5...

Object Oriented Programming in Python

Chapter Outcomes...

- Create Classes and Objects to solve the given problem.
- Write Python code for data hiding for the given problem.
- Write Python code using data abstraction for the given problem.
- Write Python program using Inheritance for the given problem.

Learning Objectives...

- To learn Object Oriented Programming Concepts in Python programming.
- To Creating Classes and Objects in Python
- To learn Method Overloading, Method Overriding, Data Hiding, Data Abstraction, Inheritance etc.

5.0 INTRODUCTION

- Python is an Object-Oriented Programming Language (OOPL) follows an Object-Oriented Programming (OOP) paradigm. It deals with declaring Python classes and objects which lays the foundation of OOPs concepts.
- Python programming offers OOP style programming and provides an easy way to develop programs. Python programming uses the OOPs concepts that makes Python more powerful to help design a program that represents real-world entities.
- Python also supports OOP concepts such as Inheritance, Method overriding, Data abstraction and Data hiding.

Important terms in OOP/Terminology of OOP:

1. **Class:** Classes are defined by the user. The class provides the basic structure for an object. It consists of data members and method members that are used by the instances, (objects) of the class.
2. **Object:** A unique instance of a data structure that is defined by its class. An object comprises both data members (class variables and instance variables) and methods. Class itself does nothing but the real functionality is achieved through their objects. Object is an instance or occurrence of the class. It takes the properties (variables) and uses the behavior (methods) defined in the class.
3. **Data Member:** A variable defined in either a class or an object; it holds the data associated with the class or object.
4. **Instance Variable:** A variable that is defined in a method; it scope is only within the object that defines it.

5. **Class Variable:** A variable that is defined in the class and can be used by all the instances of that class.
6. **Instance:** An object is an instance of the class.
7. **Instantiation:** The process of creation on an object of a class.
8. **Method:** Methods are the functions that are defined in the definition of class and are used by various instances of the class.
9. **Function Overloading:** A function defined more than one time with different behavior's is known as function overloading. The operation performed varies by the types of objects or arguments involved.
10. **Encapsulation:** Encapsulation is the process of binding together the methods and data variables as a single entity i.e., class. This keeps both the data and functionality code safe from the outside world. It hides the data within the class and makes it available only through the methods.
11. **Inheritance:** The transfer of the characteristics of a class to other classes that are derived from it. A class 'A' that can use the characteristics of another class 'B' is said to be derived class i.e. a class inherited from B. This process is called inheritance.
12. **Polymorphism:** Polymorphism allows one interface to be used for a set of actions i.e., one name may refer to different functionality. The word polymorphism means having many forms. In programming, polymorphism means same function name (but different signatures) being uses for different types.
13. **Data Abstraction**: The basic idea of data abstraction is to visible only the necessary information, unnecessary information will be hidden from the outside world. Abstraction is a process of hiding the implementation details and showing only functionality to the user. Another way, it shows only essential things to the user and hides the internal details, for example, sending SMS where we type the text and send the message. We don't know the internal processing about the message delivery.

5.1 CLASSES

- Python is an object oriented programming language. Almost everything in Python is an object, with its properties and methods.
- Object is simply a collection of data (variables) and methods (functions) that act on those data.
- A class is like an object constructor or a "blueprint" for creating objects. A class defines the properties and behavior (variables and methods) that is shared by all its objects.

5.1.1 Creating Classes

- A class is a block of statements that combine data and operations, which are performed on the data, into a group as a single unit and acts a blueprint for the creation of objects.
- To create a 'class', use the keyword class. Here's the very basic structure of python class definition.

  Syntax:

```
class ClassName:
    'Optional class documentation string'
    # list of python class variables
    # python class constructor
    # python class method definitions
```

- Following is an example of creation of an empty class:

```
class Car:
 pass
```

  Here, the pass statement is used to indicate that this class is empty.
- In a class we can define variables, functions 'tc'. While writing any function in class we have to pass atleast one argument a that is called self Parameter.

