Unit-2:

ARRAY, FUNCTION , LIST,TUPLES,DICTIONARY

Arrays

- An array is an object that stores **a group of elements (or values) of same datatype.**
- The main advantage of any array is to store and process a group of elements easily. There are two points we should remember in case of arrays in Python.
- **Arrays can store only one type of data**. It means, we can store only integer type elements or only float type elements into an array. But we cannot store one integer, one float and one character type element into the same array.
- **Arrays can increase or decrease their size dynamically**. It means, we need not declare the size of the array. When the elements are added, it will increase its size and when the elements are removed, it will automatically decrease its size in memory.

Advantages of Arrays

- The following are some advantages of arrays:
- Arrays are similar to lists. The main difference is that arrays can store only one type of elements; whereas, lists can store different types of elements. When dealing with a huge number of elements, arrays use less memory than lists and they offer faster execution than lists.
- The size of the array is not fixed in Python. Hence, we need not specify how many elements we are going to store into an array in the beginning.
- Arrays can grow or shrink in memory dynamically (during runtime).
- Arrays are useful to handle a collection of elements like a group of numbers or characters.
- Methods that are useful to process the elements of any array are available in 'array' module.

Creating an Array

- The type should be specified by using a **type code** at the time of creating the array object as:

```
arrayname = array(type code, [elements])
```

- The type code 'i', represents integer type array where we can store integer numbers.
- If the type code is 'f' then it represents float type array where we can store numbers with decimal point.

- **Example (Integer Array):**

  a = array('i', [4, 6, 2, 9]) creates an integer array named 'a'.
- **Example (Float Array):**

  arr = array('d', [1.5, -2.2, 3, 5.75]) creates a float array named 'arr'.

The Type Codes to Create Arrays

Typecode	C Type	Minimum size in bytes
‘b’	Signed integer	1
‘B’	Unsigned integer	1
‘i’	Signed integer	2
‘I’	Unsigned integer	2
‘l’	Signed integer	4
‘L’	Unsigned integer	4
‘f’	Floating point	4
‘d’	Double precision floating point	8
‘u’	Unicode character	2

Importing the Array Module

- There are three ways to import the array module into our program.
- The first way is import the entire array module using import statement as,

  import array

- When we import the array module, we are able to get the 'array' class of that module the helps us to create an array. See the following example:

  a = array.array('i', [4,6,2,9])

- Here, the first 'array' represents the module name and the next 'array' represents the class name for which the object is created. We should understand that we are creating our array as an object of array class.

- The second way of importing the array module is to give it an alias name, as:

  import array as ar

- Here, the array is imported with an alternate name 'ar'. Hence we can refer to the array class of 'ar' module as:

  a = ar.array('i', [4,6,2,9])

- The third way of importing the array module is to write:

  from array import *
- Observe the symbol that represents 'all'.
- The meaning of this statement is this: import all (classes, objects, variables etc) from the array module into our program. That means we are specifically importing the 'array' class (because of symbol) of 'array' module. So, there is no need to mention the module name before our array name while creating it.
- We can create the array as:

  a=array('i', [4,6,2,9])
- Here, the name 'array' represents the class name that is available from the 'array' module

example:

```
from array import *
arr = array('u',['a','b','c','d','e'])

print("the array elements are:")
for ch in arr:
    print(ch)
```

Indexing and Slicing on Arrays

- An ***index*** represents the position number of an element in an array.
- For example, when we create the following integer type array:

  x = array('i', [10, 20, 30, 40, 50])
- Python interpreter allocates 5 blocks of memory, each of 2 bytes size and stores the elements 10, 20, 30, 40 and 50 in these blocks.
- example:

10	20	30	40	50
X[0]	**X[1]**	**X[2]**	**X[3]**	**X[4]**

x = array('i', [10, 20, 30, 40, 50])

```
# Accessing elements by positive index
print(x[0]) # Output: 10
print(x[3]) # Output: 40

# Accessing elements by negative index
print(x[-1]) # Output: 50
print(x[-3]) # Output: 30
```

- A ***slice*** represents a piece of the array.
- When we perform 'slicing' operations on any array we can retrieve a piece of the array that contains a group of elements. Whereas **indexing** is useful to retrieve element by element from the array, **slicing** is useful to retrieve a range of elements.
- The general format of a slice is:

  arrayname [start:stop:step]
- We can eliminate any one or any two in the items: 'start', 'stop' or ‘step' from the above syntax.
- For example, **arr[1:4]**
- The above slice gives elements starting from 1st to 3rd from the array 'arr'. Counting of the elements starts from 0.
- All the items 'start', 'stop' and 'step' represent integer numbers either positive or negative.
- The item 'step' represents step size excluding the starting element.

```
x=array('I',[10,20,30,40,50,60,70])
# create array y with elements from 1st  to 3rd  from x
      y = x[1:4]
      print(y)    #array('i', [20, 30, 40])

#create array y with elements from 0th  till the last element in x
      y = x[0:]
      print(y)  # array('i', [10, 20, 30, 40, 50,60,70])

#create array y with elements from 0th  to 3rd  from x
      y = x[:4]
      print(y)  #array('i', [10, 20, 30, 40])

# create array y with last 4 elements from x
      y = x[-4:]
      print(y)    #array('i', [40, 50,60,70])

#create y with 0th  to 7th  elements from x.
#retrieve every 2nd  element from x
      y = x[0:7:2]
      print(y)   #array('i', [10, 30, 50, 70])
```

Processing the Arrays

- arrays class of arrays module in Python offers methods to process the arrays easily. The Programmers can easily perform certain **operations** by using these methods.
- Methods are generally called as: objectname.method().

Method	Description
a.append(x)	Adds an element x at the end of the existing array a.
a.count(x)	Returns the numbers of occurrences of x in the array a.
a.extend(x)	Appends x at the end of the array a. 'x' can be another array or an iterable object.
a.index(x)	Returns the position number of the first occurrence of x in the array Raises ValueError' if not found.
a.insert(i, x)	Inserts x in the position i in the array.
a.pop()	Remove last item from the array a.
a.remove(x)	Removes the first occurrence of x in the array a.
a.tolist()	Converts the array ‘a’ into a list.

