Object-Oriented Programming

Contents

- Classes and Objects
- Inheritance

Classes and Objects

- Data hiding is an important principle underlying object-oriented programming:
  - As much implementation detail as possible is hidden
- Object consists of two things:
  - Encapsulated data
    - Unauthorized access to some of an object's components is prevented
  - Methods that act on the data
    - Used to retrieve and modify the values within the object
- Programmer using an object is concerned only with
  - Tasks that the object can perform
  - Parameters used by these tasks (i.e., methods)

User-Defined Classes

- A class is a template from which objects are created
  - Specifies the properties and methods that will be common to all objects that are instances of that class
  - The data types **`str`**, **`int`**, **`float`**, **`list`**, **`tuple`**, **`dictionary`**, and **`set`**, are built-in Python classes
- Python allows users to create their own classes (i.e., data types)
  - Each class defined will have a specified set of methods
  - Each object (instance) of the class will have its own value(s)
- Class definitions have the general form:

```
class ClassName:
    indented list of methods for the class
```

User-Defined Classes

- Methods have **`self`** as their first parameter
  - When an object is created, each method's **`self`** parameter references the object
  - The **`__init__`** method (aka constructor) is automatically called when an object is created, assigning values to the instance variables (also called properties of the class)

```
class Rectangle:
    def __init__(self, width=1, height=1):
        self._width = width
        self._height = height

    def setWidth(self, width):
        self._width = width

    def setHeight(self, height):
        self._height = height
```

instance variables

initializer method

mutator methods

User-Defined Classes

```
    def getWidth(self):
        return self._width

    def getHeight(self):
        return self._height

    def area(self):
        return self._width * self._height

    def perimeter(self):
        return 2 * (self._width + self._height)

    def __str__(self):
        return ("Width: " + str(self._width)
         + "\nHeight: " + str(self._height))
```

accessor methods

other methods

state-representation methods

- The **`__str__`** method provides a customized way to represent the state (values of the instance variables) of an object as a string

User-Defined Classes

- Classes can be typed directly into programs or stored in modules and brought into programs with an import statement
- An object, which is an instance of a class, is created with a statement of the form

```
objectName = ClassName(arg1, arg2, . . . )
```

or

```
objectName = modulName.ClassName(arg1, arg2, . . . )
```

User-Defined Classes

```
import rectangle

# Create a rectangle of width 4 and height 5
r = rectangle.Rectangle(4, 5)
print(r)  # Uses the __str__ method to report the state
# Create a rectangle with the default values for width and
# height
r = rectangle.Rectangle()
print(r)
# Create a rectangle of width 4 and default height 1
r = rectangle.Rectangle(4)
print(r)
```

[Run]

```
Width: 4
Height: 5
Width: 1
Height: 1
Width: 4
Height: 1
```

User-Defined Classes

```
import rectangle

r = rectangle.Rectangle()
# Use the mutator methods.
r.setWidth(4)
r.setHeight(5)
print("The rectangle has the following measurements:")
# Use the accessor methods.
print("Width is", r.getWidth())
print("Height is", r.getHeight())
# Use other methods.
print("Area is", r.area())
print("Perimeter is", r.perimeter())
```

[Run]

```
The rectangle has the following measurements:
Width is 4
Height is 5
Area is 20
Perimeter is 18
```

User-Defined Classes

- Note:
  - `r.setWidth(4)` and `r.setHeight(5)` can be replaced by `r._width = 4` and `r._height = 5`, respectively
  - `print("Width is", r.getWidth())` and `print("Height is", r.getHeight())` can be replaced by `print("Width is", r._width)` and `print("Height is", r._height)`, respectively
- However, such replacement is considered poor programming style
- Instance variable names start with a single underscore so that they cannot be directly accessed from outside of the class definition
  - Object-oriented programming hides the implementation of methods from the users of the class

Other Forms of the Initializer Method

- There are three other ways the initializer can be defined:

```
def __init__(self):
    self._width = 1
    self._height = 1

def __init__(self, width=1):
    self._width = width
    self._height = 1

def __init__(self, width, height):
    self._width = width
    self._height = height
```

- With the third form, the constructor statement creating an instance must provide two arguments

