Classes II

Chapter 11

Department of Computer Science and Engineering, ITER
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India.

Contents

Introduction

- The classes lie at the heart of a programming methodology called object-oriented paradigm or Object-oriented Programming (OOP).
- It includes several concepts which are listed below.
  1. Polymorphism
  2. Encapsulation
  3. Data Hiding
  4. Data Abstraction
  5. Inheritance

Polymorphism

Definition

A method/operator may be applied to objects of different types (classes). This feature of object oriented programming is called polymorphism.

Example

- len(): len function operates on various types of objects such as str, list, and tuple.

Operator Overloading

- Use of the same syntactic operator for objects of different classes (types) is called **operator overloading**.
- Python also provides special methods such as __eq__,__lt__,__le__, __gt__, and __ge__ for overloading comparison operators $==$, $<$, $<=$, $>$, and $>=$ respectively.
- Overload the special method __add__ by defining an implementation of addition for objects of type Point given in example below.

Operator Overloading Cont..

- Output of given code is:(5,7)

```
'''Python Program to perform addition of two points in
    coordinate plane using binary + operator overloading.''
    '
class Point:
    def init (self, x, y):
        self.x = x
        self.y = y
    ''' adding two Point objects'''
    def add (self, other):
        x = self.x + other.x
        y = self.y + other.y
        return Point(x,y)

point1 = Point(3,6)
point2 = Point(2,1)
print(point1 + point2)
```

Function Overloading

- **Function overloading** provides the ability of writing different functions having the same name, but with a different number of parameters, possibly of the various types.
- Example: We may like to define two functions by the name area to compute the area of a circle or rectangle:

```
1 def area(radius):
2     """Computes the area of circle"""
3     areaCirc = 3.14 * radius * radius
4     return areaCirc
5 def area(length, breadth):
6     """Computes the area of rectangle"""
7     areaRect = length * breadth
8     return areaRect
```

Function Overloading Cont...

- Python does not support function overloading. Indeed, Python cannot distinguish between parameters based on their type as the type of a parameter is inferred implicitly when a function is invoked with actual arguments.
- Thus, whereas invoking the function area with a single argument results in error, invoking it with two arguments yields the expected output.
- In python function overloading can be achieve by indirect implementation. An example is shown below where if the function area is invoked with two arguments, it computes the area of the rectangle, and if with only one argument, it computes the area of the circle.

Function Overloading Cont...

```
1  def area(a, b = None):
2      """
3      Objective: To compute area of a circle or rectangle
4                     depending on the number of parameters.
5      Inputs:   a: radius, in the case of a circle side1, in case
6                  of rectangle
7                b: None, in the case of a circle side2, in the
       case
8                  of rectangle
9      Output: area of circle or rectangle as applicable"""
10     """Computes the area of circle"""
11     if b == None:
12         areaCirc = 3.14 * a * a
13         return areaCirc
14     """Computes the area of rectangle"""
15     else:
16         areaRect = a * b
17         return areaRect
```

Encapsulation, Data Hiding, and data Abstraction

- **Encapsulation** enables us to group together related data and its associated functions under one name.
- Classes provide an **abstraction** where essential features of the real world are represented in the form of interfaces, hiding the lower-level complexities of implementation details, i.e.
  Abstraction: representing essential features of the real world, hiding lower level details.
- Technique of restricting access to private members from outside the class is known as **name mangling**.
- The restriction on the use of private variables from outside the scope of the class may be bypassed using the syntax:

$$< \textit{instance} > ._{-} < \textit{className} >< \textit{attributeName} >$$

Modifier and Accessor Methods

- The methods in the class definition that modify the value of one or more arguments are known as **modifiers**.
- The methods in the class definition that can only access(not modify) the value of one or more arguments are known as **accessors**.

Static Method

- An **instance method** typically defines operations on the data members of an instance of a class.
- **Static methods** are used for modifying class data members and do not require passing object as the first parameter.
- To tell Python that a method is a static method, the function decorator **@staticmethod** precedes the method definition.

Adding Methods Dynamically

- Python allows us to add methods dynamically to a class using the syntax:
  $< \textit{className} > . < \textit{newMethodName} > = <$ *existingFunctionName* $>$
- If we were to associate a method with a particular instance of a class, we need to import the function MethodType from the module types. Subsequently, we use the following syntax to add a method to an instance of a class:
  $< \textit{instance} > . < \textit{newMethodName} > = \textit{MethodType}(<$ *existingFunctionName* $>, <$ *instance* $>)$

Composition

The process of using objects of the other classes as attribute values is called object **composition**.

Inheritance

- **Inheritance** is an important feature of object oriented programming that imparts ability to a class to inherit properties and behavior of another class.
- Class which inherits properties is called **drived**, sub, or child class and class from which properties are inherited is called **base**, super, or parent class.
- **Object** class is the base class of all classes.
- Example: Person is base class of employee. The notion of inheritance allows us to express the parent–child relationship among the classes as shown in figure (1).