Understand various methods of arrays class.

```
#operations on arrays
from array import*
# create an array with int values
arr = array('i', [10,20,30,40,50])
print('Original array:’,arr)       #Original array('i', [10, 20, 30, 40, 50])

# append 30 to the array arr
arr.append(30)
arr.append(60)
print('After appending 30 and 60:', arr)    #After appending 30 and 60: array('', [10, 20, 30, 40, 50, 30, 60])

# insert 99 at position number 1 in arr
arr.insert(1,99)
print('After inserting 99 in 1st position:', arr)   #After inserting 99 in 1st position: array('i', [10, 99, 20, 30, 40, 50, 30, 60])
```

```
#remove an element from arr
arr.remove(20)
print('After removing 20: ', arr)   #After removing 20: array('i', [10, 99, 30,40,50,30, 60])

# remove last element using pop() method
n=arr.pop()
print('Array after using pop(): ', arr)
print('Popped element:',n)  #Array after using pop(): array('i', [10, 99, 30, 40, 50, 30])
Popped element: 60

# finding position of element using index() method
n = arr.index (30)
print('First occurrence of element 30 is at: ', n)  # First occurrence of element 30 is at: 2

#convert an array into a list using to list() method
Lst=arr.tolist()
print('List:', lst)
print('Array:' arr)   #List: [10, 99, 30, 40, 50, 30]
                        #Array: array('i', [10, 99, 30, 40, 50, 30])
```

Functions

- A function is a group of statements designed to perform a specific task, similar to a mini-program within a larger program.
- Programmers use multiple functions to handle various tasks.
- There are several tasks to be performed, the programmer will write several functions.
- There are several **built-in' functions** in Python to perform various tasks.
- For example, to display output, Python has print() function, to calculate square root value, there is sqrt() function and to calculate power value, there is power() function.
- Similar to these functions, a programmer can also create his own functions which are called **'user-defined'** functions.

- The following are the advantages of functions:
- Functions are important in programming because they are **used to process data, make calculations or perform any task** which is required in the software development.
- functions are also called **reusable code**. Because of this reusability, the programmer can avoid code redundancy. It means it is possible to avoid writing the same code again and again.
- 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**. Modular programming makes programming easy.
- **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. Similarly, when a particular feature is no more needed by the user, the corresponding function can be deleted or, put into comments.
- When there is an error in the software, the corresponding function can be modified without disturbing the other functions in the software. Thus code debugging will become easy.
- The use of functions in a program will **reduce the length** of the program.

Difference between a Function and a Method

- **A function is called using its name**.
- When a function is **written inside a class, it becomes a 'method'**.
- A method is called using one of the following ways:

  objectname.methodname()

  Classname.methodname()
- So, please remember that a function and a method are same except their placement and the way they are called.

```
# User-defined function
def Subtract (a, b):
    return (a-b)

print( Subtract(10, 12) ) # prints -2

print( Subtract(15, 6) ) # prints 9
```

```
# Python 3  User-Defined  Method
class ABC :
    def method_abc (self):
        print("I am in method_abc of ABC
class. ")

class_ref = ABC() # object of ABC class
class_ref.method_abc()
```

Defining a Function

- We can define a function using the keyword ***def*** followed by function name.
- After the function name, we should write parentheses () which may contain parameters.

function definition

```
def  functionname( para1, para2,...):
     """ function docstring """“
     function statements
```

example

```
def sum(a, b):
     “””This function finds sum of two no.”””
     c=a+b
     print(c)
```

Calling a Function

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

  sum(10, 15)

- The values passed to a function are called 'arguments'. So, 10 and 15 are arguments.
- A function that accepts two values and finds their sum.

Example:

```
#a function to add two numbers
def sum(a, b):
    '''This function finds sum of two numbers'''
    c = a+b
    print('Sum=', c)

#call the function
sum(10, 15)
sum(1.5, 10.75) # call second time
```

Output:

Sum=25

Sum= 12.25

- During run time, 'a' and 'b' may assume any type of data, may be int or may be float or may be strings. This is called **'dynamic typing'.**
- In dynamic typing, the type of data is determined only during runtime, not at compile time.
- Dynamic typing is one of the features of Python that is not available in languages like C or Java.

Returning Results from a Function

- We can return the result or output from the function using a 'return' statement in the body of the function.
- For example,

  return c # returns c value out of function

  return 100 #returns 100

  return x, y, c   # returns 3 values
- When a function does not return any result, we need not write the return statement in the body of the function.

```
# a function to add two numbers
def sum(a, b):
    """This function finds sum of two numbers """
    c = a+b
    return c #return result
# call the function
x = sum(10, 15)
print('The sum is', x)
y = sum(1.5, 10.75)
print('The sum is: ', y)
```

Output:

The sum is: 25

The sum is: 12.25

Returning Multiple Values from a Function

- In Python, a function can return multiple values. When a function calculates multiple results and wants to return the results, we can use the return statement as:

  return a, b, c

- To grab these values, we can use three variables at the time of calling the function as:

  x, y, z = function()

```
def sum_sub(a, b):
        c = a + b
        d = a - b
        return c, d
x, y = sum_sub(10, 5)
print("Result of addition: ", x)
print("Result of subtraction: y)
```

Output:

```
C:\>python fun.py
Result of addition: 15
Result of subtraction: 5
```

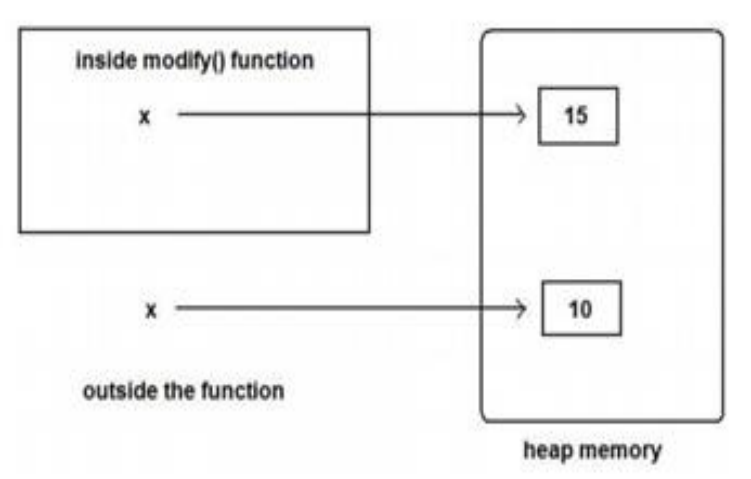
Pass by Object Reference

- We pass values to a function in two ways:
  - Pass by value or call by value
  - Pass by reference or call by reference
- 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.
- 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.
- In Python, the values are sent to functions **by means of object references**. We know everything is considered as an object in Python.
- **Mutable vs. Immutable Objects:**
- **Mutable Objects (e.g., lists, dictionaries, sets):** If you modify the contents of a mutable object within a function (e.g., appending to a list, changing a dictionary value), these changes will be reflected outside the function because both the original variable and the function parameter are pointing to the same object.
- **Immutable Objects (e.g., integers, strings, tuples):** If you try to "modify" an immutable object within a function, you are actually creating a new object and reassigning the function parameter to point to this new object. The original variable in the caller's scope will still point to the original, unchanged object.

