File Operations: Reading config files in python, Writing log files in python, Understanding read functions, read(), readline() and readlines(), Understanding write functions, write() and writelines(), Manipulating file pointer using seek, Programming using file operations.
Object Oriented Programming: Concept of class, object and instances, Constructor, class attributes and destructors, Real time use of class in live projects, Inheritance , overlapping and overloading operators, Adding and retrieving dynamic attributes of classes, Programming using Oops support.
Design with Classes: Objects and Classes, Data modeling Examples, Case Study An ATM, Structuring Classes with Inheritance and Polymorphism.

read( )

To perform read operation on file we have to open the file in read mode i.e. “r” mode. Here is the code for opening myfile.txt for reading:
The simplest way to read data from a file is to use the file method read to input the entire contents of the file as a single string. If the file contains multiple lines of text, the newline characters will be embedded in this string.
Syntax

```
file.read(size)
```

Parameter	Description
size	It is Optional. The number of bytes to return. Default value is -1, which means the whole file.

Example :

```
f = open("myfile.txt", 'r')
        # assume file has following data
        # First line
        # Second line
text = f.read()
print(text)
```

output:

```
First line.
Second line.
```

After read is finished, another call to read would return an empty string. It indicates that the end of the file has been reached. To repeat reading, the file must be reopened, in order to “rewind” it for another input process. It is not necessary to close the file.
Alternatively, an application might read and process the text one line at a time. A for loop do this nicely. The for loop views a file object as a sequence of lines of text. On each pass through the loop, the loop variable is bound to the next line of text in the sequence. Here is a program that reopens our file and visits the lines of text in it:
Example:

```
f = open("myfile.txt", 'r')
        # assume file has following data
        # First line
        # Second line
for line in f:
  print(line)
```

output:

```
First line.
Second line.
```

Readline( )

If you might want to read a specified number of lines from a file, you can use the file method readline. The readline method consumes a line of input and returns this string, including the newline. If readline encounters the end of the file, it returns the empty string.

Syntax

file.readline(size)

Parameter	Description
size	It is Optional. The number of bytes from the line to return. Default value is -1, which means the whole line.

Example:

```
f = open("myfile.txt", 'r')
        # assume file has following data
        # First line
        # Second line
while True:
  line = f.readline()
  if line == "":
        break
  print(line)
```

output:

```
First line.
Second line.
```

Readlines( )

The readlines() method returns a list containing each line in the file as a list item.
Use the hint parameter to limit the number of lines returned. If the total number of bytes returned exceeds the specified number, no more lines are returned.

Syntax

file.readlines(hint)

Parameter	Description
hint	Optional. If the number of bytes returned exceed the hint number, no more lines will be returned. Default value of hint is -1, which means all lines will be returned.

Example :

```
f= open("myfile.txt", "r")
        # assume file has following data
        # first line
        # second line
        # third line
        # fourth line
list1=f.readlines(39)
for line in list1:
   print(line,end="")
```

output :

first line
second line
third line

Write( )

The write() method writes a specified text to the file. Where the specified text will be inserted depends on the file mode and stream position.
When we open a file in append mode i.e. "a" mode then the text will be inserted at the current file stream position. The default stream position is at the end of the file.
When we open a file in write mode i.e. "w" mode then the text will be inserted at the current file stream position. The default stream position is beginning of the file i.e 0.
Syntax
file.write(byte)

Parameter	Description
byte	The text or byte object that will be inserted.

Example 1:

```
f = open("myfile.txt", "w")
f.write("\nGood Morning.")
f.close()
```

Note : It creates myfile.txt and write Good Morning at beginning of file.
Example 2:

```
f = open("myfile.txt", "a")
f.write("\nWelcome to python")
f.close()
```

Note : It opens myfile.txt and write Good Morning at end of the file.

Writelines( )

The writelines() method writes the items of a list to the file. Where the text will be inserted depends on the file mode and stream position.
When we open a file in append mode i.e. "a" mode then the text will be inserted at the current file stream position. The default stream position is at the end of the file.
When we open a file in write mode i.e. "w" mode then the text will be inserted at the current file stream position. The default stream position is beginning of the file i.e 0.
Syntax

file.writelines(list)

Parameter	Description
list	The list of texts or byte objects that will be inserted.

Example 1:

```
f = open("myfile.txt", "w")
f.writelines(["\nGood Morning", "\nGood Afternoon",”\nGood Night”])
f.close()
```

Note : It creates myfile.txt and write data at beginning of the file.
Example 2:

```
f = open("myfile.txt", "a")
f.writelines(["\nGood Morning", "\nGood Afternoon",”\nGood Night”])
f.close()
```

Note : It opens myfile.txt and write Good Morning at end of the file.

Manipulating file pointer using seek

Manipulating file pointer using seek

The seek() is used to move the file pointer from one location to another location within the file. When we open a file initially the file pointer position is at 0 i.e. beginning of file. The seek() method sets the current file pointer position at offset value. The seek() method also returns the new file pointer position.

Syntax

file.seek(offset)

Parameter	Description
offset	Offset is the position of the read/write pointer within the file. Offset value is Required. Offset value is a number. Offset value representing the position to set the current file pointer position.

Example :

```
f = open("myfile.txt", "r")
        # assume file has following data
        # one two three four five
f.seek(4)
data=f.read(3)
print( data )
f.seek(0)
data=f.read(3)
print( data )
f.close()
```

output :

```
two
one
```

Tell()

The tell() method returns the current file position in a file stream. Tell() gives the current file pointer position in file. It is a one of file handling function.