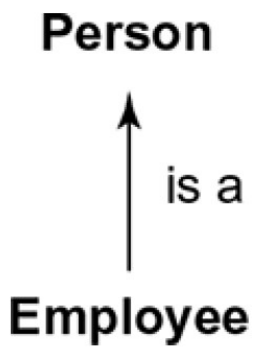

Figure 1: Parent and child class

Types of Inheritance

- There are different types of inheritance based on how base classes and derived classes are defined. These types are given below:
    1. Single Inheritance
    2. Multilevel Inheritance
    3. Hierarchical Inheritance
    4. Multiple Inheritance

Single Inheritance

- When inheritance involves a derived class that derives its properties from a single base class as shown in figure (1) is known as **Single Inheritance**.
- **Syntax:**

```
class drived_class_name (base_class_name):
    body of drived class
```

- Example:

```
class Employee (Person):
    body of drived class
```

- All the data and method attributes defined in the base class become available to the derived class.
- When an object of a derived class makes a call to a method that is also defined in the base class, Python invokes the method of derived class. Thus, the derived class methods **override** the base class methods, and this process is known as method **overriding**.

Single Inheritance Cont...

- Using **super function** for accessing a method of a base class:
  super(derived_class_name, self).
  __init__(parameters of base class)
  or
  super().__init__(parameters of base class)
- **Scope Rule**:
  To access an instance attribute, Python will first look for it in the instance's namespace, then in the class namespace, and then in the superclass namespace recursively going up in the hierarchy.
- **__bases__**: used to find base class(es) of a class.
  Example:
  >>> str.__bases__
  (<class 'object'>,)
- **__name__**: used to retrieve the name of a class
  Example:
  >>> 'Python'.__class__.__name__
  'str'

Hierarchical Inheritance

- Derived class is like any other class and may serve as base class for another class derived from it and so on.
- Each derived class may define its new attributes. Thus, inheritance allows us to create a class hierarchy, also called type hierarchy.
- When inheritance involves more than one level of hierarchy such as $A->B->C$, it is called **multilevel inheritance**, and inheritance involving multiple classes deriving from single base class is known as **hierarchical inheritance** as shown in figure (2).
- Hierarchical inheritance may be combined with multiple and multilevel inheritance, to yield hybrid inheritance.

Hierarchical Inheritance Cont...

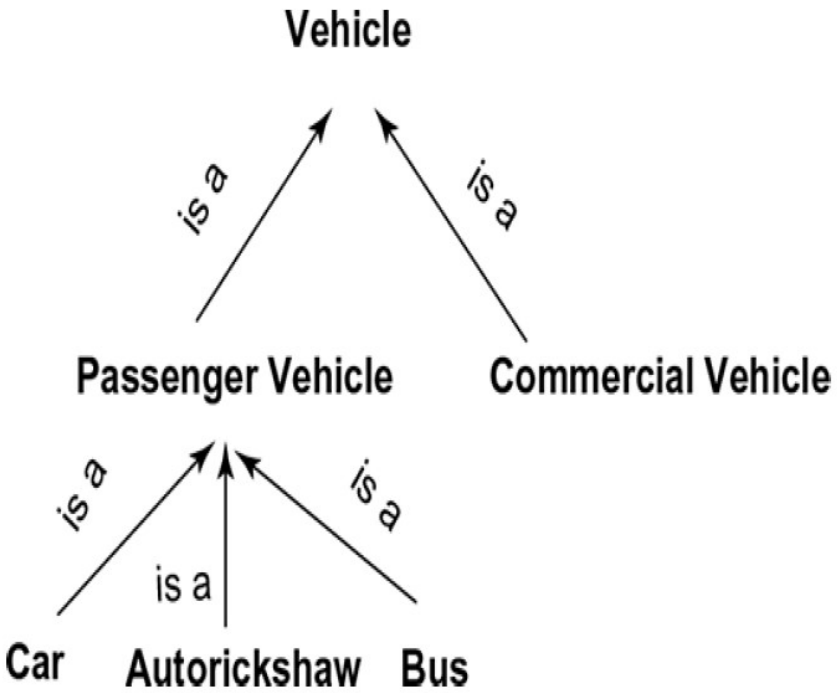

Figure 2: Example of inheritance hierarchy

Multiple Inheritance

- Subclass derives its attributes from two or more classes then such inheritance is called **multiple inheritance** as shown in figure (3).