Other Methods in a Class Definition

```
def main():
    ## Calculate and display a student's semester letter
    ## grade.
    name = input("Enter student's name: ")
    midterm = float(input("Enter grade on midterm exam: "))
    final = float(input("Enter grade on final exam: "))
    # Create an instance of an LGstudent object.
    st = LGstudent(name, midterm, final)
    print("\nNAME\tGRADE")
    # Display student's name and semester letter grade.
    print(st)

class LGstudent:
    def __init__(self, name="", midterm=0, final=0):
        self._name = name
        self._midterm = midterm
        self._final = final

    def setName(self, name):
        self._name = name
```

Other Methods in a Class Definition

```
def setMidterm(self, midterm):
     self._midterm = midterm

def setFinal(self, final):
     self._final = final

def calcSemGrade(self):
    grade = (self._midterm + self._final) / 2
    grade = round(grade)
    if grade >= 90:
        return "A"
    elif grade >= 80:
        return "B"
    elif grade >= 70:
        return "C"
    elif grade >= 60:
        return "D"
    else:
        return "F"
```

Other Methods in a Class Definition

```
    def __str__(self):
        return self._name + "\t" + self.calcSemGrade()

main()
```

[Run]

```
Enter student's name: Fred
Enter grade on midterm exam: 87
Enter grade on final exam: 92

NAME    GRADE
Fred    A
```

Lists of Objects

- Items of a list can be any data type including a user-defined class
- The following program uses a list where each item is an **`LGstudent`** object

```
import lgStudent

def main():
    ## Calculate and display students' semester letter grades.
    listOfStudents = []  # To holds objects each for a student
    carryOn = 'Y'
    while carryOn == 'Y':  # Repeat until user says 'N'
        st = lgStudent.LGstudent()
        # Obtain student's name and grades.
        name = input("Enter student's name: ")
        midterm = float(input
                ("Enter student's grade on midterm exam: "))
        final = float(input
                ("Enter student's grade on final exam: "))
```

Lists of Objects

```
        # Create an instance of an LGstudent object.
        st = lgStudent.LGstudent(name, midterm, final)
        listOfStudents.append(st)    # Insert object into list.
        carryOn = input("Do you want to continue (Y/N)? ")
        carryOn = carryOn.upper()
    print("\nNAME\tGRADE")
    # Display students, names and semester letter grades.
    for pupil in listOfStudents:
        print(pupil)

main()
```

Lists of Objects

[Run]

```
Enter student's name: Alice
Enter student's grade on midterm exam: 88
Enter student's grade on final exam: 94
Do you want to continue (Y/N)? Y
Enter student's name: Bob
Enter student's grade on midterm exam: 82
Enter student's grade on final exam: 85
Do you want to continue (Y/N)? N

NAME    GRADE
Alice   A
Bob     B
```

Inheritance

- Inheritance allows us to define a modified version of an existing class (superclass, parent class, or base class)
  - The new class is called the subclass, child class, or derived class
- Subclass inherits properties and methods of its superclass
  - It can have its own properties and methods overriding some of the superclass' methods
  - No initializer method is needed if the child class does not have its own properties (i.e., instance variables)

Example: A Semester Grade Class

```
class Student:  # Superclass
    def __init__(self, name="", midterm=0, final=0):
        self._name = name
        self._midterm = midterm
        self._final = final

    def setName(self, name):
        self._name = name

    def setMidterm(self, midterm):
        self._midterm = midterm

    def setFinal(self, final):
        self._final = final

    def getName(self):
        return self._name

    def __str__(self):
        return self._name + "\t" + self.calcSemGrade()
```

Example: A Semester Grade Class

```
class LGstudent(Student): # Subclass of Student
    def calcSemGrade(self):
        average = round((self._midterm + self._final) / 2)
        if average >= 90:
            return "A"
        elif average >= 80:
            return "B"
        elif average >= 70:
            return "C"
        elif average >= 60:
            return "D"
        else:
            return "F"

class PFstudent(Student): # Subclass of Student
    def calcSemGrade(self):
        average = round((self._midterm + self._final) / 2)
        if average >= 60:
            return "Pass"
        else:
            return "Fail"
```

Example: A Semester Grade Class

- The following function creates a list of both types of students and uses the list to display the names of the students and their semester grades

```
import student

def main():
    # students and grades
    listOfStudents = obtainListOfStudents()
    displayResults(listOfStudents)

def obtainListOfStudents():
    listOfStudents = []
    carryOn = 'Y'
    while carryOn == 'Y':
        name = input("Enter student's name: ")
        midterm = float(input("Enter grade on midterm: "))
        final = float(input("Enter grade on final: "))
        category = input("Enter category (LG or PF): ")
```

