Set in python

A set is an unordered collection (Hence no index) of items. Every element is unique and must be immutable. However, the set itself is mutable. We can add or remove items from it. A set is created by placing all the items (elements) inside curly braces {}, separated by comma or by using the built-in function set().

```
# set of integers
my_set = {1, 2, 3}
print(my_set)

# set of mixed datatypes
my_set = {1.0, "Hello", (1, 2, 3)}
print(my_set)
```

The above code will produce following output

```
{1, 2, 3}
{1.0, 'Hello', (1, 2, 3)}
```

Empty curly braces {} will make an empty dictionary in Python. To make a set without any elements we use the set() function without any argument.

We can add single element using the add() method and multiple elements using the update() method. The update() method can take tuples, lists, strings or other sets as its argument. In all cases, duplicates are avoided.

An Example

```
# initialize my_set
my_set = {1,3}
print(my_set)

# add an element
my_set.add(2)
print(my_set)

# add multiple elements
# Output: {1, 2, 3, 4}
my_set.update([2,3,4])
print(my_set)
# add list and set
# Output: {1, 2, 3, 4, 5, 6, 8}
my_set.update([4,5], {1,6,8})
print(my_set)
```

The above code will produce following output

```
{1, 3}
{1, 2, 3}
{1, 2, 3, 4}
{1, 2, 3, 4, 5, 6, 8}
```

A particular item can be removed from set using methods, discard() and remove(). The only difference between the two is that, while using discard() if the item does not exist in the set, it remains unchanged. But remove() will raise an error in such condition.

An Example

```
my_set = {1, 3, 4, 5, 6}
print(my_set)

my_set.discard(4)
print(my_set)

my_set.remove(6)
```

```
print(my_set)

my_set.discard(2)
print(my_set)

#my_set.remove(2) this line will produce KeyError
```

The above code will produce following output

```
{1, 3, 4, 5, 6}
{1, 3, 5, 6}
{1, 3, 5}
{1, 3, 5}
```

Some more methods related to python sets are as follow

Function	Description
isdisjoint()	isdisjoint() method returns True if two sets are disjoint sets. If not, it returns False.
issubset()	issubset() method returns True if all elements of a set are present in another set (passed as an argument). If not, it returns False.

Python Frozenset

Frozenset is a new class that has the characteristics of a set, but its elements cannot be changed once assigned. While tuples are immutable lists, frozensets are immutable sets. Frozensets can be created using the function frozenset().

```
# initialize A and B
A = frozenset([1, 2, 3, 4])
B = frozenset([3, 4, 5, 6])
```

Defining functions in Python

A function is a named block of statement that is used to perform a particular task. A function enhances modularity of the program and it provides reusability of the code.

Python provides inbuilt function like print() which are known as library functions. We can also define functions that fulfils the need of our application such functions are user defined functions. To write a user defined function, we need to write following syntax

```
def functionname( parameters ):
   function_suite
   return [expression]
```

To call a function, we need to specify function name with arguments to be passed for parameters.

An Example

```
#function name: myfun
#paramater-list: empty
#return: none
def myfun():
    print("Inside myfun")

#calling function
myfun()
```

The above code will produce following output

```
Inside myfun
```

Passing parameter to the function

All parameters (arguments) in the Python language supports passed by reference mechanism hence changes made in parameter refers to both callee function and the calling function.

```
def change_list(mylist):
    i = 0
    total = len(mylist)
    while i < total:
        mylist[i] = mylist[i] + 10
        i+=1

ls = [11,22,33,44]
print("Actual list is",ls)
change_list(ls)
print("list after changes is",ls)
```

The above example will produce following output

```
Actual list is [11, 22, 33, 44]
list after changes is [21, 32, 43, 54]
```

A function can have four categories of parameters that are as follow

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

Let us take a look at all. **Required arguments** are the arguments passed to a function in the same order that was specified while defining function. Like in the previous example `mylist` is required argument. If required argument is missing then program will show error.

When **keyword arguments** is used in a function call then the caller identifies the arguments by the parameter name so arguments can be placed out of order because the Python interpreter uses the keywords provided to match the values with parameters.

An Example

```
def printDetails(name, age):
    print("Student name",name,"and age is",age)

#calling funtion
printDetails("Rahul",17)
printDetails(name = "Ajay", age = 18)
printDetails(age = 19, name = "Tanisha")
printDetails("Sanjay",age = 25)
# printDetails(name ="Sharda",26) can't place required argument after keyword argument
```

The above code will produce following output

```
Student name Rahul and age is 17
Student name Ajay and age is 18
Student name Tanisha and age is 19
Student name Sanjay and age is 25
```

A **default argument** is an argument that has a default value. if a value is not provided in the function call then the default value is assumed.

An Example

```
def area(rad, pi = 3.14):
    ar = pi*rad**2
    print("area is",ar)

area(rad = 7.0)
area(4.0,3.1428)
```

```
area(rad = 5.0, pi = 3.1428)
area(pi = 3.1428, rad = 6.0)
```

The above code will produce following output

```
area is 153.86
area is 50.2848
area is 78.57
area is 113.1408
```

Sometime we are in the need of functions that can accept any number of arguments, To do so we have to use **variable arguments**. These arguments are not named in the function definition.

