1. **Python data structures**

   Python has various data structures that give greater flexibility in storing different types of data and faster processing in python.

2.1 Python Lists

A list is used to store the various types of data. Lists are used to store multiple items in a single variable.

Syntax: list_name = [“item1”, “item2”, “item3”**]**

For example: mylist = ["AI", "Bigdata", "Cloud"]
print (mylist)

2.1.1 Characteristics of List

i. **List items are ordered**: In list elements have a defined and fixed order and that order will not be change. If we add any new element in the list then new element will be placed at the end of the list.

ii. **List Items are changeable:** we can add, remove and change any element from the list after it has created.

iii. **Allows duplicate Elements:** In list we can store similar elements.

For Example: mylist = ["nike", "puma", "woodland", "nike"]
print (mylist)

iv. **The elements of the list can be access by using the index**: The index of a list starts from 0 and goes to [length – 1]. The first element of the list is stored at the 0th index, the second element of the list is stored at the 1st index and so on.

Mylist = [“Big data”,“Cloudtechnology”, “Artificial Intelligence**”, “**Database”]

Index	0	1	2	3
Data	Bigdata	Cloud technology	Artificial Intelligence	Database

Mylist [0] = Bigdata, Mylist [1] = Cloud Technology**, Mylist [3] =** [Database]

Syntax: list_name (start: stop: step)

- The **start** indicates the starting index position of the list.
- The **stop** indicates the last index position of the list.
- The **step** is used to skip the nth element within a **start: stop**

2.1.2 Operations on the list

There are various logical operation can be performed on the list as mentioned below:

I. Access the elements of the list

- **By using the index no.**

```
mytlist = ["BCA", "BBA", "MBA"]
print(mylist[1])
```

- **Negative Indexing method**

```
mytlist = ["BCA", "BBA", "MBA"]
print(mylist[-1])
```

- **Range of Indexes**

```
mylist = ["apple", "banana", "cherry", "orange", "kiwi", "melon"]
```

[Note: index 1 (included) and end index 4 (not included).]

```
print (mylist[1:4])
```

- **Check if Item Exists**

To check any specified item is present the given list. We use [**’in’**] keyword:

```
mylist = ["apple", "banana", "cherry", "orange", "kiwi", "melon"]
if  “orange”  in mylist :
print ("Yes, 'apple' is in the fruits list")
```

II. Change the elements of the List

By using the index no we can change the element of the list.

```
mylist = ["BBA", "BCA", "MTech"]
mylist [2] = "BTech"
print (mylist)
```

- We can also change the more than one elements by using the range method. If you want to replace two elements, the first new elements will be inserted where you specified, and the remaining items will move accordingly:

  For example:

```
mylist = ["apple", "banana", "cherry", "orange", "kiwi", "mango"]
mylist [1:3] = ["blackcurrant", "watermelon"]
print (mylist)
```

Output:

```
['apple', 'blackcurrant', 'watermelon', 'orange', 'kiwi', 'mango']
```

III. Add elements in the List

To insert or add any new elements in the list at specific location/ index, we use **insert ()** method.

Syntax: list_name. Insert (index_no, “item”)

mylist = ["BBA", "BCA", "MTech"]

mylist.insert (2, "BTech")

print (mylist)

Output:

```
['BBA', 'BCA', 'BTech', 'MTech']
```

❖ **Add element in the list by using append () method:** append () method will insert the elements in the last index of the list.

mylist = ["BBA", "BCA", "MCA"]

mylist.append("BTech")

print(mylist)

Output:

```
['BBA', 'BCA', 'MCA', 'BTech']
```

I. **Extend the List:** To append elements from another list to the current list, by using **extend ()** method.

course = ["BBA", "BCA", "MCA"] // **List1**

newcourse = ["B.Com", "B.Tech", "M.Tech"] // **List2**

course. Extend (newcourse) // **extended list2 into list1**

print (course)

Output:

```
['BBA', 'BCA', 'MCA', 'B.Com', 'B.Tech', 'M.Tech']
```

IV. Remove elements from the List

I. By using element name

We can remove specific item from the list by using the remove ( ) method. We need to pass the element name as argument in remove method.

Syntax: list_name. remove (“element”)

course = ["BBA", "BCA", "MCA","B.Com"]

course.remove("B.Com") // Remove B.Com from the list

print(course)

Output:

```
['BBA', 'BCA', 'MCA']
```

II. By using Index method

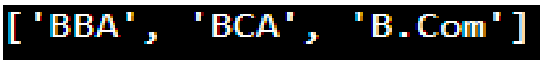
The POP () method is used to remove the specific item from the list. We need to specify the index no.

course = ["BBA", "BCA", "MCA","B.Com"]

course. pop (2)   // remove the 2nd index element from the list

print(course)

Output:



```
['BBA', 'BCA', 'B.Com']
```

[Note: if **index no.** is not given in pop () method then it will remove the last element from the list.]

III. Delete the list completely

We can delete list completely by using the **'del'** keyword.

Syntax: **del** list_name

For Example:

course = ["BBA", "BCA", "MCA","B.Com"]

del course   // delete the list

IV. Empty / clear the list

The clear () method can be used to empties the list. The list still remains, but there will be no content in the list.

Syntax: list_name. Clear ()

course = ["BBA", "BCA", "MCA","B.Com"]

course. clear ()

V. Print the items by using for loop

We can print the items of the list by using **for** loop i.e. Print all items in the list, one by one:

Syntax: **for** **[**variable_name**]** **in** list_name:

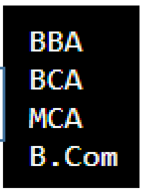
For Example:

course = ["BBA", "BCA", "MCA","B.Com"]

for x in course:

print(x)

Output



```
BBA
BCA
MCA
B.Com
```

VI. Sorting the list in Ascending or Descending order

Sort () method can be used to sort the items of the list. By default a list will be in ascending order.

Syntax: list_name. Sort ( )

For Example:

course = ["BBA", "BCA", "MCA","B.Com"]

Output

course. sort()   // sort method

```
['B.Com', 'BBA', 'BCA', 'MCA']
```

print(course)

VII. To sort a list in descending:

Use the argument in sort (**[reverse = True])** method.

course = ["BBA", "BCA", "MCA","B.Com"]

course. sort(reverse = True)

print(course)

Output

```
['MCA', 'BCA', 'BBA', 'B.Com']
```

VIII. Create a copy of the list

We can make a copy of the list by using the copy ( ) method. It will return a new list.