We can change the current file position with the seek() method.

Syntax

```
file.tell()
```

Example :-

```
f = open("myfile.txt", "r")
posn= f.tell()
print("Position of file pointer is ",posn)
```

output :

```
Position of file pointer is 0
```

Practicing Programs

1. WAP to read the data from a file using read().

```
file1=open( "myfile.txt","r")
        # myfile.txt has following data
        # This is vvit college.
        # It is located in Naburu.
        # it is a good engineering college
```

```
data=file1.read()
print( data )
output :
        This is vvit college.
        It is located in Naburu.
        it is a good engineering college
```

2. WAP to read the data from a file using readline().

```
file1 = open("tarun.txt", "r")
        # myfile.txt has following data
        # This is vvit college.
        # It is located in Naburu.
        # it is a good engineering college
while True:
   line=file1.readline()
   if line=="":
      break
   print( line,end="" )
file1.close()
output :
        This is vvit college.
        It is located in Naburu.
        it is a good engineering college
```

3. WAP to read the data from a file using readlines().

```
file1 = open("tarun.txt", "r")
        # myfile.txt has following data
        # This is vvit college.
        # It is located in Naburu.
        # it is a good engineering college
lines=file1.readlines(25)
for line in lines:
   print( line,end="" )
file1.close()
output :
        This is vvit college.
        It is located in Naburu.
        it is a
```

4. WAP to print/write data to a file.

```
file1=open("myfile.txt","w")
file1.write("guntur ")
file1.write("andhra pradesh ")
file1.write("india")
file1.close()
print("Data written to myfile.txt")
output :
        Data written to myfile.txt
```

5. WAP to print all palindrome words of a file.

```
def palindrome( str ):
   if str==str[::-1]:
      print( str )
file1=open("myfile.txt","r")
```

```
        # file has the following data
        # dad uses liril soap
data=file1.read()
words=data.split()
for word in words:
   palindrome(word)
output :
        dad
        liril
```

6. WAP to print whether the file contains unique elements or not.

```
file1=open("myfile.txt","r")
        #file has following data
        # 10 20 30 40 50 60
data=file1.read()
list1=data.split(", ")
s1=set( list1 )
if len( list1 ) == len(s1) :
   print("File contains all unique elements")
else:
   print("File contains some duplicate elements")
output :
        File contains all unique elements
```

7. WAP to copy a file to another file.

```
file1=open("myfile.txt","r")
file2=open("Mahidhar.txt","w")
data=file1.read()
file2.write( data )
file1.close()
file2.close()
print("file copied")
output :
        file copied
```

8. WAP to merge 2 files.

```
file1=open("myfile.txt","r")
file2=open("Mahidhar.txt","r")
file3=open("Dutt.txt","w")
data=file1.read()
file3.write( data )
data=file2.read()
file3.write( data )
file1.close()
file2.close()
file3.close()
print("Files merged")
output :
        Files merged
```

9. WAP to count number of words in a file.

```
file1=open("myfile.txt","r")
        # file has following data
        # rama is a good boy.
```

```
        # Rama passed X class in I class.
data=file1.read()
list1=data.split()
print("No of words=",len( list1 ))
output :
        No of words= 12
```

10. WAP to find longest word of a file.

```
file1=open("myfile.txt","r")
        # file has following data
        # rama is a good boy.
        # Rama passed X class in 1st grade.
data=file1.read()
list1=data.split()
longestword=""
length=0
for word in list1:
    if len(word)>length:
        longestword=word
        length=len(word)
print("Longest word=",longestword)
output :
        Longest word= passed
```

11. WAP to find longest line of the file.

```
file1=open("myfile.txt","r")
        # file has following data
        # rama is a cricket player.
        # ramana is also a cricket player.
        # sita is a tennis player.
length=0
longestline=""
for line in file1:
    if len( line )>length:
        longestline=line
        length=len( line )
print("longest line is : ",longestline)
output :
        longest line is :  ramana is also a cricket player.
```

12. WAP to separate vowels , consonants and digits of a file.

```
file1=open("myfile.txt","r")
vowels=open("Vowels.txt","w")
cons=open("Consonants.txt","w")
digs=open("digits.txt","w")
data=file1.read()
for ch in data:
    if ch>='0' and ch<='9':
        digs.write( ch )
    else:
        if ch in "AEIOUaeiou":
            vowels.write( ch )
        else:
```

```
            cons.write( ch )
    print("Data separated")
    file1.close()
    vowels.close()
    cons.close()
    digs.close()
    output :
            Data separated
```

Object Oriented Programming OOP

Python has been an **object-oriented language** since it existed. Because of this, creating and using classes and objects are downright easy. This chapter helps you become an expert in using Python's object-oriented programming support.

