

A sequence is a datatype that represents a group of elements. The purpose of any sequence is to store and process group elements. In python, strings, lists, tuples and dictionaries are very important sequence datatypes.

LIST:

A list is similar to an array that consists of a group of elements or items. Just like an array, a list can store elements. But, there is one major difference between an array and a list. An array can store only one type of elements whereas a list can store different types of elements. Hence lists are more versatile and useful than an array.

Creating a List:

Creating a list is as simple as putting different comma-separated values between square brackets.

```
student = [556, "Mothi", 84, 96, 84, 75, 84 ]
```

We can create empty list without any elements by simply writing empty square brackets as:

```
student=[ ]
```

We can create a list by embedding the elements inside a pair of square braces []. The elements in the list should be separated by a comma (,).

Accessing Values in list:

To access values in lists, use the square brackets for slicing along with the index or indices to obtain value available at that index. To view the elements of a list as a whole, we can simply pass the list name to print function.

negative Indexing	-7	-6	-5	-4	-3	-2	-1
Positive Indexing	0	1	2	3	4	5	6
Student	556	"Mothi"	84	96	84	75	84

Ex:

```
student = [556, "Mothi", 84, 96, 84, 75, 84 ]
print(student)
print(student[0]) # Access 0th element
print(student[0:2]) # Access 0th to 1st elements
print(student[2: ]) # Access 2nd to end of list elements
print(student[:3]) # Access starting to 2nd elements
print(student[: ]) # Access starting to ending elements
print(student[-1]) # Access last index value
print(student[-1:-7:-1]) # Access elements in reverse order
```

Output:

```
[556, "Mothi", 84, 96, 84, 75, 84]
Mothi
```

```
[556, "Mothi"]
[84, 96, 84, 75, 84]
[556, "Mothi", 84]
[556, "Mothi", 84, 96, 84, 75, 84]
84
[84, 75, 84, 96, 84, "Mothi"]
```

Creating lists using range() function:

We can use range() function to generate a sequence of integers which can be stored in a list.

To store numbers from 0 to 10 in a list as follows.

```
numbers = list( range(0,11) )
print(numbers) # [0,1,2,3,4,5,6,7,8,9,10]
```

To store even numbers from 0 to 10 in a list as follows. `numbers = list(range(0,11,2))`

```
print(numbers) # [0,2,4,6,8,10]
```

Looping on lists:

We can also display list by using for loop (or) while loop. The len() function is useful to know the numbers of elements in the list. while loop retrieves starting from 0th to the last element i.e. n-1

Ex-1:

```
numbers = [1,2,3,4,5]
for i in numbers:
    print(i,end=" ")
```

Output:

```
1 2 3 4 5
```

Updating and deleting lists:

Lists are *mutable*. It means we can modify the contents of a list. We can append, update or delete the elements of a list depending upon our requirements.

Appending an element means adding an element at the end of the list. To, append a new element to the list, we should use the append() method.

Example:

```
lst=[1,2,4,5,8,6]
print(lst) # [1,2,4,5,8,6]
lst.append(9)
print(lst) # [1,2,4,5,8,6,9]
```

Updating an element means changing the value of the element in the list. This can be done by accessing the specific element using indexing or slicing and assigning a new value.

Example:

```
lst=[4,7,6,8,9,3]
print(lst) # [4,7,6,8,9,3]
lst[2]=5 # updates 2nd element in the list
print(lst) # [4,7,5,8,9,3]
lst[2:5]=10,11,12 # update 2nd element to 4th element in the list
print(lst) # [4,7,10,11,12,3]
```

Deleting an element from the list can be done using 'del' statement. The *del* statement takes the position number of the element to be deleted.

Example:

```
lst=[5,7,1,8,9,6]
del lst[3] # delete 3rd element from the list i.e., 8
print(lst) # [5,7,1,9,6]
If we want to delete entire list, we can give statement like del lst.
```

Concatenation of Two lists:

We can simply use „+“ operator on two lists to join them. For example, “x” and “y” are two lists. If we write x+y, the list “y” is joined at the end of the list “x”.

Example:

```
x=[10,20,32,15,16]
y=[45,18,78,14,86]
print(x+y) # [10,20,32,15,16,45,18,78,14,86]
```

Repetition of Lists:

We can repeat the elements of a list “n” number of times using “*” operator.

```
x=[10,54,87,96,45]
Print(x*2) # [10,54,87,96,45,10,54,87,96,45]
```

Membership in Lists:

We can check if an element is a member of a list by using “in” and “not in” operator.

If the element is a member of the list, then “in” operator returns **True** otherwise returns **False**.

If the element is not in the list, then “not in” operator returns **True** otherwise returns **False**.

Example:

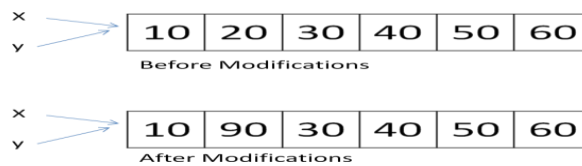
```
x=[10,20,30,45,55,65]
a=20
print(a in x) # True
a=25
print( a in x) # False
a=45
print( a not in x) # False
a=40
print( a not in x) # True
```

Aliasing and Cloning Lists:

Giving a new name to an existing list is called '*aliasing*'. The new name is called '*alias name*'. To provide a new name to this list, we can simply use assignment operator (=).

Example:

```
x = [10, 20, 30, 40, 50, 60]
y=x # x is aliased as y
print(x) # [10,20,30,40,50,60]
print(y) # [10,20,30,40,50,60]
x[1]=90 # modify 1st element in x
print(x) # [10,90,30,40,50,60]
print(y) # [10,90,30,40,50,60]
```



In this case we are having only one list of elements but with two different names “x” and “y”. Here, “x” is the original name and “y” is the alias name for the same list. Hence, any modifications done to “x” will also modify “y” and vice versa.

Obtaining exact copy of an existing object (or list) is called “cloning”. To Clone a list, we can take help of the slicing operation [:].

Example:

```
x = [10, 20, 30, 40, 50, 60]
y=x[:] # x is cloned as y
print(x) # [10,20,30,40,50,60]
print(y) # [10,20,30,40,50,60]
x[1]=90 # modify 1st element in x
print(x) # [10,90,30,40,50,60]
print(y) # [10,20,30,40,50,60]
```



When we clone a list like this, a separate copy of all the elements is stored into “y”. The lists ‘x’ and ‘y’ are independent lists. Hence, any modifications to “x” will not affect “y” and vice versa.

Methods in Lists:

Method	Description
<i>lst.index(x)</i>	Returns the first occurrence of x in the list.
<i>lst.append(x)</i>	Appends x at the end of the list.
<i>lst.insert(i,x)</i>	Inserts x to the list in the position specified by i.
<i>lst.copy()</i>	Copies all the list elements into a new list and returns it.
<i>lst.extend(lst2)</i>	Appends lst2 to list.
<i>lst.count(x)</i>	Returns number of occurrences of x in the list.
<i>lst.remove(x)</i>	Removes x from the list.
<i>lst.pop()</i>	Removes the ending element from the list.
<i>lst.sort()</i>	Sorts the elements of list into ascending order.
<i>lst.reverse()</i>	Reverses the sequence of elements in the list.
<i>lst.clear()</i>	Deletes all elements from the list.
<i>max(lst)</i>	Returns biggest element in the list.
<i>min(lst)</i>	Returns smallest element in the list.