pass by object value

Example:

```
def modify(x):
        """ reassign a value to the
        variable"""
        x=15
        print(x,id(x))
x=10
modify(x)
print(x,id(x))
```



output:

```
15   16178459
10  16178452
```

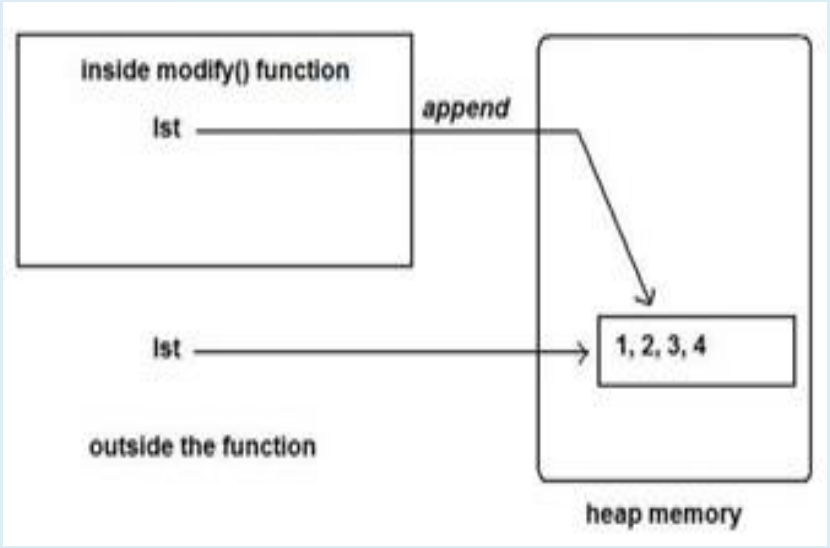
pass by object reference

Example:

```
def modify(lst):
        lst.append(9)
        print(lst, id(lst))
lst = [1,2,3,4]
modify(lst)
print(lst, id(lst))
```

Output:

[1,2,3,4,9] 37776258
[1,2,3,4,9] 37776258

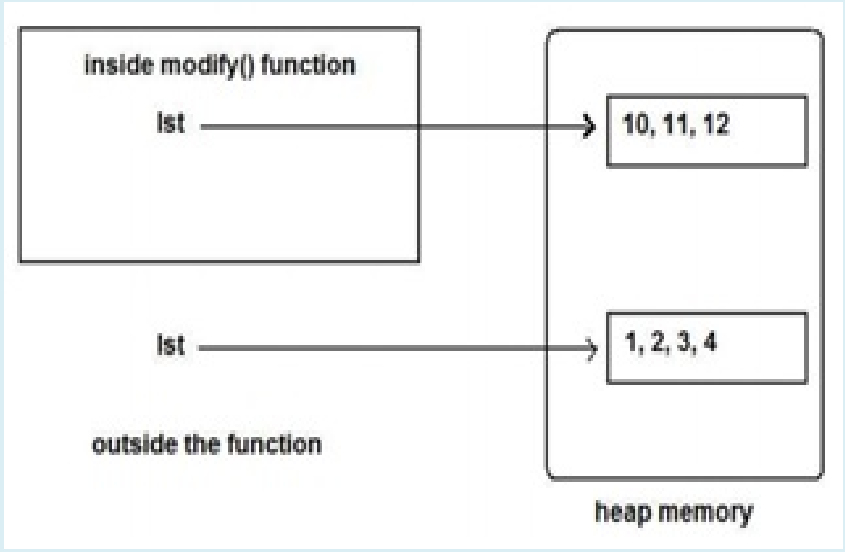


A Python program to create a new object inside the function does not modify outside object.

```
def modify(lst):
        lst = [10, 11, 12] #to create a new list
        print(lst, id(lst))
lst = [1,2,3,4]
modify(lst)
print(lst, id(lst))
```

Output:

[10,11,12] 29525989
[1,2,3,4] 29507548



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.

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

- **The actual arguments used in a function call are of 4 types:**
  - Positional arguments
  - Keyword arguments
  - Default arguments
  - Variable length arguments

1) **Positional Arguments**

- These are the arguments passed to a function **in correct positional order**. Here, the number of arguments and their positions in the function definition should match exactly with the number and position of the argument in the function call.
- For example, take a function definition with two arguments as:

```
def attach(s1, s2)
attach('New', 'York')
```

- Also, if we try to pass more than or less than 2 strings, there will be an error. For example, if we call the function by passing 3 strings as:

```
attach('New', 'York', 'City') #results error
```

- **A Python program to understand the positional arguments of a function.**

```
def attach(s1, s2):
    s3 = s1+s2
    print('Total string: '+s3)
attach('New', 'York')
```

Output : NewYork

2) 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.
- We can change the order of the arguments as:

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

- **Python program to understand the keyword arguments of a function.**

```
def grocery(item, price):
    print('Item = %s'% item)
    print('Price = %.2f'% price)
grocery(item='Sugar', price=50.75)
grocery(price=88.00, item='Oil')
```

Output:

```
Item =sugar
Price= 50.75
Item= oil
Price = 88.00
```

3) 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):

- The second argument is 'price' and its default value is mentioned to be 40.00.
- A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument.

