Programming lab :python & oop

ref:**Introduction to Programming Using Python ,**Y. Daniel Liang ,Pearson

Object-oriented programming (OOP) involves the use of objects to create programs. An *object* represents an entity in the real world that can be distinctly identified. For example, a student, a desk, a circle, a button, and even a loan can all be viewed as objects. An object has a unique identity, state, and behavior.

- An object's *identity* is like a person's Social Security number. Python automatically assigns each object a unique id for identifying the object at runtime.

- An object's *state* (also known as its *properties* or *attributes*) is represented by variables, called *data fields*. A circle object, for example, has a data field `radius`, which is a property that characterizes a circle. A rectangle object has the data fields `width` and `height`, which are properties that characterize a rectangle.

- Python uses methods to define an object's *behavior* (also known as its *actions*). Recall that methods are defined as functions. You make an object perform an action by invoking a method on that object. For example, you can define methods named `getArea()` and `getPerimeter()` for circle objects. A circle object can then invoke the `getArea()` method to return its area and the `getPerimeter()` method to return its perimeter.

Objects of the same kind are defined by using a common class. The relationship between classes and objects is analogous to that between an apple-pie recipe and apple pies. You can make as many apple pies (objects) as you want from a single recipe (class).

A Python *class* uses variables to store data fields and defines methods to perform actions. A class is a *contract*—also sometimes called a *template* or *blueprint*—that defines what an object's data fields and methods will be.

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.

Figure 7.2 shows a class named `Circle` and its three objects.

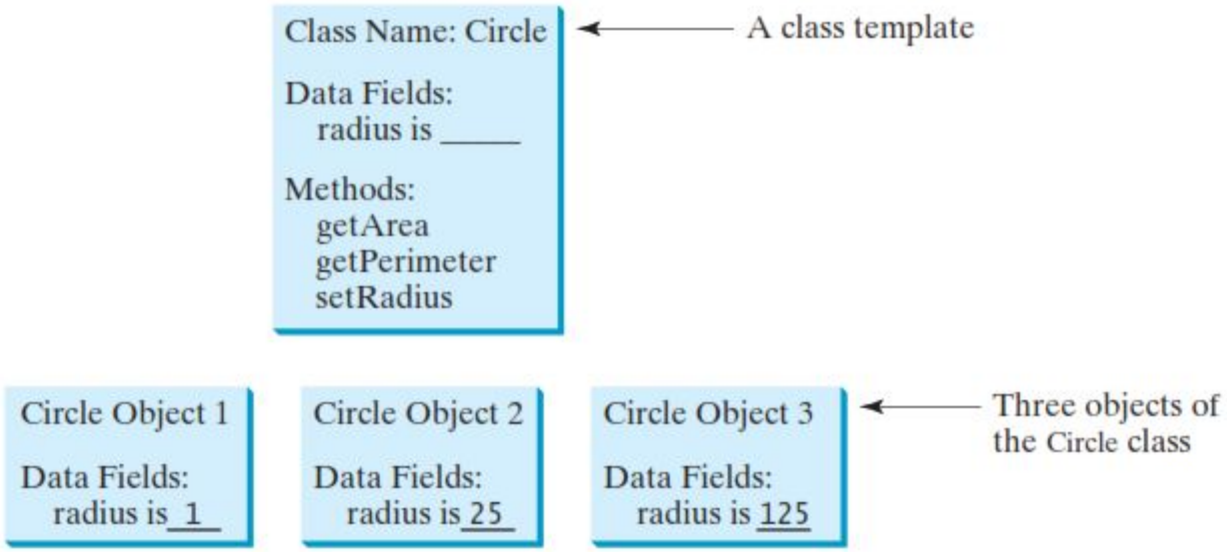
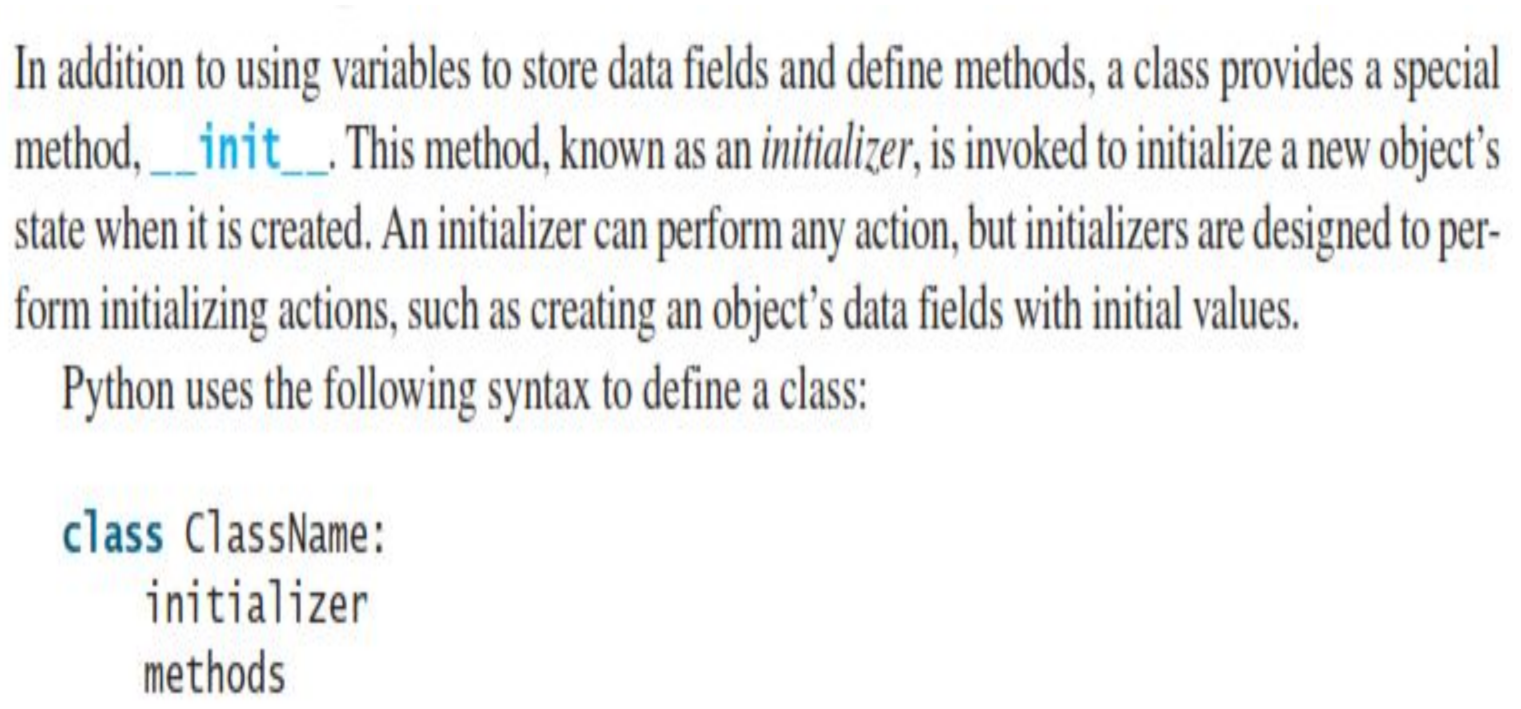