Example:

```
lst=[10,25,45,51,45,51,21,65]
lst.insert(1,46)
print(lst) # [10,46,25,45,51,45,51,21,65]
print(lst.count(45)) # 2
```

Finding Common Elements in Lists:

Sometimes, it is useful to know which elements are repeated in two lists. For example, there is a scholarship for which a group of students enrolled in a college. There is another scholarship for which another group of students got enrolled. Now, we can know the names of the students who enrolled for both the scholarships so that we can restrict them to take only one scholarship. That means, we are supposed to find out the common students (or elements) both the lists.

First of all, we should convert the lists into sets, using `set()` function, as: `set(list)`.

Then we should find the common elements in the two sets using `intersection()` method.

Example:

```
scholar1=["mothi", "sudheer", "vinay", "narendra", "ramakoteswararao" ]
scholar2=["vinay", "narendra", "ramesh"]
s1=set(scholar1)
s2=set(scholar2)
s3=s1.intersection(s2)
common =list(s3)
print(common) # display [ "vinay", "narendra" ]
```

Nested Lists:

A list within another list is called a *nested list*. We know that a list contains several elements.

When we take a list as an element in another list, then that list is called a nested list.

Example:

```
a=[10,20,30]
b=[45,65,a]
print(b) # display [ 45, 65, [ 10, 20, 30 ] ]
print(b[1]) # display 65
print(b[2]) # display [ 10, 20, 30 ]
print(b[2][0]) # display 10
print(b[2][1]) # display 20
print(b[2][2]) # display 30
for x in b[2]:
    print(x,) # display 10 20 30
```

Nested Lists as Matrices:

Suppose we want to create a matrix with 3 rows 3 columns, we should create a list with 3 other lists as:

```
mat = [ [ 1, 2, 3 ] , [ 4, 5, 6 ] , [ 7, 8, 9 ] ]
```

Here, "mat" is a list that contains 3 lists which are rows of the "mat" list. Each row contains again 3 elements as:

```
[ [ 1, 2, 3] ,           # first row
[ 4, 5, 6] ,           # second row
[ 7, 8, 9] ]          # third row
```

Example:

```
mat=[[1,2,3],[4,5,6],[7,8,9]]
for r in mat:
    print(r)
print("")

m=len(mat)
n=len(mat[0])
for i in range(0,m):
    for j in range(0,n):
        print(mat[i][j],end= ' ')
    print("")
print("")
```

[1,	2,	3]
[4,	5,	6]
[7,	8,	9]

1	2	3
4	5	6
7	8	9

One of the main uses of nested lists is that they can be used to represent matrices. A matrix represents a group of elements arranged in several rows and columns. In python, matrices are created as 2D arrays or using matrix object in numpy. We can also create a matrix using nested lists.

Q) Write a program to perform addition of two matrices.

```
a=[[1,2,3],[4,5,6],[7,8,9]]
b=[[4,5,6],[7,8,9],[1,2,3]]
c=[[0,0,0],[0,0,0],[0,0,0]]
m1=len(a)
n1=len(a[0])
m2=len(b)
n2=len(b[0])
for i in range(0,m1):
    for j in range(0,n1):
        c[i][j]= a[i][j]+b[i][j]
for i in range(0,m1):
    for j in range(0,n1):
        print(c[i][j],end='\t')
    print()
```

5	7	9
11	13	15
8	10	12

Q) Write a program to perform multiplication of two matrices.

```
a=[[1,2,3],[4,5,6]]
b=[[4,5],[7,8],[1,2]]
c=[[0,0],[0,0]]
m1=len(a)
n1=len(a[0])
m2=len(b)
n2=len(b[0])
for i in range(0,m1):
    for j in range(0,n2):
        for k in range(0,n1):
            c[i][j] += a[i][k]*b[k][j]
```

21 27
57 72

```
for i in range(0,m1):
    for j in range(0,n2):
        print(c[i][j],end=' ')
    print("")
```

List Comprehensions:

List comprehensions represent creation of new lists from an iterable object (like a list, set, tuple, dictionary or range) that satisfy a given condition. List comprehensions contain very compact code usually a single statement that performs the task.

We want to create a list with squares of integers from 1 to 100. We can write code as:

```
squares=[ ]
for i in range(1,11):
    squares.append(i**2)
```

The preceding code will create “squares” list with the elements as shown below:

```
[ 1, 4, 9, 16, 25, 36, 49, 64, 81, 100 ]
```

The previous code can be rewritten in a compact way as:

```
squares=[x**2 for x in range(1,11)]
```

This is called list comprehension. From this, we can understand that a list comprehension consists of square braces containing an expression (i.e., $x**2$). After the expression, a for loop and then zero or more if statements can be written.

```
[ expression    for    item1    In   iterable   if   statement1
                               for   item1    In   iterable   if   statement2
                               for   item1    In   iterable   if   statement3   ....]
```

Example:

```
Even_squares = [ x**2 for x in range(1,11) if x%2==0 ]
```

It will display the list even squares as list.

```
[ 4, 16, 36, 64, 100 ]
```

Dictionary:

A dictionary represents a group of elements arranged in the form of key-value pairs.

The first element is considered as “key” and the immediate next element is taken as its “value”. The key and its value are separated by a colon (:). All the key-value pairs in a dictionary are inserted in curly braces { }.

```
d= { "Regd.No": 556, "Name": "Mothi", "Branch": "CSE" }
```

Here, the name of dictionary is “dict”. The first element in the dictionary is a string “Regd.No”. So, this is called “key”. The second element is 556 which is taken as its “value”.

Example:

```
d={"Regd.No": 556,"Name": "Mothi", "Branch": "CSE"}  
print( d["Regd.No"]) # 556  
print( d["Name"]) # Mothi  
print( d["Branch"]) # CSE
```

To access the elements of a dictionary, we should not use indexing or slicing. For example, dict[0] or dict[1:3] etc. expressions will give error. To access the value associated with a key, we can mention the key name inside the square braces, as: dict[“Name”].

If we want to know how many key-value pairs are there in a dictionary, we can use the len() function, as shown

```
d={"Regd.No": 556,"Name": "Mothi", "Branch": "CSE"}  
print len(d) # 3
```

We can also insert a new key-value pair into an existing dictionary. This is done by mentioning the key and assigning a value to it.

```
d={'Regd.No':556,'Name':'Mothi','Branch':'CSE'}  
print(d) #{'Branch': 'CSE', 'Name': 'Mothi', 'Regd.No': 556}  
d['Gender']="Male"
```

```
print(d) # {'Gender': 'Male', 'Branch': 'CSE', 'Name': 'Mothi', 'Regd.No': 556}
```

Suppose, we want to delete a key-value pair from the dictionary, we can use *del* statement as:

```
del dict["Regd.No"] #{'Gender': 'Male', 'Branch': 'CSE', 'Name': 'Mothi'}
```

To Test whether a “key” is available in a dictionary or not, we can use “in” and “not in” operators. These operators return either True or False.

```
"Name" in d # check if „Name“ is a key in d and returns True / False
```

We can use any datatypes for value. For example, a value can be a number, string, list, tuple or another dictionary. But keys should obey the rules:

➤ Keys should be unique. It means, duplicate keys are not allowed. If we enter same key again, the old key will be overwritten and only the new key will be available.


```
emp={'nag':10,'vishnu':20,'nag':20}
```

```
print(emp)# {'nag': 20, 'vishnu': 20}
```