Example: A Semester Grade Class

```
        if category.upper() == "LG":
            st = student.LGstudent(name, midterm, final)
        else:
            st = student.PFstudent(name, midterm, final)
        listOfStudents.append(st)
        carryOn = input("Do you want to continue (Y/N)? ")
        carryOn = carryOn.upper()
    return listOfStudents

def displayResults(listOfStudents):
    print("\nNAME\tGRADE")
    # Sort students by name.
    listOfStudents.sort(key = lambda x: x.getName())
    for pupil in listOfStudents:
        print(pupil)

main()
```

[Run]

```
Enter student's name: Bob
```

Example: A Semester Grade Class

```
Enter grade on midterm: 79
Enter grade on final: 85
Enter category (LG or PF): LG
Do you want to continue (Y/N)? Y
Enter student's name: Alice
Enter grade on midterm: 92
Enter grade on final: 96
Enter category (LG or PF): PF
Do you want to continue (Y/N)? Y
Enter student's name: Carol
Enter grade on midterm: 75
Enter grade on final: 76
Enter category (LG or PF): LG
Do you want to continue (Y/N)? N

NAME    GRADE
Alice   Pass
Bob     B
Carol   C
```

“is-a” Relationship

- Child classes are specializations of their parent’s class
  - Have all the characteristics of their parents
  - But, more functionality
  - Each child satisfies the “is-a” relationship with the parents
- E.g., each letter-grade student *is a* student, and each pass-fail student *is a* student

The *isinstance* Function

- A statement of the form

  `isinstance(object, className)`

  returns **`True`** if **`object`** is an instance of the named class or any of its subclasses, and otherwise returns **`False`**

- Some expressions involving the **`isinstance`** function

Expression	Value	Expression	Value
`isinstance("Hello", str)`	`True`	`isinstance((), tuple)`	`True`
`isinstance(3.4, int)`	`False`	`isinstance({'b':"be"}, dict)`	`True`
`sinstance(3.4, float)`	`True`	`isinstance({}, dict)`	`True`
`isinstance([1, 2, 3], list)`	`True`	`isinstance({1, 2, 3}, set)`	`True`
`isinstance([], list)`	`True`	`isinstance({}, set)`	`False`
`isinstance((1, 2, 3), tuple)`	`True`	`isinstance(set(), set)`	`True`

The *isinstance* Function

- The following function is an extension of the **`displayResults`** function on page 22
  - The **`isinstance`** function is used to count the number of letter-grade students

```
def displayResults(listOfStudents):
    print("\nNAME\tGRADE")
    numberOfLGstudents = 0
    listOfStudents.sort(key = lambda x: x.getName())
    for pupil in listOfStudents:
        print(pupil)
        # Keep track of number of letter-grade students.
        if isinstance(pupil, student.LGstudent):
            numberOfLGstudents += 1
    # Display number of students in each category.
    print("Number of letter-grade students:",
          numberOfLGstudents)
    print("Number of pass-fail students:",
          len(listOfStudents) - numberOfLGstudents)
```

The *isinstance* Function

```
main()
```

[Run]

```
NAME    GRADE
Alice   Pass
Bob     B
Carol   C
Number of letter-grade students: 2
Number of pass-fail students: 1
```

Adding New Instance Variables to a Subclass

- Child classes can also add properties (i.e., instance variables)
- Child class must contain an initializer method
  - Draws in the parent's properties
  - Then adds its own new properties
- The parameter list in the header of the child's initializer method should begin with **`self`**, list the parent's parameters, and add on its own new parameters
  - The first line of the block should have the form

```
super().__init__(parentPar1, . . . , parentParN)
```

  - This line should be followed by standard declaration statements for the new parameters of the child

Adding New Instance Variables to a Subclass

```
class PFstudent(Student):
    # A new Boolean parameter fullTime is added
    def __init__(self, name="", midterm=0, final=0,
                 fullTime=True):
        super().__init__(name, midterm, final)
        self._fullTime = fullTime

    def setFullTime(self, fullTime):
        self._fullTime = fullTime

    def getFullTime(self):
        return self._fullTime

    def calcSemGrade(self):
        average = round((self._midterm + self._final) / 2)
        if average >= 60:
            return "Pass"
        else:
            return "Fail"
```