- **A Python program to understand the use of default arguments in a function.**

```
def grocery(item, price=40.00):
    print('Item = %s'% item)
    print('Price = %.2f'% price)
grocery(item='Sugar', price=50.75)
grocery(item='Sugar')
```

Output:

```
Item = Sugar
Price = 50.75
Item= Sugar
Price =40.00
```

4) Variable Length Arguments

- If the programmer wants 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.
- **A Python program to show variable length argument and its use.**

```
#variable length argument demo
def add(farg, *args):
    print('Formal argument=', farg)
    sum=0
    for i in args:
        sum+=i
    print('Sum of all numbers= ',(farg+sum))
add(5, 10)
add(5, 10, 20, 30)
```

Output:

```
Formal argument =5
Sum of all numbers = 15
Formal argument =5
Sum of all argument =65
```

- **A keyword variable length argument** is an argument that can accept any number of values provided in the **format of keys and values.**
- If we want to use a keyword variable length argument, we can declare it with ' ** ' before the argument as:

  def display(kwargs):**
- Here '**kwargs' is called keyword variable argument. This argument **internally represents a dictionary object.** A dictionary stores data in the form of key and value pairs. It means, when we provide values for '**kwargs', we can pass multiple pairs of values using keywords as:

  display(rno=10)
- Here, 5 is stored into 'farg' which is formal argument. 'rno' is stored as key and its value '10' is stored as value in the dictionary that is referenced by 'kwargs'.

A Python program to understand keyword variable argument.

```
def display_info(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")


display_info(name="Alice", age=30,
city="New York")
```

Output:

name: Alice

age: 30

city: New York

Anonymous Functions

- **A function without a name is called 'anonymous function'** . So far, the functions we wrote were defined using the keyword 'def'.But anonymous functions are not defined using 'def'.
- They are **defined using the keyword lambda** and hence they are also called **'Lambda functions'** . Let's take a normal function that returns square of a given value.

  def square(x):

  return x*x
- The same function can be written as anonymous function as:

  lambda x: x*x
- The format of lambda functions is:

  lambda argument_list: expression
- Normally, if a function returns some value, we assign that value to a variable as:

  y = square(5)
- But, lambda functions return a function and hence they should be assigned to a function as:

  f = lambda x: x*x
- Here, 'f' is the function name to which the lambda expression is assigned. Now, if we call the function f() as:

  value = f(5)
- Now, 'value' contains the square value of 5, i.e. 25

A Python program to create a lambda function that returns a square value of a given number.

```
f = lambda x: x*x
value = f(5)
print('Square of 5 =', value)
```

Add two numbers using Lambda.

```
f = lambda x, y: x+y
result = f(1.55, 10)
print('Sum =', result)
```

- Lambda functions contain only one expression and they return the result implicitly. Hence we should not write any 'return' statement in the lambda functions.

Lambda with filter() Function

- The filter() function is useful **to filter out the elements of a sequence** depending on the result of a function. We should supply a function and a sequence to the filter() function as:

  filter(function, sequence)

- Here, the 'function' represents a function name that may return either True or False; and 'sequence' represents a list, string or tuple.
- The 'function' is applied to every element of the 'sequence' and when the function returns True, the element is extracted otherwise it is ignored. Before using the filter() function, let's first write a function that tests whether a given number is even or odd.

```
def is_even(x):
    if x%2==0:
        return True
    else:
        return False
```

- Now, we can use this function inside filters() to test the elements of a list 'lst' as: filter(is_even, lst)
- A Python program using filter() to filter out even numbers from a list.

```
def is_even(x):
    if x%2==0:
        return True
    else:
        return False
lst = [10, 23, 45, 46, 70, 99]
lst1 = list(filter(is_even, lst))
print(lst1)
```

Output:

[10,46,70]

Passing lambda function to filter() function is more elegant.

A lambda that returns even numbers from a list.

```
lst = [10, 23, 45, 46, 70, 99]
lst1 = list(filter(lambda x: (x%2 == 0), lst))
print(lst1)
```

Output:

[10,46,70]

Lambda with map() function

- Function The map() function is similar to filter() function but **it acts on each element of the sequence and perhaps changes the elements**.
- The format of map() function is:

  map(function, sequence)
- The 'function' performs a specified operation on all the elements of the sequence and the modified elements are returned which can be stored in another sequence.
- **A Python program to find squares of elements in a list.**

```
def squares(x):
    return x*x
lst = [1, 2, 3, 4, 5]
lst1 = list(map(squares, lst))
print(lst1)
```

- **A lambda function that returns squares of elements in a list.**

```
lst = [1, 2, 3, 4, 5]
lst1 = list(map(lambda x: x*x, lst))
print(lst1)
```

Output:
[1,4,9,16,25]

- It is possible to use map() function **on more than one list** if the lists are of same length.
- In this case, map() function takes the lists as arguments of the lambda function and does the operation. For example,

map(lambda x, y: x*y, lst1, lst2)

- Here, lambda function has two arguments 'x' and 'y'. Hence, 'x' represents 'lst1' and 'y' represents 'lst2'. Since lambda is showing x*y, the respective elements from lst1 and lst2 are multiplied and the product is returned.

- **A Python program to find the products of elements of two different lists using lambda function.**

```
lst1 = [1, 2, 3, 4, 5]
lst2 = [10, 20, 30, 40, 50]
lst3 = list(map(lambda x, y: x*y, lst1, lst2))
print(lst3)
```

Output:

[10,40,90,160,250]

Lambda with reduce() function

- The reduce() function **reduces a sequence of elements to a single value by processing the elements** according to a function supplied. The reduce() function is used in the format:

  reduce(function, sequence)

- For example, we write the reduce() function with a lambda expression, as:

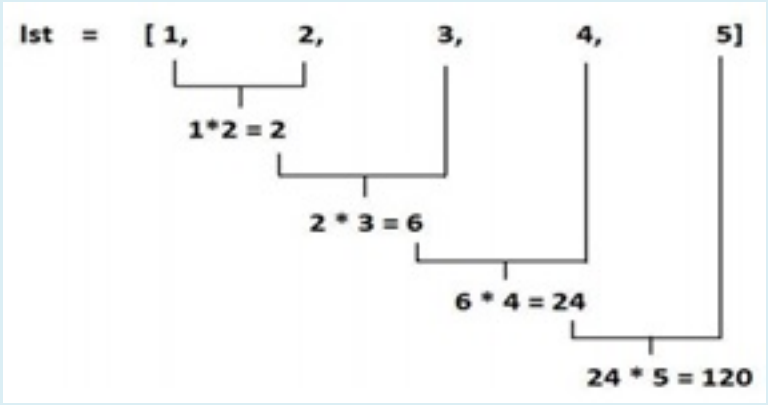
  lst = [1, 2, 3, 4, 5]

  reduce(lambda x, y: x*y, lst)

- The lambda function is taking two arguments and returning their product. Hence, starting from the 0th element of the list 'lst', the first two elements are multiplied and the product is obtained. Then this product is multiplied with the third element and the product is obtained. Again this product is multiplied with the fourth element and so on. The final product value is returned, as shown in Figure

A lambda function to calculate products of elements of a list.

```
from functools import *
lst = [1, 2, 3, 4, 5]
result = reduce(lambda x, y: x*y, lst)
print(result)
```



List, Tuples and Dictionary

- 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.
- lists are the most used data type in Python programs.
- **The list appears as given below:**

  [10, 'Venn herry', 'M', 50, 55, 62, 74, 66]
- Indexing and slicing operations are commonly done on lists.
- Indexing represents accessing elements by their position numbers in the list. The position numbers start from 0 onwards and are written inside square braces.