FIGURE 7.2 A class is a template—or contract—for creating objects.

Defining classes



In addition to using variables to store data fields and define methods, a class provides a special method, `__init__`. This method, known as an *initializer*, is invoked to initialize a new object's state when it is created. An initializer can perform any action, but initializers are designed to perform initializing actions, such as creating an object's data fields with initial values.

Python uses the following syntax to define a class:

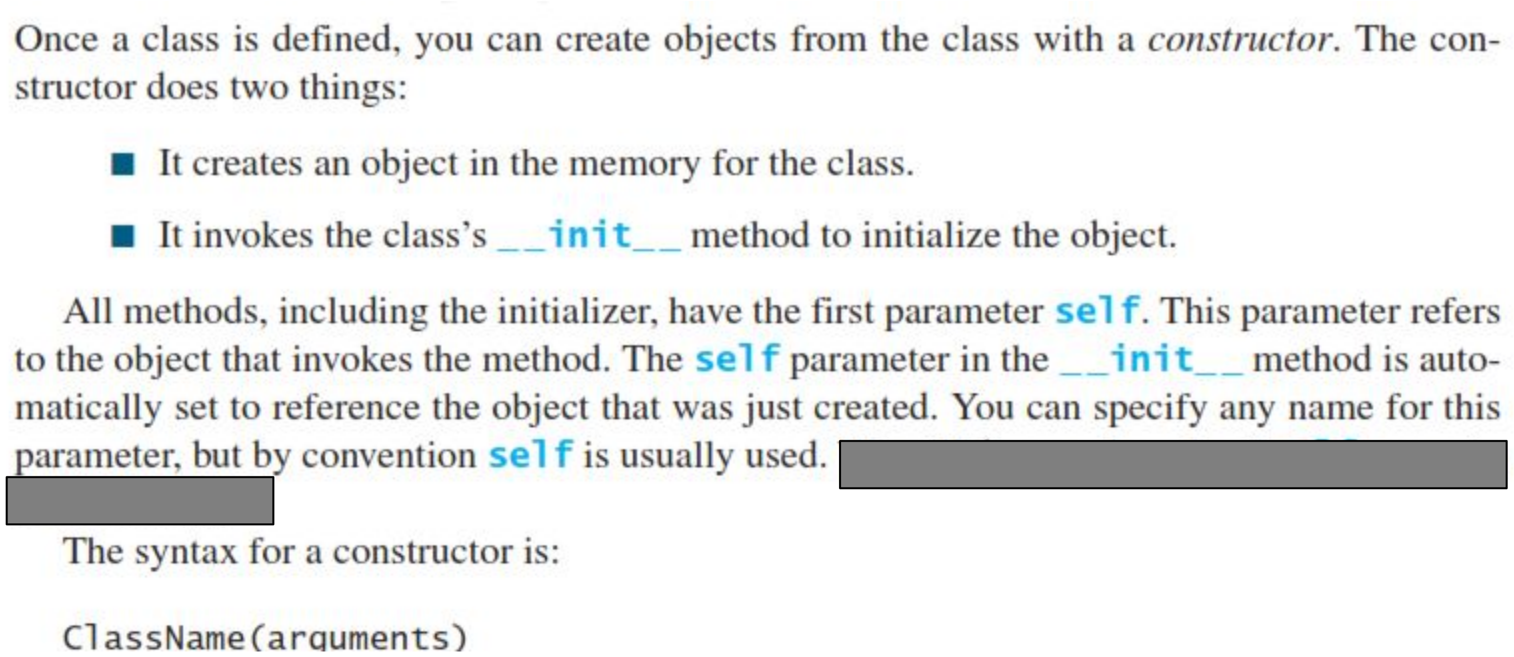
```
class ClassName:
    initializer
    methods
```