However, here is small introduction of Object-Oriented Programming (**OOP**):

Overview of OOP Terminology

- **Class:** A user-defined prototype for an object that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.
  - **Class variable**: A variable that is shared by all instances of a class. Class variables are defined within a class but outside any of the class's methods. Class variables are not used as frequently as instance variables are.
- **Instance variable:** A variable that is defined inside a method and belongs only to the current instance of a class.
- **Data member:** Data member may be a class variable or instance variable that holds data associated with a class and its objects.
- **Method:** A special kind of function that is defined in a class definition.
- **Object :** A unique instance of a data structure that's defined by its class. An object comprises both data members (class variables and instance variables) and methods. for example,C is an instance of the class Circle.
- **Instantiation:** It is a process of creation of an instance of a class.
- **Inheritance:** The transfer of the characteristics of a class to other classes that are derived from it.
- **Function overloading:** The assignment of more than one behaviour to a particular function. The operation performed varies the types of objects or arguments involved.
- **Operator overloading:** The assignment of more than one function to a an operator.

Creation a Class

A class can be created using the keyword **class. Class** must follow the ClassName. ClassName must follow the ':' symbol. The simplest form of class definition is like this:

Syntax :

```
class ClassName:
        <statement-1>
                ……
         <Statement-N>
```

Example:

```
class Student:
        def __init__(self):
            self.name="hari"
            self.branch="CSE"
        def display(self):
            print(self.name)
            print(self.branch)
```

- For example, In "Student" class, we can write code which specifies the attributes and actions performed by any student.

- Observer that the keyword **class** is used to declare a class. After this, we should write the class name. So, "Student" is our class name. Generally, a class name should start with a capital letter; hence **"S"** is a capital in "Student".
- In the class, we have written the variables and methods. Since in python, we cannot declare variables, we have written the variables inside a special method, i.e. __init__(). This method is used to initialize the variables. Hence the name **"init"**.
- The method name has two underscores before and after. This indicates that this method is internally defined and we cannot call this method explicitly. It is called magic function.
- Observe the parameter **"self"** written after the method name in the parentheses. **"self"** is a variable that refers to current class instance.
- When we create an instance for the Student class, a separate memory block is allocated on the heap and that memory location is stored in **"self"**.
- The variables **"name"** and **"branch"** are called instance variables. To refer to instance variables, we can use the dot operator notation along with self as **"self.name"** and **"self.branch"**.
- The method display ( ) also takes the **"self"** variable as parameter. This method displays the values of variables by referring them using **"self"**.
- The methods that act on instances/objects of a class are called instance methods. Instance methods use **"self"** as the first parameter that refers to the instance.
- Writing a class like this is not sufficient. It should be used. To use a class, we should create an instance to the class. Instance creation allots memory necessary to store the actual data of the variables, i.e., **"hari", "CSE"**.

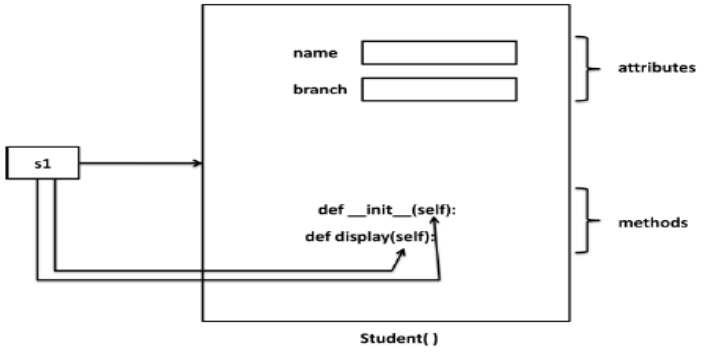
Instance / Object

- Instance of a class is called object.
- Object is the real world thing which we can see around us.
- To create an instance, the following syntax is used:

  InstanceName = ClassName( )
- So, to create an instance to the Student class, we can write as:

  s1 = Student ( )
- Here **"s1"** represents the instance name. When we create an instance like this, the following steps will take place internally:

1. First of all, a block of memory is allocated on heap. How much memory is to be allocated is decided from the attributes and methods available in Student class.
2. After allocating the memory, the special method by the name **"__init__(self)"** is called internally. This method stores the initial data into the variables. Since this method is useful to construct the instance, it is called **"constructor"**.
3. Finally, the allocated memory location address of the instance is returned into **"s1"** variable. To see this memory location in decimal number format, we can use id( ) function as id(s1).



Self variable

"Self" is a default variable that contains the memory address of the instance of the current class. When an instance to the class is created, the instance name contains the memory location of the instance. This memory location is internally passed to **"self"**.

For example, we create an instance to student class as:

```
s1 = Student( )
```

Here, **"s1"** contains the memory address of the instance. **"self"** can refer to all the members of the instance.

We use **"self"** in two Ways:

- The self variable is used as first parameter in the constructor as:

```
def __init__(self):
```

In thi s case, **"self"** can be used to refer to the instance variables inside the constructor.

- **"self"** can be used as first parameter in the instance methods as:

```
def display(self):
```

Here, **display( )** is instance method as it acts on the instance variables. If this method wants to act on the instance variables, it should know the memory location of the instance variables. That memory location is by default available to the **display( )** method through **"self".**

Constructor

A constructor is a special method that is used to initialize the instance variables of a class. In the constructor, we create the instance variables and initialize them with some starting values. The first parameter of the constructor will be **"self"** variable that contains the memory address of the instance.

```
def __init__( self ):
    self.name = "hari"
    self.branch = "CSE"
```

Here, the constructor has only one parameter, i.e. **"self".** We can access instance variables of a class as **"self.name"** and **"self.branch"**. A constructor is called at the time of creating an instance. So, the above constructor will be called whenever we create an instance as:

```
s1 = Student( )
```

Let's take another example, we can write a constructor with some parameters in addition to **"self"** as:

```
def __init__( self , n = " " , b = " " ):
    self.name = n
    self.branch = b
```

Here, the formal arguments are **"n"** and **"b"** whose default values are given as **""** (None) and **""** (None). Hence, if we do not pass any values to constructor at the time of creating an instance, the default values of those formal arguments are stored into name and branch variables. For example,

```
s1 = Student( )
```

Since we are not passing any values to the instance, None and None are stored into name and branch. Suppose, we can create an instance as:

```
s1 = Student( “mothi”, “CSE”)
```

In this case, we are passing two actual arguments: **“mothi”** and **“CSE”** to the Student instance.