L = [10, ‘Venn herry', 'M', 50, 55, 62, 74, 66]

```
L[1]          #’Venn herry'
L[0:3:1]      # [10, ‘Venn herry', 'M']
L[:3:]        # [10, ‘Venn herry', 'M']
L[::]         # [10, ‘Venn herry', 'M', 50, 55, 62, 74, 66]
L[-2]         #74
L[-5:-1]      # [50, 55, 62, 74]
L[::-1]       # [66, 74, 62, 55, 50, 'M', ‘Venn herry', 10]
L[2:]         # ['M', 50, 55, 62, 74, 66]
```

A Python program to create lists with different types of elements.

```
num = [10, 20, 30, 40, 50]
print('Total list = ', num)
print('First = %d, Last = %d' % (num[0], num[4]))

names = ["Raju", "Vani", "Gopal", "Laxmi"]
print('Total list = ', names)
print('First = %s, Last = %s '% (names[0],
names[3]))

x = [10, 20, 10.5, 2.55, "Ganesh", 'Vishnu']
print('Total list = ', x)
print('First= %d, Last= %s'% (x[0], x[5]))
```

Output:

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

First =10, Last=50

Total list ["Raju", "Vani", "Gopal", "Laxmi“]

First=Raju, Last=Laxmi

Total list=[10, 20, 10.5, 2.55, "Ganesh", 'Vishnu']

First=10, Last=Vishnu

Create Lists using range() Function:

- We can use range() function to generate a sequence of integers which can be stored in a list. The format of the range() function is:

  range(start, stop, stepsize)

- If we do not mention the 'start', it is assumed to be 0 and the 'stepsize' is taken as 1.

A Python program to create lists using range() function.

```
list1 = range(10)
for i in list1:
  print(i,',', end='')
print() #throw cursor to next line
#create list with integers from 5 to 9
list2 = range(5, 10)
for i in list2:
  print(i,',', end='')
print()
#create a list with odd numbers from 5 to 9
list3 = range(5, 10, 2) #step size is 2
for i in list3:
  print(i, ',' , end='')
```

Output:

0,1,2,3,4,5,6,7,8,9,

5,6,7,8,9,

5,7,9,

Update elements of List

- **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.

```
lst = list(range(1,5))  #create a list using list() and range()
print(lst)      #[1,2,3,4]
lst.append(9)  #append a new element to lst
print(lst)  #[1,2,3,4,9]
```

- 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.
- Consider the following statements:

```
lst[1] = 8 #update 1st element of lst
print(lst)    #[1,8,3,4,9]
lst[1:3] = 10, 11 #update 1st and 2nd elements of lst
print(lst)     #[1,10,11,4,9]
```

Delete elements of List

- 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.

  del lst[1] #delete 1st element from lst

  print(lst) #[1,11,4,9]
- We can also delete an element using the **remove() method**. In this method, we should pass the element to be deleted.

  lst.remove(11) #delete 11 from lst

  print(lst) #[1,4,9]

- To retrieve the elements of a list in reverse order.
- This can be done easily by using the **reverse()** method, as:

  list.reverse()
- This will reverse the order of elements in the list and the reversed elements are available in the list

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'.
- **lists concatenation**

```
x = [10, 20, 30, 40, 50]
y = [100, 110, 120]
print(x+y)    # concatenate x and y
```

Output:

[10,20,30,40,50,100,110,120]

Repetition of lists

- We can repeat the elements of a list 'n' number of times using '*' operator. For example, if we write x*n, the list 'x' will be repeated for n times as:

print(x*2) # repeat the list x for 2 times

Output:

[10,20,30,40,50,10,20,30,40,50]

Membership in Lists

We can check if an element **is a member of a list or not by using 'in' and 'not in' operator**. If the element is a member of the list, then 'in' operator returns True else False. If the element is not in the list, then 'not in' operator returns True else False. See the examples below:

```
x = [10, 20, 30, 40, 50]
a = 20
print(a in x)
```

The preceding statements will give: **True**

If you write,

```
print(a not in x)    #check if a is not a member of x
```

Then, it will give **False**

Aliasing and Cloning lists

List aliasing

- Giving a new name to an existing list is called 'aliasing'. The new name is called 'alias name'. For example, take a list 'x' with 5 elements as

  x = [10, 20, 30, 40, 50]

- To provide a new name to this list, we can simply use assignment operator as:

  y = x

- 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. Observe the following statements where an element x[1] is modified with a new value.

```
x = [10,20,30,40,50]
y = x    #x is aliased as y
print(x) will display [10,20,30,40,50]
print(y) will display [10,20,30,40,50]

x[1] = 99    #modify 1st element in x
print(x) will display [10,99,30,40,50]
print(y) will display [10,99,30,40,50]
```

