CHAPTER 3

Python Dictionary

Python Dictionary is used to store the data in a key-value pair format. The dictionary is the data type in Python, which can simulate the real-life data arrangement where some specific value exists for some particular key. It is the mutable data-structure. The dictionary is defined into element Keys and values.

- Keys must be a single element
- Value can be any type such as list, tuple, integer, etc.

In other words, we can say that a dictionary is the collection of key-value pairs where the value can be any Python object. In contrast, the keys are the immutable Python object, i.e., Numbers, string, or tuple.

Creating the dictionary

The dictionary can be created by using multiple key-value pairs enclosed with the curly brackets {}, and each key is separated from its value by the colon (:).The syntax to define the dictionary is given below.

Syntax:

```
1. Dict = {"Name": "Tom", "Age": 22}
```

In the above dictionary **Dict**, The keys **Name** and **Age** are the string that is an immutable object.

Let's see an example to create a dictionary and print its content.

```
1. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
2. print(type(Employee))
3. print("printing Employee data .... ")
4. print(Employee)
```

Output

```
<class 'dict'>
Printing Employee data ....
{'Name': 'John', 'Age': 29, 'salary': 25000, 'Company': 'GOOGLE'}
```

Python provides the built-in function **dict()** method which is also used to create dictionary. The empty curly braces {} is used to create empty dictionary.

```
1. # Creating an empty Dictionary
2. Dict = {}
3. print("Empty Dictionary: ")
4. print(Dict)
5. # Creating a Dictionary
6. # with dict() method
7. Dict = dict({1: 'Java', 2: 'T', 3:'Point'})
8. print("\nCreate Dictionary by using  dict(): ")
```

```
print(Dict)
# Creating a Dictionary
# with each item as a Pair
Dict = dict([(1, 'Devansh'), (2, 'Sharma')])
print("\nDictionary with each item as a pair: ")
print(Dict)
```

Output:

```
Empty Dictionary:
{}
Create Dictionary by using dict():
{1: 'Java', 2: 'T', 3: 'Point'}
Dictionary with each item as a pair:
{1: 'Devansh', 2: 'Sharma'}
```

Accessing the dictionary values

We have discussed how the data can be accessed in the list and tuple by using the indexing.

However, the values can be accessed in the dictionary by using the keys as keys are unique in the dictionary.

The dictionary values can be accessed in the following way.

```
Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
print(type(Employee))
print("printing Employee data .... ")
print("Name : %s" %Employee["Name"])
print("Age : %d" %Employee["Age"])
print("Salary : %d" %Employee["salary"])
print("Company : %s" %Employee["Company"])
```

Output:

```
<class 'dict'>
printing Employee data ....
Name : John
Age : 29
Salary : 25000
Company : GOOGLE
```

Python provides us with an alternative to use the get() method to access the dictionary values. It would give the same result as given by the indexing.

Adding dictionary values

The dictionary is a mutable data type, and its values can be updated by using the specific keys. The value can be updated along with key **Dict[key] = value**. The update() method is also used to update an existing value.

Note: If the key-value already present in the dictionary, the value gets updated. Otherwise, the new keys added in the dictionary.

Let's see an example to update the dictionary values.

Example - 1:

```
1. # Creating an empty Dictionary
2. Dict = {}
3. print("Empty Dictionary: ")
4. print(Dict)
5. # Adding elements to dictionary one at a time
6. Dict[0] = 'Peter'
7. Dict[2] = 'Joseph'
8. Dict[3] = 'Ricky'
9. print("\nDictionary after adding 3 elements: ")
10. print(Dict)
11. # Adding set of values
12. # with a single Key
13. # The Emp_ages doesn't exist to dictionary
14. Dict['Emp_ages'] = 20, 33, 24
15. print("\nDictionary after adding 3 elements: ")
16. print(Dict)
17. # Updating existing Key's Value
18. Dict[3] = 'JavaTpoint'
19. print("\nUpdated key value: ")
20. print(Dict)
```

Output:

```
Empty Dictionary:
{}

Dictionary after adding 3 elements:
{0: 'Peter', 2: 'Joseph', 3: 'Ricky'}

Dictionary after adding 3 elements:
{0: 'Peter', 2: 'Joseph', 3: 'Ricky', 'Emp_ages': (20, 33, 24)}
```

```
Updated key value:
{0: 'Peter', 2: 'Joseph', 3: 'JavaTpoint', 'Emp_ages': (20, 33, 24)}
```

Example - 2:

```
1. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
2. print(type(Employee))
3. print("printing Employee data .... ")
4. print(Employee)
5. print("Enter the details of the new employee....");
6. Employee["Name"] = input("Name: ");
7. Employee["Age"] = int(input("Age: "));
8. Employee["salary"] = int(input("Salary: "));
9. Employee["Company"] = input("Company:");
10. print("printing the new data");
11. print(Employee)
```

Output:

```
Empty Dictionary:
{}

Dictionary after adding 3 elements:
{0: 'Peter', 2: 'Joseph', 3: 'Ricky'}

Dictionary after adding 3 elements:
{0: 'Peter', 2: 'Joseph', 3: 'Ricky', 'Emp_ages': (20, 33, 24)}

Updated key value:
{0: 'Peter', 2: 'Joseph', 3: 'JavaTpoint', 'Emp_ages': (20, 33, 24)}
```

Deleting elements using del keyword

The items of the dictionary can be deleted by using the **del** keyword as given below.

```
1. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
2. print(type(Employee))
3. print("printing Employee data .... ")
4. print(Employee)
5. print("Deleting some of the employee data")
6. del Employee["Name"]
7. del Employee["Company"]
8. print("printing the modified information ")
```

```
9. print(Employee)
10. print("Deleting the dictionary: Employee");
11. del Employee
12. print("Lets try to print it again ");
13. print(Employee)
```

Output:

```
<class 'dict'>
printing Employee data ....
{'Name': 'John', 'Age': 29, 'salary': 25000, 'Company': 'GOOGLE'}
Deleting some of the employee data
printing the modified information
{'Age': 29, 'salary': 25000}
Deleting the dictionary: Employee
Lets try to print it again
NameError: name 'Employee' is not defined
```

The last print statement in the above code, it raised an error because we tried to print the Employee dictionary that already deleted.