➤ Keys should be immutable type. For example, we can use a number, string or tuples as keys since they are immutable. We cannot use lists or dictionaries as keys. If they are used as keys, we will get “TypeError”.

```
emp={['nag']:10,'vishnu':20,'nag':20}
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    emp={['nag']:10,'vishnu':20,'nag':20}
TypeError: unhashable type: 'list'
```

Dictionary Methods:

Method	Description
d.clear()	Removes all key-value pairs from dictionary "d".
d2=d.copy()	Copies all elements from "d" into a new dictionary d2.
d.fromkeys(s [,v])	Create a new dictionary with keys from sequence "s" and values all set to "v".
d.get(k [,v])	Returns the value associated with key "k". If key is not found, it returns "v".
d.items()	Returns an object that contains key-value pairs of "d". The pairs are stored as tuples in the object.
d.keys()	Returns a sequence of keys from the dictionary "d".
d.values()	Returns a sequence of values from the dictionary "d".
d.update(x)	Adds all elements from dictionary "x" to "d".
d.pop(k [,v])	Removes the key „k" and its value from "d" and returns the value. If key is not found, then the value "v" is returned. If key is not found and "v" is not mentioned then "KeyError" is raised.
d.setdefault(k [,v])	If key "k" is found, its value is returned. If key is not found, then the k, v pair is stored into the dictionary "d".

Using for loop with Dictionaries:

for loop is very convenient to retrieve the elements of a dictionary. Let's take a simple dictionary that contains color code and its name as:

```
colors = { 'r':"RED", 'g':"GREEN", 'b':"BLUE", 'w':"WHITE" }
```

Keys are "r", "g", "b" and "RED", "GREEN", "BLUE" and "WHITE" indicate values.

```
colors = { 'r':"RED", 'g':"GREEN", 'b':"BLUE", 'w':"WHITE" }
for k in colors:
    print(k) # displays only
keysfor k in colors:
    print(colors[k]) # keys to dictionary and display the values
```

Converting Lists into Dictionary:

When we have two lists, it is possible to convert them into a dictionary. For example, we have two lists containing names of countries and names of their capital cities.

There are two steps involved to convert the lists into a dictionary. The first step is to create a “zip” class object by passing the two lists to zip() function. The zip() function is useful to convert the sequences into a zip class object. The second step is to convert the zip object into a dictionary by using dict() function.

Example:

```
countries = [ 'USA', 'INDIA', 'GERMANY', 'FRANCE' ]  
cities = [ 'Washington', 'New Delhi', 'Berlin', 'Paris' ]  
z=zip(countries, cities)  
d=dict(z)  
print d
```

Output:

```
{'GERMANY': 'Berlin', 'INDIA': 'New Delhi', 'USA': 'Washington', 'FRANCE': 'Paris'}
```

Converting Strings into Dictionary:

When a string is given with key and value pairs separated by some delimiter like a comma (,) we can convert the string into a dictionary and use it as dictionary.

```
test_str = 'gfg:1, is:2, best:3'  
# printing original string  
print("The original string is : " + str(test_str))  
# Convert key-value String to dictionary  
# Using map() + split() + loop  
res = []  
for sub in test_str.split(', '):  
    if ':' in sub:  
        res.append(map(str.strip, sub.split(':', 1)))  
res = dict(res)  
# printing result  
print("The converted dictionary is : " + str(res))
```

Output:

```
The original string is : gfg:1, is:2, best:3
```

```
The converted dictionary is : {'gfg': '1', 'is': '2', 'best': '3'}
```

Q) A Python program to create a dictionary and find the sum of values.

```
d={'m1':85,'m3':84,'eng':86,'c':91}
sum=0
for i in d.values():
    sum+=i
print(sum) # 346
```

Q) A Python program to create a dictionary with cricket player's names and scores in a match. Also we are retrieving runs by entering the player's name.

```
n=input("Enter How many players? ")
d={ }
for i in range(0,n):
    k=input("Enter Player name: ")
    v=input("Enter score: ")
    d[k]=v
print(d)
name=input("Enter name of player for score: ")
print("The Score is",d[name])
```

Output:

```
Enter How many players? 3
Enter Player name: "Sachin"
Enter score: 98
Enter Player name: "Sehwag"
Enter score: 91
Enter Player name: "Dhoni"
Enter score: 95
{'Sehwag': 91, 'Sachin': 98, 'Dhoni': 95}
Enter name of player for score: "Sehwag"
The Score is 91
```

FUNCTIONS:

A function is a block of organized, reusable code that is used to perform a single, related action.

- Once a function is written, it can be reused as and when required. So, functions are also called reusable code.
- Functions provide modularity for programming. A module represents a part of the program. Usually, a programmer divides the main task into smaller sub tasks called modules.
- Code maintenance will become easy because of functions. When a new feature has to be added to the existing software, a new function can be written and integrated into the software.
- When there is an error in the software, the corresponding function can be modified without disturbing the other functions in the software.
- The use of functions in a program will reduce the length of the program.

As you already know, Python gives you many built-in functions like `sqrt()`, etc. but you can also create your own functions. These functions are called *user-defined functions*.

Difference between a function and a method:

A function can be written individually in a python program. A function is called using its name. When a function is written inside a class, it becomes a „method“. A method is called using object name or class name. A method is called using one of the following ways:

Objectname.methodname()Classname.methodname()

Defining a Function

You can define functions to provide the required functionality. Here are simple rules to define a function in Python.

- Function blocks begin with the keyword **def** followed by the function name and parentheses ().
- Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
- The first statement of a function can be an optional statement - the documentation string of the function or *docstring*.
- The code block within every function starts with a colon (:) and is indented.
- The statement `return [expression]` exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as `return none`.

Syntax:

```
def functionname (parameters):  
    """function_docstring"""  
    function_suite  
    return
```

By default, parameters have a positional behavior and you need to inform them in the same order that they were defined.

Example:

```
def add(a,b):  
    """This function sum the  
    numbers"""  
    c=a+b  
    print(c)  
    return
```

Here, “def” represents starting of function. “add” is function name. After this name, parentheses () are compulsory as they denote that it is a function and not a variable or something else. In the parentheses we wrote two variables “a” and “b” these variables are called “parameters”. A parameter is a variable that receives data from outside a function. So, this function receives two values from outside and those are stored in the variables “a” and “b”. After parentheses, we put colon (:) that represents the beginning of the function body. The function body contains a group of statements called “suite”.

Calling Function:

A function cannot run by its own. It runs only when we call it. So, the next step is to call function using its name. While calling the function, we should pass the necessary values to the function in the parentheses as:

```
add(5,12)
```

Here, we are calling “add” function and passing two values 5 and 12 to that function.

When this statement is executed, the python interpreter jumps to the function definition and copies the values 5 and 12 into the parameters „a” and „b” respectively.

Example:

```
def add(a,b):  
    """This function sum the numbers"""  
    c=a+b  
    print(c)  
add(5,12) # 17
```

Returning Results from a function:

We can return the result or output from the function using a “return” statement in the function body. When a function does not return any result, we need not write the return statement in the body of the function.

Q) Write a program to find the sum of two numbers and return the result from the function.

```
def add(a,b):  
    """This function sum the numbers"""  
    c=a+b  
    return c  
print(add(5,12)) # 17  
print(add(1.5,6)) #6.5
```

Returning multiple values from a function:

A function can return a single value in the programming languages like C, C++ and JAVA. But, in python, a function can return multiple values. When a function calculates multiple results and wants to return the results, we can use return statement as:

return a, b, c

Here, three values which are in "a", "b" and "c" are returned. These values are returned by the function as a tuple. To grab these values, we can use three variables at the time of calling the function as:

x, y, z = functionName()

Here, "x", "y" and "z" are receiving the three values returned by the function.