Syntax: **new_list = old_list. Copy ()**

course = ["BBA", "BCA", "MCA","B.Com"]

newlist = course.copy()

print(new_list)

Output

```
['BBA', 'BCA', 'MCA', 'B.Com']
```

[Note: We can also use **list ()** method to create a copy of list.**]**

IX. Join the lists

Joining refers to merge or concatenation of the two list. We can join the two list by using following methods:

- **By using the (+) operator**

Syntax: list3 = list1 + list 2

oldcourse = ["BBA", "BCA", "MCA","B.Com"] **// list 1**

newcourse = ["MBA","MTech","PHD"] **// list 2**

final = oldcourse + newcourse **// list3 = list1+ list2**

print (final)

Output

```
['BBA', 'BCA', 'MCA', 'B.Com', 'MBA', 'MTech', 'PHD']
```

- **By using append() method**

list1 = ["BBA", "BCA", "MCA","B.Com"]

list2 = ["MBA","MTech","PHD"]

for x in list2:

  list1.append(x) **// use of append () method two merge the list**

print (list1)

Output

```
['BBA', 'BCA', 'MCA', 'B.Com', 'MBA', 'MTech', 'PHD']
```

- **By using the extend() :**

We can add elements from one list to another list. The syntax of extend method is below:

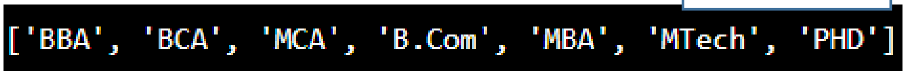
list1 = ["BBA", "BCA", "MCA","B.Com"]

list2 = ["MBA","MTech","PHD"]

list1.extend (list2)   // **extend () method**

print(list1)

Output



```
['BBA', 'BCA', 'MCA', 'B.Com', 'MBA', 'MTech', 'PHD']
```

Key methods and there use in the list		
S.No	**Method**	**Description**
1.	append()	Adds an element at the end of the list
2.	clear()	Removes all the elements from the list
3.	copy()	Returns a copy of the list
4.	count()	Returns the number of elements with the specified value
5.	extend()	Add the elements of a list to the end of the current list
6.	index()	Returns the index of the first element with the specified value
7.	insert()	Adds an element at the specified position
8.	pop()	Removes the element at the specified position
9.	remove()	Removes the item with the specified value
10.	reverse()	Reverses the order of the list
11.	sort()	Sorts the list in the Ascending or Descending order

Points to be remember: List Vs Array

1. *List can store multiple data types i.e. integer, string, float and Boolean.*
2. *Array can store only similar kind of data types.*
3. *List cannot manage arithmetic operations While on array can be performed.*
4. *It consumes a larger memory while array take less memory.*

Difference between List and Array

List	Array
Contains elements of different data types.	Contains elements of the same data types.
Explicitly importing modules is not required to declare a list.	Need to import the module explicitly to declare an array.
Cannot handle arithmetic operations.	Can handle arithmetic operations.
Can be nested inside another list.	Must contain all elements of the same size.
Mostly used in the shorter sequence of data elements.	Mostly used in the longer sequence of data elements.
Easy modifications like addition, deletion, and update of data elements are done.	It is difficult to modify an array since addition, deletion, and update operation is performed on a single element at a time.
We can print the entire list without the help of an explicit loop.	To print or access array elements, we will require an explicit loop.
For easy addition of elements, large memory storage is required.	In comparison to the list, it is more compact in-memory size.

2.2 Tuples

Tuples is a data structure in python which are used to store multiple items with different data types in a single variable. The key difference between list and tuples are that, we cannot change the elements of the tuples after it created. We can create any tuples by using () bracket.

Syntax: tuple_name = (‘item1’, ’item2’,’item3’)

For example: mytup = ('physics', 'chemistry', 2005, 2010);

2.2.1 Characteristic of tuples

Tuples has following characteristics

I. **Tuples are ordered**

Tuples have a defined order, and that order will not change.

II. **Tuples are unchangeable**

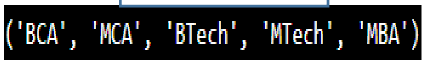
We cannot change, add, and remove the elements once tuples is created.

III. **Tuples allowed duplicate item**

Tuples follow the index method so we can store same elements in the tuples**.**

```
mytup = ("BCA", "MCA", "BTech", "MTech", "MBA")
print(mytup)
```

Output



```
('BCA', 'MCA', 'BTech', 'MTech', 'MBA')
```

2.2.2 Creating tuples with one item:

A tuple can be created with only one item but we need to add a comma after the item, otherwise Python will not recognize it as a tuple.

```
mytuple = ("PGDBDA",)                    # Valid tuple
print(type(mytuple))
mytuple = ("apple")                        #NOT a tuple
print(type(mytuple))
```

2.2.3 Operations on the tuples

There are following operation can be performed on a tuple:

I. **Access the items of tuples:** Items of the tuple can be accessed by using the index.

Syntax: tuple_name = (‘item1’,’item2’...)

```
mytup = ("Orange", "Bringle", "cherry")
print (mytup[1])            // here we want to access the 1st index item
```

We can use range index method for accessing the tuples:

```
mytuple = ("apple", "banana", "cherry", "orange", "kiwi", "mango")
```

print (thistuple[3:5])

[Note: start index will be **included** and end index will be **excluded.]**

II. **Change the items of tuples:** It was earlier discussed that if a tuple has create then you cannot change or modified or remove any elements in the tuple. But if we want to change the items then firstly we need to change tuple in to the list and then we can change/add/remove element in the list.

For Example:

```
x = ("BCA", "MCA", "BTech", "MTech", "MBA")
y = list(x)    # Converting tuple into list
y [2] = "B.Com"  # change the item
x = tuple (y)
print(x)
```

III. **Unpacking a Tuple**

When we create a tuple, we assign the values to it. This process is called **"packing"** a tuple**.** In Python, we are also allowed to extract the values back into variables. This is called "unpacking".

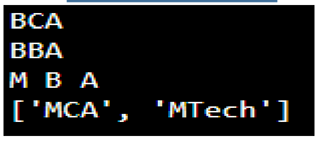
```
fruits = ("apple", "banana", "cherry")          // Packing process
(green, yellow, red) = fruits                   // unpacking
print (green)
print (yellow)
print (red)
```

Note: The number of variables must be equal to the number of values in the tuple, if not, then use an **asterisk** (*) to collect the remaining values as a list.

For Example:

```
mycourse = ("BCA", "BBA", "MBA", "MCA", "MTech")
(amit, sumit,*vaibhav, tarun) = mycourse
print(amit)
print(sumit)
print(*vaibhav)
print(tarun)
```

Output

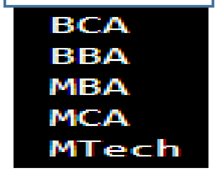


```
BCA
BBA
M B A
['MCA', 'MTech']
```

IV. **Print items using for loop**

```
mycourse = ("BCA", "BBA", "MBA", "MCA", "MTech")
for x in mycourse:                    # for loop
print(x)
```

Output



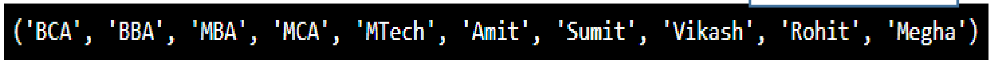
```
BCA
BBA
MBA
MCA
MTech
```

V. **Join the Two Tuples**

By using (+) operator we can merge two tuples as mentioned below:

```
mycourse = ("BCA", "BBA", "MBA", "MCA", "MTech")   # tuple1
student = ("Amit","Sumit","Vikash","Rohit","Megha")  # tuple2
list = mycourse + student                    # (+) operator adding two tuples
print (list)
```

Output



```
('BCA', 'BBA', 'MBA', 'MCA', 'MTech', 'Amit', 'Sumit', 'Vikash', 'Rohit', 'Megha')
```

2.3. Set

Python also has another rich data structure which also used to store multiple items. Set is just like similar to others data types like list and tuples. A set can be created by using the **{‘item’}** curly brackets.