Example:

```
class Student:
    def __init__(self,n=' ',b=' '):
        self.name=n
        self.branch=b
    def display(self):
        print("Hi",self.name)
        print("Branch", self.branch)
s1=Student()
s1.display()
print("_____")
s2=Student("mothi","CSE")
s2.display()
print("_____")
```

Output:

```
Hi
Branch
------------------------------
Hi mothi
Branch CSE
------------------------------
```

Types of Variables

The variables which are written inside a class are of 2 types:

a) Instance Variables
b) Class Variables or Static Variables

a) Instance Variables

Instance variables are the variables whose separate copy is created for every instance. For example, if **“x”** is an instance variable and if we create 3 instances, there will be 3 copies of **“x”** in these 3 instances. When we modify the copy of **“x”** in any instance, it will not modify the other two copies.

Example:

```
class Sample:
    def __init__(self):
        self.x = 10
    def modify(self):
        self.x = self.x + 1
s1=Sample()
s2=Sample()
print("x in s1=",s1.x)
print("x in s2=",s2.x)
print("----------------")
s1.modify()
print("x in s1=",s1.x)
print ("x in s2=",s2.x)
print("----------------")
```

Output:

```
x in s1= 10
x in s2= 10
----------------
x in s1= 11
x in s2= 10
----------------
```

Instance variables are defined and initialized using a constructor with **“self”** parameter. Also, to access instance variables, we need instance methods with **“self”** as first parameter. It is possible that the instance methods may have other parameters in addition to the **“self”** parameter. To access the instance variables, we can use **self.variable** as shown in program. It is also possible to access the instance variables from outside the class, as: instancename.variable, **e.g. s1.x.**

b) Class Variables or Static Variables

Class variables are the variables whose single copy is available to all the instances of the class. If we modify the copy of class variable in an instance, it will modify all the

copies in the other instances. For example, if **"x"** is a class variable and if we create 3 instances, the same copy of **"x"** is passed to these 3 instances. When we modify the copy of **"x"** in any instance using a class method, the modified copy is sent to the other two instances.

Example:

```
class Sample:
   x=10
   @classmethod
   def modify(cls):
      cls.x = cls.x + 1
s1=Sample()
s2=Sample()
print("x in s1=",s1.x)
print("x in s2=",s2.x)
print("----------")
s1.modify()
print("x in s1=",s1.x)
print("x in s2=",s2.x)
print("----------")
```

Output:

```
x in s1= 10
x in s2= 10
----------
x in s1= 11
x in s2= 11
----------
```

Destructor

The users call Destructor for destroying the object. In Python, developers might not need destructors as much it is needed in the C++ language. This is because Python has a garbage collector whose function is handling memory management automatically.

The **__del__()** function is used as the destructor function in Python. The user can call the **__del__()** function when all the references of the object have been deleted, and it becomes garbage collected.

Syntax :

```
def __del__(self):
   # the body of destructor will be written here.
```

In the following example, we will use the __del__() function and the del keyword for deleting all the references of the object so that the destructor will involve automatically.

Example

```
class Player:
        def __init__(self):
                self.pname="Dhoni"
                self.game="Cricket"
        def show(self):
                print("Player name=",self.pname)
                print("Game name=",self.game)
        def __del__(self):
                print('The destructor is called for deleting the properties of player.')
p1 = Player()
p1.show()
del p1
output :
--------------------------
Player name= Dhoni
Game name= Cricket
The destructor is called for deleting the Animals.
```

Note :
The destructor is called after the ending of the program.

```
Example
   class Player:
            def __init__(self):
                    self.pname=”Dhoni”
                    self.game=”Cricket”
            def show(self):
                    print(“Player name=”,self.pname)
                    print(“Game name=”,self.game)
            def __del__(self):
                    print('The destructor is called for deleting the Animals.')
     p1 = Player()
     p1.show()
     output :
     -------------------------
     Player name= Dhoni
     Game name= Cricket
     The destructor is called for deleting the Animals.
```

Inheritance

- Software development is a team effort. Several programmers will work as a team to develop software.
- When a programmer develops a class, he will use its features by creating an instance to it. When another programmer wants to create another class which is similar to the class already created, then he need not create the class from the scratch. He can simply use the features of the existing class in his own new class.
- Deriving new class from the super class is called inheritance.
- The child class inherits the attributes of its parent class, and you can use those attributes as if they were defined in the child class.
- A child class can also override data members and methods from the parent.

```
Syntax:
class Subclass(BaseClass):
        <class body>
```

- When an object is to SubClass is created, it contains a copy of BaseClass within it. This means there is a relation between the BaseClass and SubClass objects.
- We do not create BaseClass object,but still a copy of it is available to SubClass object.
- By using inheritance, a programmer can develop classes very easilt. Hence programmer’s productivity is increased. Productivity is a term that refers to the code developed by the programmer in a given span of time.
- If the programmer used inheritance, he will be able to develop more code in less time.
- In inheritance, we always create only the sub class object. Generally, we do not create super class object. The reason is clear. Since all the members of the super class are available to sub class, when we crate an object, we can access the members of both the super and sub classes.
  - **Parent class** is a class being inherited from, also called base class or super class.