List cloning

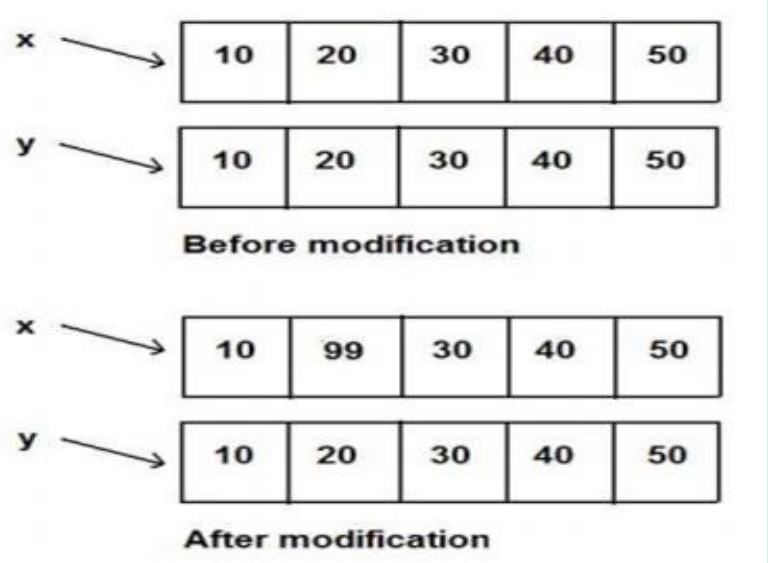
- If the programmer wants two independent lists, he should not go for aliasing. On the other hand, he should use **cloning or copying.**
- Obtaining exact copy of an existing object (or list) is called 'cloning'. To clone a list, we can take help of the slicing operation as:

  y = x[:] #x is cloned as y
- We can observe that in cloning, modifications to a list are confined only to that list. The same can be achieved by copying the elements of one list to another using **copy() method**. For example, consider the following statement:

  y = x.copy() #x is copied as y
- When we copy a list like this, a separate copy of all the elements is stored into 'y'. The lists 'x' and 'y' are independent. Hence, any modifications to 'x' will not affect 'y' and vice versa.

```
x = [10,20,30,40,50]
y = x[:] #x is cloned as y
print(x) will display [10,20,30,40,50]
print(y) will display [10,20,30,40,50]

x[1] = 99 #modify 1st element in x
print(x) will display [10,99,30,40,50]
print(y) will display [10,20,30,40,50]
```



Methods to process List

- The function len() is useful to find the number of elements in a list. We can use this function as:

  n = len(list)

- There are various other methods provided by Python to perform various operations on lists.

Method	Example	Description
sum()	list.sum()	Returns sum of all elements in the list.
index()	list.index(x)	Returns the first occurrenceof x in the list
append()	list.append(x)	Appends x at the end of the list
insert()	list.insert(i,x)	Inserts x in to the list in the position specified by i
copy()	list.copy()	copies all the list elements into a new list and returns it

List methods and details

Method	Example	Description
extend()	list.extend(list1)	Appends list 1 to list
count()	list.count(x)	Returns number of occurences of x in the list
remove()	list.remove(x)	Removes x from the list
pop()	list.pop()	Removes the ending element from the list
sort()	list.sort()	Sorts the elements of the list into ascending order
reverse()	list.reverse()	Reverses the sequence of elements in the list
clear()	list.clear()	Deletes all elements from the list

A Python program to understand list processing methods.

```
num = [10,20,30,40,50]
n = len(num)
print('No. of elements in num:', n)
```

o/p: No. of elements in num : 5

```
num.append(60)
print('num after appending 60:', num)
```

o/p: number after appending: [10,20,30,40,50,60]

```
num.insert(0,5)
print('num after inserting 5 at 0th position:', num)
```

o/p: num after inserting 5 at 0th position: [5,10,20,30,40,50,60]

```
num1 = num.copy()
print('Newly created list num1:', num1)
```

o/p: Newly created list num1: [5,10,20,30,40,50,60]

```
num.extend(num1)
print('num after appending num1:', num)
```

o/p: num after appending num1: [5,10,20,30,40,50,60,5,10,20,30,40,50,60]

```
n = num.count(50)
print('No. of times 50 found in the list num:', n)
```

o/p: No. of times 50 found in the list num: 2

```
num.remove(50)
print('num after removing 50:', num)
```

o/p: num after removing 50: [5,10,20,30,40,60,5,10,20,30,40,50,60]

```
num.pop()
print('num after removing ending element:', num)
```

o/p: num after removing ending element: [5,10,20,30,40,60,5,10,20,30,40,50]

```
num.sort()
print('num after sorting:', num)
```

o/p: num after sorting: [5,5,10,10,20,20,30,30,40,40,50,60]

```
num.reverse()
print('num after reversing:',num)
```

o/p: num after reversing: [60,50,40,40,30,30,20,20,10,10,5,5]

```
num.clear()
print('num after removing all elements:', num)
```

o/p: num after removing all elements:[]

Nested Lists

- **A list within another list is called a *nested list***. When we take a list as an element in another list, then that list is called a nested list.
- For example, we have two lists 'a' and 'b' as:

  a = [80, 90]

  b = [10, 20, 30, a]
- Observe that the list 'a' is inserted as an element in the list 'b' and hence 'a' is called a nested list.

  print(b)
- The elements of b appears:

  [10, 20, 30, [80, 90]]
- The last element [80, 90] represents a nested list. So, 'b' has 4 elements and they are:

  b[0] = 10

  b[1] = 20

  b[2] = 30

  b[3] = [80, 90]
- So, b[3] represents the nested list and if we want to display its elements separately, we can use a for loop as:

```
for x in b[3]:
    print(x)
```

A Python program to create a nested list and display its elements.

```
#To create a list with another list as element
list = [10, 20, 30, [80, 90]]
print('Total list=', list)     #display entire list
print('First element=', list[0])     #display first element
print('Last element is nested list=', list[3])
for x in list[3]:      #display all elements in nested list
  print(x)
```

Output:

```
Total list = [10, 20, 30, [80, 90]]
First element= 10
Last element is nested list= [80,90]
80
90
```

Nested Lists as Matrices

- 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.
- If a matrix contains ‘m’ rows and ‘n’ columns, then it is called m X n matrix.
- In python, matrices are created as 2D arrays or using matrix object in numpy.
- We can also create a matrix using nested lists.
- Suppose we want to create a matrix with 3 rows and 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
- If we use a for loop to retrieve the elements from 'mat', it will retrieve row by row, as:

```
for r in mat:
  print(r) #display row by row
```

- But we want to retrieve columns (or elements) in each row; hence we need another for loop inside the previous loop, as:

```
for r in mat:
  for c in r:     #display columns in each row
    print(c, end=‘ ‘)
  print()
```

- Another way to display the elements of a matrix is using indexing. For example, mat[i][j] represents the i^{th} row and j^{th} column element. Suppose the matrix has 'm' rows and 'n' columns, we can retrieve the elements as:

```
for i in range(len(mat)):    #i values change from 0 to m-1
  for j in range(len(mat[i])):     #j values change from 0 to n-1
    print('%d ' %mat[i][j], end=‘ ‘)
  print()
```

