tuple' object does not support item assignment
```

Mathematical operators for tuple:

We can apply + and * operators for tuple

1. **Concatenation Operator(+):**
   It concatenates the tuple mentioned on either side of the operator.

```
t1=(10,20,30)
t2=(40,50,60)
t3=t1+t2
print(t3).....(10,20,30,40,50,60)
```

2. **Multiplication operator or repetition operator(*)**
   The repetition operator enables the tuple elements to be repeated multiple times.

```
t1=(10,20,30)
t2=t1*3
print(t2).....(10,20,30,10,20,30,10,20,30)
```

Important functions of Tuple:

1. len(): To return number of elements present in the tuple

```
t=(10,20,30,40)
print(len(t)) ..... 4
```

2. count(): To return number of occurrences of given element in the tuple

```
t=(10,20,10,10,20)
print(t.count(10)).....3
```

3. index():returns index of first occurrence of the given element.
If the specified element is not available then we will get ValueError.

```
t=(10,20,10,10,20)
print(t.index(10)).....0
print(t.index(30)).....ValueError: tuple.index(x): x not in tuple
```

4.sorted(): To sort elements based on default natural sorting order

```
t=(40,10,30,20)
t1=sorted(t)
print(t1)…… [10, 20, 30, 40]
print(t) ..... (40, 10, 30, 20)
```

We can sort according to reverse of default natural sorting order as follows

```
t1=sorted(t, reverse=True)
print(t1)..... [40, 30, 20, 10]
```

5. min() and max() functions:

These functions return min and max values according to default natural sorting order.

```
t=(40,10,30,20)
print(min(t))..... 10
print(max(t)) ..... 40
```

6. cmp():

- It compares the elements of both tuples.
- If both tuples are **equal then returns 0**
- If the **first tuple is less than second tuple then it returns -1**
- If the **first tuple is greater than second tuple then it returns +1**

```
t1=(10,20,30)
t2=(40,50,60)
t3=(10,20,30)
print(cmp(t1,t2)) …… -1
print(cmp(t1,t3)) ..... 0
print(cmp(t2,t3)) ..... +1
```

Note: cmp() function is available only in Python2 but not in Python 3

Tuple Packing and Unpacking:

- We can create a tuple by packing a group of variables.

```
a=10
b=20
c=30
d=40
t=a,b,c,d
print(t) ..... (10, 20, 30, 40)
```

Here a,b,c,d are packed into a tuple t. This is nothing but tuple packing.

- Tuple unpacking is the reverse process of tuple packing

We can unpack a tuple and assign its values to different variables

```
t=(10,20,30,40)
a,b,c,d=t
print("a=",a,"b=",b,"c=",c,"d=",d) ..... a= 10 b= 20 c= 30 d= 40
```

Note: At the time of tuple unpacking the number of variables and number of values should be same. ,otherwise we will get ValueError.

Eg: t=(10,20,30,40)

a,b,c=t ..... ValueError: too many values to unpack (expected 3)

Tuple Comprehension:

Tuple Comprehension is not supported by Python.

```
t= ( x**2 for x in range(1,6))
```

Here we are not getting tuple object and we are getting generator object.

```
t= ( x**2 for x in range(1,6))
print(type(t))
for x in t:
print(x)
```

Output: <class 'generator'>

1
4
9
16
25

Write a program to take a tuple of numbers from the keyboard and print its sum and average?