Example:

```
def calc(a,b):
    c=a+b
    d=a-b
    e=a*b
    return c,d,e
x,y,z=calc(5,8)
print("Addition=",x)
print("Subtraction=",y )
print("Multiplication=",z)
```

Functions are First Class Objects (or) Higher Order Functions:

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. Since functions are objects, we can pass a function to another function just like we pass an object (or value) to a function. The following possibilities are:

- 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.

To understand these points, we will take a few simple programs.

- A python program to see how to assign a function to a variable.

```
def display(st):
    return "hai"+st
x=display("cse")
print(x)
```

Output: haicse

- A python program to know how to define a function inside another function.

```
def display(st):
    def message():
        return "how r u?"
    res=message()+st
    return res
x=display("cse")
print(x)
```

The scope of inner function is limited to outer function.

Output: how r u?cse

- A python program to know how to pass a function as parameter to another function.

```
def display(f):
    return "hai"+f
def message():
    return "how r u?"
fun=display(message())
print(fun)
```

Output: haihow r u?

- A python program to know how a function can return another function.

```
def display():
    def message():
        return "how r u?"
    return message
fun=display()
print(fun())
```

Output: how r u?

Managing a Program's Namespace:

```
replacements = {"I":"you", "me":"you", "my":"my""your", "we":"you", "us":"you", "mine":"yours"}
```

```
def changePerson(sentence):
    """Replaces first person pronouns with second person pronouns."""
    words = sentence.split()
    replyWords = []
    for word in words:
        replyWords.append(replacements.get(word, word))
    return " ".join(replyWords)
```

This code appears in the file **doctor.py**(assume), so its module name is doctor. The names in this code fall into four categories, depending on where they are introduced:

1. **Module variables.** The names `replacements` and `changePerson` are introduced at the level of the module. Although `replacements` names a dictionary and `changePerson` names a function, they are both considered variables. You can see the module variables of the `doctor` module by importing it and entering `dir(doctor)` at a shell prompt. When module variables are introduced in a program, they are immediately given a value.
2. **Parameters.** The name `sentence` is a parameter of the function `changePerson`. A parameter name behaves like a variable and is introduced in a function or method header. The parameter does not receive a value until the function is called.
3. **Temporary variables.** The names `words`, `replyWords`, and `word` are introduced in the body of the function `changePerson`. Like module variables, temporary variables receive their values as soon as they are introduced.
4. **Method names.** The names `split` and `join` are introduced or defined in the `str` type. As mentioned earlier, a method reference always uses an object, in this case, a string, followed by a dot and the method name.

Pass by Value: Pass by value represents that a copy of the variable value is passed to the function and any modifications to that value will not reflect outside the function. In python, the values are sent to functions by means of object references. We know everything is considered as an object in python. All numbers, strings, tuples, lists and dictionaries are objects.

If we store a value into a variable as:

x=10

In python, everything is an object. An object can be imagined as a memory block where we can store some value. In this case, an object with the value “10” is created in memory for which a name “x” is attached. So, 10 is the object and “x” is the name or tag given to that object. Also, objects are created on heap memory which is a very huge memory that depends on the RAM of our computer system.

Example: A Python program to pass an integer to a function and modify it.

```
def modify(x):  
    x=15  
    print("inside",x,id(x))  
x=10  
modify(x)  
print("outside",x,id(x))
```

Output:

```
inside 15 6356456  
outside 10 6356516
```

From the output, we can understand that the value of “x” in the function is 15 and that is not available outside the function. When we call the modify() function and pass “x” as:

modify(x)

We should remember that we are passing the object references to the modify() function. The object is 10 and its references name is “x”. This is being passed to the modify() function. Inside the function, we are using:

x=15

This means another object 15 is created in memory and that object is referenced by the name “x”. The reason why another object is created in the memory is that the integer objects are immutable (not modifiable). So in the function, when we display “x” value, it will display 15. Once we come outside the function and display “x” value, it will display numbers of “x” inside and outside the function, and we see different numbers since they are different objects.

In python, integers, floats, strings and tuples are immutable. That means their data cannot be modified. When we try to change their value, a new object is created with the modified value.

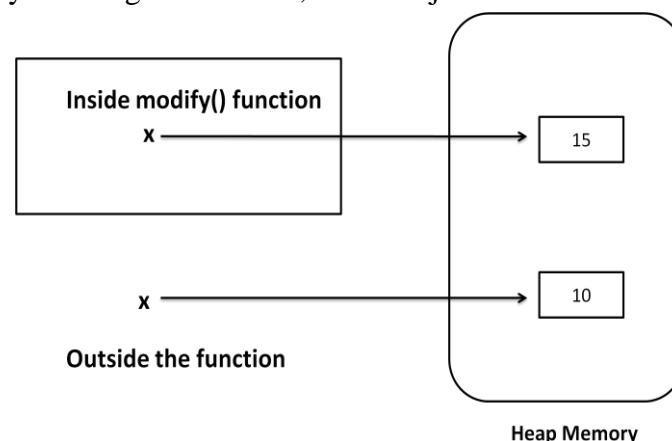


Fig. Passing Integer to a Function

Pass by Reference: Pass by reference represents sending the reference or memory address of the variable to the function. The variable value is modified by the function through memory address and hence the modified value will reflect outside the function also.

In python, lists and dictionaries are mutable. That means, when we change their data, the same object gets modified and new object is not created. In the below program, we are passing a list of numbers to modify () function. When we append a new element to the list, the same list is modified and hence the modified list is available outside the function also.

Example: A Python program to pass a list to a function and modify it.

```
def modify(a):
    a.append(5)
    print("inside",a,id(a))
a=[1,2,3,4]
modify(a)
print("outside",a,id(a))
```

Output:

```
inside [1, 2, 3, 4, 5] 45355616
outside [1, 2, 3, 4, 5] 45355616
```

In the above program the list “a” is the name or tag that represents the list object. Before calling the modify() function, the list contains 4 elements as: **a=[1,2,3,4]**

Inside the function, we are appending a new element “5” to the list. Since, lists are mutable, adding a new element to the same object is possible. Hence, append() method modifies the same object.

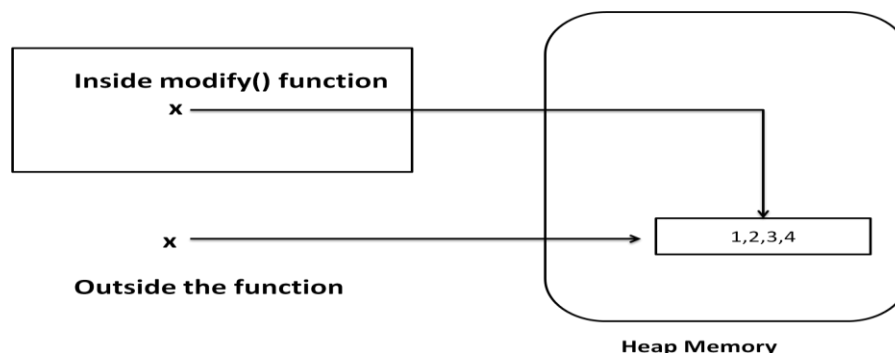


Fig. Passing a list to the function

Formal and Actual Arguments:

When a function is defined, it may have some parameters. These parameters are useful to receive values from outside of the function. They are called “formal arguments”. When we call the function, we should pass data or values to the function. These values are called “actual arguments”. In the following code, “a” and “b” are formal arguments and “x” and “y” are actual arguments.