- **Using pop() method**

The **pop()** method accepts the key as an argument and remove the associated value. Consider the following example.

```
1. # Creating a Dictionary
2. Dict = {1: 'JavaTpoint', 2: 'Peter', 3: 'Thomas'}
3. # Deleting a key
4. # using pop() method
5. pop_ele = Dict.pop(3)
6. print(Dict)
```

Output:

```
{1: 'JavaTpoint', 2: 'Peter'}
```

Python also provides a built-in methods popitem() and clear() method for remove elements from the dictionary. The popitem() removes the arbitrary element from a dictionary, whereas the clear() method removes all elements to the whole dictionary.

Iterating Dictionary

A dictionary can be iterated using for loop as given below.

Example 1

 # for loop to print all the keys of a dictionary

```
1. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
2. for x in Employee:
```

3. **print**(x)

Output:

```
Name
Age
salary
Company
```

Example 2

#for loop to print all the values of the dictionary

1. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
2. **for** x **in** Employee:
3. **print**(Employee[x])

Output:

```
John
29
25000
GOOGLE
```

Example - 3

#for loop to print the values of the dictionary by using values() method.

1. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
2. **for** x **in** Employee.values():
3. **print**(x)

Output:

```
John
29
25000
GOOGLE
```

Example 4

#for loop to print the items of the dictionary by using items() method.

1. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
2. **for** x **in** Employee.items():
3. **print**(x)

Output:

```
('Name', 'John')
('Age', 29)
('salary', 25000)
('Company', 'GOOGLE')
```

Properties of Dictionary keys

1. In the dictionary, we cannot store multiple values for the same keys. If we pass more than one value for a single key, then the value which is last assigned is considered as the value of the key.

Consider the following example.

```
1. Employee={"Name":"John","Age":29,"Salary":25000,"Company":"GOOGLE","Name":"Jo
   hn"}
2. for x,y in Employee.items():
3.     print(x,y)
```

Output:

```
Name John
Age 29
Salary 25000
Company GOOGLE
```

2. In python, the key cannot be any mutable object. We can use numbers, strings, or tuples as the key, but we cannot use any mutable object like the list as the key in the dictionary.

Consider the following example.

```
1. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE",[100,201
   ,301]:"Department ID"}
2. for x,y in Employee.items():
3.     print(x,y)
```

Output:

```
Traceback (most recent call last):
  File "dictionary.py", line 1, in
    Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE",[100,201,301]:"Department ID"}
TypeError: unhashable type: 'list'
```

Built-in Dictionary functions

The built-in python dictionary methods along with the description are given below.

SN	Function	Description
1	cmp(dict1, dict2)	It compares the items of both the dictionary and returns true if the first dictionary values are greater than the second dictionary, otherwise it returns false.
2	len(dict)	It is used to calculate the length of the dictionary.
3	str(dict)	It converts the dictionary into the printable string representation.
4	type(variable)	It is used to print the type of the passed variable.

Built-in Dictionary methods

The built-in python dictionary methods along with the description are given below.

SN	Method	Description
1	dic.clear()	It is used to delete all the items of the dictionary.
2	dict.copy()	It returns a shallow copy of the dictionary.
3	dict.fromkeys(iterable, value = None, /)	Create a new dictionary from the iterable with the values equal to value.
4	dict.get(key, default = "None")	It is used to get the value specified for the passed key.
5	dict.has_key(key)	It returns true if the dictionary contains the specified key.
6	dict.items()	It returns all the key-value pairs as a tuple.
7	dict.keys()	It returns all the keys of the dictionary.
8	dict.setdefault(key,default= "None")	It is used to set the key to the default value if the key is not specified in the dictionary
9	dict.update(dict2)	It updates the dictionary by adding the key-value pair of dict2 to this dictionary.
10	dict.values()	It returns all the values of the dictionary.
11	len()	
12	popItem()	
13	pop()	
14	count()	
15	index()	

Class and Objects

We have already discussed in previous tutorial, a class is a virtual entity and can be seen as a blueprint of an object. The class came into existence when it instantiated. Let's understand it by an example.

Suppose a class is a prototype of a building. A building contains all the details about the floor, rooms, doors, windows, etc. we can make as many buildings as we want, based on these details. Hence, the building can be seen as a class, and we can create as many objects of this class.

On the other hand, the object is the instance of a class. The process of creating an object can be called instantiation.

In this section of the tutorial, we will discuss creating classes and objects in Python. We will also discuss how a class attribute is accessed by using the object.

Creating classes in Python

In Python, a class can be created by using the keyword class, followed by the class name. The syntax to create a class is given below.

Syntax

```
class ClassName:
    #statement_suite
```

In Python, we must notice that each class is associated with a documentation string which can be accessed by using **<class-name>.__doc__.** A class contains a statement suite including fields, constructor, function, etc. definition.