- **Child class** is a class that inherits from another class, also called derived class or sub class.

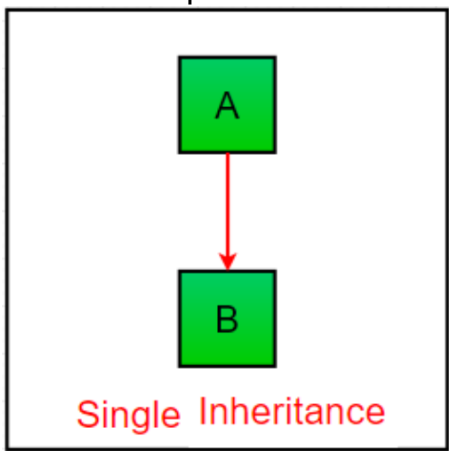
Types of Inheritance

Inheritances are mainly classified into 5 types.

a) Single inheritance
b) Multiple inheritance
c) Multi Level Inheritance
d) Heirarchical inheritance
e) Hybrid inheritance

1) Single inheritance

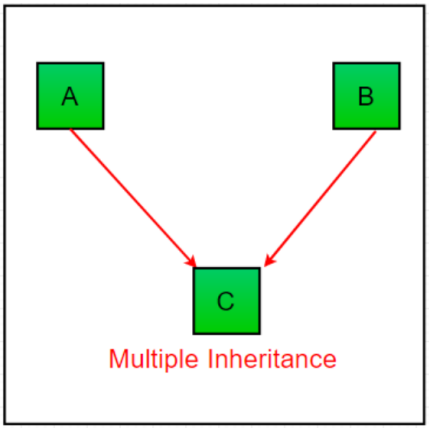
Single inheritance enables a derived class to inherit properties from a single parent class, thus enabling code reusability and the addition of new features to existing code. It is also called as simple inheritance. It must be built with maximum and minimum of 2 classes.




```
class Subject:
    def __init__(self):
                self.scode="R191104"
                self.sname="Python"
    def show(self):
                print("Sub Code=",self.scode)
                print("Sub Name=",self.sname)
class Teacher(Subject):
    def __init__(self):
                Subject.__init__(self)
                self.tname="Sudhakar"
                self.tqua="M.Tech"
    def disp(self):
                print("Teacher=",self.tname)
                print("Qualification=",self.tqua)
t=Teacher()
t.show()
t.disp()
```

2. Multiple inheritance :

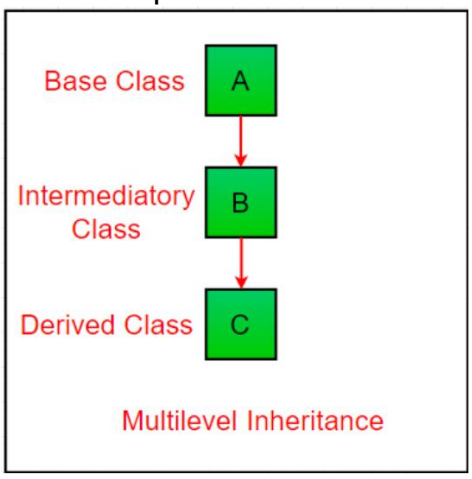
When a class can be derived from more than onebase class this type of inheritance is called multiple inheritance. In multiple inheritance, all the features of the base classes are inherited into the derived class. The classes has many to one relationship.




```
class Father:
    def __init__(self):
        self.fname="Rajesh"
        self.foccu="Doctor"
    def show(self):
        print("Father name=",self.fname)
        print("Father occupation=",self.foccu)
class Mother:
    def __init__(self):
        self.mname="Suhasini"
        self.moccu="Teacher"
    def disp(self):
        print("Mother name=",self.mname)
        print("Mother occupation=",self.moccu)
class Child(Mother,Father):
    def __init__(self):
        Father.__init__(self)
        Mother.__init__(self)
        self.cname="Ramana"
        self.coccu="Student"
    def display(self):
        print("Child name=",self.cname)
        print("Child Occupation=",self.coccu)
c=Child()
c.show()
c.disp()
c.display()
```

3. Multi Level Inheritance

In multilevel inheritance, features of the base class and the derived class are further inherited into the new derived class. It is extension to single inheritance. This is similar to a relationship representing a child and grandfather. The classes has one to one relationship. It can be build with minimum of 3 classes.

