Object-Oriented Programs

1 Definition of Classes

A class definition includes the data structures and the behavior of the objects in that class and consists of:

- The definitions of the attributes of the class
- Descriptions of the operations, known as methods of the class

The general structure of a class is illustrated in Figure 1 and shows the attributes and the definitions of three methods: Method A, Method B, and Method C. Each of these operations consists of local data and its instructions.

The software implementation of a program is carried out writing the code in a suitable programming language. Detailed design is often written in pseudo-code, which is a high-level notation at the level between the modeling diagrams and the programming language.

In object-oriented programming, there are two general categories of variables:

- Variables of elementary or primitive type
- Object reference variables

© 2015 J. M. Garrido
Reference variables are defined when creating objects of a class. An object reference variable refers to an object.

2 Class Definitions In Python

A python program can include zero or more class definitions, function definitions, and instructions to create objects and to manipulate the objects created. A class definition is implemented with a class header and additional statements; these are:

1. The **class** statement is used for the class header in the class definition. This includes a name of the class and other information related to the class.

2. An optional comment string that serves as class documentation and which may be used to include a textual documentation or description of the class.

3. Data definitions

4. One or more member functions of the class and these are known as *methods*.

In Python, the general syntax of a class definition is:

```python
class ⟨class_name⟩:
    [data definitions]
    [method definitions]
```

2.1 Data Definitions in a Class

In a class definition, two categories of variables can be defined:

- **Class variables**
- **Instance variables**

*Class variables* are defined usually at the top and their values are shared by all objects or instances of the class. When a class variable is used outside the class, its name has to be prefixed by the class name. For example, if a class variable `xx` is defined in class `Point`, the prefix `Point.` is used with the name of the variable. To display the value of `xx` the statement is:

```python
print "Variable xx is: ", Point.xx
```

*Instance variables* are data definitions that appear inside a method and their values will be unique to each instance or object of the class. The name of all instance variables in a class have the prefix `self`.

© 2015 J. M. Garrido
2.2 Methods in a Class Definition

A method can have zero or more parameter declarations, the first parameter is always self. Four categories of methods can be defined in a class:

1. Constructor methods
2. Accessor methods
3. Mutator methods
4. Auxiliary methods

A class may have one or more constructor methods and these are used to assign initial values to instance variables and to perform any computation. A constructor method is used to create a new instance or object of the class. The name of this method is __init__ and is always the same for any class.

An accessor method returns the value of an instance variable or a class variable.

A mutator method changes the value of an instance variable and, in effect, changes the state of the object referred.

An auxiliary method is one that is only called internally in the class and it called with the self prefix.

2.3 Example of a Class Definition

In the following example, class Circle is defined and the statements that appear after the class are the instructions to perform computing with the objects created of the class.

Line 2 has a documentation string for the class. In Line 3, class variable circount is defined with an initial value of zero. Lines 5–7 defines a constructor method, which is used to initialize the instance variables and create a new instance of the class.

In line 6 the value of parameter ir is assigned to instance variable radius. The name of all instance variables in a class have the prefix self. and can be referenced in one or more methods of the class. In line 7, the class variable circount is incremented. The value of this variable is the number of objects of this class that have been created. The name of this variable is prefixed with the name of class Circle.

In line 9 a mutator method is defined that sets a new value to instance variable radius. In line 12, a mutator method is defined that computes the value of instance variable cir. In line 16 another mutator method is defined that computes the value of instance variable area. In line 20, an accessor method is defined that returns the value of instance variable radius.

© 2015 J. M. Garrido
3 Creating and Manipulating Objects

After a class has been defined, objects of the class can be created by invoking one of
the constructor methods of the class. The general form of the assignment statement
used to create an object is:

\[
\text{ref\_name} = \text{class\_name}(\text{arguments})
\]

\text{ref\_name} is a reference variable that is used as a reference to the newly created
object. For example, to create an object of class \textit{Circle} with a radius of 2.35 and a
reference variable \textit{cirobj}, the statement is:

\texttt{cirobj = Circle(2.35)}

The reference variable is used to manipulate the object by invoking one or more
of its methods. For example, the following statements are used to compute the area
of the object referenced by variable \textit{cirobj} and to display this value.

