Chapter 10 Class and Object

We already learned about variables and functions. Sometimes it is desirable to put related variables and user-defined functions in one place to make it easier to work with.

Suppose, we need to store the radius and height of a cylinder and calculate its area and volume. To handle this task, we can create two variables, say, radius and height along with the functions `calculate_area`() and `calculate_volume`(). In Python, rather than creating separate variables and functions, we can also wrap these related variables and functions in a single place (called **objects**). This programming paradigm is known as object-oriented programming (OOP).

OOP is one of the most efficient and most commonly used programming methodologies. Understanding the basics of OOP is a must for developers.

OOP is neither a tool nor a programming language—it is just a concept. Some programming languages are designed to follow this concept. Python is one of them. There are other object-oriented languages, such as C++, Java, C#, PHP, JavaScript, Dart, and so on.

1. Class and Object

An **object** represents an entity in the real world such as a student, a table, a circle. An object is simply a collection of attributes (aka. data fields, fields, or states) and methods (aka. behaviors or actions).

- A circle object, for example, has an attribute `radius`, which is a property that characterizes a circle. A rectangle object has the attributes `width` and `height`, which are properties that characterize a rectangle.
- You make an object perform actions by calling its methods. For example, you can define a method named `get_area`() for circle objects. A circle object can then call the `get_area`() method to return its area.

A **class** is a blueprint for creating objects. A class uses attributes to store values and defines methods to perform actions.

2. Defining a Class

A class in Python can be defined using the `class` keyword.

```
class ClassName:
    # attributes
    # methods
```

Let's create a class named `Person`, with two attributes named `name` and `age`.

```
class Person:
    name = ""
    age = 0
```

Here, `Person` is the name of the class. The `name` and `age` are the attributes inside the class with default values of `""` and `0` respectively. **Note**: you cannot not assign any value to the attributes of a class.

3. Creating Objects

An object is an instance of a class, and you can create many instances of a class. Creating an instance of a class is referred to as **instantiation**. The terms object and instance are often used interchangeably; an object is an instance and an instance is an object.

When an object of a class is created, the class is said to be **instantiated**. All the objects of a class share the same attributes and methods. But the values of those attributes are unique for each object. A single class may have any number of objects.

Syntax for creating an object:

```
obj = Constructor
```

For example, we will create an object named p1 of our class `Person` as below:

```
class Person:
    name = ""
    age = 0

# Create a Person object
p1 = Person()
```

4. Accessing Members of Objects

An object's member refers to its attributes and methods. You can access the object's attributes and invoke its methods by using the dot operator (.), also known as the "object member access operator".

Syntax:

```
object.attribute
object.method(parameter1, parameter2, ...)
```

Example 01: Create a class and its objects.

```
# Define a class
class Person:
    name = ""  # Default name is an empty string
    age = 0    # Default age is 0

# Create a person object and set the values for its attributes
p1 = Person()
p1.name = "John"
p1.age = 22

# Create another person object and set the values for its attributes
p2 = Person()
p2.name = "Lucy"
p2.age = 20
```

```
# Display the names and ages of the two person objects
print(p1.name, p1.age)
print(p2.name, p2.age)
```

Output:

```
John 22
Lucy 20
```

5. The `__init__()` Method

All classes have a special method called `__init__()`, which is always executed when the class is being initiated. Note that `init` needs to be preceded and followed by two underscores.

The `__init__()` method, known as an **initializer**, is called automatically every time the class initialize a new object. An initializer can perform any action, but initializers are designed to perform initializing actions; creating an object's attributes with initial values.

All methods, including the initializer, MUST have the first parameter `self`. self represents the current active object of the class, and is used to access object's members in a class definition.

Example 02: Create a class called `Person`, and use the `__init__()` method to assign the values for name and age.

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

# Create and initialize two person objects
p1 = Person("John", 22)
p2 = Person("Lucy", 20)

# Display the names and ages of the two person objects
print(p1.name, p1.age)
print(p2.name, p2.age)
```

Output:

```
John 22
Lucy 20
```

`self` is a required parameter for every method; not just for the `__init__()` method. Method definition is pretty much the usual function definition like we do without a class, using the `def` keyword, except for a new parameter named `self`. This is what makes class's method different from the usual function.

Note that the arguments of the constructor match the parameters in the `__init__()` method without `self`.

`self` is a convention and not a Python keyword. You can use another name in place of it. But it is advisable to use `self` because it increases the readability of code, and it is also a good programming practice.

Example 03: In this example, we use the name `this` instead of `self`.