Consider the following example to create a class **Employee** which contains two fields as Employee id, and name.

The class also contains a function **display()**, which is used to display the information of the **Employee.**

Example

```
1. class Employee:
2.     id = 10
3.     name = "Devansh"
4.     def display (self):
5.         print(self.id,self.name)
```

Here, the **self** is used as a reference variable, which refers to the current class object. It is always the first argument in the function definition. However, using **self** is optional in the function call.

The self-parameter

The self-parameter refers to the current instance of the class and accesses the class variables. We can use anything instead of self, but it must be the first parameter of any function which belongs to the class.

Creating an instance of the class

A class needs to be instantiated if we want to use the class attributes in another class or method. A class can be instantiated by calling the class using the class name.

The syntax to create the instance of the class is given below.

<object-name> = <**class**-name>(<arguments>)

The following example creates the instance of the class Employee defined in the above example.

Example

```
1. class Employee:
```

```
    id = 10
    name = "John"
    def display (self):
        print("ID: %d \nName: %s"%(self.id,self.name))
# Creating a emp instance of Employee class
emp = Employee()
emp.display()
```

Output:

```
ID: 10
Name: John
```

In the above code, we have created the Employee class which has two attributes named id and name and assigned value to them. We can observe we have passed the self as parameter in display function. It is used to refer to the same class attribute.

We have created a new instance object named **emp.** By using it, we can access the attributes of the class.

Delete the Object

We can delete the properties of the object or object itself by using the del keyword. Consider the following example.

Example

```
class Employee:
    id = 10
    name = "John"

    def display(self):
        print("ID: %d \nName: %s" % (self.id, self.name))
    # Creating a emp instance of Employee class

emp = Employee()

# Deleting the property of object
del emp.id
# Deleting the object itself
del emp
emp.display()
```

It will through the Attribute error because we have deleted the object **emp**.

Python Constructor

A constructor is a special type of method (function) which is used to initialize the instance members of the class.

In C++ or Java, the constructor has the same name as its class, but it treats constructor differently in Python. It is used to create an object.

Constructors can be of two types.

1. Parameterized Constructor
2. Non-parameterized Constructor

Constructor definition is executed when we create the object of this class. Constructors also verify that there are enough resources for the object to perform any start-up task.

Creating the constructor in python

In Python, the method the **__init__()** simulates the constructor of the class. This method is called when the class is instantiated. It accepts the **self**-keyword as a first argument which allows accessing the attributes or method of the class.

We can pass any number of arguments at the time of creating the class object, depending upon the **__init__()** definition. It is mostly used to initialize the class attributes. Every class must have a constructor, even if it simply relies on the default constructor.

Consider the following example to initialize the **Employee** class attributes.

Example

```
1. class Employee:
2.     def __init__(self, name, id):
3.         self.id = id
4.         self.name = name
5.
6.     def display(self):
7.         print("ID: %d \nName: %s" % (self.id, self.name))
8.  emp1 = Employee("John", 101)
9.  emp2 = Employee("David", 102)
10.
11. # accessing display() method to print employee 1 information
12.
13. emp1.display()
14.
15. # accessing display() method to print employee 2 information
16. emp2.display()
```

Output:

```
ID: 101
Name: John
ID: 102
Name: David
```

Counting the number of objects of a class

The constructor is called automatically when we create the object of the class. Consider the following example.

Example

```
1. class Student:
2.     count = 0
3.     def __init__(self):
4.         Student.count = Student.count + 1
5. s1=Student()
6. s2=Student()
7. s3=Student()
8. print("The number of students:",Student.count)
```

Output:

```
The number of students: 3
```

Python Non-Parameterized Constructor

The non-parameterized constructor uses when we do not want to manipulate the value or the constructor that has only self as an argument. Consider the following example.

Example

```
1. class Student:
2.     # Constructor - non parameterized
3.     def __init__(self):
4.         print("This is non parametrized constructor")
5.     def show(self,name):
6.         print("Hello",name)
7. student = Student()
8. student.show("John")
```

Python Parameterized Constructor

The parameterized constructor has multiple parameters along with the **self**. Consider the following example.

Example

```
1. class Student:
2.     # Constructor - parameterized
3.     def __init__(self, name):
4.         print("This is parametrized constructor")
5.         self.name = name
6.     def show(self):
7.         print("Hello",self.name)
8. student = Student("John")
9. student.show()
```

Output:

```
This is parametrized constructor
Hello John
```

Python Default Constructor

When we do not include the constructor in the class or forget to declare it, then that becomes the default constructor. It does not perform any task but initializes the objects. Consider the following example.

Example

```
1. class Student:
2.     roll_num = 101
3.     name = "Joseph"
4.
5.     def display(self):
6.         print(self.roll_num,self.name)
7.
8. st = Student()
9. st.display()
```

Output:

```
101 Joseph
```

More than One Constructor in Single class

Let's have a look at another scenario, what happen if we declare the two same constructors in the class.

Example

```
1. class Student:
2.     def __init__(self):
3.         print("The First Constructor")
```

```
4.      def __init__(self):
5.          print("The second contructor")
6.
7.  st = Student()
```

Output:

```
The Second Constructor
```

In the above code, the object **st** called the second constructor whereas both have the same configuration. The first method is not accessible by the **st** object. Internally, the object of the class will always call the last constructor if the class has multiple constructors.

Note: The constructor overloading is not allowed in Python.

Python built-in class functions

The built-in functions defined in the class are described in the following table.