Tuples

- 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**.
- Since tuples are immutable, 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 of same datatype or different types.
- For example, to create an empty tuple, we can simply write empty parenthesis, as:

  tup1 = () #empty tuple
- If we want to create a tuple with only one element, we can mention that element in parentheses and after that a comma is needed, as:

  tup2 = (10,)

- Here is a tuple with different types of elements:

  tup3 = (10, 20, -30.1, 40.5, 'Hyderabad', 'New Delhi')
- We can create a tuple with only one of type of elements also, like the following:

  tup4 = (10, 20, 30) #tuple with integers
- If we do not mention any brackets and write the elements separating them by commas, then they are taken by default as a tuple. See the following example:

  tup5 = 1, 2, 3, 4 #no braces
- The point to remember is that if do not use any brackets, it will become a tuple and not a list or any other datatype.
- It is also possible to create a tuple from a list. This is done by converting a list into a tuple using the tuple() function.

  list = [1, 2, 3] #take a list

  tpl = tuple(list) #convert list into tuple

  print(tpl) #display tuple (1,2,3)
- Another way to create a tuple is by using range() function that returns a sequence.

  tpl= tuple(range(4, 9, 2)) #numbers from 4 to 8 in steps of 2

  print(tpl)

  The preceding statements will give: (4, 6, 8)

Access Tuple Elements:

- Accessing the elements from a tuple can be done using indexing or slicing. This is same as that of a list.

  tup = (50,60,70,80,90,100)

- Indexing represents the position number of the element in the tuple.
- Now, tup[0] represents the 0^{th} element, tup[1] represents the 1^{st} element and so on.

```
print(tup[0])          #50
print(tup[:])          #(50,60,70,80,90,100)
print(tup[1:4])        #(60,70,80)
print(tup[::2])        #(50,70,90)
print(tup[-4:-1])      #(70,80,90)

student = (10, 'Naresh kumar', 50,60,65,61,70)
rno, name = student[0:2]
print(rno, name)       #10, Nareshkumar

marks = student[2:7]
for i in marks:
    print(i)    #50 60 65 61 70
```

Python Tuple Operations

- In Python,there are 5 basic operations:
- **finding length**, **concatenation**, **repetition**, **membership** and **iteration** operations can be performed on any sequence may be it is a string, list, tuple or a dictionary.
- To find length of a tuple, we can use len() function.
- This returns the number of elements in the tuple.

```
student = (30, 'Abhinav Siddarth', 74,85,56,52,36)
len(student)            #7
```

- We can concatenate or join two tuples and store the result in a new tuple.
- For example, the student paid a fees of Rs. 25,000.00 every year for 4 terms, then we can create a 'fees' tuple as:

```
fees = (25000.00,)*4                #repeat the tuple elements for 4 times.
print(fees)
Difference b/w fees = (25000.00,)*4 and fees = (25000.00)*4
```

- Now, we can concatenate the 'student' and 'fees' tuples together to form a new tuple 'student1' as:

```
student1 = student+fees
print(student1)        #(30, 'Abhinav Siddarth', 74,85,56,52,36,25000.00, 25000.00, 25000.00, 25000.00)

Membership in Tuple
name='Abhinav Siddarth'
name in student1     #True
```

Tuple operation functions:

Function	Example	Description
len()	len(tpl)	Returns the number of elements in the tuple.
min()	min(tpl)	Returns the smallest element in the tuple.
max()	max(tpl)	Returns the biggest element in the tuple.
count()	tpl.count(x)	Returns how many times the element 'x' is found in tpl.
index()	tpl.index(x)	Returns the first occurrence of the element 'x' in tpl. Raises ValueError if 'x' is not found in the tuple.
sorted()	sorted(tpl)	Sorts the elements of the tuple into ascending order.sorted(tpl,reverse=True) will sort in reverse order.

Python Nested Tuples

- A tuple inserted **inside another tuple is called nested tuple**.
- For example,

  tup = (50,60,70,80,90, (200, 201))
- The tuple (200, 201) is called **nested tuple** as it is inside another tuple.
- The nested tuple with the elements (200, 201) is treated as an element along with other elements in the tuple 'tup'.
- To retrieve the nested tuple, we can access it as an ordinary element as tup[5] as its index is 5.
- Every nested tuple can represent a specific data record.
- For example, to store 4 employees data in 'emp' tuple, we can write:

  emp = ((100, "Varun", 9000.90), (20, "Nihaarika", 5500.75), (30, "Vanaja",8900.00), (40, "Karthik", 5000.50))
- Here, 'emp' is the tuple name. It contains 4 nested tuples each of which represents the data of an employee. Every employee's identification number, name and salary are stored as a nested tuples.

Sorting Nested Tuples

- To sort a tuple, we can use **sorted() function**.
- This function sorts by default into ascending order.
- For example,
- **print(sorted(emp))** will sort the tuple 'emp' in ascending order of the 0^{th} element in the nested tuples, i.e. identificaton number.
- If we want to sort the tuple based on employee name, which is the 1^{st} element in the nested tuples,
- we can use a lambda expression as:

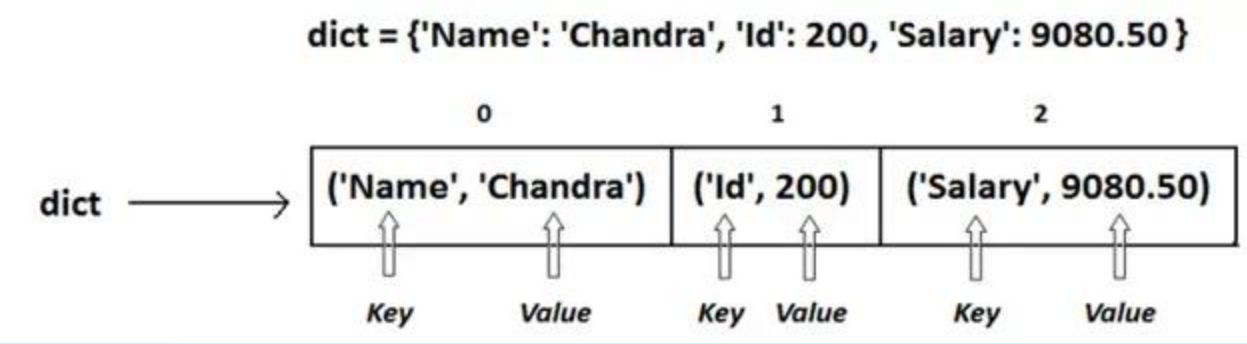
  print(sorted(emp, key=lambda x: x[1]))
- Here, key indicates the key for the sorted() function that tells on which element sorting should be done.
- The lambda function: **lambda x: x[1]** indicates that x[1] should be taken as the key that is nothing but 1st element. If we want to sort the tuple based on salary, we can use the lambda function as: **lambda x: x[2]** .

```
print(sorted(emp))   #sorts by default on id
print(sorted(emp, reverse=True))          #reverses on id
print(sorted(emp, key=lambda x: x[1]))  #sort on name
print(sorted(emp, key=lambda x: x[1],reverse=True))
print(sorted(emp, key=lambda x: x[2]))  #sort on salary
print(sorted(emp, key=lambda x: x[2],reverse=True))
```