Example:

```
def add(a,b): # a, b are formal arguments
    c=a+b
    print(c)
x,y=10,15
add(x,y) # x, y are actual arguments
```

The actual arguments used in a function call are of 4 types:

- a) Positional arguments
 - b) Keyword arguments
 - c) Default arguments
 - d) Variable length arguments
- a) **Positional Arguments:** These are the arguments passed to a function in correct positional order. Here, the number of arguments and their position in the function definition should match exactly with the number and position of argument in the function call.

```
def attach(s1,s2):  
    s3=s1+s2  
    print(s3)  
attach("New","Delhi") #Positional arguments
```

This function expects two strings that too in that order only. Let's assume that this function attaches the two strings as s1+s2. So, while calling this function, we are supposed to pass only two strings as: **attach("New", "Delhi")**

The preceding statements displays the following output NewDelhi

Suppose, we passed "Delhi" first and then "New", then the result will be: "DelhiNew". Also, if we try to pass more than or less than 2 strings, there will be an error.

- b) **Keyword Arguments:** Keyword arguments are arguments that identify the parameters by their names. For example, the definition of a function that displays grocery item and its price can be written as:

def grocery(item, price):

At the time of calling this function, we have to pass two values and we can mention which value is for what. For example,

grocery(item='sugar', price=50.75)

Here, we are mentioning a keyword "item" and its value and then another keyword "price" and its value. Please observe these keywords are nothing but the parameter names which receive these values. We can change the order of the arguments as:

grocery(price=88.00, item='oil')

In this way, even though we change the order of the arguments, there will not be any problem as the parameter names will guide where to store that value.

```
def grocery(item,price):  
    print("item=",item )  
    print("price=",price)  
grocery(item="sugar",price=50.75) # keyword arguments  
grocery(price=88.00,item="oil") # keyword arguments
```

Output:

item= sugar price= 50.75

item= oil price= 88.0

- c) **Default Arguments:** We can mention some default value for the function parameters in the definition.

Let's take the definition of grocery() function as:

def grocery(item, price=40.00)

Here, the first argument is “item” whose default value is not mentioned. But the second argument is “price” and its default value is mentioned to be 40.00, at the time of calling this function, if we do not pass “price” value, then the default value of 40.00 is taken. If we mention the “price” value, then mentioned value is utilized. So, a default argument is an argument that assumes a default value if a value is not provided in the function call for that argument.

Example:

```
def grocery(item,price=40.00):  
    print("item=",item)  
    print("price=",price)  
grocery(item="sugar",price=50.75)  
grocery(item="oil")
```

Output:

```
item= sugar  
price= 50.75  
item= oil  
price= 40.0
```

d) Variable Length Arguments: Sometimes, the programmer does not know how many values a function may receive. In that case, the programmer cannot decide how many arguments to be given in the function definition. for example, if the programmer is writing a function to add two numbers, he/she can write: **add(a,b)**

But, the user who is using this function may want to use this function to find sum of three numbers. In that case, there is a chance that the user may provide 3 arguments to this function as: **add(10,15,20)**

Then the add() function will fail and error will be displayed. If the programmer want to develop a function that can accept „n” arguments, that is also possible in python. For this purpose, a variable length argument is used in the function definition. a variable length argument is an argument that can accept any number of values. The variable length argument is written with a „*” symbol before it in the function definition as:

def add(farg, *args):

here, “farg” is the formal; argument and “*args” represents variable length argument. We can pass 1 or more values to this “*args” and it will store them all in a tuple.

Example:

```
def add(farg,*args):  
    sum=0  
    for i in args:  
        sum=sum+i  
    print("sum is",sum+farg)  
add(5,10)  
add(5,10,20)  
add(5,10,20,30)
```

Output:

sum is 15

sum is 35

sum is 65

Local and Global Variables:

When we declare a variable inside a function, it becomes a local variable. A local variable is a variable whose scope is limited only to that function where it is created. That means the local variable value is available only in that function and not outside of that function.

When the variable “a” is declared inside myfunction() and hence it is available inside that function. Once we come out of the function, the variable „a” is removed from memory and it is not available.

Example-1:

```
def myfunction():
    a=10
    print("Inside function",a) #display 10
myfunction()
print("outside function",a ) # Error, not available
```

Output:

Inside function 10

outside function

NameError: name 'a' is not defined

When a variable is declared above a function, it becomes global variable. Such variables are available to all the functions which are written after it.

Example-2:

```
a=11
def myfunction():
    b=10
    print("Inside function",a ) #display global var
    print("Inside function",b )#display local var
myfunction()
print("outside function",a) # available
print("outside function",b) # error
```

Output:

Inside function 11

Inside function 10

outside function 11

outside function

NameError: name 'b' is not defined

The Global Keyword:

Sometimes, the global variable and the local variable may have the same name. In that case, the function, by default, refers to the local variable and ignores the global variable. So, the global variable is not accessible inside the function but outside of it, it is accessible.

Example-1:

```
a=11
def myfunction():
    a=10
    print("Inside function",a) # display local variable
myfunction()
print("outside function",a) # display global variable
```

Output:

```
Inside function 10
outside function 11
```

When the programmer wants to use the global variable inside a function, he can use the keyword “global” before the variable in the beginning of the function body as:

global a

Example-2:

```
a=11
def myfunction():
    global a
    a=10
    print("Inside function",a) # display global variable
myfunction()
print("outside function",a) # display global variable
```

Output:

```
Inside function 10
outside function 10
```

Recursive Functions or Design with Recursive Functions:

A function that calls itself is known as “recursive function”. For example, we can write the factorial of 3 as:

```
factorial(3) = 3 * factorial(2)
Here, factorial(2) = 2 * factorial(1)
And, factorial(1) = 1 * factorial(0)
```

Now, if we know that the factorial(0) value is 1, all the preceding statements will evaluate and give the result as:

```
factorial(3) = 3 * factorial(2)
              = 3 * 2 * factorial(1)
              = 3 * 2 * 1 * factorial(0)
              = 3 * 2 * 1 * 1
              = 6
```

From the above statements, we can write the formula to calculate factorial of any number “n” as:

```
factorial(n) = n * factorial(n-1)
```

Example-1:

```
def factorial(n):
    if n==0:
        result=1
    else:
        result=n*factorial(n-1)
    return result
for i in range(1,5):
    print("factorial of ",i,"is",factorial(i))
```

Output:

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

For example, the Fibonacci sequence is a series of values with a recursive definition. The first and second numbers in the Fibonacci sequence are 1. Thereafter, each number in the sequence is the sum of its two predecessors, as follows: 1 1 2 3 5 8 13 . . .

More formally, a recursive definition of the n th **Fibonacci number** is the following:

$\text{Fib}(n) = 1$, when $n = 1$ or $n = 2$

$\text{Fib}(n) = \text{Fib}(n - 1) + \text{Fib}(n - 2)$, for all $n > 2$

Given this definition, you can construct a recursive function that computes and returns the n th Fibonacci number. Here it is:

```
def fib(n):
    """Returns the nth Fibonacci number."""
    if n < 3:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
```

Note that the base case as well as the two recursive steps return values to the caller.

Functions as Abstraction Mechanism: An abstraction hides detail and thus allows a person to view many things as just one thing. A function is a block of organized, reusable code that is used to perform a single, related action.