- The self parameter is a reference to the class itself and is used to access variables that belongs to the class. It does not have to be named self, we can call it whatever we like, but it has to be the first parameter of any function in the class.
- We can write a class on interactive interpreter or in a.py file

Class on Interactive Interpreter:

```
>>> class student:
    def display(self):          # defining method in class
        print("Hello Python")
```

Class in a .py file:

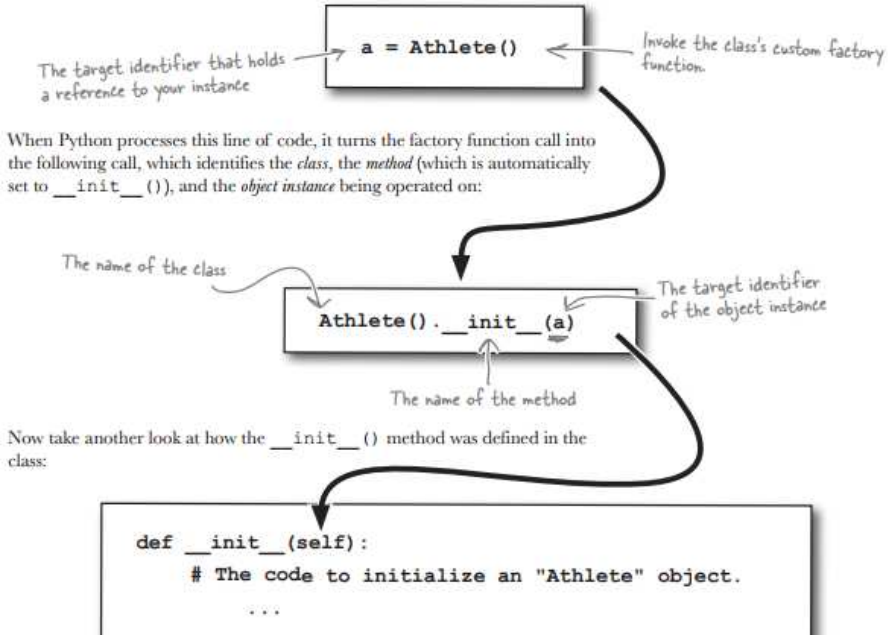
```
class student:
 def display(self):             # defining method in class
 print("Hello Python")
```

The Importance of Self:

- In Python programming self is a default variable that contains the memory address of the instance of the current class. So we can use self to refer to all the instance variables and instance methods.
- **To confirm:** When we define a class we are, in effect, defining a custom factory function that we can then use in the code to create instances:

- When Python processes this line of code, it turns the factory function call into the following call, which identifies the class, the method (which is automatically set to _ _init _ _()) and the object instance being operated on:

- Now take another look at how _ _ int _ _ () method was defined in the class:



5.1.2 Objects and Creating Objects

- An object is an instance of a class that has some attributes and behavior.
- Objects can be used to access the attributes of the class.

Syntax: `obj_name=class_name()`

Example:

```
s1=student()
s1.display()
```

- Complete program with class and objects on interactive interpreter is given below:

```
>>> class student:
    def display(self):          # defining method in class
        print("Hello Python")
>>> s1=student()               # creating object of class
>>> s1.display()               # calling method of class using object
Hello Python
```

- Complete program with class and objects on interactive interpreter in .py file is given below:

```
class student:
 def display(self):
 print("Hello Python")
s1=student()
s1.display()
```

Output:

```
Hello Python
```

Example : Class with get and put method.

```
class Car:
 def get(self, color, style):
 self.color = color
 self.style = style
 def put(self):
 print(self.color)
 print(self.style)
c = Car()
c.get('Sedan', 'Black')
c.put()
```

Output:

```
Sedan
Black
```

5.1.3 Instance Variable and Class Variable

- Instance variable is defined in a method and its scope is only within the object that defines it. Instance attribute is unique to each object (instance). Every object of that class has its own copy of that variable. Any changes made to the variable don't reflect in other objects of that class.
- Class variable is defined in the class and can be used by all the instances of that class. Class attribute is same for all objects. And there's only one copy of that variable that is shared with all objects. Any changes made to that variable will reflect in all other objects.
- Instance variables are unique for each instance, while class variables are shared by all instances. Following example demonstrates the use of instance and class variable.

Example: For instance and class variables.

```
class Sample:
 x = 2                      # x is class variable
 def get(self, y):          # y is instance variable
 self.y = y
s1 = Sample()
s1.get(3)                   # Access attributes
print(s1.x ," ",s1.y)
s2 = Sample()
s2.y=4                      # Modify attribute
print(s2.x," ",s2.y)
```

Output:

```
2 3
2 4
```

5.2 DATA HIDING

- Data hiding is a software development technique specifically used in Object-Oriented Programming (OOP) to hide internal object details (data members).
- Data hiding ensures exclusive data access to class members and protects object integrity by preventing unintended or intended changes.
- Data hiding is also known as information hiding. An object's attributes may or may not be visible outside the class definition.
- We need to name attributes with a double underscore (_ _) prefix, and those attributes then are not be directly visible to outsiders. Any variable prefix with double underscore is called private variable which is accessible only with class where it is declared.