```
t=eval(input("Enter Tuple of Numbers:"))
l=len(t)
sum=0
for x in t:
    sum=sum+x
print("The Sum=",sum)
print("The Average=",sum/l)
```

output: Enter Tuple of Numbers:(10,20,30,40)
The Sum= 100
The Average= 25.0

Differences between List and Tuple:

- List and Tuple are exactly same except small difference: List objects are mutable whereas Tuple objects are immutable.
- In both cases insertion order is preserved, duplicate objects are allowed, heterogenous objects are allowed, index and slicing are supported

List	Tuple
1) List is a Group of Comma separated Values within Square Brackets and Square Brackets are mandatory. Eg: i = [10, 20, 30, 40]	1) Tuple is a Group of Comma separated Values within Parenthesis and Parenthesis are optional. Eg: t = (10, 20, 30, 40) t = 10, 20, 30, 40
2) List Objects are Mutable i.e. once we creates List Object we can perform any changes in that Object. Eg: i[1] = 70	2) Tuple Objects are Immutable i.e. once we creates Tuple Object we cannot change its content. t[1] = 70 …….ValueError: tuple object does not support item assignment.
3.If the Content is not fixed and keep on changing then we should go for List.	3) If the content is fixed and never changes then we should go for Tuple.
4) List Objects can not used as Keys for Dictionaries because Keys should be Hashable and Immutable.	4) Tuple Objects can be used as Keys for Dictionaries because Keys should be Hashable and Immutable

Inserting elements in a tuple

In Python, tuples are immutable i.e. once created we cannot change its contents. But sometimes we want to modify the existing tuple; in that case we need to create a new tuple with updated elements only from the existing tuple.

Append an element in Tuple at end

- Suppose we have a tuple i.e.
  # Create a tuple
  tupleObj = (12 , 34, 45, 22, 33)
- Now to append an element in this tuple, we need to create a copy of existing tuple and then add new element to it using + operator i.e.
  # Append 19 at the end of tuple
  tupleObj = tupleObj + (19 ,)
- We will assign the new tuple back to original reference, hence it will give an effect that new element is added to existing tuple.

Contents of tuple will be now, (12, 34, 45, 22, 33, 19)
A new Element is appended at the end of tuple.

Ex. t=eval(input("enter a tuple values"))
print("t=",t)
print("id(t)=",id(t))
t1=eval(input("enter new tuple to append"))
print("t1=",t1)
print("id(t1)=",id(t1))
t=t+t1
print("t=",t)
print("id(t)=",id(t))

output:

enter a tuple values 10,2,3
t= (10, 2, 3)
id(t)= 1394983717208
enter new tuple to append (2,3,4,5)
t1= (2, 3, 4, 5)
id(t1)= 1394983872456
t= (10, 2, 3, 2, 3, 4, 5)
id(t)= 1394982526200

Insert an element at specific position in tuple

To insert an element at a specific index in the existing tuple we need to create a new tuple by slicing the existing tuple and copying contents from it.

- Suppose we have a tuple i.e.
  # Create a tuple tupleObj = (12 , 34, 45, 22, 33)
- As indexing starts from 0 in tuple, so to insert an element at **index n** in this tuple, we will create two sliced copies of existing tuple from **(0 to n)** and **(n to end)** i.e.
  # Sliced copy containing elements from **0 to n-1**
  tupleObj[: n]
  # Sliced copy containing elements from **n to end**
  tupleObj[n :]
- Now join these two sliced copies with new elements in between i.e.
  n = 2
  # Insert 19,20 in tuple at index 2
  tupleObj = tupleObj[: n] + (19 ,20) + tupleObj[n :]
- A new Element is inserted at index n. (12, 34, 19, 20,45, 22, 33)

Ex:
```
t=eval(input("enter a tuple values"))
print("t=",t)
print("id(t)=",id(t))
new=eval(input("enter a new tuple value to insert"))
print("new=",new)
n=int(input("enter position to insert a new elements:"))
t=t[0:n]+new+t[n:]
print("t=",t)
print("id(t)=",id(t))
```

Output:

```
enter a tuple values(12,34,45,22,33)
t= (12, 34, 45, 22, 33)
id(t)= 1394982928840
enter a new tuple value to insert(19,20)
new= (19, 20)
enter position to insert a new elements: 2
t= (12, 34, 19, 20, 45, 22, 33)
id(t)= 1394982526200
```

Delete an element at specific index in tuple

- Suppose we have a tuple i.e.
  # Create a tuple
  tupleObj = (12 , 34, 45, 22, 33)
- To delete the element at index n in tuple we will use the same slicing logic as above, but we will slice the tuple from from (0 to n-1) and (n+1 to end) i.e.
  # Sliced copy containing elements from 0 to n-1
  tupleObj[: n]
  # Sliced copy containing elements from n to end
  tupleObj[n + 1 :]
- Delete the element at index 2, the above sliced copies contains existing element at index n. Now join these two sliced copies i.e.
  tupleObj = tupleObj[: n] + tupleObj[n+1 :]
- new tuple after deleting element at index 2 is : (12, 34, 22, 33)

Ex:

```
t=eval(input("enter a tuple values"))
print("t=",t)
print("id(t)=",id(t))
n=int(input("enter position to delete an element:"))
t=t[:n]+t[n+1:]
print("t=",t)
print("id(t)=",id(t))
```

output:

```
enter a tuple values1,34,45,22,33
t= (12, 34, 45, 22, 33)
id(t)= 1394982928840
enter position to delete an element:2
t= (1, 34, 22, 33)
id(t)= 1394984444776
```

Set Data Structure

- If we want to represent a group of unique values as a single entity then we should go for set.
- Duplicates are not allowed.
- Insertion order is not preserved. But we can sort the elements.
- Indexing and slicing not allowed for the set.
- Heterogeneous elements are allowed.
- Set objects are mutable i.e once we creates set object we can perform any changes in that object based on our requirement.
- We can represent set elements within curly braces and with comma separation
- We can apply mathematical operations like union,intersection,difference etc on set
- objects.

Creation of Set objects:

Eg:

```
s= {10,20,30,40}
print(s)..... {40, 10, 20, 30}
print(type(s)) ..... <class 'set'>
```

1.We can create set objects by using set() function

```
s=set(any sequence)
```

Eg 1:

```
l = [10,20,30,40,10,20,10]
s=set(l)
print(s)……. {40, 10, 20, 30}
```

Eg 2:

```
 s=set(range(5))
print(s)……{0, 1, 2, 3, 4}
```

Note: While creating empty set we have to take special care. Compulsory we should use set() function.

s={} ==>It is treated as dictionary but not empty set

Eg:

```
s={}
print(s)..... {}
print(type(s)) ..... <class 'dict'>
```

Eg:

```
s=set()
print(s)..... set()
print(type(s)) ..... <class 'set'>
```

functions of set:

1. add(x): Adds item x to the set

Eg:

```
s={10,20,30}
s.add(40);
print(s).....40, 10, 20, 30}
```

2. update(x,y,z): To add multiple items to the set.

Arguments are not individual elements and these are Iterable objects like List,range etc.All elements present in the given Iterable objects will be added to the set.

Eg:

```
s={10,20,30}
l=[40,50,60,10]
s.update(l,range(5))
print(s)
```

Output

```
{0, 1, 2, 3, 4, 40, 10, 50, 20, 60, 30}
```

Q. What is the difference between add() and update() functions in set?

- We can use add() to add individual item to the Set, where as we can use update() functionto add multiple items to Set.
- add() function can take only one argument whereas update() function can take any number of arguments but all arguments should be iterable objects.

3. copy(): Returns copy of the set. It is cloned object.

```
s={10,20,30}
s1=s.copy()
print(s1).....{10, 20, 30}
```

4. pop(): It removes and returns some random element from the set.

```
s={40,10,30,20}
print(s).....{40, 10, 20, 30}
print(s.pop()) ..... 40
print(s).....{10, 20, 30}
```

5. remove(x): It removes specified element from the set.

If the specified element not present in the Set then we will get KeyError

```
s={40,10,30,20}
s.remove(30)
print(s)..... {40, 10, 20}
s.remove(50) ==>KeyError: 50
```

6. discard(x): It removes the specified element from the set.

If the specified element not present in the set then we won't get any error.

```
s={10,20,30}
s.discard(10)
print(s)..... {20, 30}
s.discard(50)
print(s)..... {20, 30}
```

7.clear(): To remove all elements from the Set.

```
s={10,20,30}
print(s).....{10, 20, 30}
s.clear()
print(s).....set()
```

Mathematical operations on the Set:

1.union(): We can use this function to return all elements present in both sets

x.union(y) or x|y

Eg:

```
x={10,20,30,40}
y={30,40,50,60}
print(x.union(y)).....{10, 20, 30, 40, 50, 60}
print(x|y) .....{10, 20, 30, 40, 50, 60}
```

2. intersection(): Returns common elements present in both x and y

x.intersection(y) or x&y

Eg:

```
x={10,20,30,40}
y={30,40,50,60}
print(x.intersection(y))……{40, 30}
print(x&y) .....{40, 30}
```

3. difference():returns the elements present in x but not in y

x.difference(y) or x-y

Eg:

```
x={10,20,30,40}
y={30,40,50,60}
print(x.difference(y))……{10, 20}
print(x-y).....{10, 20}
print(y-x).....{50, 60}
```

4.symmetric_difference(): Returns elements present in either x or y but not in both

x.symmetric_difference(y) or x^y

Eg:

```
x={10,20,30,40}
y={30,40,50,60}
print(x.symmetric_difference(y)) ..... {10, 50, 20, 60}
print(x^y) .....{10, 50, 20, 60}
```

Membership operators: (in , not in)

Eg:

```
s=set("naresh")
print(s).....{'a', 's', 'e', 'h', 'r', 'n'}
print('a' in s).....True
print('z' in s).....False
```

Set Comprehension: Set comprehension is possible.

creating set objects from any iterable objects(like list,tuple,dictionary,range etc) based on some condition

Syntax: set={expression for item in list if condition}

```
s={x*x for x in range(5)}
print(s).....{0, 1, 4, 9, 16}
s={2**x for x in range(2,10,2)}
print(s) .....{16, 256, 64, 4}
```

set objects won't support indexing and slicing:

Eg:

```
s={10,20,30,40}
print(s[0]) ..... TypeError: 'set' object does not support indexing
print(s[1:3]) ..... TypeError: 'set' object is not subscriptable
```

Dictionary

- We can use List, Tuple and Set to represent a group of individual objects as a single entity.
- If we want to represent a group of objects as key-value pairs then we should go for Dictionary.

Eg:

rollno--- name
phone number--address
ipaddress-- domain name

- Duplicate keys are not allowed but values can be duplicated.
- Heterogeneous objects are allowed for both key and values.
- insertion order is not preserved
- Dictionaries are mutable
- Dictionaries are dynamic
- indexing and slicing concepts are not applicable

Note: In C++ and Java Dictionaries are known as "Map" where as in Perl and Ruby it is known as "Hash"

How to create Dictionary?

We are creating empty dictionary

```
d={} or d=dict()
```

We can add entries as follows

```
d[100]="prasanna"
d[200]="vamsi"
d[300]="naresh"
print(d) ..... {100: 'prasanna', 200: 'vamsi', 300: 'naresh'}
```

If we know data in advance then we can create dictionary as follows

```
d={key:value, key:value}
d={100:'prasanna' ,200:'vamsi',
300:'naresh'}
```

How to access data from the dictionary?

- We can access data by using keys.

```
d={100:'prasanna' ,200:'vamsi',
300:'naresh'}
print(d[100]) ..... prasanna
print(d[300]) ..... naresh
```

- If the specified key is not available then we will get KeyError

```
print(d[400]) ..... KeyError: 400
```

- We can prevent this by checking whether key is already available or not by using has_key() function or by using in operator.

```
d.has_key(400) ==> returns 1 if key is available otherwise returns 0
```

But has_key() function is available only in Python 2 but not in Python 3. Hence compulsory we have to use in operator.

```
if 400 in d:
    print(d[400])
```

How to update dictionaries?

- We can update data **by using keys**.

  d[key]=value
- If the key is not available then a new entry will be added to the dictionary with the specified key-value pair
- If the key is already available then old value will be replaced with new value.

```
d={100:"prasanna",200:"ravi",300:"shiva"}
print(d).....{100: 'prasanna', 200: 'vamsi', 300: 'naresh'}
d[400]="vivaan"
print(d).....{100: 'prasanna', 200: 'vamsi', 300: 'naresh', 400: 'vivaan'}
d[100]="dileep"
print(d).....{100: 'dileep', 200: 'vamsi', 300: 'naresh', 400: 'vivaan'}
```

How to delete elements from dictionary?

1) **del d[key]: to remove particular key-value pair**

- It deletes entry associated with the specified key.
- If the key is not available then we will get KeyError

Eg:
```
d={100:"prasanna",200:"vamsi",300:"naresh"}
print(d)..... {100: 'prasanna', 200: 'vamsi', 300: 'naresh'}
del d[100]
print(d)..... {200: 'vamsi', 300: 'naresh'}
del d[400]
print(d).....KeyError: 400
```

2) **d.clear():**To remove all entries from the dictionary

Eg:
```
d={100:"prasanna",200:"vamsi",300:"naresh"}
print(d).....{100: 'prasanna', 200: 'vamsi', 300: 'naresh'}
d.clear()
print(d).....{}
```

3) **del d:**To delete total dictionary. Now we cannot access dictionnary d

Eg:
```
d={100:"prasanna",200:"vamsi",300:"naresh"}
print(d).....{100: 'prasanna', 200: 'ravi', 300:
'shiva'}
del d
print(d)..... name 'd' is not defined
```

functions of dictionary:

1. **dict():**To create a dictionary

   d=dict()..... It creates empty dictionary

   d=dict({100:"prasanna",200:"ravi"})……t creates dictionary with specified elements

   d=dict([(100,"prasanna"),(200,"shiva"),(300,"ravi")]).....It creates dictionary with the given list of tuple elements.
2. **len():** Returns the number of items in the dictionary

   d={100:"prasanna",200:"ravi",300:"shiva"}

   print(len(d)).....3
3. **clear():**To remove all elements from the dictionary

4. **get():** To get the value associated with the key

- **d.get(key):**If the key is available then **returns the corresponding value** otherwise returns None. It wont raise any error.
- **d.get(key,defaultvalue):**If the key is available then returns the corresponding value otherwise returns default value.