Adding New Instance Variables to a Subclass

```
    def __str__(self):
        if self._fullTime:
            status = "Full-time student"
        else:
            status = "Part-time student"
        return (self._name + "\t" + self.calcSemGrade()
                + "\t" + status)
```

Adding New Instance Variables to a Subclass

- The following program uses new definition of **`PFstudent`** on page 29

```
import studentWithStatus  # Contains new PFstudent definition

def main():
    ## Calculate and display a student's semester letter grade
    ## and status. Obtain student's name, grade on midterm
    ## exam, and grade on final.
    name = input("Enter student's name: ")
    midterm = float(input("Enter grade on midterm: "))
    final = float(input("Enter grade on final: "))
    category = input("Enter category (LG or PF): ")
    if category.upper() == "LG":
        st = studentWithStatus.LGstudent(name, midterm, final)
    else:
        question = input("Is " + name
                         + " a full time student (Y/N)? ")
        if question.upper() == 'Y':
            fullTime = True
        else:
            fullTime = False
```

Adding New Instance Variables to a Subclass

```
        st = studentWithStatus.PFstudent(name, midterm,
                                         final, fullTime)
    # Display student's name, semester letter grade, and
    # status.
    print("\nNAME\tGRADE\tSTATUS")
    print(st)

main()
```

[Run]

```
Enter student's name: Alice
Enter grade on midterm: 92
Enter grade on final: 96
Enter category (LG or PF): PF
Is Alice a full time student (Y/N)? N

NAME    GRADE   STATUS
Alice   Pass    Part-time student
```

Overriding a Method

- If a method defined in the subclass has the same name as a method in its superclass, the child's method will override the parent's method
- Instead of the three classes **`student`**, **`LGstudent`**, and **`PFstudent`** as defined on p. 19, the following program has only two classes, **`LGstudent`** and its subclass **`PFstudent`**
  - New definition is shorter and easier to read

```
def main():
    # Students and grades
    listOfStudents = obtainListOfStudents()
    displayResults(listOfStudents)

def obtainListOfStudents():
    listOfStudents = []
    carryOn = 'Y'
    while carryOn == 'Y':
        name = input("Enter student's name: ")
```

Overriding a Method

```
        midterm = float(input("Enter grade on midterm: "))
        final = float(input("Enter grade on final: "))
        category = input("Enter category (LG or PF): ")
        if category.upper() == "LG":
            st = LGstudent(name, midterm, final)
        else:
            st = PFstudent(name, midterm, final)
        listOfStudents.append(st)
        carryOn = input("Do you want to continue (Y/N)? ")
        carryOn = carryOn.upper()
    return listOfStudents

def displayResults(listOfStudents):
    print("\nNAME\tGRADE")
    listOfStudents.sort(key = lambda x: x.getName())
    for pupil in listOfStudents:
        print(pupil)

class LGstudent:
    def __init__(self, name="", midterm=0, final=0):
```

Overriding a Method

```
        self._name = name
        self._midterm = midterm
        self._final = final

    def setName(self, name):
        self._name = name

    def setMidterm(self, midterm):
        self._midterm = midterm

    def setFinal(self, final):
        self._final = final

    def getName(self):
        return self._name

    def calcSemGrade(self):
        average = round((self._midterm + self._final) / 2)
        if average >= 90:
            return "A"
```

Overriding a Method

```
        elif average >= 80:
            return "B"
        elif average >= 70:
            return "C"
        elif average >= 60:
            return "D"
        else:
            return "F"

    def __str__(self):
        return self._name + "\t" + self.calcSemGrade()

class PFstudent(LGstudent):
    def calcSemGrade(self):
        average = round((self._midterm + self._final) / 2)
        if average >= 60:
            return "Pass"
        else:
            return "Fail"

main()
```

Polymorphism

- A feature of all object-oriented programming languages
- Allows two classes to use the same method name but with different implementations
  - **`calcSemGrade`** on pages 20 and 36

Multiple Inheritance

- A class can be derived from more than one base class
  - The features of all the base classes are inherited into the derived class

```
class Base1:
    pass

class Base2:
    pass

class MultiDerived(Base1, Base2):
    pass
```

- Method resolution order (MRO):
  - Any specified attribute is searched first in the current class
  - If not found, the search continues into parent classes in depth-first, left-right fashion without searching the same class twice