Example: For data hinding.

```
class Counter:
 _ _secretCount = 0 #private variable
 def count(self): #public method
 self._ _secretCount += 1
 print ("count=",self._ _secretCount) #can be accessible in the same class

c1= Counter()
c1.count() #invoke method
c1.count()
print ("Total count=",c1._ _secretCount) #cannot access private variable directly
```

Output:

```
count= 1
count= 2
AttributeError: 'Counter' object has no attribute '_ _secretCount'
```

5.3 DATA ENCAPSULATION AND DATA ABSTRACTION

- We can restrict access of methods and variables in a class with the help of encapsulation. It will prevent the data from being modified by accident.
- Encapsulation is used to hide the values or state of a structured data object inside a class, preventing unauthorized parties' direct access to them.
- Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.
- The terms encapsulation and abstraction (data hiding) are often used as synonyms. Data abstraction we can achieve through encapsulation. Encapsulation is a process to bind data and functions together into a single unit i.e., class while as traction is a process in which the data inside the class is the hidden from the outside used. It hides the sensitive information.
- Hiding internal details and showing functionality is known as data abstraction.
- To support encapsulation, declare the methods or variables as private in the class. The private methods cannot be called by the object directly. It can be called only from within the class in which they are defined.
- Any function prefix with double underscore is called private method which is accessible only with class where it is declared.
- Following table shows the access modifiers for variables and methods:

Sr. No.	Types	Description
1.	Public methods	Accessible from anywhere i.e. inside the class in which they are defined, in the sub class, in the same script file as well as outside the

		script file.
2.	Private methods	Accessible only in their own class. starts with two underscores.
3.	Public variables	Accessible from anywhere.
4.	Private variables	Accessible only in their own class or by a method if defined. starts with two underscores.

Example: For access modifiers with data abstraction.

```
class student:
 _ _a=10 #private variable
 b=20 #public variable
 def _ _private_method(self): #private method
 print("private method is called")
 def public_method(self): #public method
 print("public method is called")
 print("a=",self._ _a) #can be accessible in same class
s1=student()
# print("a=",s1._ _a) #generate error
print("b=",s1.b)
# s1._ _private_method() #generate error
s1.public_method()
```

Output:

```
b=20
public method is called
a=10
```

Constructor:

- A constructor is a special method i.e., is used to initialize the instance variable of a class.

Creating Constructor in Class:

- A constructor is a special type of method (function) which is used to initialize the instance members of the class.
- Constructors are generally used for instantiating an object. The task of constructors is to initialize (assign values) to the data members of the class when an object of class is created.
- Python class constructor is the first piece of code to be executed when we create a new object of a class.
- In Python the _ _init_ _() method is called the constructor and is always called when an object is created.
- Primarily, the constructor can be used to put values in the member variables. We may also print messages in the constructor to be confirmed whether the object has been created.

Syntax:

```
def _ _init_ _(self):
 # body of the constructor
```

- _ _init_ _ is a special method in Python classes, which is the constructor method for a class. In the following example you can see how to use it.

Example 1: For creating constructor.

```
class Person:
 def _ _init_ _(self, rollno, name, age):
 self.rollno=rollno
 self.name = name
 self.age = age
 print("Student object is created")
```

```
p1 = Person(11,"Vijay", 40)
print("Rollno of student= ", p1.rollno)
print("Name of student= ",p1.name)
print("Age of student= ",p1.age)
```

Output:

```
Student object is created
Rollno of student= 11
Name of student= Vijay
Age of student= 40
```

Example 2: Define a class named Rectangle which can be constructed by a length and width. The Rectangle class has a method which can compute the area.

```
class Rectangle(object):
 def _ _init_ _(self, l, w):
 self.length = l
 self.width = w

 def area(self):
 return self.length*self.width

r = Rectangle(2,10)
print(r.area())
```

Output:

```
20
```

Example 3: Create a Cricle class and intialize it with radius. Make two methods getArea and getCircumference inside this class.

```
class Circle():
 def _ _init_ _(self,radius):
 self.radius = radius
 def getArea(self):
 return 3.14*self.radius*self.radius
 def getCircumference(self):
 return self.radius*2*3.14
c=Circle(5)
print("Area",c.getArea())
print("Circumference",c.getCircumference())
```

Output:

```
Area 78.5
Circumference 31.400000000000002
```

- The **types of constructors** includes default constructor and parameterized constructor.

1. Default constructor:

- The default constructor is simple constructor which does not accept any arguments. It's definition has only one argument which is a reference to the instance being constructed.