SN	Function	Description
1	getattr(obj,name,default)	It is used to access the attribute of the object.
2	setattr(obj, name,value)	It is used to set a particular value to the specific attribute of an object.
3	delattr(obj, name)	It is used to delete a specific attribute.
4	hasattr(obj, name)	It returns true if the object contains some specific attribute.

Example

```
1.  class Student:
2.      def __init__(self, name, id, age):
3.          self.name = name
4.          self.id = id
5.          self.age = age
6.
7.      # creates the object of the class Student
8.  s = Student("John", 101, 22)
9.
10. # prints the attribute name of the object s
11. print(getattr(s, 'name'))
12.
13. # reset the value of attribute age to 23
14. setattr(s, "age", 23)
15.
16. # prints the modified value of age
```

```
17. print(getattr(s, 'age'))
18.
19. # prints true if the student contains the attribute with name id
20.
21. print(hasattr(s, 'id'))
22. # deletes the attribute age
23. delattr(s, 'age')
24.
25. # this will give an error since the attribute age has been deleted
26. print(s.age)
```

Output:

```
John
23
True
AttributeError: 'Student' object has no attribute 'age'
```

Built-in class attributes

Along with the other attributes, a Python class also contains some built-in class attributes which provide information about the class.

The built-in class attributes are given in the below table.

SN	Attribute	Description
1	__dict__	It provides the dictionary containing the information about the class namespace.
2	__doc__	It contains a string which has the class documentation
3	__name__	It is used to access the class name.
4	__module__	It is used to access the module in which, this class is defined.
5	__bases__	It contains a tuple including all base classes.

Example

```
1. class Student:
2.     def __init__(self,name,id,age):
3.         self.name = name;
4.         self.id = id;
5.         self.age = age
6.     def display_details(self):
7.         print("Name:%s, ID:%d, age:%d"%(self.name,self.id))
8. s = Student("John",101,22)
```

9. **print**(s.__doc__)
10. **print**(s.__dict__)
11. **print**(s.__module__)

Output:

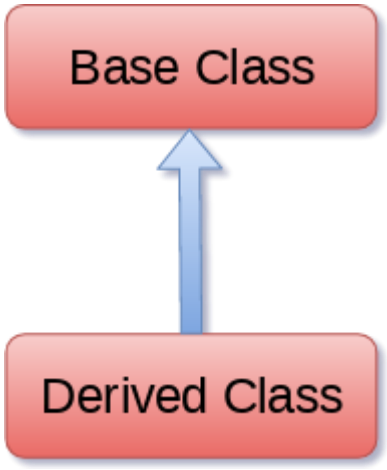
```
None
{'name': 'John', 'id': 101, 'age': 22}
__main__
```

Python Inheritance

Inheritance is an important aspect of the object-oriented paradigm. Inheritance provides code reusability to the program because we can use an existing class to create a new class instead of creating it from scratch.

In inheritance, the child class acquires the properties and can access all the data members and functions defined in the parent class. A child class can also provide its specific implementation to the functions of the parent class. In this section of the tutorial, we will discuss inheritance in detail.

In python, a derived class can inherit base class by just mentioning the base in the bracket after the derived class name. Consider the following syntax to inherit a base class into the derived class.



Syntax

1. **class** derived-**class**(base **class**):
2. <**class**-suite>

A class can inherit multiple classes by mentioning all of them inside the bracket. Consider the following syntax.

Syntax

```
class derive-class(<base class 1>, <base class 2>, ..... <base class n>):
    <class - suite>
```

Example 1

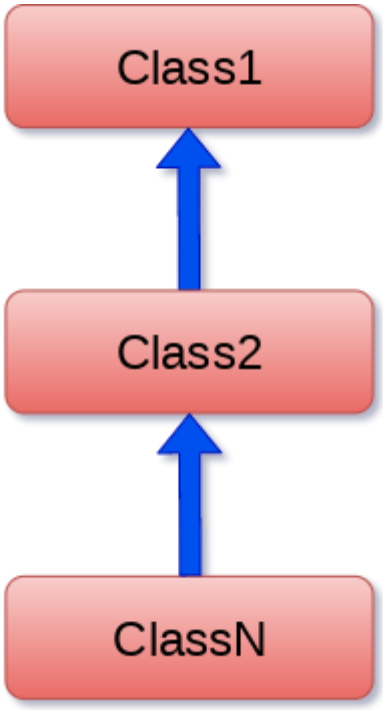
```
class Animal:
    def speak(self):
        print("Animal Speaking")
#child class Dog inherits the base class Animal
class Dog(Animal):
    def bark(self):
        print("dog barking")
d = Dog()
d.bark()
d.speak()
```

Output:

```
dog barking
Animal Speaking
```

Python Multi-Level inheritance

Multi-Level inheritance is possible in python like other object-oriented languages. Multi-level inheritance is archived when a derived class inherits another derived class. There is no limit on the number of levels up to which, the multi-level inheritance is archived in python.



The syntax of multi-level inheritance is given below.