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

An asterisk (*) is placed before the variable name that holds the values of all nonkeyword variable arguments.

```
def var_fun_demo( a, *b ):
    print("Required argument has value",a)
    print("Variable argument(s) has value(s)")
    for c in b:
        print(c,end = ' ')

var_fun_demo(1)
print('-----------------------------------')
var_fun_demo(2,3,4)
```

The code will produce following output

```
Required argument has value 1
Variable argument(s) has value(s)
-----------------------------------
Required argument has value 2
Variable argument(s) has value(s)
3 4
```

Python also supports **Anonymous Function**, These are the functions that doesn't have any name also these functions are not declared in the standard manner i.e. by using the def keyword. To create small anonymous functions, we have to use **lambda keyword**. We can assign the function to a variable to give it a name. To define anonymous function syntax is as follow

```
lambda arguments : expression
```

An anonymous function cannot contain commands or multiple expressions because it can have only one expression. Anonymous function is required when some nameless function is required for a short period of time.

```
sum = lambda a,b: a + b;
# Now you can call sum as a function
print("sum(10, 20) is ",sum(10, 20))
print("sum(25, 56) is ",sum(25, 56))
```

The above code will produce following output

```
sum(10, 20) is  30
sum(25, 56) is  81
```

Similar to other programming languages, **return** keyword is used to return value from a function.

Note: In python It is okay to create a function inside another function.

Types of variables of behalf of scope

A variable declared inside the function's body or in the local scope is known as **local variable**. A local variable is not accessible outside its scope.

An Example

```
def foo():
    y = "Python"

foo()
print(y)
```

The code will produce following error

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

A variable declared outside of the function or in global scope (means outside all blocks) is known as **global variable**. It is accessed by whole program means inside or outside of the function.

An Example

```
a = 10

def hello():
    b = 20;
    print("a = ",a)
    print("b = ",b)
    print("-----------------")
    a = 30

hello()
```

When we try to modify value of variable a in function hello() then it will produce error because global variables are read only in local scope. To make changes in the global variables we have to use `global` keyword inside function. It is used to create a global variable and make changes to the variable in a local context.

```
a = 10

def hello():
    global a
    global c
    b = 20
    c = 40
    print("a = ",a)
    print("b = ",b)
    print("c = ", c)
    print("-----------------")
    a = 30
    print("a = ",a)
    print("b = ",b)

hello()
print(c)
```

The above code will produce following output

```
a =  10
b =  20
c =  40
-----------------
a =  30
b =  20
```

Here in the above example, we have a global variable a and another global variable c is created inside function hello() using `global` keyword. Because a is with global keyword so it can be modified inside local scope, Also c is also global so accessible in hello() function as well as outside hello() function.

If a global variable and local variable share same name then the function that contains local variable will always access its local variable by outside that function global variable will be accessible.

An Example

```
x = "Global"

def hello():
    x = "Local"
    print("x is",x)

hello()
```

The above code will produce following output

```
x is Local
```

Apart from the two scopes, python also supports **nonlocal variables**. Before moving forward look at the following example

```
def outer():
    msg = "from outer!"
    def inner():
        msg = "from inner!"
        print(msg)
    inner()
    print(msg)

outer();
```

The above code will print following output

```
from inner!
from outer!
```

In the above example when inner() function is called then it created another variable msg which is different from variable msg of outer() function, that's why we are having different output both times, if we want to use the same variable msg of outer() function inside inner() function then we have to use nonlocal keyword in outer() function.

```
def outer():
    msg = "from outer!"
    def inner():
        nonlocal msg
        msg = "from inner!"
        print(msg)
    inner()
    print(msg)

outer();
```

The above code will print following output

```
from inner!
from inner!
```

Concept of namespace, locals() and globals() function

A namespace is a dictionary in which variable names are keys and their corresponding objects are values. A Python statement can access variables in a local namespace and in the global namespace. If a local and a global variable have the same name, the local variable shadows the global variable. Each function has its own local namespace. Class methods follow the same scoping rule as ordinary functions. If we want a function to search for a variable in global namespace then we have use global keyword before that variable.

Python provides a locals() function If it is called from within a function, it will return all the names that can be accessed locally from that function. similarly Python provides a globals() function If it is called from within a function, it will return all the names that can be accessed globally from that function. Both functions returns dictionary in which variable-names of the namespace work as keys.

An Example

```
a = "global"
def outer():
    b = "local"
    loc = locals()
    print(loc.keys())
    glb = globals()
    print(glb.keys())

outer()
```

The above code will produce following output

```
dict_keys(['b'])
dict_keys(['__name__',  '__doc__',  '__package__',  '__loader__',  '__spec__',
'__annotations__', '__builtins__', '__file__', '__cached__', 'a', 'outer'])
```