Example 1: Display Hello message using default constructor.

```
class Student:
 def _ _init_ _(self):
 print("This is non parametrized constructor")
 def show(self,name):
 print("Hello",name)
s1 = Student()
```

```
s1.show("Meenakshi")
```

Output:

```
This is non parametrized constructor
Hello Meenakshi
```

Example 2: Counting the number of objects of a class.

```
class Student:
 count=0;
 def _ _init_ _(self):
 Student.count=Student.count+1

s1=Student()
s2=Student()
print("The number of student objects",Student.count)
```

Output:

```
The number of student objects: 2
```

---

2. Parameterized Constructor:

- Constructor with parameters is known as parameterized constructor.
- The parameterized constructor take its first argument as a reference to the instance being constructed known as self and the rest of the arguments are provided by the programmer.

Example: For parameterized constructor.

```
class Student:
 def _ _init_ _(self,name):
 print("This is parametrized constructor")
 self.name = name
 def show(self):
 print("Hello",self.name)

s1 = Student("Meenakshi")
s1.show()
```

Output:

```
This is parametrized constructor
Hello Meenakshi
```

Destructor:

- A class can define a special method called a destructor with the help of _ _del_ _(). It is invoked automatically when the instance (object) is about to be destroyed.
- It is mostly used to clean up any non-memory resources used by an instance (object).

Example: For destructor.

```
class Student:
 def __init__(self):
 print('non parameterized constructor-student created ')
 def __del__(self):
 print('Destructor called, student deleted.')
s1=Student()
s2=Student()
del s1
```

Output:

```
non parameterized constructor-student created
non parameterized constructor-student created
Destructor called, student deleted.
```

Built-in Class Attributes:

- Every Python class keeps following built-in attributes and they can be accessed using dot operator like any other attribute:
  - _ _dict_ _: It displays the dictionary in which the class's namespace is stored.
  - _ _name_ _: It displays the name of the class.
  - _ _bases_ _: It displays the tuple that contains the base classes, possibly empty. It displays them in the order in which they occur in the base class list.
  - _ _doc_ _: It displays the documentation string of the class. It displays none if the docstring isn't given.
  - _ _module_ _: It displays the name of the module in which the class is defined. Generally the value of this attributes is "_ _main_ _" in interactive mode.

Example: For default built-in class attribute.

```
class test:
 'This is a sample class called Test.'
def _ _init_ _(self):
print("Hello from _ _init_ _ method.")

# class built-in attribute
print(test._ _doc_ _)
print(test._ _name_ _)
print(test._ _module_ _)
print(test._ _bases_ _)
print(test._ _dict_ _)
```

Output:

```
This is a class called Test.
test
_ _main_ _
(<class 'object'>,)
{'_ _module_ _': '_ _main_ _', '_ _doc_ _': 'This is a sample class called Test.',
'_ _init_ _': <function test._ _init_ _ at 0x013AC618>, '_ _dict_ _': <attribute
'_ _dict_ _' of 'test' objects>, '_ _weakref_ _': <attribute '_ _weakref_ _' of
'test' objects>}
```

5.4 METHOD OVERLOADING

- Method overloading is the ability to define the method with the same name but with a different number of arguments and data types.
- With this ability one method can perform different tasks, depending on the number of arguments or the types of the arguments given.
- Method overloading is a concept in which a method in a class performs operations according to the parameters passed to it.

Example: With a method to perform different operations using method overloading.

```
class operation:
def add(self,a,b):
        return a+b

op1=operation()
# To add two integer numbers
print("Addition of integer numbers=",op1.add(10,20))
# To add two floting point numbers
print("Addition of integer numbers=",op1.add(11.12,12.13))
# To add two strings
```

```
print("Addition of integer numbers=",op1.add("Hello","Python"))
```

Output:

```
Addition of integer numbers= 30
Addition of integer numbers= 23.25
Addition of integer numbers= HelloPython
```

- As in other languages we can write a program having two methods aith same name but with different number of arguments or order of arguments but in python if we will try to do the same we will get the following issue with method overloading in Python:

```
# to calculate area of rectangle
def area(length, breadth):
calc = length * breadth
    print calc
#to calculate area of square
def area(size):
calc =  size * size
    print calc

area(3)
area(4,5)
```

Output:

```
9
TypeError: area() takes exactly 1 argument (2 given)
```

- Python does **not** support method overloading, that is, it is not possible to define more than one method with the same name in a class in Python.
- This is because method arguments in python do not have a type. A method accepting one argument can be called with an integer value, a string or a double as shown in next example.

```
class Demo:
def method(self, a):
print(a)
obj= Demo()
obj.method(50)
obj.method('Meenakshi')
obj.method(100.2)
```

Output:

```
50
Meenakshi
100.2
```

- Same method works for three different data types. Thus, we cannot define two methods with the same name and same number of arguments but having different type as shown in the above example. They will be treated as the same method.
- It is clear that method overloading is not supported in python but that does not mean that we cannot call a method with different number of arguments. There are a couple of alternatives available in python that make it possible to call the same method but with different number of arguments.