Syntax: set_name = {‘item1’,’item2’,}

For Example:

```
myset = {"apple", "banana", "cherry"}
```

2.3.1 Characteristics of a set

I. **Set is unordered i.e.** cannot be sure in which order the elements will appear.

II. **Set is un-indexed i.e**. it did not use index to represent the elements.

III. **Set is unchangeable i.e.** elements can be add/remove but cannot change**.**

IV. **Duplicity not allowed in a set i.e.** cannot store two same elements.

set1 = {"**Amit",** “Tarun”,”**Amit”,** 55, 'M'}   # not allowed

print (set1)

V. **Can contains different data types i.e**. Integer, Float, String, Boolean in a set.

```
set1 = {"Amit", 34, True, 152.55, 'M'}
print (set1)
```

2.3.2 Operations on Set

I. **Access Items of the Set**

In set we cannot access items by referring to an index or a key. But by using **for** loop we can access the items of set.

```
myset = {"BBA", "BCA", "MCA"}
for x in myset:
print(x)
```

❖ We can also check whether an element exist in the set or not by using **in** keyword. **For Example**: print("BBA" **in** myset)

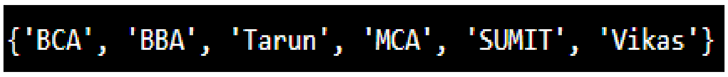
II. Add Set Items or set into a set

Once a set is created, you cannot change its items, but you can add new item By using **add ()** method.

```
myset = {"BBA", "BCA", "MCA"}
myset.add ("M.Tch")           # add an element
print (myset)
```

❖ **We can also add a set in to another set**

```
myset = {"BBA", "BCA", "MCA"}              # set1
newset = {"SUMIT", "Tarun", "Vikas"}       # set2
myset.update(newset)                 # newest add in to myset
print(myset)
```

Output: 

```
{'BCA', 'BBA', 'Tarun', 'MCA', 'SUMIT', 'Vikas'}
```

III. Remove the set items

Elements can be removed from a set by using **remove ()** or **discard ()** or **pop ()** method. Only single argument will be accepted by **remove ( )** and **discard ()** method. **Pop ()** method will not take any argument.

Syntax: set_name . remove ("Item")

```
myset = {"BBA", "BCA", "BTech"}
myset.remove("BCA")          # remove( ) method
print(myset)
```

[**Note:** if the requested item does not exist then it will raise an error]

```
myset = {"BBA", "BCA", "B.Tech"}
myset.discard("BCA")         # discard( ) method
print(myset)
```

[**Note:** if the requested item does not exist then it will not raise an error]

```
myset = {"BBA", "BCA", "B.Tech"}
myset.pop( )    # pop( ) method
print(myset)
```

[**Note:** Set is unordered. Pop () method will remove a random item from set, so we cannot be sure which item will be removed.]

IV. Empty the Set

Clear () method can be used to empty the Set. I.e. the items of Set will be deleted but set will remains. **Syntax:** Set_name . Clear ()

V. Delete a Set completely

To delete a set we can use **Del** keyword.

Syntax: **Del** Set_name

```
myset = {"BBA", "BCA", "MCA"}
del   myset
```

VI. Join Sets

There are two ways to join the set. **(I)** by using **union ( ) & (II) update ( )** method. The key difference in these method is that **union ( )** method will return a new set after joining process, while update ( ) method will add the items from one set to another set.

```
set1 = {"PDGBA", "PGDAC" , "PGDIT"}
set2 = {'Amit', "Tarun", "Mohit"}
set3 = set1.union (set2)                    # union () method
print (set3)
set1 = {"PDGBA", "PGDAC", "PGDIT"}
set2 = {'Amit', "Tarun", "Mohit"}
set2 = set1.update (set2)                   # update ( ) method
print (set1)
```

List of key methods and there use

Method	**Description**
add()	Adds an element to the set
clear()	Removes all the elements from the set
copy()	Returns a copy of the set
difference()	Returns a set containing the difference between two or more sets
difference_update()	Removes the items in this set that are also included in another, specified set
discard()	Remove the specified item
intersection()	Returns a set, that is the intersection of two other sets
intersection_update()	Removes the items in this set that are not present in other, specified set(s)
isdisjoint()	Returns whether two sets have a intersection or not
issubset()	Returns whether another set contains this set or not
issuperset()	Returns whether this set contains another set or not
pop()	Removes an element from the set
remove()	Removes the specified element
symmetric_difference()	Returns a set with the symmetric differences of two sets
symmetric_difference_update()	inserts the symmetric differences from this set and another
union()	Return a set containing the union of sets
update()	Update the set with the union of this set and others

2.4. Dictionary

Dictionary is also a key data structure in which we also can store multiple items with different data types. Dictionary store the data by using **key: value pairs** methodology. In python 3.7 dictionary is **ordered** while till 3.6 version it was **unordered.**

Syntax:

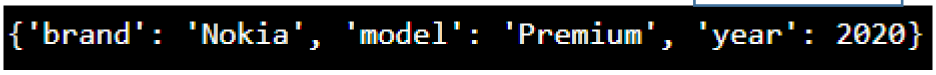
```
mydict = {
  "brand": "Nokia",     # key value pair
  "model": "premium",
  "year": 2010
  }
```

2.4.1 Characteristics of a Dictionary

- Dictionary items are ordered
- Changeable i.e. items can be change /add/ remove after the creation.
- Does not allow duplicates values.

```
tmydict = {
  "Brand": "Nokia",
  "model": "premium",
  "year": 2018,                    # year is used to time so in output it will
  "year": 2020                     be eliminated.
}
print (thisdict)
```

Output



```
{'brand': 'Nokia', 'model': 'Premium', 'year': 2020}
```

2.4.2 Operations on dictionary

There are following operations may be performed on the dictionary:

I. Access the items

We can access the elements from dictionary by using **‘key’ or get ()** method. Both will return the same result.

```
mydict = {
  "brand": "Nokia",
  "model": "premium",
  "year": 2018
}
x = mydict ["model"]        # using key method, here model is key
```

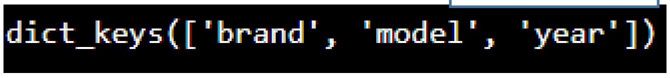
```
x = mydict.get ("model")   # using get method
print (x)
```

II. Finding the keys of dictionary

The **keys ( )** method can be used to list all the keys in the dictionary.

```
mydict = {
 "brand": "Nokia",
 "model": "Premium",
 "year": 2018,
 }
x= mydict. Keys ()
print(x)
```

Output



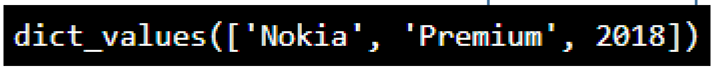
```
dict_keys(['brand', 'model', 'year'])
```

III. Get the values of dictionary

The values () method can be used to list of all the values in the dictionary.

```
mydict = {
 "brand": "Nokia",
 "model": "Premium",
 "year": 2018,
 }
x= mydict. values ()
print(x)
```

Output



```
dict_values(['Nokia', 'Premium', 2018])
```

IV. Change Dictionary Items

We can change the value of a specific item by referring to its key name.

```
mydict = {
 "brand": "Nokia",
 "model": "Premium",
 "year": 2018,                    # change the product year 2018 to 2020
 }
mydict["year"] = 2020
print(mydict)
```

V. Adding Items in the dictionary

We can add the item to the dictionary by using a new index key and assigning a value to it.