```
class Person:
    def __init__(this, name, age):
        this.name = name
        this.age = age

# Create a person
p1 = Person("John", 20)

# Display the name and age of the person
print(p1.name)
print(p1.age)
```

Output:

```
John
20
```

Example 04: Define a Circle class and calculate the areas of two circles

```
import math

class Circle:
    def __init__(self, radius = 1):
        self.radius = radius

    def get_area(self):
        return round(self.radius * self.radius * math.pi, 2)

# Create a circle and display its area
circle1 = Circle()
print("Area 1:", circle1.get_area())

# Create another circle and display its area
circle2 = Circle(5)
print("Area 2:", circle2.get_area())
```

Output:

```
Area 1: 3.14
Area 2: 78.54
```

In the above example, to construct a Circle object with radius of 1, we use `Circle()`. And to construct a Circle object with radius 5, we use `Circle(5)`.

6. The `self` parameter

As mentioned earlier, the first parameter for each method defined is **`self`**. This parameter is used in the implementation of the method, but it is not used when the method is called. So, what is this parameter **`self`** for? Why does Python need it? Let's look at an example:

```
class Person:
    def __init__(self):
        self.name = "John" # instance variable
        age = 20           # local variable

p1 = Person()

print(p1.name)
print(p1.age) # AttributeError: 'Person' object has no attribute 'age'
```

All variables that are declared inside a method using the **`self`** keyword are **instance variables**. These variables **belong to objects**.

And all variables that are declared inside a method without the **`self`** keyword are **local variables**. These variables **belong to methods**. In the example above, `age` is a local variable, and the scope of the variable `age` is only inside the **`__init__()`** method.

A figure below shows the scope of instance variables inside a class:

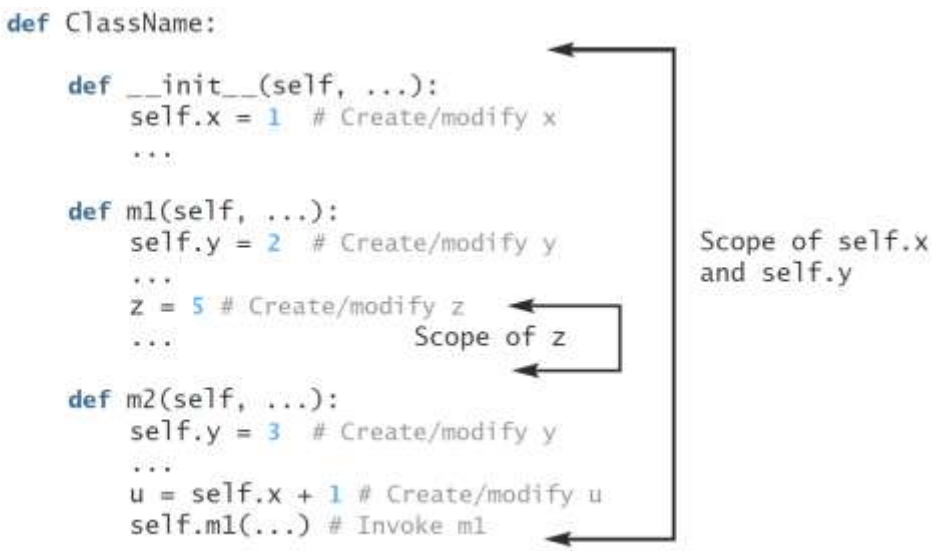

FIGURE 7.5 The scope of an instance variable is the entire class.

The scope of an instance variable is the entire class once it is created. In the above figure, **`self.`**`x` is an instance variable created in the **`__init__()`** method. It is accessed in method m2. The instance variable **`self.`**`y` is set to `2` in method `m1` and set to `3` in `m2`.

You can also create local variables in a method. **The scope of a local variable is within the method**. The local variable `z` is created in method m1 and its scope is from its creation to the end of method `m1`.

7. Class Variables, Instance Variables and Local Variables

Objects of a class can access instance variables, class variables, but not local variable of a method. A **class variable** is a variable that is declared inside of class, but outside of any instance method or `__init__()` method. These variables **belong to class**.

Let's look at an example below:

```
class Person:
    id = ""   # class variable

    def __init__(self, name, age):
        self.name = name  # instance variable
        age = age         # local variable

    def set_gender(self, gender):
        self.gender = gender      # instance variable

p1 = Person("John", 20)
p1.id = 1
p1.set_gender("M")

print(p1.id)
print(p1.name)
print(p1.age) # AttributeError: 'Person' object has no attribute 'age'
print(p1.gender)
```

Class variables can be accessed using class name or objects. There is a difference in the result:

- If you change the class variable's value by using an object, a new instance variable is created for that particular object, which shadows the class variables. The change made only affect this current object.
- To change all objects of a class at once, use the class name to access to the class variable and change the value of the variable.

Example 05: Change the value of a class variable using its object.

```
class Student:
    dept = "ITE"  # class variable

    def __init__(self, name):
        self.name = name  # instance variable

    def show_info(self):
        print("Student Name:", self.name, ", Department:", self.dept)

# Create two student objects
student1 = Student("John")
student2 = Student("Lucy")
```

```
# Update the department using a student object
student1.dept = "DSE"

# Display the information of the students
student1.show_info()
student2.show_info()
```

Output:

```
Student Name: John , Department: DSE
Student Name: Lucy, Department: ITE
```

Example 06: Change the value of a class variable using the class name. This will affect all objects of the class.

```
class Student:
    dept = "ITE"  # class variable

    def __init__(self, name):
        self.name = name  # instance variable

    def show_info(self):
        print("Student Name:", self.name, ", Department:", self.dept)

# Create two student objects
student1 = Student("John")
student2 = Student("Lucy")

# Update the department using the class name
Student.dept = "DSE"

# Display the information of the students
student1.show_info()
student2.show_info()
```

Output:

```
Student Name: John , Department: DSE
Student Name: Lucy , Department: DSE
```

8. None Keyword

None is used to define a null value or null object in Python. It is not the same as an empty string, False, or a zero.

Let's look at an example:

```
class Person:
    name = None
    age = None

    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Using **None** prevents the IDE to infer a better type for the variable, so it is better to use a sensible default value instead when possible. So, for the example above, we might change it to:

```
class Person:
    name = "Unknown"
    age = 0

    def __init__(self, name, age):
        self.name = name
        self.age = age
```

9. Anonymous Objects

Usually, you create a named object and later access its members through its name. Occasionally, you may create an object and use it only once. In this case, you do not have to name it. Such objects are called **anonymous objects**.

Example 07: Using anonymous objects.

```
class Wall:
    def __init__(self, length, height):
        self.length = length
        self.height = height

    def calculate_area(self):
        return self.length * self.height

print("Area of the wall: ", Wall(10.5, 8.6).calculate_area())
```

Output:

```
Area of the wall: 90.3
```

10. Naming Convention

Naming conventions are very important and useful in any programming language. The intent is to allow useful information to be deduced from the names.

- Names of classes are in MixedCase starting with a capital letter. If the most natural name for the class is a phrase, start each word with a capital letter. For example, `Circle`, `LinearEquation`, `StudentCard`.
- Names of constants are all uppercase letters separated each word by underscores. For example, AGE, `MIN_VALUE`, `ALL_UPPER_CASE`.
- Other names (functions, methods, variables, objects, modules) are all in lowercase letters separated each word by underscores to. For example, `compute_area`(), `insert_book`().

The Google Python Style Guide has the following convention:

module_name, package_name, ClassName, method_name, ExceptionName, function_name, GLOBAL_CONSTANT_NAME, global_var_name, instance_var_name, function_parameter_name, local_var_name.

Exercises

Write the following programs and their test programs. Note that you can add more attributes, methods or constructor if needed.

1. Design a class named `Rectangle` that contains:
   - Two attributes named `width` and `height`.
   - Constructors that create a rectangle with the specified width and height. The default values are `1` and `2` for the width and height, respectively.
   - A method named `getArea`() that returns the area of this rectangle.
   - A method named `getPerimeter`() that returns the perimeter.