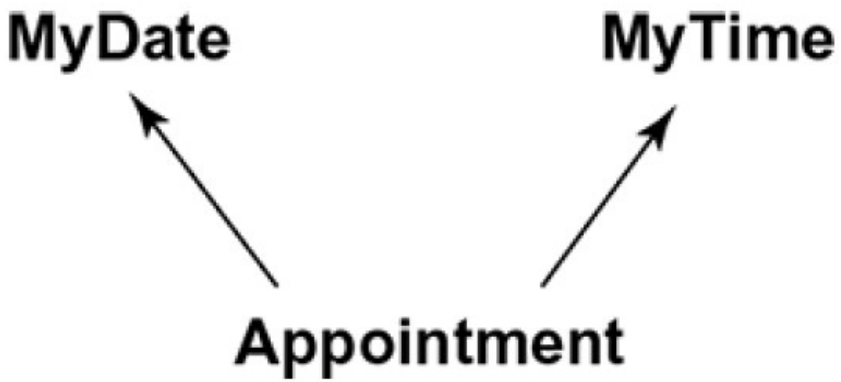

Figure 3: Example of Multiple Inheritance

Abstract Methods

- An **abstract method** in a base class identifies the functionality that should be implemented by all its subclasses.
- The implementation of an abstract method would differ from one subclass to another, often the method body comprises just a **pass** statement.
- A class containing abstract methods is called **abstract class**.
- To use Python **Abstract Base Classes** (ABCs), one needs to import **ABCMeta** and **abstractmethod** from the abc module.
- To indicate that a method is abstract it is by preceding its definition by **@abstractmethod (function decorator)**.
- The definition of an abstract class begins with **__metaclass__ = ABCMeta**.
- Exmaple:
  Shape is base class of rectangle and circle and area is abstract method because area of rectangle and circle is different.

Abstract Methods Cont...

```
1  from abc import ABCMeta, abstractmethod
2  class Shape:
3      metaclass = ABCMeta
4      def __init__ (self, shapetype):
5          self.shapetype = shapetype
6      @abstractmethod
7      def area(self):
8          pass
9
10 class rectangle(Shape):
11     def __init__ (self, length, breadth):
12         Shape.__init__(self,'rectangle')
13         self.length = length
14         self.breadth = breadth
15     def area(self):
16         return self.length * self.breadth
```

Abstract Methods Cont...

- Output of following code is:
  Area of ractangle with length = 30 and breadth = 15 is: 450
  Area of circle with radius = 5 is: 78.5

```
class circle(Shape):
    pi = 3.14
    def __init__ (self, radius):
        Shape.__init__(self,'circle')
        self.radius = radius
    def area(self):
        return round(circle.pi*(slef.radius**2),2)

r = rectangle(30,15)
area_rectangle = r.area()
print("Area of ractangle with length = 30 and breadth = 15
    is: ",area_ractangle)
c = circle(5)
area_circle = c.area()
print("Area of circle with radius = 5 is: ",area_circle)
```

Attribute resolution order for Inheritance

- A new-style class inherits from the built-in class object.
- In the new style classes, to access an attribute of an object, Python typically looks for it in the namespace of the object itself, then in the namespace of the class, then in the namespace of the superclasses in the order in which they are derived and so on.
- For determining the order in which methods of classes will be accessed, Python provides the attribute __**mro**__ which stands for **method resolution order**.
- __**mro**__ returns a list ordered by the classes in which search for methods is to take place.

Attribute resolution order for Inheritance Cont...

- The newStyleClasses, class C inherits from classes B1 and B2, which further inherit from the class A. It is important to point out that the class C inherits from classes B1 and B2 in the order B1 and B2. **The order of resolution** would be the object c1 of class C, class C, class B1, class B2, class A, class object. The output of this example is 50.

```
1  class A:
2      test = 20
3
4  class B1(A):
5      pass
6
7  class B2(A):
8      test = 50
9
10 class C(B1,B2):
11     def str (self):
12         return str(self.test)
13 c1 = C()
14 print(c1)
```

Built-In functions for classes

- To find whether a class is a subclass of another class, one may use the function **issubclass** that takes two arguments: sub and super:

  issubclass(sub, super)

  The function issubclass returns True if sub is the subclass of class super, and False otherwise.
- To find whether the instance obj is an object of class class1, we use the function **isinstance**:

  isinstance(obj, class1)

  This function returns True if either obj is an instance of class class1 or it is an instance of a subclass of class class1, and False otherwise.

Built-In functions for classes Cont...

- To find whether an instance obj contains an attribute attr, we use the function **hasattr**:

  hasattr(obj, attr)

  This function returns True if instance obj contains an attribute attr, and False otherwise.
- The functions **getattr** and **setattr** may be used to retrieve and set respectively, the value val of an attribute attr of an instance obj, as illustrated below:

  getattr(obj, attr)

  setattr(obj, attr, val)
- The function **delattr** used to delete an attribute attr of instance obj as follows:

  delattr(obj, attr)

References

[1] Python Programming: A modular approach by Taneja Sheetal, and Kumar Naveen, *Pearson Education India, Inc.*, 2017.

Thank You
Any Questions?