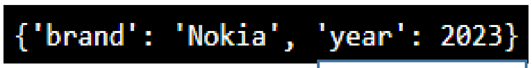
```
mydict = {
 "brand": "Nokia",
 "model": "Premium",
 "year": 2018,
 }
mydict ["price"] = 5000          # adding price in the dictionary
print (mydict)
```

VI. Remove Dictionary Items

Dictionary items can be remove by using various methods. There are following method can be used:

❖ **By using pop ("items") method:** it will remove the specific item by using they item keys.

```
 mydict =        {
"brand": "Nokia",
 "model": "Premium",
 "year": 2023
           }
mydict. Pop("model")          # pop (  ) method use
print (mydict)
```



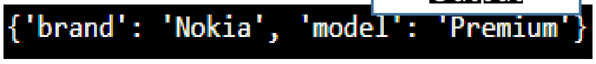
```
{'brand': 'Nokia', 'year': 2023}
```

Output

❖ **By using the popitem ( ):** This method will remove the last inserted item. [In version **3.7** will do similarly while below **3.7 version** it will remove the random item from the dictionary.]

```
mydict =        {
"brand": "Nokia",
"model": "Premium",
"year": 2023
 }
mydict . popitem()
print(mydict)
```

Output

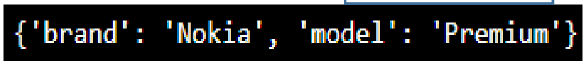


```
{'brand': 'Nokia', 'model': 'Premium'}
```

❖ **By using the del keyword: Del** keyword is removes the item with the specified key name:

```
mydict =        {
"brand": "Nokia",
"model": "Premium",
"year": 2023
        }
del mydict["year"]    # use of del keyword
print(mydict)
```

Output



```
{'brand': 'Nokia', 'model': 'Premium'}
```

VII. **Delete the Dictionary**

By using **Del** keyword we can dele the dictionary completely. Just we need to use the dictionary name without its key.

Syntax: Del dictionary-name

```
  mydict =       {
 "brand": "Nokia",
 "model": "Premium",
 "year": 2023
   }
del   mydict     # use of del keyword for deletion of dictionary.
print (mydict)
```

VIII. **Empty the dictionary: Clear ()** method can be used to empty the dictionary.

```
  thisdict = {
 "brand": "Ford",
 "model": "Mustang",
 "year": 1964
  }
 thisdict.clear ()   # use of clear ( ) method
 print(thisdict)
```

IX. **Empty the dictionary:** We can create a copy by using the **copy ( )** method.

```
 mydict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
   }
mydict1 = mydict.copy()          # use of copy method
print(mydict1)
```

[Note: dict ( ) function also can be used for creating a copy of the dictionary.]

List of key methods in dictionary

Method	**Description**
clear()	Removes all the elements from the dictionary
copy()	Returns a copy of the dictionary
fromkeys()	Returns a dictionary with the specified keys and value
get()	Returns the value of the specified key
items()	Returns a list containing a tuple for each key value pair
keys()	Returns a list containing the dictionary's keys
pop()	Removes the element with the specified key
popitem()	Removes the last inserted key-value pair
setdefault()	Returns the value of the specified key. If the key does not exist: insert the key, with the specified value
update()	Updates the dictionary with the specified key-value pairs
values()	Returns a list of all the values in the dictionary

2. **Conditional statements and loops in Python:**

3.1 if –else

Decision making is important aspect in computer programming. So we have **if ( ) and if - else** statement to making this type of decision on the basis of given condition.

A. if ( ) statement

If ( ) statement is use to perform the execution of block of code according to the given condition. If condition **is true** then block of code will be execute. In case of **false** condition there will be **no** execution of the block of code.

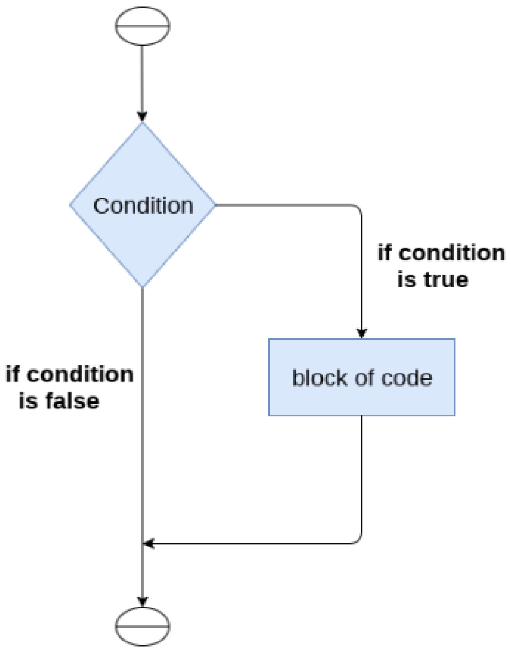
Syntax:

If (**condition statement):**

  Block of code statement

Indentation: Python relies on indentation (Whitespace at the beginning of a line) to define scope in the code.

```
a = 33
b = 200
if  b > a:
print ("b is greater than a")
 # if space is not given you will get an error
```



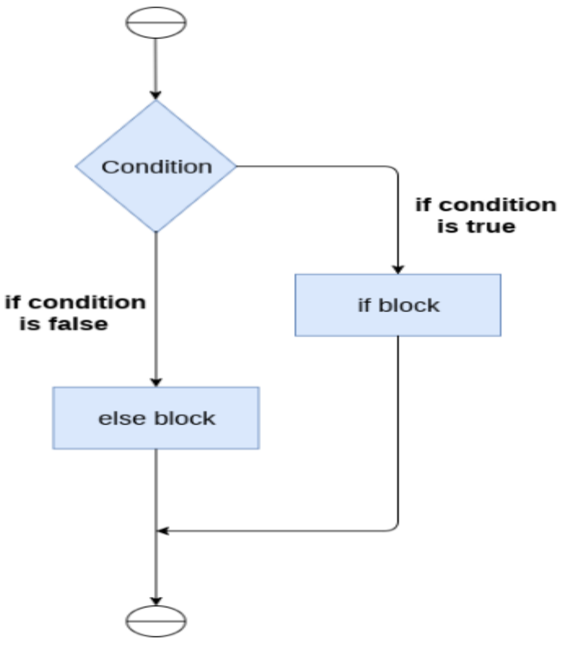
B. if – else statement

If -else statement is use to perform the execution of block of code according to the given condition. If condition **is true** then if-block code will be execute. If condition is **false** then **else-block** code will be executed.

```
If condition:
#block1 of statements
Else:
# block2 of statements (else-block)
```

For Example:

```
a = 33
b = 33
if  b > a:
  print ("b is greater than a")
elif a == b:
 print("a and b are equal")
```

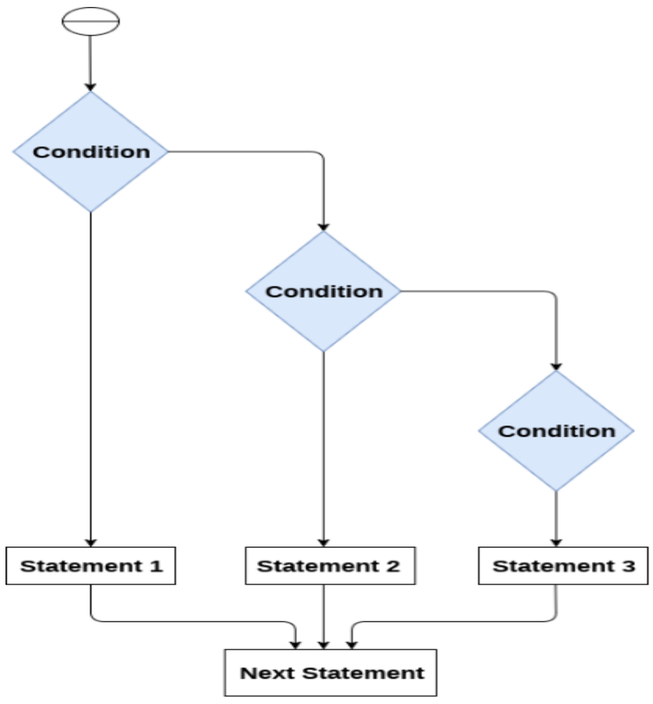


C. Nested If

Nested if is very important in case we have to make the decision on the basis of many given conditions.