Syntax

```
class class1:
    <class-suite>
class class2(class1):
    <class suite>
class class3(class2):
    <class suite>
.
.
```

Example

```
class Animal:
    def speak(self):
        print("Animal Speaking")
#The child class Dog inherits the base class Animal
class Dog(Animal):
    def bark(self):
        print("dog barking")
#The child class Dogchild inherits another child class Dog
class DogChild(Dog):
    def eat(self):
        print("Eating bread...")
```

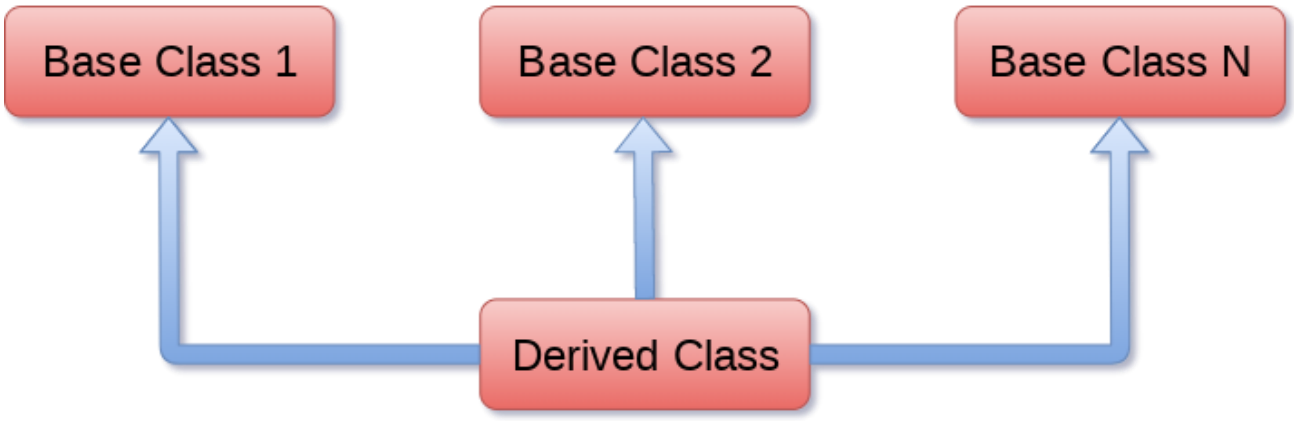
```
12. d = DogChild()
13. d.bark()
14. d.speak()
15. d.eat()
```

Output:

```
dog barking
Animal Speaking
Eating bread...
```

Python Multiple inheritance

Python provides us the flexibility to inherit multiple base classes in the child class.



The syntax to perform multiple inheritance is given below.

Syntax

```
1. class Base1:
2.     <class-suite>
3.
4. class Base2:
5.     <class-suite>
6. .
7. .
8. .
9. class BaseN:
10.    <class-suite>
11.
12. class Derived(Base1, Base2, ..... BaseN):
13.    <class-suite>
```

Example

```
class Calculation1:
    def Summation(self,a,b):
        return a+b;
class Calculation2:
    def Multiplication(self,a,b):
        return a*b;
class Derived(Calculation1,Calculation2):
    def Divide(self,a,b):
        return a/b;
d = Derived()
print(d.Summation(10,20))
print(d.Multiplication(10,20))
print(d.Divide(10,20))
```

Output:

```
30
200
0.5
```

The issubclass(sub,sup) method

The issubclass(sub, sup) method is used to check the relationships between the specified classes. It returns true if the first class is the subclass of the second class, and false otherwise.

Consider the following example.

Example

```
class Calculation1:
    def Summation(self,a,b):
        return a+b;
class Calculation2:
    def Multiplication(self,a,b):
        return a*b;
class Derived(Calculation1,Calculation2):
    def Divide(self,a,b):
        return a/b;
d = Derived()
print(issubclass(Derived,Calculation2))
```

```
12. print(issubclass(Calculation1,Calculation2))
```

Output:

```
True
False
```

The isinstance (obj, class) method

The isinstance() method is used to check the relationship between the objects and classes. It returns true if the first parameter, i.e., obj is the instance of the second parameter, i.e., class.

Consider the following example.

Example

```
1. class Calculation1:
2.     def Summation(self,a,b):
3.         return a+b;
4. class Calculation2:
5.     def Multiplication(self,a,b):
6.         return a*b;
7. class Derived(Calculation1,Calculation2):
8.     def Divide(self,a,b):
9.         return a/b;
10. d = Derived()
11. print(isinstance(d,Derived))
```

Output:

```
True
```

Method Overriding

We can provide some specific implementation of the parent class method in our child class. When the parent class method is defined in the child class with some specific implementation, then the concept is called method overriding. We may need to perform method overriding in the scenario where the different definition of a parent class method is needed in the child class.

Consider the following example to perform method overriding in python.

Example

```
1. class Animal:
2.     def speak(self):
3.         print("speaking")
4. class Dog(Animal):
5.     def speak(self):
6.         print("Barking")
7. d = Dog()
8. d.speak()
```

Output:

```
Barking
```

Real Life Example of method overriding

```
1. class Bank:
2.     def getroi(self):
3.         return 10;
4. class SBI(Bank):
5.     def getroi(self):
6.         return 7;
7.
8. class ICICI(Bank):
9.     def getroi(self):
10.        return 8;
11. b1 = Bank()
12. b2 = SBI()
13. b3 = ICICI()
14. print("Bank Rate of interest:",b1.getroi());
15. print("SBI Rate of interest:",b2.getroi());
16. print("ICICI Rate of interest:",b3.getroi());
```

Output:

```
Bank Rate of interest: 10
SBI Rate of interest: 7
ICICI Rate of interest: 8
```

File Handling

Till now, we were taking the input from the console and writing it back to the console to interact with the user.

Sometimes, it is not enough to only display the data on the console. The data to be displayed may be very large, and only a limited amount of data can be displayed on the console since the memory is volatile, it is impossible to recover the programmatically generated data again and again.

The file handling plays an important role when the data needs to be stored permanently into the file. A file is a named location on disk to store related information. We can access the stored information (non-volatile) after the program termination.