Using Default Arguments:

- It is possible to provide default values to method arguments while defining a method. If method arguments are supplied default values, then it is not mandatory to supply those arguments while calling method as shown in next example.

```
class Demo:
```

```
def arguments(self, a = None, b = None, c = None):
if(a != None and b != None and c != None):
print("3 arguments")
elif (a != None and b != None):
print("2 arguments")
elif a != None:
print("1 argument")
else:
print("0 arguments")
obj = Demo()
obj.arguments("Meenakshi","Anurag","Thalor")
obj.arguments("Anurag","Thalor")
obj.arguments("Thalor")
obj.arguments()
```

Output:

```
3 arguments
2 arguments
1 argument
0 arguments
```

5.5 INHERITANCE AND COMPOSITION CLASS

- The inheritance feature allows us to make it possible to use the code of the misting class by simply creating a new class and inherits the code of the existing class.

5.5.1 Inheritance

- In inheritance objects of one class procure the properties of objects of another class. Inheritance provide code usability, which means that some of the new features can be added to the code while using the existing code. The mechanism of designing or constructing classes from other classes is called inheritance.
- The new class is called derived class or child class and the class from which this derived class has been inherited is the base class or parent class.
- 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.

Syntax:

```
class A:
 # properties of class A
class B(A):
# class B inheriting property of class A
# more properties of class B
```

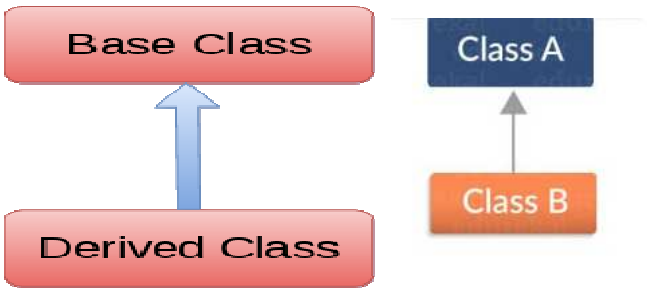


Fig. 5.1: Concept of Inheritance (Single Inheritance)

Example 1: Inheritance without using constructor.

```
class Vehicle: #parent class
 name="Maruti"
 def display(self):
 print("Name= ",self.name)
class Category(Vehicle): #derived class
 price=2000
 def disp_price(self):
 print("Price=$",self.price)
car1=Category()
car1.display()
car1.disp_price()
```

Output:

```
Name= Maruti
Price=$ 2000
```

Example 2: Inheritance using constructor.

```
class Vehicle: #parent class
 def __init__(self,name):
 self.name=name
 def display(self):
 print("Name= ",self.name)
class Category(Vehicle): #derived class
 def __init__(self,name,price):
 Vehicle.__init__(self,name) # passing data to base class constructor
 self.price=price
 def disp_price(self):
 print("Price=$ ",self.price)
car1=Category("Maruti",2000)
car1.display()
car1.disp_price()
car2=Category("BMW",5000)
car2.display()
car2.disp_price()
```

Output:

```
Name= Maruti
Price=$ 2000
Name= BMW
Price=$ 5000
```

Multilevel Inheritance:

- 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.
- In multilevel inheritance (See Fig. 5.2), we inherit the classes at multiple separate levels. We have three classes A, B and C, where A is the super class, B is its sub(child) class and C is the sub class of B.