Syntax:

```
if expression 1:
  # Block of statements
 elif expression 2:
  # Block of statements
 elif expression 3:
  # Block of statements
 else:
  # Block of statements
```



For Example:

```
 x = 51
 if x > 20:
  print ("Above twenty,")
 if x > 30:
  print ("and also above 30!")
 else:
   print ("but not above 30.")
```

3.2 While loop

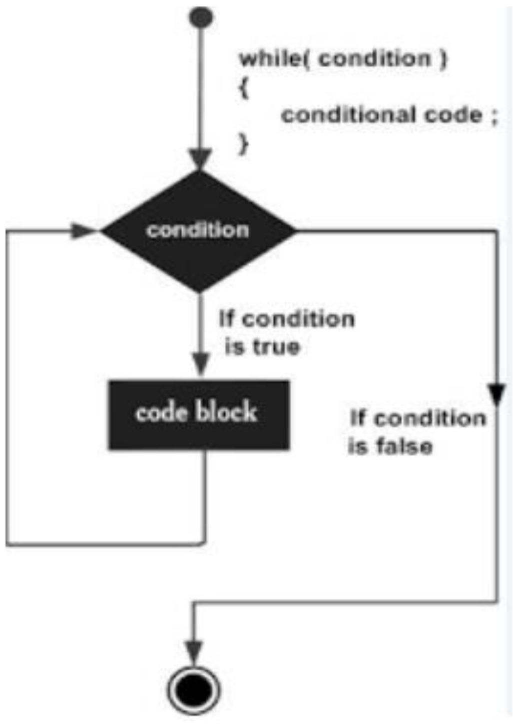
While statement is also known as primitive statement. This statement has the ability to work continuously until the condition is false.

Syntax: **while <condition>:**

{Code block}

For Example:

```
a = 2
while a < 6:
 print (a)
break
a += 1
```



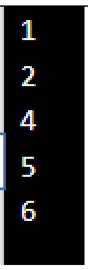
Break statement: The break Statement is used to **Stop** the loop even if the while condition is true:

Continue Statement: The continue statement can stop the current iteration, and continue with the next iteration.

```
i = 0
 while i < 6:
  i += 1
 if  i == 3:
 continue    # continue statement
  print (i)
```

In the output 3 is missing due to continue statement.

Output



```
1
2
4
5
6
```

3.3 For loop

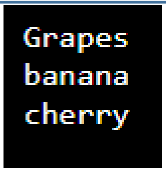
For loop is also used for printing a sequence of items like list, set, dictionary and string.

Syntax: for x in name_of_sequence:

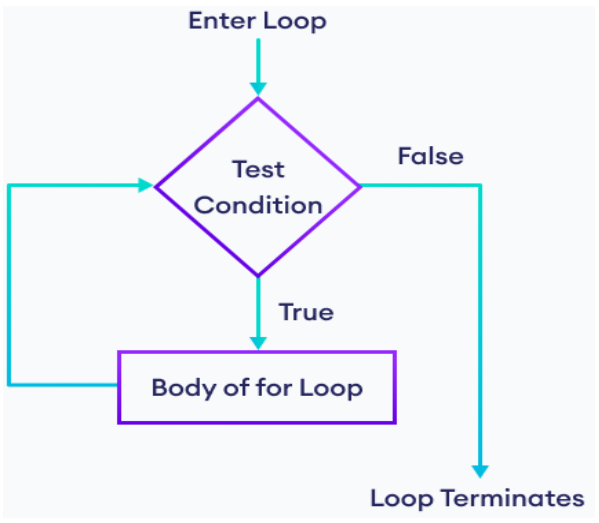
For example:

```
mybasket = ["Grapes", "banana", "cherry"]
for x in mybasket:
  print(x)
```

Output



```
Grapes
banana
cherry
```



3.4 Nested loops

Nested loop is a process in which a loop works inside other loop. This concept is very important when we have multiple condition and according to these condition we have to make execution of the statement.

```
course = ["Car", "Motorcycle", "Bicycle"]
student = ["John", "Shelly", "Mike"]
For x in course:
For y in student:
Print(x, y)
```

Pass Statement: For loops cannot be empty, but due to any reason if you have **for loop** with no content, put in the **pass** statement to avoid getting an error.

For Example

```
For x in [0, 1, 2]:
 Pass            # use of pass statement
```

3. Introduction to Python functions

4.1 Defining functions

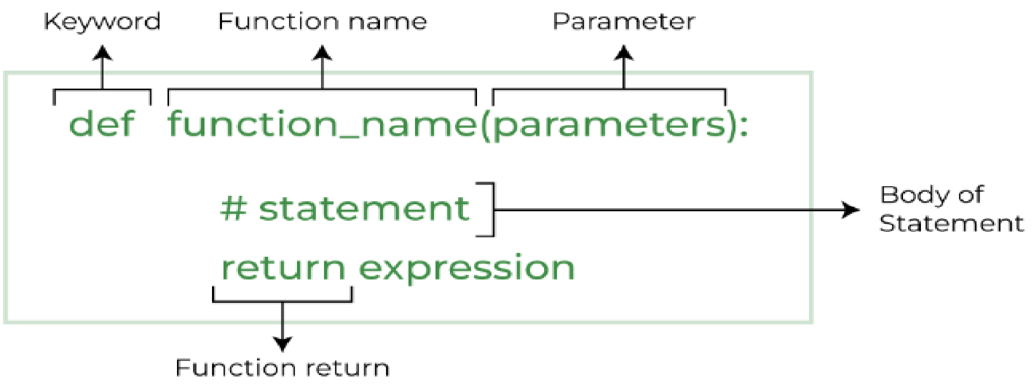
A function is a block of code that runs only when it is called. By using parameter/ arguments we can input data in to a function. A function can return result as a data.

There are two types of functions in python:

A. **Built – in functions:** These functions are the predefined functions that can be use as per the requirement.

B. **User defined functions:** These are the kind of function that's created by the user as per his real time requirement.