The file-handling implementation is slightly lengthy or complicated in the other programming language, but it is easier and shorter in Python.

In Python, files are treated in two modes as text or binary. The file may be in the text or binary format, and each line of a file is ended with the special character.

Hence, a file operation can be done in the following order.

- Open a file
- Read or write - Performing operation
- Close the file

Opening a file

Python provides an **open()** function that accepts two arguments, file name and access mode in which the file is accessed. The function returns a file object which can be used to perform various operations like reading, writing, etc.

Syntax:

file object = open(<file-name>, <access-mode>, <buffering>)

The files can be accessed using various modes like read, write, or append. The following are the details about the access mode to open a file.

SN	Access mode	Description
1	r	It opens the file to read-only mode. The file pointer exists at the beginning. The file is by default open in this mode if no access mode is passed.
2	rb	It opens the file to read-only in binary format. The file pointer exists at the beginning of the file.
3	r+	It opens the file to read and write both. The file pointer exists at the beginning of the file.
4	rb+	It opens the file to read and write both in binary format. The file pointer exists at the beginning of the file.
5	w	It opens the file to write only. It overwrites the file if previously exists or creates a new one if no file exists with the same name. The file pointer exists at the beginning of the file.
6	wb	It opens the file to write only in binary format. It overwrites the file if it exists previously or creates a new one if no file exists. The file pointer exists at the beginning of the file.
7	w+	It opens the file to write and read both. It is different from r+ in the sense that it overwrites the previous file if one exists whereas r+ doesn't overwrite the

		previously written file. It creates a new file if no file exists. The file pointer exists at the beginning of the file.
8	wb+	It opens the file to write and read both in binary format. The file pointer exists at the beginning of the file.
9	a	It opens the file in the append mode. The file pointer exists at the end of the previously written file if exists any. It creates a new file if no file exists with the same name.
10	ab	It opens the file in the append mode in binary format. The pointer exists at the end of the previously written file. It creates a new file in binary format if no file exists with the same name.
11	a+	It opens a file to append and read both. The file pointer remains at the end of the file if a file exists. It creates a new file if no file exists with the same name.
12	ab+	It opens a file to append and read both in binary format. The file pointer remains at the end of the file.

Let's look at the simple example to open a file named "file.txt" (stored in the same directory) in read mode and printing its content on the console.

Example

```
1. #opens the file file.txt in read mode
2. fileptr = open("file.txt","r")
3.
4. if fileptr:
5.     print("file is opened successfully")
```

Output:

```
<class '_io.TextIOWrapper'>
file is opened successfully
```

In the above code, we have passed **filename** as a first argument and opened file in read mode as we mentioned **r** as the second argument. The **fileptr** holds the file object and if the file is opened successfully, it will execute the print statement

The close() method

Once all the operations are done on the file, we must close it through our Python script using the **close()** method. Any unwritten information gets destroyed once the **close()** method is called on a file object.

We can perform any operation on the file externally using the file system which is the currently opened in Python; hence it is good practice to close the file once all the operations are done.

The syntax to use the **close()** method is given below.

Syntax

```
fileobject.close()
```

Consider the following example.

```
1. # opens the file file.txt in read mode
```

```
2. fileptr = open("file.txt","r")
3. if fileptr:
4.     print("file is opened successfully")
5. #closes the opened file
6. fileptr.close()
```

After closing the file, we cannot perform any operation in the file. The file needs to be properly closed. If any exception occurs while performing some operations in the file then the program terminates without closing the file.

We should use the following method to overcome such type of problem.

```
1. try:
2.     fileptr = open("file.txt")
3.     # perform file operations
4. finally:
5.     fileptr.close()
```

The with statement

The **with** statement was introduced in python 2.5. The with statement is useful in the case of manipulating the files. It is used in the scenario where a pair of statements is to be executed with a block of code in between.

The syntax to open a file using with the statement is given below.

```
with open(<file name>, <access mode>) as <file-pointer>:
    #statement suite
```

The advantage of using with statement is that it provides the guarantee to close the file regardless of how the nested block exits.

It is always suggestible to use the **with** statement in the case of files because, if the break, return, or exception occurs in the nested block of code then it automatically closes the file, we don't need to write the **close()** function. It doesn't let the file to corrupt.

Consider the following example.

Example

```
1. with open("file.txt",'r') as f:
2.     content = f.read();
3.     print(content)
```

Writing the file

To write some text to a file, we need to open the file using the open method with one of the following access modes.

w: It will overwrite the file if any file exists. The file pointer is at the beginning of the file.

a: It will append the existing file. The file pointer is at the end of the file. It creates a new file if no file exists.

Consider the following example.

Example

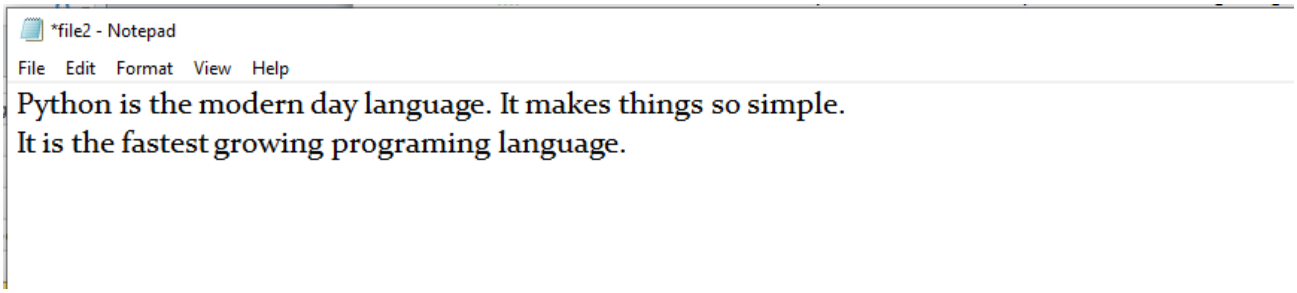
```
1. # open the file.txt in append mode. Create a new file if no such file exists.
2. fileptr = open("file2.txt", "w")
3. # appending the content to the file
4. fileptr.write('''''Python is the modern day language. It makes things so simple.
5. It is the fastest-growing programing language''')
6. # closing the opened the file
7. fileptr.close()
```