Functions Eliminate Redundancy: The first way that functions serve as abstraction mechanisms is by eliminating redundant, or repetitious, code.

Functions Hide Complexity: Another way that functions serve as abstraction mechanisms is by hiding complicated details.

Once a function is written, it can be reused as and when required. So, functions are also called reusable code.

Functions provide modularity for programming. A module represents a part of the program. Usually, a programmer divides the main task into smaller sub tasks called modules.

Code maintenance will become easy because of functions. When a new feature has to be added to the existing software, a new function can be written and integrated into the software.

When there is an error in the software, the corresponding function can be modified without disturbing the other functions in the software.

The use of functions in a program will reduce the length of the program.

Modules: A module is a file containing Python definitions and statements. The file name is the module name with the suffix.py appended. Within a module, the module's name (as a string) is available as the value of the global variable `__name__`. For instance, create a file called *fibonacci.py* in the current directory with the following contents:

```
# Fibonacci numbers module

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

def fib2(n): # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

Now enter the Python interpreter and import this module with the following command:

```
>>> import fibo
```

This does not enter the names of the functions defined in fibo directly in the current symbol table; it only enters the module name fibo there. Using the module name you can access the functions:

```
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

from statement:

- A module can contain executable statements as well as function definitions. These statements are intended to initialize the module. They are executed only the first time the module name is encountered in an import statement. (They are also run if the file is executed as a script.)
- Each module has its own private symbol table, which is used as the global symbol table by all functions defined in the module. Thus, the author of a module can use global variables in the module without worrying about accidental clashes with a user's global variables. On the other hand, if you know what you are doing you can touch a module's global variables with the same notation used to refer to its functions, `modname.itemname`.
- Modules can import other modules. It is customary but not required to place all import statements at the beginning of a module (or script, for that matter). The imported module names are placed in the importing module's global symbol table.
- There is a variant of the import statement that imports names from a module directly into the importing module's symbol table. For example:

```
>>> from fibo import fib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This does not introduce the module name from which the imports are taken in the local symbol table (so in the example, fibo is not defined).

There is even a variant to import all names that a module defines:

```
>>> from fibo import *
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Packages in Python: A package is a hierarchical file directory structure that defines a single Python application environment that consists of modules and subpackages and subsubpackages, and so on.

Third Party Packages:

The Python has got the greatest community for creating great python packages. There are more than 1,00,000 Packages available at <https://pypi.python.org/pypi>.

Python Package is a collection of all modules connected properly into one form and distributed PyPI, the Python Package Index maintains the list of Python packages available.

Now when you are done with pip setup Go to command prompt / terminal and say

pip install <package name>

Note: In windows, pip file is in “Python27\Scripts” folder. To install package you have to go to the path C:\Python27\Scripts in command prompt and install.

The requests and flask Packages are downloaded from internet. To download install the packages follow the commands

- **Installation of requests Package:**
 - ∞ **Command:** cd C:\Python27\Scripts
 - ∞ **Command:** pip install requests
- **Installation of flask Package:**
 - ∞ **Command:** cd C:\Python27\Scripts
 - ∞ **Command:** pip install flask

Example: Write a script that imports requests and fetch content from the page.

```
import requests
r = requests.get('https://www.google.com/')
print(r.status_code)
print(r.headers['content-type'])
print(r.text)
```



```
Python 2.7.13 Shell
File Edit Shell Debug Options Window Help
Python 2.7.13 (v2.7.13:a06454b1afal, Dec 17 2016, 20:42:59) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:/Python27/progs/12b.py =====
status code= 200
content type= text/html; charset=ISO-8859-1
Content is <!doctype html><html itemscope="" itemtype="http://schema.org/WebPage" lang="en-IN"><head>
<meta content="text/html; charset=UTF-8" http-equiv="Content-Type"><meta content="/images/branding/goog
gle/1x/google_standard_color_l28dp.png" itemprop="image"><title>Google</title><script>(function() {wi
ndow.google={kEI:'DzzJWYeWLSnvwgSutL2ABw',kEXPI:'1352614,1353383,1353746,1354277,1354401,1354443,13546
19,1354625,1354749,1354875,1355205,1355218,1355324,3700281,3700476,4029815,4031109,4043492,4045841,404
8347,4063220,4072776,4076999,4078430,4081039,4081165,4095910,4097153,4097922,4097929,4098733,4098740,4
098752,4101430,4101437,4102111,4102237,4103475,4103845,4103861,4104258,4104414,4109316,4109490,4110656
,4111590,4113217,4113604,4115697,4116724,4116731,4117327,4117539,4117980,4118103,4118227,4118798,41190
32,4119034,4119036,4119121,4119272,4119740,4119797,4119799,4119806,4120414,4120660,4120916,4121035,412
1174,4121350,4122025,4123641,4124173,4124220,4124411,4124850,4125477,4125837,4125963,4126204,4126242,4
126246,4126671,4127232,4127473,4127744,4127775,4127890,4128378,4128586,4129520,4129555,4130572,4130823
,4131073,4131247,4131419,4131646,4131647,4131871,4131943,4132309,4132420,4132451,4132588,4132618,41327
83,4132985,4133114,4133117,10200083,19003656,19003743,19003770,19003791,21060965',authuser:0,kscs:'c9c
918f0_41',u:'c9c918f0'};google.kHL='en-IN';})();(function(){google.lc=[];google.li=0;google.getEI=func
tion(a){for(var b;a&&(!a.getAttribute)||!(b=a.getAttribute("eid")));)a=a.parentNode;return b|google.kE
I};google.getLEI=function(a){for(var b=null;a&&(!a.getAttribute)||!(b=a.getAttribute("leid")));)a=a.par
entNode;return b};google.https=function(){return"https://"+window.location.protocol};google.ml=function
(){return null};google.wl=function(a,b){try{google.ml(Error(a),'l,b)}catch(c){};google.time=function(
){return(new Date).getTime();google.log=function(a,b,c,d,g){if(a=google.logUrl(a,b,c,d,g)){b=new Imag
e;var e=google.lc,f=google.li;e[f]=b;b.onerror=b.onload=b.onabort=function(){delete e[f]};google.vel&&
google.vel.lu&&google.vel.lu(a);b.src=a;google.li=f+1};google.logUrl=function(a,b,c,d,g){var e=""",f=g
oogle.ls|""",c||-1!=b.search("&ei=")|| (e="&ei="+google.getEI(d),-1==b.search("&lei=")&&(d=google.getLE
I(d))&&(e+="&lei="+d));d="";!c&&google.cshid&&-1==b.search("&cshid=")&&(d="&cshid="+google.cshid);a=c|
Ln:13 Col:4
```


There are some libraries in python:

- **Requests:** The most famous HTTP Library. It is a must and an essential criterion for every Python Developer.
- **Scrapy:** If you are involved in webscripting then this is a must have library for you. After using this library you won't use any other.
- **Pillow:** A friendly fork of PIL (Python Imaging Library). It is more user-friendly than PIL and is a must have for anyone who works with images.
- **SQLAlchemy:** It is a database library.
- **BeautifulSoup:** This xml and html parsing library.
- **Twisted:** The most important tool for any network application developer.
- **NumPy:** It provides some advanced math functionalities to python.
- **SciPy:** It is a library of algorithms and mathematical tools for python and has caused many scientists to switch from ruby to python.
- **Matplotlib:** It is a numerical plotting library. It is very useful for any data scientist or anydata analyzer.