- **Creation and declaration of a user defined Function:** we can create a function in python by using the **def** keyword.



```
def   my_function():          # function name is my_function()
  print ("Hello i am here")
   my_function()                  # calling a function
```

- **Function arguments / parameters:**

  By using function argument we can input the data to a functions.

```
def my_function(coursename):
  print (coursename + " BTech")
my_function ("Civil")
my_function ("Electronics")
my_function ("Mechanical")
```

- **Arbitrary Arguments, *args :** This is the key feature in python, when we did not have confirmation about the parameter or how many parameter or arguments will be passes to a function then we can use this feature to declare a function.

```
def   my_function(*data):
    print ("The details is " + data[1])
```

```
my_function ("Email", "Contact_no", "Address")
```

4.2 Function Arguments / Parameters

By using function argument we can input the data to a functions.

```
def my_function(coursename):         # Function Arguments
 print (coursename + " BTech")
my_function ("Civil")
my_function ("Electronics")
my_function ("Mechanical")
```

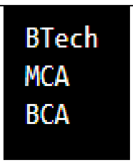
- **Arbitrary Arguments, *args :** This is the key feature in python, when we did not have confirmation about the parameter or how many parameter or arguments will be passes to a function then we can use this feature to declare a function.

```
def   my_function(*data):
     print ("The details is " + data[1])
     my_function ("Email", "Contact_no", "Address")
```

- **Passing a List as an Argument:** We can send any data types of argument to a function like **string, number, list, dictionary etc.** It will be treated as the same data type inside the function.

```
def    my_function(course):
   for x in course:
    print(x)
    list_course = ["BTech", "MCA", "BCA"]
    my_function (list_course)
```

Output



```
BTech
MCA
BCA
```

4.3 Return statement

A return statement is written to exit the function and return the calculated value. A declared function will return an empty string if no return statement is written.

```
Def my_function(x):
   return 5 * x                # Use of return statement
   print (my_function(3))
```

- **Pass Statement:** We cannot put function definition empty. But in exception case if we want to put function definition with no content then we can use **pass** statement. It will avoid the error.

```
def myfunction():
 pass
```

- **Anonymous Functions in Python**

  An anonymous function in Python is one that has no name when it is defined. We use the lambda keyword to define the anonymous functions

  # using lambda function

  ```
  cube_v2 = lambda x : x*x*x
  print (cube_v2(7))
  ```

4.4 Functions vs Methods

Functions in Python	Methods in Python
Functions are created outside a class.	Methods are created inside a class.
Functions are not linked to anything.	Methods are linked with the classes they are created in.
Functions can be executed just by calling with its name.	To execute methods, we need to use either an object name or class name and a dot operator.
Functions can have zero parameters.	Methods should have a default parameter either self or cls to get the objects or class's address.
Functions cannot access or modify class attributes.	Methods can access and modify class attributes.
Functions are independent of classes.	Methods are dependent on classes.

For Example:

```
Class vegetables:
         def __init__(self, name):   # method1
                   self.name = name
                   self.veggies_eaten = 0
        def eat_one(self):                        # Method2
                   self.veggies_eaten += 1
        return "Eaten one {self.name}"
                   v1 = vegetables("Radish")
                   v2 = vegetables("Beetroot")
        def eat_many(vegetable, count):   # Function
          for i in range(count):
             vegetable.eat_one()
                   eat_many(v1, 20)
                   eat_many(v2, 1)
                   print(v1.veggies_eaten)
                   print(v2.veggies_eaten)
```

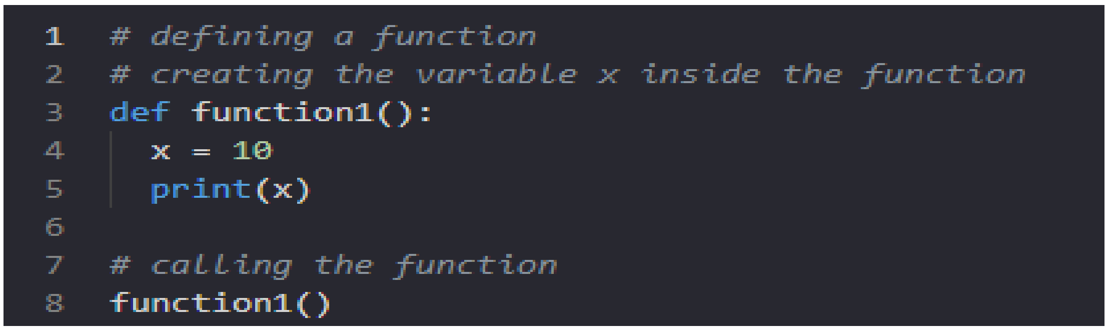
4.5 Scope of variables

The scope of a variable is the location where we can declare a variable and access it if necessary. A variable can be found in the location where it was created, which is its scope.

Types of scope: In Python, there are two types of scope:

A.Local scope/local variables

A local scope or local variables are the variable which is created inside a function. This variable belongs to a local scope because it is only available inside the function it was created. The following example describe how to use local variable/ local scope:

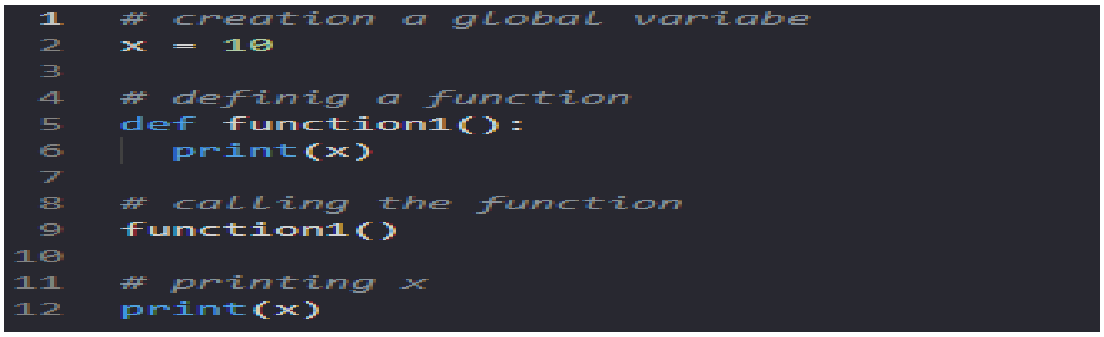


```python
# defining a function
# creating the variable x inside the function
def function1():
  x = 10
  print(x)

# calling the function
function1()
```

B.Global scope/global variable

A global scope/global variable is the variable which is created outside of a function. The function is created in the main body of the Python code, so it is called a global scope and the variable is referred to as a global variable.



```python
# creation a global variabe
x = 10

# definig a function
def function1():
  print(x)

# calling the function
function1()

# printing x
print(x)
```

4. Introduction to Data Manipulation

Data manipulation is the process of altering or transforming data to extract, organize, modify, or analyze it in order to derive meaningful insights or meet specific requirements. We can perform various operations on the data like as filtering, sorting, aggregating, joining, merging, and transforming etc. Data manipulation is commonly performed in data analysis, database management, and programming contexts. There are following common techniques and operations involved in data manipulation:

A. **Filtering**: Selecting a subset of data from the database based on specified required criteria. For example, selecting all sales records where the purchase amount is greater than a certain value.

B. **Sorting:** Arranging the data in a specific order based on one or more attributes. For instance, sorting a list of customer names in alphabetical order.

C. **Aggregating**: Combining multiple data records to create summary statistics or metrics. This can include operations like calculating the total sales, average price, or maximum value.

D. **Joining/Merging**: Combining data from multiple sources based on a common attribute or key. This is often used to consolidate information from different tables or datasets.

E. **Transforming:** Modifying the structure or format of data. This can involve operations like converting data types, renaming columns, or splitting and combining values.