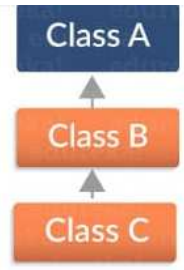


Fig. 5.2

Syntax:

```
class A:
 # properties of class A
class B(A):
 # class B inheriting property of class A
```

```
 # more properties of class B
class C(B):
 # class C inheriting property of class B
 # thus, class C also inherits properties of class A
 # more properties of class C
```

Example: For multilevel inheritance.

```
class Grandfather:
 def display1(self):
 print("Grand Father")
#The child class Father inherits the base class Grandfather
class Father(Grandfather):
 def display2(self):
 print("Father")
#The child class Son inherits another child class Father
class Son(Father):
 def display3(self):
 print("Son")
s1=Son()
s1.display3()
s1.display2()
s1.display1()
```

Output:

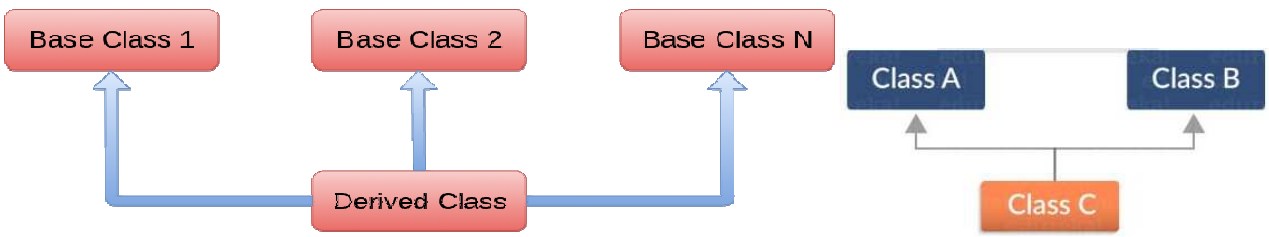
```
Son
Father
Grand Father
```

Multiple Inheritance:

- Python provides us the flexibility to inherit multiple base classes in the child class.
- Multiple Inheritance means that we are inheriting the property of multiple classes into one. In case we have two classes, say A and B, and we want to create a new class which inherits the properties of both A and B.
- So it just like a child inherits characteristics from both mother and father, in python, we can inherit multiple classes in a single child class.

Syntax:

```
class A:
 # variable of class A
 # functions of class A
class B:
 # variable of class A
 # functions of class A
class C(A, B):
 # class C inheriting property of both classA and B
 # add more properties to class C
```



(a) (b)

Fig. 5.3

Example:

```
# base class
class Father:
 def display1(self):
 print("Father")
#base class
class Mother:
 def display2(self):
 print("Mother")
#derived class
class Son(Father,Mother):
 def display3(self):
 print("Son")
s1=Son()
s1.display3()
s1.display2()
s1.display1()
```

Output:

```
Son
Mother
Father
```

Hierarchical Inheritance:

- When more than one derived classes are created from a single base – it is called hierarchical inheritance.

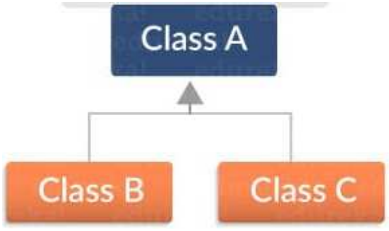


Fig. 5.4

- In following program, we have a parent (base) class name Email and two child (derived) classes named Gmail and yahoo.

Example: For hierarchical inheritance.

```
class Email:
 def send_email(self, msg):
 print()

class Gmail(Email):
 def send_email(self, msg):
 print( "Sending `{}` from Gmail".format(msg) )

class Yahoo(Email):
 def send_email(self, msg):
 print( "Sending `{}` from Yahoo".format(msg) )

client1 = Gmail()
client1.send_email("Hello!")
```

```
client2 = Yahoo()
client2.send_email("Hello!")
```

Output:

```
Sending `Hello!` from Gmail
Sending `Hello!` from Yahoo
```

5.5.2 Method Overriding

- Overriding is the ability of a class to change the implementation of a method provided by one of its base class. Method overriding is thus a strict part of the inheritance mechanism.
- To override a method in the base class, we must define a new method with same name and same parameters in the derived class.
- Overriding is a very important part of OOP since it is the feature that makes inheritance exploit its full power. Through method overriding a class may "copy" another class, avoiding duplicated code, and at the same time enhance or customize part of it.

Example: For method overloading.

```
class A:                        # parent class
 "Parent Class"
 def display(self):
 print ('This is base class.')
class B(A):      #derived class
 "Child/Derived class"
 def display(self):
 print ('This is derived class.')
obj = B()                       # instance of child
obj.display()                   # child calls overridden method
```

Output:

```
This is derived class.
```

5.5.3 Composition Classes

- In composition, we do not inherit from the base class but establish relationships between classes through the use of instance variables that are references to other objects.
- Composition also reflects the relationships between parts, called a “has-a” relationships. Some OOP design texts refer to composition as aggregation.
- It enables creating complex types by combining objects of other types. This means that a class Composite can contain an object of another class Component.
- The composite class generally provides an interface all its own and implements it by directing the embedded objects.
- UML represents composition as shown in Fig. 5.5.