```
d={100:"prasanna",200:"ravi",300:"shiva"}
print(d).....{100:"prasanna",200:"ravi",300:"shiva"}
print(d.get(100))..... prasanna
print(d.get(400))……None
print(d.get(100,"Dileep"))
 ..... prasan
na
print(d.get(400,"Dileep"))……Dileep
```

5.pop():

- **d.pop(key):**It removes the entry associated with the specified key and returns the corresponding value

If the specified key is not available then we will get **KeyError**

```
d={100:"prasanna",200:"ravi",300:"shiva"}
print(d) ..... {100:"prasanna",200:"ravi",300:"shiva"}
print(d.pop(100))
print(d)
 ..... prasan
na
print(d.pop(400))
print(d) ..... KeyError: 400
```

6. **popitem():**It removes last item(key-value) from the dictionary and returns it**.**

```
d={100:"prasanna",200:"ravi",300:"shiva"}
print(d) ..... {100:"prasanna",200:"ravi",300:"shiva"}
print(d.popitem()) .....(300, 'shiva')
print(d) ..... {100: 'prasanna', 200: 'ravi'}
print(d.popitem()) .....(200, 'ravi')
print(d) ..... {100: 'prasanna'}
```

If the dictionary is empty then we will get KeyError

```
d={}
print(d.popitem()) ..... KeyError: 'popitem(): dictionary is empty'
```

7.**keys():** It returns all keys associated with dictionary.

Eg:

```
d={100:"prasanna",200:"ravi",300:"
shiva"}print(d.keys())
for i in d.keys():
    print(i)
```

output: dict_keys([100, 200, 300])

```
100
200
300
```

8.**values():** It returns all values associated with the dictionary

```
Eg:     d={100:"prasanna",200:"ravi",300:"shiva"}
        print(d.values())
        for i in d.values():
                print(i)
```

output: dict_values(['prasanna', 'ravi', 'shiva'])

```
        prasanna
        ravi
        shiva
```

9.**items():** It returns list of tuples representing key-value pairs.

```
                [(k,v),(k,v),(k,v)]
```

Eg:

```
                d={100:"prasanna",200:"ravi",300:"
        shiva"}for k,v in d.items():
                print(k,"=",v)
```

output: 100 = prasanna

```
         200 = ravi
         300 = shiva
```

10. **copy():** To create exactly duplicate dictionary(cloned copy)

```
        d1=d.copy()
```

eg:

```
                d={100:"prasanna",200:"ravi",300:"
        shiva"}d1=d.copy()
        print(d) ..... {100:"prasanna",200:"ravi",300:"shiva"}
        print(d1) ..... {100:"prasanna",200:"ravi",300:"shiva"}
```

11. setdefault():

d.setdefault(k,v): If the key is already available then this function returns the corresponding value.

If the key is not available then the specified key-value will be added as new item to the dictionary.

```
Eg:     d={100:"prasanna",200:"ravi",300:"shiva"}
        print(d.setdefault(400,"pavan")).....pavan
        print(d) ..... {100: 'prasanna', 200: 'ravi', 300: 'shiva', 400: 'pavan'}
        print(d.setdefault(100,"dileep")) ..... prasanna
        print(d) ..... {100: 'prasanna', 200: 'ravi', 300: 'shiva', 400: 'pavan'}
```

12. update():

d.update(x): All items present in the dictionary x will be added to dictionary d.

```
        d={100:"prasanna",200:"ravi",300:"shiva"}
        x={101:"prasanna",201:"ravi
        "}d.update(x)
        print(d) ..... {100: 'prasanna', 200: 'ravi', 300: 'shiva', 101: 'prasanna', 201: 'ravi'}
```