Extra Topics (Not in Academic Syllabus):

TUPLE:

A Tuple is a python sequence which stores a group of elements or items. Tuples are similar to lists but the main difference is tuples are immutable whereas lists are mutable. Once we create a tuple we cannot modify its elements. Hence, we cannot perform operations like append(), extend(), insert(), remove(), pop() and clear() on tuples. Tuples are generally used to store data which should not be modified and retrieve that data on demand.

Creating Tuples:

We can create a tuple by writing elements separated by commas inside parentheses ().

The elements can be same datatype or different types.

To create an empty tuple, we can simply write empty parenthesis, as:

```
tup=( )
```

To create a tuple with only one element, we can, mention that element in parenthesis and after that a comma is needed. In the absence of comma, python treats the element as ordinary data type.

tup = (10)	tup = (10,)
print tup # display 10	print tup # display 10
print type(tup) # display <type „int“>	print type(tup) # display<type„tuple“>

To create a tuple with different types of elements:

```
tup=(10, 20, 31.5, „Gudivada“)
```

If we do not mention any brackets and write the elements separating them by comma, then they are taken by default as a tuple.

```
tup= 10, 20, 34, 47
```

It is possible to create a tuple from a list. This is done by converting a list into a tuple using tuple function.

```
n=[1,2,3,4]
tp=tuple(n)
print tp        # display (1,2,3,4)
```

Another way to create a tuple by using range() function that returns a sequence.

```
t=tuple(range(2,11,2))
print t        # display (2,4,6,8,10)
```

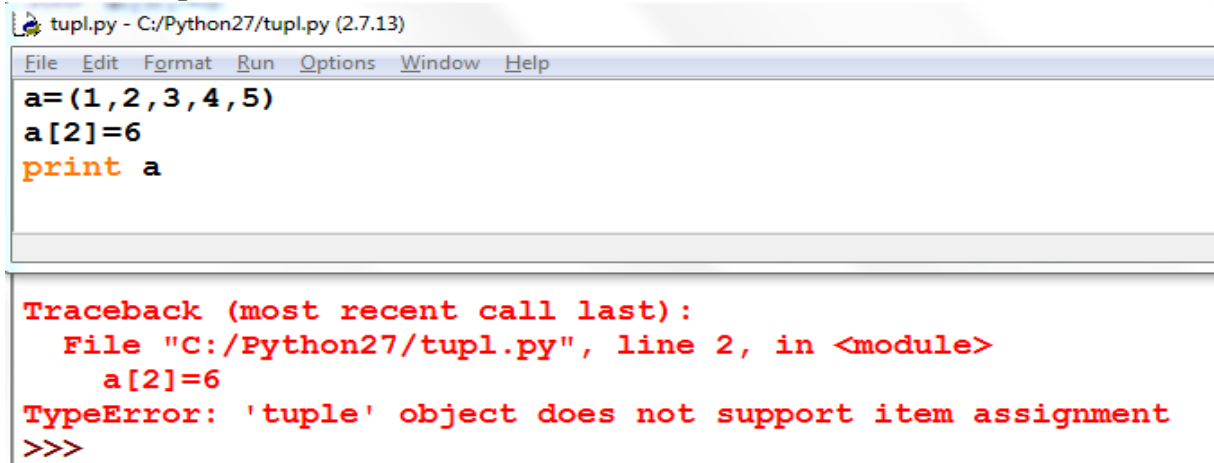
Accessing the tuple elements:

Accessing the elements from a tuple can be done using indexing or slicing. This is same as that of a list. Indexing represents the position number of the element in the tuple. The position starts from 0.

```
tup=(50,60,70,80,90)
print tup[0]        # display 50
print tup[1:4]        # display (60,70,80)
print tup[-1]        # display 90
print tup[-1:-4:-1]    # display (90,80,70)
print tup[-4:-1]        # display (60,70,80)
```

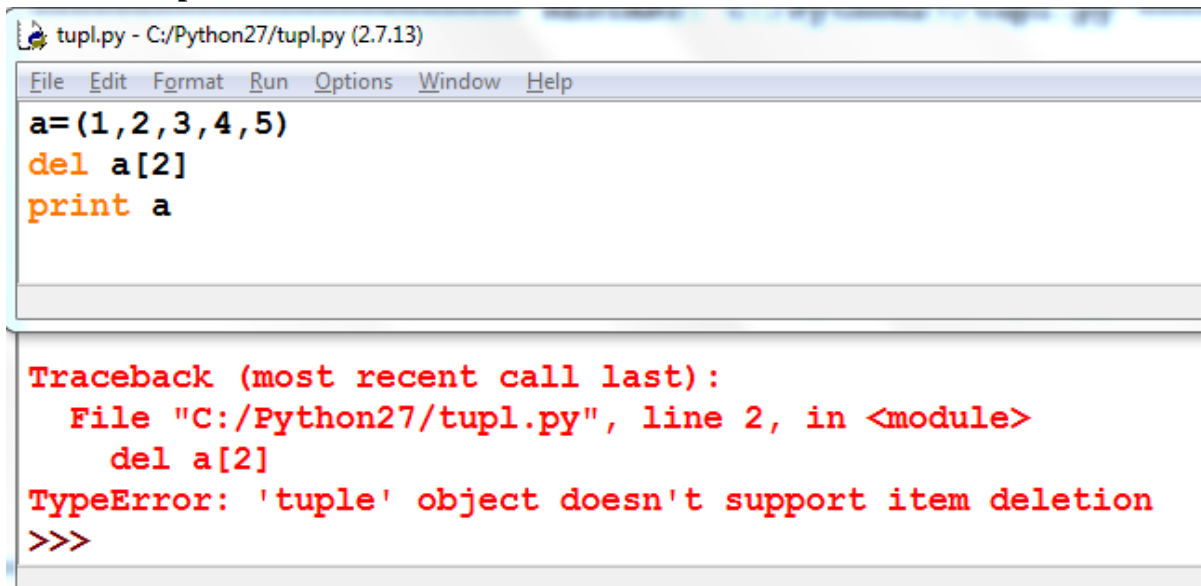
Updating and deleting elements:

Tuples are immutable which means you cannot update, change or delete the values of tuple elements.

Example-1:

```
tupl.py - C:/Python27/tupl.py (2.7.13)
File Edit Format Run Options Window Help
a=(1,2,3,4,5)
a[2]=6
print a

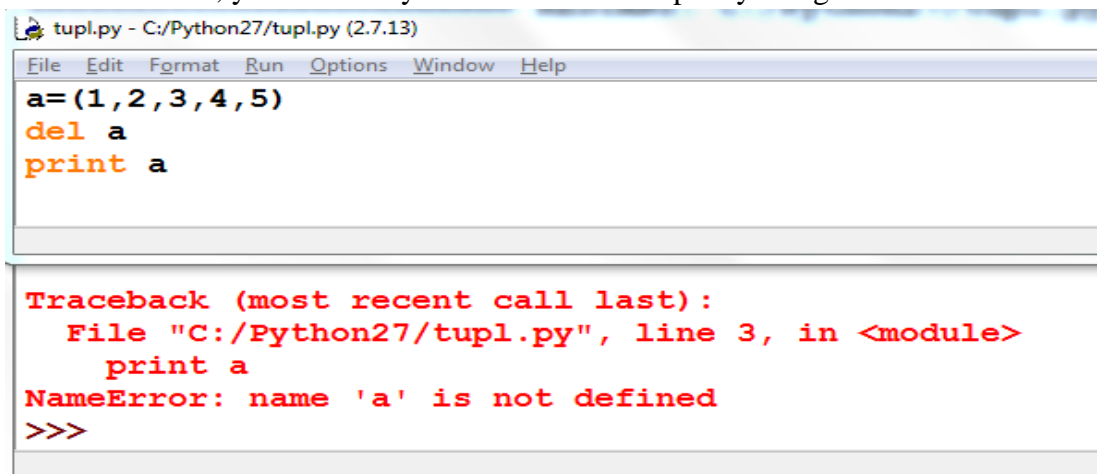
Traceback (most recent call last):
  File "C:/Python27/tupl.py", line 2, in <module>
    a[2]=6
TypeError: 'tuple' object does not support item assignment
>>>
```