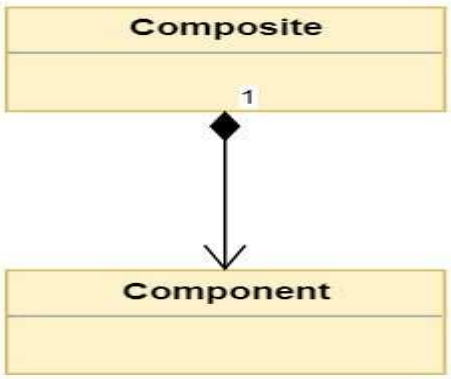


Fig. 5.5

- The composite side can express the cardinality of the relationship. The cardinality indicates the number or valid range of Component instances the Composite class will contain. Cardinality can be expressed in the following ways:
- A number indicates the number of Component instances that are contained in the Composite. The * symbol indicates that the Composite class can contain a variable number of Component instances.
- A range 1..4 indicates that the Composite class can contain a range of Component instances. The range is indicated with the minimum and maximum number of instances, or minimum and many instances like in 1..*.

Syntax:

```
Class GenericClass:
define some attributes and methods
class ASpecificClass:
Instance_variable_of_generic_class=GenericClass
# use this instance somewhere in the class
some_method(Instance_variable_of_generic_class)
```

- In following program, we have three classes Email, Gmail and yahoo. In email class we are referring the Gmail and using the concept of Composition.

Example: For composition.

```
class Gmail:
 def send_email(self, msg):
 print("Sending `{}` from Gmail".format(msg))
class Yahoo:
 def send_email(self, msg):
 print( "Sending `{}` from Yahoo".format(msg) )
class Email:
 provider=Gmail()
 def set_provider(self,provider):
 self.provider=provider
 def send_email(self, msg):
 print(self.provider.send_email(msg))
client1 = Email()
client1.send_email("Hello!")
client1.set_provider(Yahoo())
client1.send_email("Hello!")
```

Output:

```
Sending `Hello!` from Gmail
None
Sending `Hello!` from Yahoo
None
```

5.6 CUSTOMIZATION VIA INHERITANCE SPECIALIZING INHERITED METHODS

- In Python, every time we use an expression of the form object.attr, (where object is an instance or class object), Python searches the namespace tree from bottom to top, beginning with object, looking for the first attr it can find.
- This includes references to self attributes in the methods. Because lower definitions in the tree override higher ones, inheritance forms the basis of specialization.
- Program code in Fig. 5.6 create a tree of objects in memory to be searched by attribute inheritance.
- Calling a class creates a new instance that remembers its class, running a class statement creates a new class and superclasses are listed in parentheses in the class statement header.

- Each attribute reference triggers a new bottom - up tree search - even references to self attributes within a class's methods.

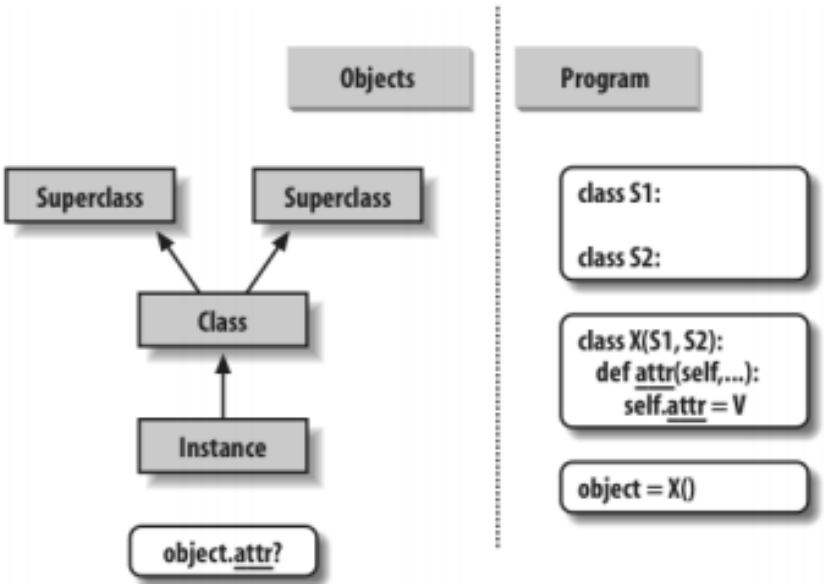


Fig. 5.6

- Fig. 5.6 summarizes the way namespace trees are constructed and populated with names. Generally:
  1. Instance attributes are generated by assignments to self attributes in methods.
  2. Class attributes are created by statements (assignments) in class statements.
  3. Superclass links are made by listing classes in parentheses in a class statement header.
- The net result is a tree of attribute namespaces that leads from an instance, to the class it was generated from, to all the superclasses listed in the class header.
- Python searches upward in this tree, from instances to superclasses, each time we use qualification to fetch an attribute name from an instance object.