Output:

File2.txt

Python is the modern-day language. It makes things so simple. It is the fastest growing programming language.

Snapshot of the file2.txt



We have opened the file in **w** mode. The **file1.txt** file doesn't exist, it created a new file and we have written the content in the file using the **write()** function.

Example 2

```
1. #open the file.txt in write mode.
2. fileptr = open("file2.txt","a")
3. #overwriting the content of the file
4. fileptr.write(" Python has an easy syntax and user-friendly interaction.")
5. #closing the opened file
6. fileptr.close()
```

Output:

Python is the modern day language. It makes things so simple.

It is the fastest growing programing language Python has an easy syntax and user-friendly interaction.

Snapshot of the file2.txt

We can see that the content of the file is modified. We have opened the file in **a** mode and it appended the content in the existing **file2.txt**.

To read a file using the Python script, the Python provides the **read()** method. The **read()** method reads a string from the file. It can read the data in the text as well as a binary format.

The syntax of the **read()** method is given below.

Syntax:

```
fileobj.read(<count>)
```

Here, the count is the number of bytes to be read from the file starting from the beginning of the file. If the count is not specified, then it may read the content of the file until the end.

Consider the following example.

Example

```
1. #open the file.txt in read mode. causes error if no such file exists.
2. fileptr = open("file2.txt","r")
3. #stores all the data of the file into the variable content
4. content = fileptr.read(10)
5. # prints the type of the data stored in the file
6. print(type(content))
7. #prints the content of the file
8. print(content)
9. #closes the opened file
10. fileptr.close()
```

Output:

```
<class 'str'>
Python is
```

In the above code, we have read the content of **file2.txt** by using the **read()** function. We have passed count value as ten which means it will read the first ten characters from the file.

If we use the following line, then it will print all content of the file.

```
1. content = fileptr.read()
2. print(content)
```

Output:

Python is the modern-day language. It makes things so simple.

It is the fastest-growing programing language Python has easy an syntax and user-friendly interaction.

Read file through for loop

We can read the file using for loop. Consider the following example.

```
1. #open the file.txt in read mode. causes an error if no such file exists.
2. fileptr = open("file2.txt","r");
3. #running a for loop
4. for i in fileptr:
5.     print(i) # i contains each line of the file
```

Output:

Python is the modern day language.

It makes things so simple.

Python has easy syntax and user-friendly interaction.

Read Lines of the file

Python facilitates to read the file line by line by using a function **readline()** method. The **readline()** method reads the lines of the file from the beginning, i.e., if we use the readline() method two times, then we can get the first two lines of the file.

Consider the following example which contains a function **readline()** that reads the first line of our file **"file2.txt"** containing three lines. Consider the following example.

Example 1: Reading lines using readline() function

```
1. #open the file.txt in read mode. causes error if no such file exists.
2. fileptr = open("file2.txt","r");
3. #stores all the data of the file into the variable content
4. content = fileptr.readline()
5. content1 = fileptr.readline()
6. #prints the content of the file
7. print(content)
8. print(content1)
9. #closes the opened file
10. fileptr.close()
```

Output:

Python is the modern day language.

It makes things so simple.

We called the **readline()** function two times that's why it read two lines from the file.

Python provides also the **readlines()** method which is used for the reading lines. It returns the list of the lines till the end of **file(EOF)** is reached.

Example 2: Reading Lines Using readlines() function

```
1. #open the file.txt in read mode. causes error if no such file exists.
2. fileptr = open("file2.txt","r");
3.
4. #stores all the data of the file into the variable content
5. content = fileptr.readlines()
6.
7. #prints the content of the file
8. print(content)
9.
10. #closes the opened file
11. fileptr.close()
```

Output:

['Python is the modern day language.\n', 'It makes things so simple.\n', 'Python has easy syntax and user-friendly interaction.']

Creating a new file

The new file can be created by using one of the following access modes with the function open().

x: it creates a new file with the specified name. It causes an error a file exists with the same name.

a: It creates a new file with the specified name if no such file exists. It appends the content to the file if the file already exists with the specified name.

w: It creates a new file with the specified name if no such file exists. It overwrites the existing file.

Consider the following example.

Example 1

```
1. #open the file.txt in read mode. causes error if no such file exists.
2. fileptr = open("file2.txt","x")
3. print(fileptr)
4. if fileptr:
5.     print("File created successfully")
```

Output:

<_io.TextIOWrapper name='file2.txt' mode='x' encoding='cp1252'>

File created successfully

File Pointer positions

Python provides the tell() method which is used to print the byte number at which the file pointer currently exists. Consider the following example.

```
1. # open the file file2.txt in read mode
2. fileptr = open("file2.txt","r")
3.
4. #initially the filepointer is at 0
5. print("The filepointer is at byte :",fileptr.tell())
6.
7. #reading the content of the file
8. content = fileptr.read();
9.
10. #after the read operation file pointer modifies. tell() returns the location of the fileptr.

11.
12. print("After reading, the filepointer is at:",fileptr.tell())
```

Output:

```
The filepointer is at byte : 0
After reading, the filepointer is at: 117
```

Modifying file pointer position

In real-world applications, sometimes we need to change the file pointer location externally since we may need to read or write the content at various locations.