Introduction to Dictionaries

- A dictionary represents **a group of elements arranged in the form of key-value pairs.**
- In the dictionary, 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 {}**. Let's take a dictionary by the name 'dict' that contains employee details:

  dict = {'Name': 'Chandra', 'Id': 200, 'Salary': 9080.50}



A Python program to create a dictionary with employee details and retrieve the values upon giving the keys.

```
dict = {'Name': 'Chandra', 'Id': 200, 'Salary': 9080.50}

print('Name of employee=', dict['Name'])
print('Id number=', dict['Id'])
print('Salary=', dict['Salary'])
```

Output:

Name of employee= chandra

Id number= 200

Salary = 9080.5

Operations on Dictionaries

- 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']**. This will return the value associated with 'Name'.

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

  dict = {'Name': 'Chandra', 'Id': 200, 'Salary': 9080.50}

  print('Length=', len(dict)) # Length=3

- We can modify the existing value of a key by assigning a new value, as shown in the following statement:

  dict['Salary'] = 10500.00 #{'Name': 'Chandra', 'Id': 200, 'Salary': 10500.00}

- 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, as shown in the following statement:

  dict['Dept'] = 'Finance‘ **#**{'Name': 'Chandra',’Dept’:’Finance’,'Id': 200, 'Salary': 10500.00}
- **This pair may be added at any place in the dictionary.**

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

  del dict['Id']
- 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.
- Consider the following statement:

  'Dept' in dict #true

  'Gender' not in dict #false
- We can use any datatypes for values.
- For example, a value can be a number, string, list, tuple or another dictionary. But keys should obey the following 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.
- Consider the following example:

  emp = {'Nag':10, 'Vishnu':20, 'Nag':30}

  print(emp) #{'Vishnu':20, 'Nag':30}

- **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, 'Raj':30}

  Output:

  TypeError: unhashable type: 'list'

Dictionary Methods

- Various methods are provided to process the elements of a dictionary.
- These methods generally retrieve or manipulate the contents of a dictionary.

Method	Example	Description
clear()	d.clear()	Removes all key-value pairs from dictionary 'd'.
copy()	d1= d.copy()	Copies all elements from 'd' into a new dictionary 'd1'.
fromkeys()	d.fromkeys(s [,v])	Create a new dictionary with keys from sequence 's' and values all set to 'v'.
get()	d.get(k [,v])	Returns the value associated with key 'k'. If key is not found, it returns 'v'.
items()	d.items()	Returns an object that contains key-value pairs of 'd'. The pairs are stored as tuples in the object.
keys()	d.keys()	Returns a sequence of keys from the dictionary 'd'.
values()	d.values()	Returns a sequence of values from the dictionary 'd'.
update()	d.update(x)	Adds all elements from dictionary 'x' to 'd'.
pop()	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.
setdefault()	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'.

Examples:

1. d = {'1': ‘Vishnu', '2': ‘Raj', '3': ‘Smit'}
   d.clear()
   print(d)

   O/p: {}

2. d1= d.copy()
   Print(d1)

   o/p: {'1': ‘Vishnu', '2': ‘Raj', '3': ‘Smit'}

3. seq = ('a', 'b', 'c')
   print(dict.fromkeys(seq, None))
   print(dict.fromkeys(seq, 1))

   o/p: {'a': None, 'b': None, 'c': None}

   o/p: {'a': 1, 'b': 1, 'c': 1}

4. d = {'Name': 'Ram', 'Age': '19', 'Country': 'India'}
   print(d.get('Name'))
   print(d.get('Gender'))

   o/p: Ram
   None

5. d = {'Name': 'Ram', 'Age': '19', 'Country': 'India'}
   print(list(d.items()))

   o/p: [('Name', 'Ram'), ('Age', '19'), ('Country', 'India')]

6. d = {'Name': 'Ram', 'Age': '19', 'Country': 'India'}
   print(list(d.keys()))

o/p: ['Name', 'Age', 'Country']

7. d = {'Name': 'Ram', 'Age': '19', 'Country': 'India'}
   print(list(d.values()))

o/p: ['Ram', '19', 'India']

8. d1 = {'Name': 'Ram', 'Age': '19', 'Country': 'India'}
   d2 = {'Name': 'Neha', 'Age': '22'}
   d1.update(d2)
   print(d1)

o/p: {'Name': 'Neha', 'Age': '22', 'Country': 'India'}

9. d = {'Name': 'Ram', 'Age': '19', 'Country': 'India'}
   d.pop('Name')
   print(d)

o/p: {'Age': '19', 'Country': 'India'}

10. d = {'Name': 'Ram', 'Age': '19', 'Country': 'India'}
    print(d.setdefault('city'))
    print(d)

o/p: {'Name': 'Ram', 'Age': '19', 'Country': 'India', 'city': None}

Converting List 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.

```
countries = ["USA", "India", "Germany", "France"]
cities = ['Washington', 'New Delhi', 'Berlin', 'Paris']
```

- We want to create a dictionary out of these two lists by taking the elements of 'countries' list as keys and of 'cities' list as values.

- 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 as:

```
z = zip(countries, cities)
```

- 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.

  d = dict(z)

- A Python program to convert the elements of two lists into key-value pairs of a dictionary.

```
countries = ["USA", "India", "Germany", "France"]
cities = ['Washington', 'New Delhi', 'Berlin', 'Paris']


z = zip(countries, cities)
d = dict(z)
print(d)
```

o/p: {'USA': 'Washington', 'India': 'New Delhi', 'Germany': 'Berlin', 'France': 'Paris'}