F. **Cleaning:** Correcting or removing errors, inconsistencies, or missing values in the data. This step ensures data quality and reliability.

G. **Grouping:** Grouping data based on specific criteria to analyze subsets of data. For instance, grouping sales data by region or product category.

H. **Summarizing:** Generating descriptive statistics or summarizing data based on certain criteria. This includes operations like calculating the mean, median, or standard deviation.

Data manipulation can be performed using various tools and technologies, like spreadsheet software (e.g., Microsoft Excel, Google Sheets), programming languages (e.g., Python, R, SQL), data manipulation libraries (e.g., pandas in Python), and database management systems (e.g., MySQL, PostgreSQL).

There are following 4 queries that can be used for the data manipulation in SQL:

- **SELECT** – to query data in the database.
- I**NSERT** – to insert data into a table.
- **UPDATE** – to update data in a table.
- **DELETE** – to delete data from a table.

5.1 Introduction to NumPy

NumPy stands for the Numerical python. It is a python package for the computation and processing of the multidimensional and single dimensional array elements. **Travis Oliphant** created NumPy package in 2005.

- It is an open source project and you can use it freely. NumPy offers various powerful data structures, implementing multi-dimensional arrays and matrices.
- NumPy is written partially in Python, but most of the parts that require fast computation are written in C or C++.
- NumPy is capable of performing Fourier Transform and reshaping the data stored in multidimensional arrays.
- NumPy provides a convenient and efficient way to handle the vast amount of data.

5.2 NumPy Package in Python

NumPy doesn't come bundled with Python. We have to install it using the **python pip installer.**

```
$ pip install NumPy
```

After the installation of the NumPy Package we can check whether package is installed or not y using the **import** keyword:

```
 Import  numpy          # use of NumPy library
arr = numpy.array([1, 2, 3, 4, 5])
print(arr)
```

- **Checking NumPy Version:** The version string is stored under __version__ attribute.

  ```
  import numpy as np
  print (np.__version__)
  ```

5.3 Importing NumPy

NumPy is imported under the **np** alias. We can use the following **syntax:**

```
Import NumPy as np            # np means NumPy package
```

For Example:

```
import  numpy as np       # import NumPy library
arr = np.array([10, 20, 30, 40, 50])
print (arr)
print (type(arr))
```

5.4 Creating different arrays using NumPy

We can create different kinds of array as mentioned below:

- **0-D Array:** It is a special type of array that contains single elements in an array.

  ```
  import numpy as np
  ```

arr = np.array (42)   # **Zero – D Array**

print (arr)

- **1-D Array**

  1-D arrays in numpy are one dimension that can be thought of a list where you can access the elements with the help of indexing.

  Syntax: array_name = np. Array ([item1, item2, item3...])

  For Example:

  import numpy as np

  arr1 = np. array ([10, 20, 30, 40, 50])

  print (arr1)

- **2-D Array**

  2-D arrays in numpy are two dimensions array that can be distinguished based on the number of square brackets used. NumPy has a whole sub module dedicated towards matrix operations called (**numpy.mat).**

  Syntax: array_name = np. Array ([item1, item2, item3],[elem1,elem2,elem3])

  import numpy as np

  my_arr = np.array ([[25, 22, 29], [14, 15, 16]])   # 2-D Array

  print (my_arr)

- **3-D Array**

  The 3-D arrays in numpy are the three-dimension array that can have three square brackets.

  Syntax: my_arr = np.array([[[iem1, item2, item3], [obj1, obj2, obj3]], [[elem1, elem2, elem3]]])

  import numpy as np

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

  print (arr)

- **Multi-Dimensional Array**

  We can create multidimensional array by using the **ndim** argument.

  Syntax: arr-name = np. array ([1, 2, 3, 4], ndmin = value)

  For Example:

  import numpy as np

  arr = np.array([1, 2, 3, 4], ndmin=5)

  print(arr)

  print('number of dimensions :', arr.ndim)

5.5 Array functions and Methods

There is a vast range of built-in operations that we can perform on these arrays.

1. Ndim – It returns the dimensions of the array.
2. Itemsize – It calculates the byte size of each element.
3. Dtype – It can determine the data type of the element.
4. Reshape – It provides a new view.
5. Slicing – It extracts a particular set of elements.
6. Linspace – Returns evenly spaced elements.
7. max/min, sum, sqrt
8. Ravel – It converts the array into a single line.

5.6 Different Mathematical Functions

NumPy contains a large number of various mathematical operations. NumPy provides standard trigonometric functions, functions for arithmetic operations, handling complex numbers, etc.

A. **Trigonometric Functions –** NumPy has standard trigonometric functions which return trigonometric ratios for a given angle in radians.

FUNCTION	**DESCRIPTION**
Tan ()	Compute tangent element-wise.
Arcsin ()	Inverse sine, element-wise.
Arccos ()	Trigonometric inverse cosine, element-wise.
Arctan ()	Trigonometric inverse tangent, element-wise.
Degrees ()	Convert angles from radians to degrees.
Radians ()	Convert angles from degrees to radians.

5.7 Different Matrix Operations

Python has powerful features that offers matrix features. The matrix can be implemented by using list or array in the python.

There are various operations can be performed on the matrix as below:

1. Add ():- This function is used to perform element wise matrix addition.

2. **Subtract ():-** This function is used to perform element wise matrix subtraction.

3. **Divide ():-** This function is used to perform element wise matrix division.

4. **Multiply ():-** This function is used to perform element wise matrix multiplication.

5. Dot ():- This function is used to compute the matrix multiplication, rather than element wise multiplication.

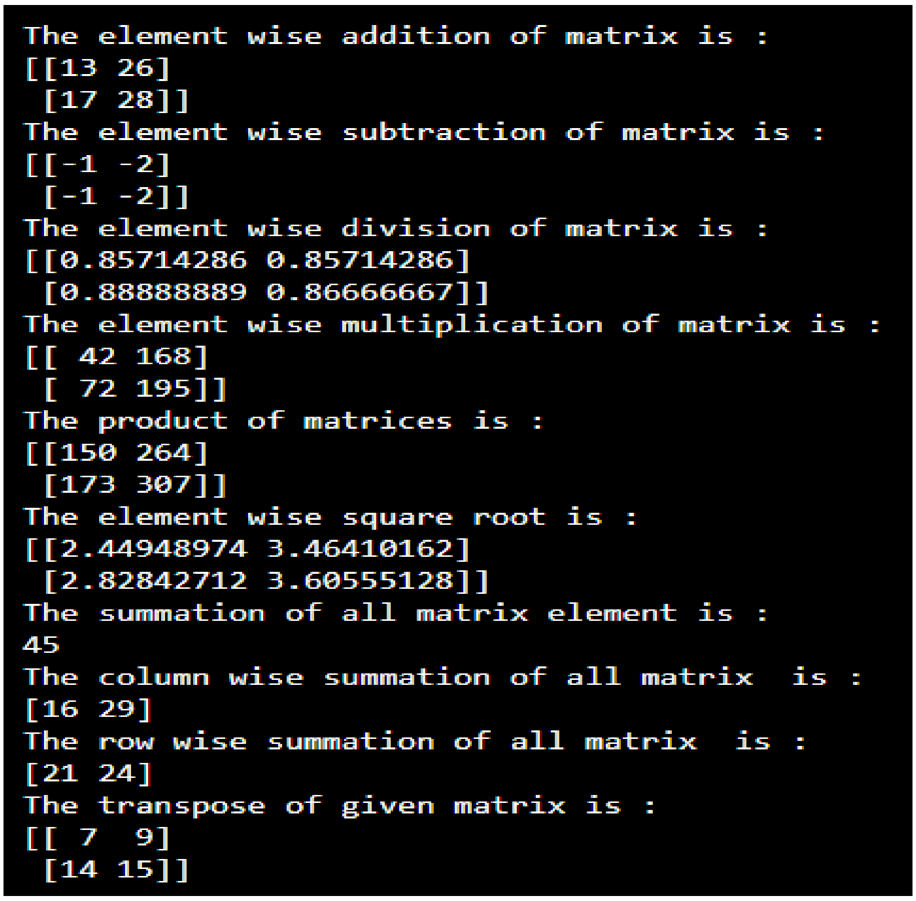
6. Sqrt ():- This function is used to compute the square root of each element of matrix.