Specializing Inherited Methods:

- The tree-searching model of inheritance just described turns out to be a great way to specialize systems. Because inheritance finds names in derived classes before it checks base classes, derived classes can replace default behavior by redefining their base classes' attributes.
- In fact, we can build entire systems as hierarchies of classes, which are extended by adding new external derived classes rather than changing existing logic in-place. The idea of redefining inherited names leads to a variety of specialization techniques.
- For instance, derived classes may replace inherited attributes completely, provide attributes that a base class expects to find, and extend base class methods by calling back to the base class from an overridden method. Here is an example that shows how extension works.

Example : For specialized inherited methods.

```
class A:                        # parent class
 "Parent Class"
 def display(self):
 print ('This is base class.')
class B(A):                     # derived class
 "Child/Derived class"
 def display(self):
 A.display(self)
 print ('This is derived class.')
obj = B()                       # instance of child
obj.display()                   # child calls overridden method
```

- The derived class replaces base's method function with its own specialized version, but within the replacement, derived calls back to the version exported by base class to carry out the default behavior.
- In other words, derived class.display just extends base class.display behavior, rather than replacing it completely.
- Extension is only one way to interface with a superclass. Following program defines multiple classes that illustrate a variety of common techniques:
  1. **Super:** Defines a method function and a delegate that expects an action in a subclass.
  2. **Inheritor:** Doesn't provide any new names, so it gets everything defined in Super.
  3. **Replacer:** Overrides Super's method with a version of its own.
  4. **Extender:** Customizes Super's method by overriding and calling back to run the default.
  5. **Provider:** Implements the action method expected by Super's delegate method.

Example: Give a feel for the various ways to customize a common superclass.

```
class Super:
 def method(self):
 print('in Super.method') # Default behavior
                                  def delegate(self):
 self.action() # Expected to be defined
class Inheritor(Super): # Inherit method verbatim
                                  pass
class Replacer(Super): # Replace method completely
 def method(self):
                                  print('in Replacer.method')
class Extender(Super): # Extend method behavior
                                  def method(self):
 Super.method(self)
 print('in Extender.method')
class Provider(Super): # Fill in a required method
                                  def action(self):
 print('in Provider.action')
for klass in (Inheritor, Replacer, Extender):
 print('\n' + klass.__name__ + '...')
 klass().method()
print('\nProvider...')
x = Provider()
x.delegate()
```

Output:

```
Inheritor...
in Super.method
Replacer...
in Replacer.method
Extender...
in Super.method
in Extender.method
Provider...
in Provider.action
```

- At the end of this program instances of three different classes are created in a for loop. Because classes are objects, you can put them in a tuple and create instances generically.
- Classes also have the special _ _name_ _ attribute, which preset to a string containing the name in the class header.

- In previous example, when we call the delegate method through a provider instance, two independent inheritance searches occur:
  1. On the initial x.delegate call, Python finds the delegate method in Super by searching the Provider instance and above. The instance x is passed into the method's self argument as usual.
  2. Inside the Super.delegate method, self.action invokes a new, independent inheritance search of self and above. Because self references a Provider instance, the action method is located in the Provider subclass.
- The superclass in this example is what is sometimes called an abstract superclass – a class that expects parts of its behavior to be provided by its subclasses.

Practice Questions

1. What is OOP?
2. List the features and explain about different Object Oriented features supported by Python.
3. List and explain built in class attributes with example.
4. Design a class that store the information of student and display the same.
5. What are basic overloading methods?
6. What is method overriding? Write an example.
7. Explain class inheritance in Python with an example.
8. How to declare a constructor method in python? Explain.
9. How operator overloading can be implemented in Python? Give an example.
10. Write a Python program to implement the concept of inheritance.
11. Create a class employee with data members: name, department and salary. Create suitable methods for reading and printing employee information.
12. What is data abstraction? Explain in detail.
13. Define the following terms:
    (i) Object
    (ii) Class
    (iii) Inheritance
    (iv) Data abstraction.
14. Describe the term composition classes with example.
15. Explain customization via inheritance specializing inherited methods.

■■■