For this purpose, the Python provides us the seek() method which enables us to modify the file pointer position externally.

The syntax to use the seek() method is given below.

Syntax:

```
1. <file-ptr>.seek(offset[, from)
```

The seek() method accepts two parameters:

offset: It refers to the new position of the file pointer within the file.

from: It indicates the reference position from where the bytes are to be moved. If it is set to 0, the beginning of the file is used as the reference position. If it is set to 1, the current position of the file pointer is used as the reference position. If it is set to 2, the end of the file pointer is used as the reference position.

Consider the following example.

Example

```
1. # open the file file2.txt in read mode
```

```
2. fileptr = open("file2.txt","r")
3.
4. #initially the filepointer is at 0
5. print("The filepointer is at byte :",fileptr.tell())
6.
7. #changing the file pointer location to 10.
8. fileptr.seek(10);
9.
10. #tell() returns the location of the fileptr.
11. print("After reading, the filepointer is at:",fileptr.tell())
```

Output:

```
The filepointer is at byte : 0
After reading, the filepointer is at: 10
```

Renaming the file

The Python **os** module enables interaction with the operating system. The os module provides the functions that are involved in file processing operations like renaming, deleting, etc. It provides us the rename() method to rename the specified file to a new name. The syntax to use the **rename()** method is given below.

Syntax:

```
1. rename(current-name, new-name)
```

The first argument is the current file name and the second argument is the modified name. We can change the file name bypassing these two arguments.

Example 1:

```
1. import os
2.
3. #rename file2.txt to file3.txt
4. os.rename("file2.txt","file3.txt")
```

Output:

The above code renamed current **file2.txt** to **file3.txt**

Removing the file

The os module provides the **remove()** method which is used to remove the specified file. The syntax to use the **remove()** method is given below.

Syntax:

```
1. remove(file-name)
```

Example 1

```
1. import os;
```

```
2. #deleting the file named file3.txt
3. os.remove("file3.txt")
```

Directory in Python

Creating the new directory

The **mkdir()** method is used to create the directories in the current working directory. The syntax to create the new directory is given below.

Syntax:

```
1. mkdir(directory name)
```

Example 1

```
1. import os
2.
3. #creating a new directory with the name new
4. os.mkdir("new")
```

The getcwd() method

This method returns the current working directory.

The syntax to use the getcwd() method is given below.

Syntax

```
1. os.getcwd()
```

Example

```
1. import os
2. os.getcwd()
```

Output:

```
'C:\\Users\\DEVANSH SHARMA'
```

Changing the current working directory

The chdir() method is used to change the current working directory to a specified directory.

The syntax to use the chdir() method is given below.

Syntax

```
1. chdir("new-directory")
```

Example

```
1. import os
2. # Changing current directory with the new directiory
3. os.chdir("C:\\Users\\DEVANSH SHARMA\\Documents")
4. #It will display the current working directory
```

5. os.getcwd()

Output:

'C:\\Users\\DEVANSH SHARMA\\Documents'

Deleting directory

The rmdir() method is used to delete the specified directory.

The syntax to use the rmdir() method is given below.

Syntax

1. os.rmdir(directory name)

Example 1

```
1. import os
2. #removing the new directory
3. os.rmdir("directory_name")
```

It will remove the specified directory.

Writing Python output to the files

In Python, there are the requirements to write the output of a Python script to a file.

The **check_call()** method of module **subprocess** is used to execute a Python script and write the output of that script to a file.

The following example contains two python scripts. The script file1.py executes the script file.py and writes its output to the text file **output.txt.**

Example

file.py

```
1. temperatures=[10,-20,-289,100]
2. def c_to_f(c):
3.     if c< -273.15:
4.         return "That temperature doesn't make sense!"
5.     else:
6.         f=c*9/5+32
7.         return f
8. for t in temperatures:
9.     print(c_to_f(t))
```

file.py

```
1. import subprocess
2.
3. with open("output.txt", "wb") as f:
```

4. subprocess.check_call(["python", "file.py"], stdout=f)

The file related methods

The file object provides the following methods to manipulate the files on various operating systems.

SN	Method	Description
1	file.close()	It closes the opened file. The file once closed, it can't be read or write anymore.
2	File.fush()	It flushes the internal buffer.
3	File.fileno()	It returns the file descriptor used by the underlying implementation to request I/O from the OS.
4	File.isatty()	It returns true if the file is connected to a TTY device, otherwise returns false.
5	File.next()	It returns the next line from the file.
6	File.read([size])	It reads the file for the specified size.
7	File.readline([size])	It reads one line from the file and places the file pointer to the beginning of the new line.
8	File.readlines([sizehint])	It returns a list containing all the lines of the file. It reads the file until the EOF occurs using readline() function.
9	File.seek(offset[,from)	It modifies the position of the file pointer to a specified offset with the specified reference.
10	File.tell()	It returns the current position of the file pointer within the file.
11	File.truncate([size])	It truncates the file to the optional specified size.
12	File.write(str)	It writes the specified string to a file
13	File.writelines(seq)	It writes a sequence of the strings to a file.

*****