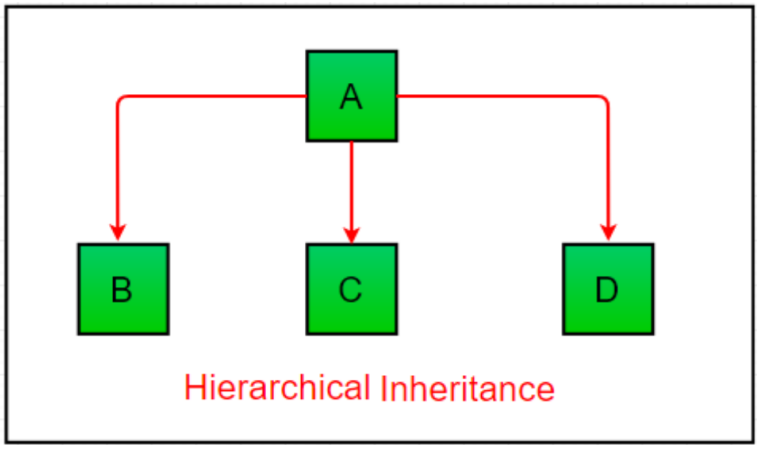


Here the class A is super class to B. Class B is super class to C. Class A is indirect super class to C. Class C is sub class to B. Class B is sub class to A. Class C is indirect sub class to A. the class B acts as super class to C and sub class to A.

```
class College:
    def __init__(self):
                self.cname="VVIT"
                self.cadd="Namburu"
    def show(self):
                print("College=",self.cname);
                print("Address=",self.cadd)
class Principal(College):
    def __init__(self):
                College.__init__(self)
                self.pname="Mallikarjuna Reddy"
                self.pqua="Ph.D"
    def disp(self):
                print("Principal=",self.pname)
                print("Qualification=",self.pqua)
class HOD(Principal):
    def __init__(self):
                Principal.__init__(self)
                self.hname="Eswar"
                self.dept="CSE"
    def display(self):
                print("HOD=",self.hname)
                print("Dept=",self.dept)
h=HOD()
h.show()
h.disp()
h.display()
```

4. Heirarchical inheritance

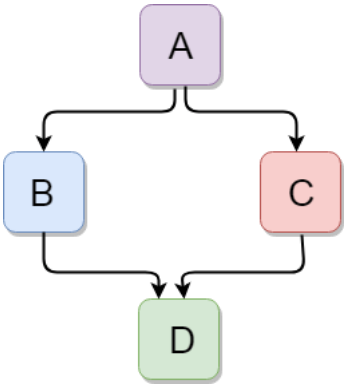
When more than one derived classes are created from a single base this type of inheritance is called hierarchical inheritance. In this program, we have a parent (base) class and two or more child (derived) classes. The classes has one to many relationship. It can be build with minimum of 3 classes. It is looking like an inverted tree shape. It has family members relationship.



```
class Subject:
    def __init__(self):
                self.subcode="R191205"
                self.subname="Data structures"
     def show(self):
                print( "Subject Code=",self.subcode)
                print("Subject Name=",self.subname)
class Internals(Subject):
    def __init__(self):
                Subject.__init__(self)
                self.max=30
                self.marks=28
    def disp(self):
                print("Max marks in internals",self.max)
                print("Marks got in internals",self.marks)
class Externals(Subject):
    def __init__(self):
                Subject.__init__(self)
                self.max=70
                self.marks=60
    def display(self):
                print("Max marks in Externals",self.max)
                print("Marks got in externals",self.marks)
obj1=Internals()
obj2=Externals()
obj1.show()
obj1.disp()
obj2.display()
print("Total marks=",obj1.marks + obj2.marks)
```

5. Hybrid Inheritance :-

Hybrid inheritance is a combination of simple, multiple inheritance and hierarchical inheritance. Usually, in multiple inheritances, a class is derived from two classes where one of the parent classes is also a derived class and not a base class. Hybrid inheritance is called as multipath inheritance. It is the process of deriving a class using more than one level or more than one mode of inheritance.



```
class Subject:
    def __init__(self):
                self.subcode="R191205"
                self.subname="Data structures"
     def show(self):
                print( "Subject Code=",self.subcode)
                print("Subject Name=",self.subname)

class Internals(Subject):
    def __init__(self):
                Subject.__init__(self)
                self.max=30
                self.marks=28
    def disp(self):
                print("Max marks in internals",self.max)
                print("Marks got in internals",self.marks)

class Externals(Subject):
    def __init__(self):
                Subject.__init__(self)
                self.max=70
                self.marks=60
    def display(self):
                print("Max marks in Externals",self.max)
                print("Marks got in externals",self.marks)

class Student(Internals,Externals):
    def __init__(self):
                Internals.__init__(self)
                Externals.__init__(self)
                self.rno=501
                self.sname="Gopi"
    def printing(self):
                print("Rno=",self.rno)
                print("Student=",self.sname)
s=Student()
s.show()
s.disp()
s.display()
s.printing()
```

Special Functions

Class functions that begin with double underscore __ are called special functions in Python.
These functions are not the typical functions that we define for a class. The __init__() function we defined is one of them. It gets called every time we create a new object of that class. There are numerous other special functions in Python.
Using special functions, we can make our class compatible with built-in functions.
Example Program:

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

p1 = Point(1, 2)
print(p1)
```

Output:

```
<__main__.Point object at 0x00000000031F8CC0>
```

Suppose we want the print() function to print the coordinates of the Point object instead of what we got. We can define a __str__() method in our class that controls how the object gets printed. Let's look at how we can achieve this:

```
class Point:
    def __init__(self, x = 0, y = 0):
        self.x = x
        self.y = y

    def __str__(self):
        return "({0},{1})".format(self.x,self.y)

p1 = Point(2, 3)
print(p1)
```

Output:

```
(2, 3)
```

Operator Overloading

Consider that we have two objects which are a physical representation of a class (user-defined data type) and we have to add two objects with binary '+' operator it throws an error, because compiler don't know how to add two objects. So we define a method for an operator and that process is called operator overloading. We can overload all existing operators but we can't create a new operator. To perform operator overloading, Python provides some special function or magic function that is automatically invoked when it is associated with that particular operator. For example, when we use + operator, the magic method __add__ is automatically invoked in which the operation for + operator is defined.