\[
\text{cirobj.\text{compArea}()}\quad \text{and}\quad \text{print(cirobj.\text{getRadius}())}
\]
area = cirobj.compArea()
print "Area of the circle: ", area

To change the value of an instance variable of an object, one of the mutator methods of the object is invoked. For example to change the value of the radius to 4.55 of the object referenced by variable \textit{cirobj}:

\begin{verbatim}
cirobj.setRadius(4.55)
\end{verbatim}

\section{Complete Program with a Class}

Listing 1 shows a Python program stored if file \texttt{circlep.py} that includes the class definition \texttt{Circle}. The statements following the class definition create and manipulate objects of class \texttt{Circle}.

\begin{verbatim}
# Program : circlep.py
# Author : Jose M Garrido, May 21 2014.
# Description : This program defines a class for circles and computes the area and circumference of circle objects
# Read the value of the radius for several circle objects from the input console, display results.

import math
class Circle:
   'Circle for computing circumference and area'
circount = 0

def __init__(self, ir):
   self.radius = ir
   Circle.circount += 1

def setRadius(self, ir):
   self.radius = ir

def compCircum(self):
   self.cir = 2.0 * math.pi * self.radius
   return self.cir
\end{verbatim}

© 2015 J. M. Garrido
def compArea(self):
    self.area = math.pi * self.radius * self.radius
    return self.area

def getRadius(self):
    return self.radius

r1 = input("Enter value of radius 1: ")
r2 = input("Enter value of radius 2: ")
r3 = input("Enter value of radius 3: ")
cobj1 = Circle(r1)
cobj2 = Circle(r2)
cobj3 = Circle(r3)
print "Value of radius1: ", cobj1.getRadius()
print "Value of radius2: ", cobj2.getRadius()
print "Value of radius3: ", cobj3.getRadius()

print "Number of objects created of class Circle: ", Circle.circount
cperim1 = cobj1.compCircum()
print "Perimeter of first circle object: ", cperim1
carea1 = cobj1.compArea()
print "Area of first circle: ", carea1

r1 = input("Enter new value of radius 1: ")
cobj1.setRadius(r1)
print "Radius of first circle: ", cobj1.getRadius()
cperim1 = cobj1.compCircum()
print "Perimeter of first circle object: ", cperim1
carea1 = cobj1.compArea()
print "Area of first circle: ", carea1

5 Scope of Variables

The scope of a variable is that portion of a program in which statements can reference that data item. Variables and constants declared as attributes of the class can be accessed from anywhere in the class. Instructions in any functioned of the class can use these data items. Local definitions define variables that can only be used by instructions in the function in which they have been declared.

The persistence of a variable is the interval of time that the data item exists—the lifetime of the data item. The lifetime of a variable declared as an attribute of a class, exists during the complete life of an object of the class. Variables declared
locally will normally have a life time only during which the function executes.

6 Class Hierarchy with Inheritance

Classes in an application that are related in some manner and are organized in the form a hierarchy of classes. Other are completely independent because they do not have any relationship with other classes.

In a class hierarchy, the most general class is placed at the top and is known as the base class, parent class, and is also known as the super class. A subclass inherits the characteristics (all attributes and operations) of its parent class. These characteristics of a class are also known as features. A subclass can be further inherited to lower-level classes.

Figure 2 illustrates the inheritance class relationship among several classes. Class University_employee is the base class, the other three classes inherit the features of this base class.

![Class Inheritance Diagram](image)

7 Defining Classes with Inheritance

The purpose of inheritance is to define a new class from an existing class and to shorten the time compared to the development of a class from scratch. Inheritance also enhances class reuse.

The base class is a more general class than its subclasses. A derived class can be defined by adding more features or modifying some of the inherited features can be defined as:

- An extension of the base class, if in addition to the inherited features, it includes its own data and operations.

© 2015 J. M. Garrido
• A specialized version of the base class, if it overrides (redefines) one or more of the features inherited from its parent class

• A combination of an extension and a specialization of the base class

Multiple inheritance is the ability of a class to inherit the characteristics from more than one parent class. Most object-oriented programming languages support multiple inheritance.

In the simple class hierarchy with inheritance shown in Figure 2, the base class is University_employee and the subclasses are: Faculty, Staff, and Short-term. All objects of class Faculty in Figure 2 are also objects of class University_employee, because this is the base class for the other classes. On the contrary, not all objects of class University_employee are objects of class Faculty.