7. Sum(x, axis):- This function is used to add all the elements in matrix. Optional "axis" argument computes the column sum if axis is 0 and row sum if axis is 1.

8. **"T" (Transpose):-** This argument is used to transpose the specified matrix.

```
Import NumPy
x = numpy.array([[6, 12], [8, 13]])  # initializing matrices
y = numpy.array([[7, 14], [9, 15]])
print ("The element wise addition of matrix is : ")
print (numpy.add(x,y))          # using add() to add matrices
print ("The element wise subtraction of matrix is : ")
print (numpy.subtract(x,y))     # using subtract() to subtract matrices
print ("The element wise division of matrix is : ")
print (numpy.divide(x,y))       # using divide() to divide matrices
print ("The element wise multiplication of matrix is : ")
print (numpy.multiply(x,y))     # using multiply() to multiply matrices element wise
print ("The product of matrices is : ")
print (numpy.dot(x,y))          # using dot() to multiply matrices
print ("The element wise square root is : ")
print (numpy.sqrt(x))           # using sqrt() to print the square root of matrix
print ("The summation of all matrix element is : ")
print (numpy.sum(y))  #using sum() to print summation of all elements of matrix
 print ("The column wise summation of all matrix  is : ")
print (numpy.sum(y,axis=0)) # using sum(axis=0) Sum of all columns of matrix
print ("The row wise summation of all matrix  is : ")
print (numpy.sum(y,axis=1))  # using sum(axis=1) Sum of all columns of matrix
print ("The transpose of given matrix is : ")
print (y.T)        # using "T" to transpose the matrix
```

Output:



```
The element wise addition of matrix is :
[[13 26]
 [17 28]]
The element wise subtraction of matrix is :
[[-1 -2]
 [-1 -2]]
The element wise division of matrix is :
[[0.85714286 0.85714286]
 [0.88888889 0.86666667]]
The element wise multiplication of matrix is :
[[ 42 168]
 [ 72 195]]
The product of matrices is :
[[150 264]
 [173 307]]
The element wise square root is :
[[2.44948974 3.46410162]
 [2.82842712 3.60555128]]
The summation of all matrix element is :
45
The column wise summation of all matrix  is :
[16 29]
The row wise summation of all matrix  is :
[21 24]
The transpose of given matrix is :
[[ 7  9]
 [14 15]]
```

5.8 Random Numbers

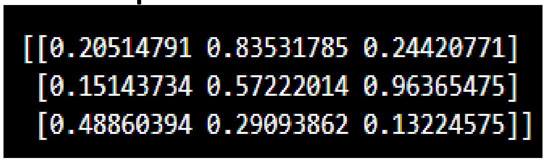
The random is a module present in the NumPy library. This module contains the functions which are used for generating random numbers. This module contains some simple random data generation methods, permutation & distribution functions, and random generator functions.

A. Simple random data

This function of random module is used to generate random numbers or values in a given range.

```
import   numpy as np
a=  np.random.rand(3,3)
print(a)
```

Output:



```
[[0.20514791 0.83531785 0.24420771]
 [0.15143734 0.57222014 0.96365475]
 [0.48860394 0.29093862 0.13224575]]
```

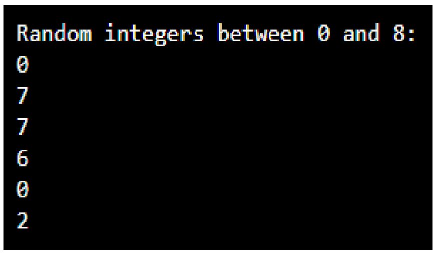
5.9 Generate Numbers between a range

Python provides a function named randrange() in the random package that can produce random numbers from a given range . There are various method that can be used for generating random numbers between a ranges:

Method 1: Generate random integers using random.randrange() method

```
import random
print("Random integers between 0 and 8: ")
for i in range(7, 13):
    y = random.randrange(8)    # use of randrange method
    print(y)
```

Output:

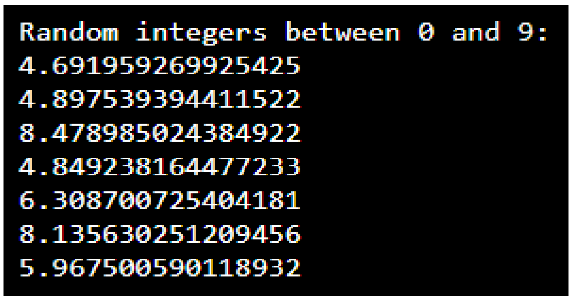


```
Random integers between 0 and 8:
0
7
7
6
0
2
```

Method 2: Generate random integers using random.uniform() method

The method, "random.uniform()" is defined in the "random" module. It Returns the generated floating-point random number between the lower limit and upper limit.

```
import random
 print("Random integers between 0 and 9: ")
for i in range(4, 11):
    y = random.uniform(4, 10)
    print(y)
```

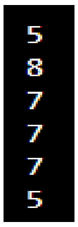


```
Random integers between 0 and 9:
4.691959269925425
4.897539394411522
8.478985024384922
4.849238164477233
6.308700725404181
8.135630251209456
5.967500590118932
```

Method 3: Generate random integers using randbelow() method

This method is used for handling crucial information including cryptographically secure passwords, account authentication, security tokens, and related secrets, the secrets module is utilized to generate random integers. We can use randbelow() function from the **secrets module** to generate random integers.

```
from secrets import randbelow
 for _ in range(3, 9):
     print(randbelow(10))  # randbelow method
```



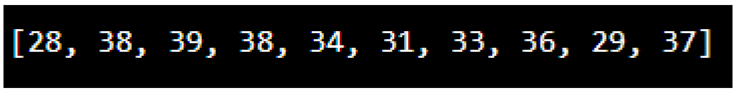
```
5
8
7
7
7
5
```

Method 4: Generate random integers using the random.randint() method

Python provides a random module to generate random numbers. To generate random numbers we have used the random function along with the use of the (random.randint) function. randint accepts two parameters, a **starting point**, and an **ending point**. Both should be integers and the first value should always be less than the second.

```
import numpy as np
def Rand(start, end, num):
    res = []
     for j in range(num):
        res.append(np.random.randint(start, end))
    return res
num = 10
start = 20
end = 40
print(Rand(start, end, num))
```

<u>**Output**</u>



```
[28, 38, 39, 38, 34, 31, 33, 36, 29, 37]
```