```
class A:
        def __init__(self, no):
                self.no = no

        # adding two objects
        def __add__(self, other):
                return self.no + other.no
ob1 = A(1)
ob2 = A(2)
ob3 = A("Rama")
ob4 = A("Rao")

print(ob1 + ob2)
print(ob3 + ob4)
output :
        3
        RamaRao
```

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __str__(self):
        return "({0},{1})".format(self.x, self.y)

    def __lt__(self, other):
        self_m = (self.x ** 2) + (self.y ** 2)
        other_m = (other.x ** 2) + (other.y ** 2)
        return self_m < other_m

p1 = Point(1,1)
p2 = Point(-2,-3)
p3 = Point(1,-1)
# use less than
print(p1<p2) #True
print(p2<p3) #False
print(p1<p3) #False
```

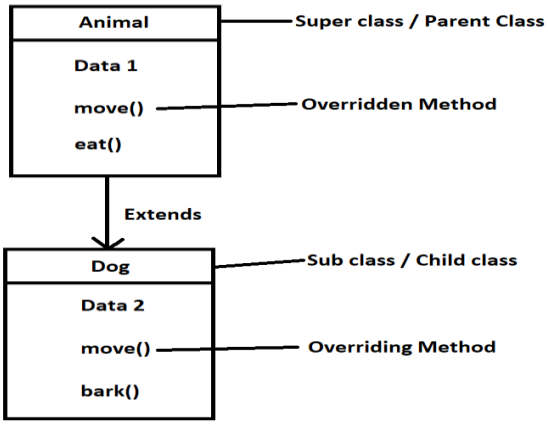
Some special functions for other operators:

Operator	Expression	Internally
Addition	p1 + p2	p1.__add__(p2)
Subtraction	p1 - p2	p1.__sub__(p2)
Multiplication	p1 * p2	p1.__mul__(p2)
Power	p1 ** p2	p1.__pow__(p2)
Division	p1 / p2	p1.__truediv__(p2)
Floor Division	p1 // p2	p1.__floordiv__(p2)
Remainder (modulo)	p1 % p2	p1.__mod__(p2)
Bitwise Left Shift	p1 << p2	p1.__lshift__(p2)
Bitwise Right Shift	p1 >> p2	p1.__rshift__(p2)
Bitwise AND	p1 & p2	p1.__and__(p2)
Bitwise OR	p1 | p2	p1.__or__(p2)
Bitwise XOR	p1 ^ p2	p1.__xor__(p2)
Bitwise NOT	~p1	p1.__invert__()
Less than	p1 < p2	p1.__lt__(p2)
Less than or equal to	p1 <= p2	p1.__le__(p2)
Equal to	p1 == p2	p1.__eq__(p2)

Not equal to	p1 != p2	p1.__ne__(p2)
Greater than	p1 > p2	p1.__gt__(p2)
Greater than or equal to	p1 >= p2	p1.__ge__(p2)

Method Overriding / Polymorphism

Method overriding is an ability of any object-oriented programming language that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes.
It is a process of defining a method in sub class which is as same as the method of super class. The method of super class overrides the method of super class. The method of sub class is called overriding method. the method of super class is called overridden method.




```
class School:
    def __init__(self):
        self.sname="Saraswathi School"
        self.sadd="Prakash Nagar"
    def show(self):
        print("School name=",self.sname)
        print("School address=",self.sadd)

class Teacher(School):
    def __init__(self):
        School.__init__(self)
        self.tname="Prasad"
        self.subname="Maths"
    def show( self ):
        School.show(self)
        print("Teacher name=",self.tname)
        print("Subject name=",self.subname)
t1=Teacher()
t1.show()
output :
        School name=Saraswathi School
        School address=Prakash Nagar
        Teacher name=Prasad
        Subject name=Maths
```

Adding and retrieving Dynamic Attributes of classes

Dynamic attributes in Python are terminologies for attributes that are **defined at runtime**, after creating the objects or instances. In Python we call all functions, methods also as an object. So you can define a dynamic instance attribute for nearly anything in Python.

Dynamic attribute of a class object:

```
class College:
    pass

c1=College()
c1.cname="VVIT"
print( "College name=",c1.cname )
output :
        College name=VVIT
```

Note :
Here c1 is an object. Dynamic attribute of c1 is cname. This is defined at runtime and not at compile time like static attributes. The class "College" and all other objects or instances of this class do not know the attribute "cname". It is only defined for the instance "c1".

Dynamic attribute of a function:

```
def show():
        pass

show.cname="VVIT"
print( "College name=",show.cname )

output :
            College name=VVIT
```

Note :
Here show() is a function. Functions are also considered as objects in python. So show() is an object. Dynamic attribute of show() is cname. This is defined at runtime and not at compile time like static attributes.

Reading config files in python:

Configuration files are well suited to specify configuration data to your program. Within each config file, values are grouped into different sections (e.g., “installation”, “debug” and “server”).

Each section then has a specific value for various variables in that section. For the same purpose, there are some prominent differences between a config file and using a Python source file.