Example-2:

```
tupl.py - C:/Python27/tupl.py (2.7.13)
File Edit Format Run Options Window Help
a=(1,2,3,4,5)
del a[2]
print a

Traceback (most recent call last):
  File "C:/Python27/tupl.py", line 2, in <module>
    del a[2]
TypeError: 'tuple' object doesn't support item deletion
>>>
```

However, you can always delete the entire tuple by using the statement.



```
tupl.py - C:/Python27/tupl.py (2.7.13)
File Edit Format Run Options Window Help
a=(1,2,3,4,5)
del a
print a

Traceback (most recent call last):
  File "C:/Python27/tupl.py", line 3, in <module>
    print a
NameError: name 'a' is not defined
>>>
```

Note that this exception is raised because you are trying print the deleted element.

Operations on tuple:

Operation	Description
len(t)	Return the length of tuple.
tup1+tup2	Concatenation of two tuples.
Tup*n	Repetition of tuple values in n number of times.
x in tup	Return True if x is found in tuple otherwise returns False.
cmp(tup1,tup2)	Compare elements of both tuples
max(tup)	Returns the maximum value in tuple.
min(tup)	Returns the minimum value in tuple.
tuple(list)	Convert list into tuple.
tup.count(x)	Returns how many times the element „x“ is found in tuple.
tup.index(x)	Returns the first occurrence of the element „x“ in tuple. Raises ValueError if „x“ is not found in the tuple.
sorted(tup)	Sorts the elements of tuple into ascending order. sorted(tup,reverse=True) will sort in reverse order.

cmp(tuple1, tuple2)

The method **cmp()** compares elements of two tuples.

Syntax

```
cmp(tuple1, tuple2)
```

Parameters

tuple1 -- This is the first tuple to be compared

tuple2 -- This is the second tuple to be compared

Return Value

If elements are of the same type, perform the compare and return the result. If elements are different types, check to see if they are numbers.

- If numbers, perform numeric coercion if necessary and compare.
- If either element is a number, then the other element is "larger" (numbers are "smallest").
- Otherwise, types are sorted alphabetically by name.

If we reached the end of one of the tuples, the longer tuple is "larger." If we exhaust both tuples and share the same data, the result is a tie, meaning that 0 is returned.

Example:

```
tuple1 = (123, 'xyz')
tuple2 = (456, 'abc')
print cmp(tuple1, tuple2)      #display -1
print cmp(tuple2, tuple1)      #display 1
```

Nested Tuples: Python allows you to define a tuple inside another tuple. This is called a *nested tuple*.

```
students=((“RAVI”, “CSE”, 92.00), (“RAMU”, “ECE”, 93.00), (“RAJA”, “EEE”, 87.00))
for i in students:
    print i
```

Output: (“RAVI”, “CSE”, 92.00)
 (“RAMU”, “ECE”, 93.00)

("RAJA", "EEE", 87.00)

SET:

Set is another data structure supported by python. Basically, sets are same as lists but with a difference that sets are lists with no duplicate entries. Technically a set is a mutable and an unordered collection of items. This means that we can easily add or remove items from it.

Creating a Set:

Set is created by placing all the elements inside curly brackets `{ }`. Separated by comma or by using the built-in function `set()`.

Syntax:

```
Set_variable_name={var1, var2, var3, var4, .....}
```

Example:

```
s={1, 2.5, "abc" }
print s          # display set( [ 1, 2.5, "abc" ] )
```

Converting a list into set:

A set can have any number of items and they may be of different data types. `set()` function is used to converting list into set.

```
s=set( [ 1, 2.5, "abc" ] )
print s          # display set( [ 1, 2.5, "abc" ] )
```

We can also convert tuple or string into set.

```
tup= ( 1, 2, 3, 4, 5 )

print set(tup) # set( [ 1, 2, 3, 4, 5 ] )
str= "MOTHILAL"
```

Operations on set:

Sno	Operation	Result
1	<code>len(s)</code>	number of elements in set <i>s</i> (cardinality)
2	<code>x in s</code>	test <i>x</i> for membership in <i>s</i>
3	<code>x not in s</code>	test <i>x</i> for non-membership in <i>s</i>
4	<code>s.issubset(t)</code> (or) <code>s <= t</code>	test whether every element in <i>s</i> is in <i>t</i>
5	<code>s.issuperset(t)</code> (or) <code>s >= t</code>	test whether every element in <i>t</i> is in <i>s</i>
6	<code>s == t</code>	Returns True if two sets are equivalent and returns False.
7	<code>s != t</code>	Returns True if two sets are not equivalent and returns False.
8	<code>s.union(t)</code> (or) <code>s t</code>	new set with elements from both <i>s</i> and <i>t</i>
9	<code>s.intersection(t)</code> (or) <code>s & t</code>	new set with elements common to <i>s</i> and <i>t</i>

Sno	Operation	Result
10	<code>s.difference(t)</code> (or) <code>s-t</code>	new set with elements in <i>s</i> but not in <i>t</i>
11	<code>s.symmetric_difference(t)</code> (or) <code>s ^ t</code>	new set with elements in either <i>s</i> or <i>t</i> but not both
12	<code>s.copy()</code>	new set with a shallow copy of <i>s</i>
13	<code>s.update(t)</code>	return set <i>s</i> with elements added from <i>t</i>
14	<code>s.intersection_update(t)</code>	return set <i>s</i> keeping only elements also found in <i>t</i>
15	<code>s.difference_update(t)</code>	return set <i>s</i> after removing elements found in <i>t</i>
16	<code>s.symmetric_difference_update(t)</code>	return set <i>s</i> with elements from <i>s</i> or <i>t</i> but not both
17	<code>s.add(x)</code>	add element <i>x</i> to set <i>s</i>
18	<code>s.remove(x)</code>	remove <i>x</i> from set <i>s</i> ; raises <u>KeyError</u> if not present
19	<code>s.discard(x)</code>	removes <i>x</i> from set <i>s</i> if present
20	<code>s.pop()</code>	remove and return an arbitrary element from <i>s</i> ; raises <u>KeyError</u> if empty
21	<code>s.clear()</code>	remove all elements from set <i>s</i>
22	<code>max(s)</code>	Returns Maximum value in a set
23	<code>min(s)</code>	Returns Minimum value in a set
24	<code>sorted(s)</code>	Return a new sorted list from the elements in the set.

Note:

To create an empty set you cannot write `s={ }`, because python will make this as a directory. Therefore, to create an empty set use `set()` function.

<code>s=set()</code>	<code>s={}</code>
<code>print type(s) # display <type „set“></code>	<code>print type(s) # display <type „dict“></code>

Updating a set:

Since sets are unordered, indexing has no meaning. Set operations do not allow users to access or change an element using indexing or slicing.