   Write a test program that creates two Rectangle objects—one with width `4` and height `40` and the other with width `3.5` and height `35.7`. Display the width, height, area, and perimeter of each rectangle

2. Define a class called `Calculator` that contains:
   - Three attributes named `num1`, `operator` and `num2`.
   - A constructor that creates a calculator with the specified `num1`, `operator` and `num2`.
   - Methods: `add`(), `subtract`(), `multiply`(), `divide`(), `power`() and `modulo`() that will do the calculation and return the result.

   Write a test program that asks the user to enter the value for the two number and the operator, then display the result. Here is a sample run:

```
Enter number1, operator and number2 separated by a comma: 5.5 + 12
5.5 + 12 = 17.5
```

3. (Algebra: quadratic equations) The two roots of a quadratic equation, for example, $ax^2 + bx + c = 0$, can be obtained using the following formula:

$$r_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \quad \text{and} \quad r_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

$b^2 - 4ac$ is called the discriminant of the quadratic equation.

Design a class named `QuadraticEquation` that contains:

- Three attributes `a`, `b`, and `c` that represent three coefficients.
- A constructor that creates a quadratic equation object with the specified a, b, and c.
- A method named `get_discriminant`() that returns the discriminant.
- A method named `display_roots`() that displays the result based on the discriminant. If the discriminant is positive, display two roots. If the discriminant is 0, display one root. Otherwise, display "The equation has no real roots".

4. Define two classes called `Subject` and `Student`. Each student can take multiple subjects.
   - The `Subject` class contains:
     - Fields: `name`, and `score`.
     - A constructor that creates a student with the specified `name` and `score`.
     - A method named `displaySubjectDetail`() that displays name and score of the subject.
   - The `Student` class contains:
     - Fields: `id`, `name`, and subjects which is an ArrayList of subjects.
     - A constructor that creates a student with the specified `id`, `name` and `subjects`.
     - A method named `displaySubjects`() that displays all subjects the student takes.

   Write a test program that creates two students, `student1` and `student2`. The `student1` takes two subjects, and the `student2` takes three subjects. Display the students and their subjects Here is a sample run:

```
001, John Smith:
    - Java, 85
    - OOAD, 70

002, Lucy Brown:
    - Java, 90
    - OOAD, 82
    - Web, 75
```

5. (💘 Cupid's Matchmaking System 💘) Create a matchmaking system that evaluates compatibility between people based on their preferences.
   - ♥ Define a class called `Person` that contains: `id`, `name`, `gender`, `favorite_color`, `favorite_fruit` and `hobby`.
   - ♥ Define a class called `Matchmaker` that stores and manages a list of person objects. This class should have the following features:
     - ♥ Attribute: `candidates` which is a list that contains person objects.
     - ♥ Method: `find_match()`:
       - ♥ Accepts a person object.
       - ♥ Compares the person's preferences (favorite color, favorite fruit, and hobby) with all stored candidates.
       - ♥ Calculate a match score (out of 10) based on similarity:
         - ♥ +4 points if favorite colors match.
         - ♥ +3 points if favorite fruits match.
         - ♥ +3 points if hobbies match.
       - ♥ Display the best-matching candidate(s) and the match score(s)

   Your program should work with Candidates.csv file to store and retrieve candidate information. When the program starts, check if the Candidates.csv file exists or not. If the file does not exist, create it with the following headers: ID, Name, Gender, Favorite_Color, Favorite_Fruit, Hobby. Next, display a menu:

   💘 Cupid's Matchmaking System 💘
   1. Add a new candidate
   2. Find a match
   3. Display all candidates
   4. Exit

   Note:
   - ♥ When asking for favorite color, favorite fruit, or hobby, please display 10 options for the user to choose from. For example, here is a sample run when asking for a favorite color:

```
1. Red    2. Green    3. Blue    4. Yellow    5. ……

Select your favorite color: 2
```

   - ♥ For "Find a match" option, asks the user to enter a name, gender, favorite color, favorite fruit, hobby, then find their best match using the `find_match()` method.