The contrast between a config record and Python code is that, in contrast to scripts, configuration files are not executed in a top-down way. Rather, the file is read completely. On the off chance that variable substitutions are made, they are done later after the fact. For instance, it doesn’t make a difference that the prefix variable is allocated after different variables that happen to utilize it.
Example:
Code #1 : Configuration File (Have to be saved with .ini extension): config.ini

```
; Sample configuration file
[installation]
library = %(prefix)s/lib
include = %(prefix)s/include
bin = %(prefix)s/bin
prefix = /usr/local

# Setting related to debug configuration
[debug]
pid-file = /tmp/spam.pid
show_warnings = False
log_errors = true
[server]
nworkers: 32
port: 8080
root = /www/root
signature:
```

Code #2 : Reading the file and extracting values.

```
from configparser import ConfigParser

configur = ConfigParser()
print (configur.read('config.ini'))

print ("Sections : ", configur.sections())
print ("Installation Library : ", configur.get('installation','library'))
print ("Log Errors debugged ? : ", configur.getboolean('debug','log_errors'))
print ("Port Server : ", configur.getint('server','port'))
print ("Worker Server : ", configur.getint('server','nworkers'))
```

Output:
['config.ini']
Sections : ['installation', 'debug', 'server']
Installation Library : '/usr/local/lib'
Log Errors debugged ? : True
Port Server : 8080
Worker Server : 32

We can modify the configuration and write it back to a file using the cfg.write() method.

Code #3 :

```
configur.set('server','port','9000')
configur.set('debug','log_errors','False')

import sys
configur.write(sys.stdout)
```

Output :
[installation]
library = %(prefix)s/lib
include = %(prefix)s/include
bin = %(prefix)s/bin
prefix = /usr/local

[debug]
log_errors = False
show_warnings = False

[server]
port = 9000
nworkers = 32
pid-file = /tmp/spam.pid
root = /www/root

Writing log files in python:

Python has a built-in module *logging* which allows writing status messages to a file or any other output streams. The file can contain the information on which part of the code is executed and what problems have been arisen.

There are built-in levels of the log message.

- Debug : These are used to give Detailed information, typically of interest only when diagnosing problems.
- Info : These are used to Confirm that things are working as expected
- Warning : These are used an indication that something unexpected happened, or indicative of some problem in the near future
- Error : This tells that due to a more serious problem, the software has not been able to perform some function

- Critical : This tells serious error, indicating that the program itself may be unable to continue running

Each built-in level has been assigned its numeric value.

Level	Numeric Value
NOTSET	0
DEBUG	10
INFO	20
WARNING	30
ERROR	40
CRITICAL	50

Logging module is packed with several features. It has several constants, classes, and methods. The items with all caps are constant, the capitalize items are classes and the items which start with lowercase letters are methods.
There are several logger objects offered by the module itself.

- Logger.info(msg) : This will log a message with level INFO on this logger.
- Logger.warning(msg) : This will log a message with level WARNING on this logger.
- Logger.error(msg) : This will log a message with level ERROR on this logger.
- Logger.critical(msg) : This will log a message with level CRITICAL on this logger.
- Logger.log(lvl,msg) : This will Logs a message with integer level lvl on this logger.
- Logger.exception(msg) : This will log a message with level ERROR on this logger.
- Logger.setLevel(lvl) : This function sets the threshold of this logger to lvl. This means that all the messages below this level will be ignored.
- Logger.addFilter(filt) : This adds a specific filter filt to the to this logger.
- Logger.removeFilter(filt) : This removes a specific filter filt to the to this logger.
- Logger.filter(record) : This method applies the logger's filter to the record provided and returns True if record is to be processed. Else, it will return False.
- Logger.addHandler(hdlr) : This adds a specific handler hdlr to the to this logger.
- Logger.removeHandler(hdlr) : This removes a specific handler hdlr to the to this logger.
- Logger.hasHandlers() : This checks if the logger has any handler configured or not.

The Basics: Basics of using the logging module to record the events in a file are very simple. For that, simply import the module from the library.

- Create and configure the logger. It can have several parameters. But importantly, pass the name of the file in which you want to record the events.
- Here the format of the logger can also be set. By default, the file works in append mode but we can change that to write mode if required.
- Also, the level of the logger can be set which acts as the threshold for tracking based on the numeric values assigned to each level.
- There are several attributes which can be passed as parameters.
- The list of all those parameters is given in Python Library. The user can choose the required attribute according to the requirement.

After that, create an object and use the various methods as shown in the example.

```
#importing module
import logging

#Create and configure logger
logging.basicConfig(filename="newfile.log",
                    format='%(asctime)s %(message)s',
                    filemode='w')

#Creating an object
logger=logging.getLogger()

#Setting the threshold of logger to DEBUG
logger.setLevel(logging.DEBUG)

#Test messages
logger.debug("Harmless debug Message")
logger.info("Just an information")
logger.warning("Its a Warning")
logger.error("Did you try to divide by zero")
logger.critical("Internet is down")
```

Output: (Stored in file)

```
2021-08-23 09:18:41,666 Harmless debug Message
2021-08-23 09:18:41,666 Just an information
2021-08-23 09:18:41,666 Its a Warning
2021-08-23 09:18:41,666 Did you try to divide by zero
2021-08-23 09:18:41,666 Internet is down
```