7.1 Inheritance with Python

The definition of a subclass in Python include one or more name of base classes. The general form of the Python statement for the header in the definition of a subclass is:

```python
class ⟨ class_name ⟩ ( ⟨ base_class_list ⟩ ) :
...
```

The header of the subclasses Faculty and Staff in Figure 2 are written:

```python
class Faculty (University_employee):

and

class Staff (University_employee):
```

7.2 Inheritance and Constructor Methods

The constructor methods of a base class are the only features that are not inherited by the subclasses. A constructor method of a subclass will normally invoke the constructor method of the base class.

The statement to call or invoke a constructor method of the base class from the subclass is:

```python
{ baseclass_name }.__init__( { arguments } )
```
The following portion of Python code from the file `univemp.py` defines the base class `University_employee`. The constructor method of this class sets initial values to the instance variables `name`, `date_start`, and `phone`.

```python
class University_employee:
    'Base class'
    empcount = 0

    def __init__(self, name, datas, phone):
        self.name = name
        self.date_start = datas
        self.phone = phone
        University_employee.empcount += 1

    def setPhone(self, nphone):
        self.phone = nphone

    def getPhone(self):
        return self.phone

    def getName(self):
        return self.name

    def getDates(self):
        return self.date_start
```

The following portion of code defines a subclass `Faculty` that inherits the features of an existing (base) class `University_employee`.

The subclass `Faculty` has one class variable `faccount` and two other instance variables `rank` and `tenure`. The constructor method of this class in lines 34–38 invokes the constructor method of the base class and sets initial values to its two instance variables. The following code shows this subclass.

```python
class Faculty (University_employee):
    'Subclass of University_employee'
    facount = 0

    def __init__(self, name, datas, phone, rank, tenure):
        University_employee.__init__(self, name, datas, phone)
        self.rank = rank
        self.tenure = tenure
        Faculty.faccount += 1
```

© 2015 J. M. Garrido
def getRank(self):
    return self.rank

def setRank(self, nrank):
    self.rank = nrank

def getTenure(self):
    return self.tenure

def setTenure(self, nten):
    self.tenure = nten

In a similar manner to the previous subclass, Staff is a subclass that inherits
the features of the (base) class University_employee.

The subclass Staff has one class variable staffcount and two other instance vari-
ables position and train. The constructor method of this class, defined in lines
56–60, invokes the constructor method of the base class and sets initial values to its
two instance variables. The following code shows the definition of this subclass.

class Staff (University_employee):
    'Subclass of University_employee'
    staffcount = 0

    def __init__(self, name, datas, phone, position, train_level):
        University_employee.__init__(self, name, datas, phone)
        self.position = position
        self.train = train_level
        Staff.staffcount += 1

    def getPos(self):
        return self.position

    def setPos(self, npos):
        self.position = npos

    def getTrain(self):
        return self.train

    def setTrain(self, ntrainl):
        self.train = ntrainl
7.3 Example Objects

In the following code, objects of the base class and the two subclasses are created and methods of these objects are called. Notice that in line 78, object referenced by `femp` of subclass `Faculty` invokes method `setPhone`, which is a method of the base class. This is perfectly legal because the features of the base class are available to the subclasses. The complete program is stored in file `univemp.py`.

```python
74 gemp = University_employee("Jose Garrido", "10 Oct 2007", 2138)
75 femp = Faculty("F. Hunt", "23 March 2010", 1121, 2, False)
76 semp = Staff("J Sommer", "12 April 1999", 6543, 12, 2)
77 print "Phone of Jose: ", gemp.getPhone()
78 femp.setPhone(4454)
79 print "Phone of Hunt: ", femp.getPhone()
80 print "Tenure status Hunt: ", femp.getTenure()
81 print "Training level Sommer: ", semp.getTrain()
```

8 Overloading and Overriding Methods

*Overloading* is an object-oriented facility that allows the definition of more than one method to be defined with the same name in a class definition. This facility is not directly supported in Python but there are some more advanced ways to implement this facility.

With inheritance, a class can be defined as a *specialized* subclass of the base class. To use this facility, one or more methods of the base class are *redefined* (or overridden) in the subclass. The subclass is said to re-implement one or more methods of the base class.