Listing 7.1 defines the `Circle` class. The class name is preceded by the keyword `class` and followed by a colon (`:`). The initializer is always named `__init__` (line 5), which is a special method. *Note that* `init` *needs to be preceded and followed by two underscores.* A data field `radius` is created in the initializer (line 6). The methods `getPerimeter` and `getArea` are defined to return the perimeter and area of a circle (lines 8–12). More details on the initializer, data fields, and methods will be explained in the following sections.

LISTING 7.1 Circle.py

```
 1  import math
 2
 3  class Circle:
 4      # Construct a circle object
 5      def __init__(self, radius = 1):
 6          self.radius = radius
 7
 8      def getPerimeter(self):
 9          return 2 * self.radius * math.pi
10
11      def getArea(self):
12          return self.radius * self.radius * math.pi
13
14      def setRadius(self, radius):
15          self.radius = radius
```

Constructing objects



Once a class is defined, you can create objects from the class with a *constructor*. The constructor does two things:

- It creates an object in the memory for the class.
- It invokes the class's `__init__` method to initialize the object.

All methods, including the initializer, have the first parameter `self`. This parameter refers to the object that invokes the method. The `self` parameter in the `__init__` method is automatically set to reference the object that was just created. You can specify any name for this parameter, but by convention `self` is usually used. [redacted]

The syntax for a constructor is:

```
ClassName(arguments)
```

Figure 7.3 shows how an object is created and initialized. After the object is created, `self` can be used to reference the object.

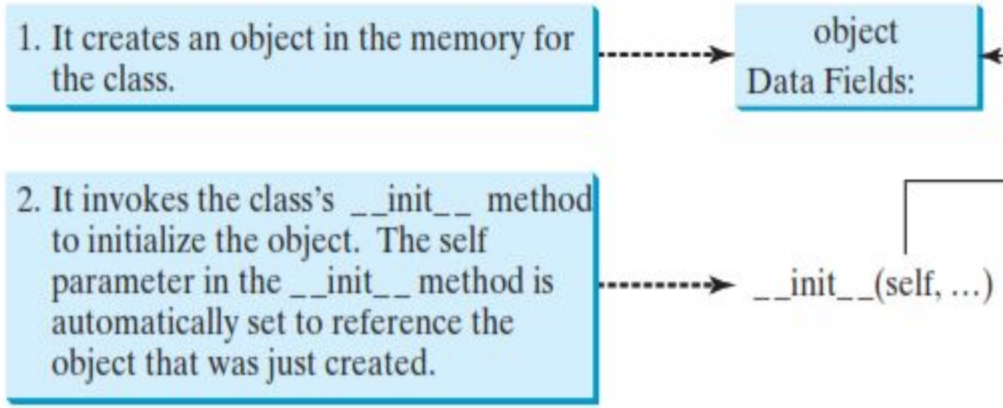

1. It creates an object in the memory for the class.
object
Data Fields:
2. It invokes the class's __init__ method to initialize the object. The self parameter in the __init__ method is automatically set to reference the object that was just created.
__init__(self, ...)

FIGURE 7.3 Constructing an object creates the object in the memory and invokes its initializer.

The arguments of the constructor match the parameters in the `__init__` method without `self`. For example, since the `__init__` method in line 5 of Listing 7.1 is defined as `__init__(self, radius = 1)`, to construct a `Circle` object with radius `5`, you should use `Circle(5)`. Figure 7.4 shows the effect of constructing a `Circle` object using `Circle(5)`. First, a `Circle` object is created in the memory, and then the initializer is invoked to set `radius` to `5`.

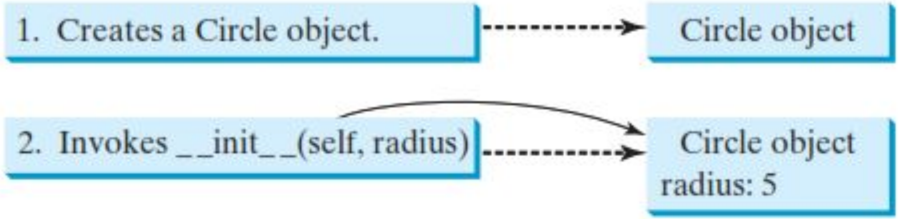
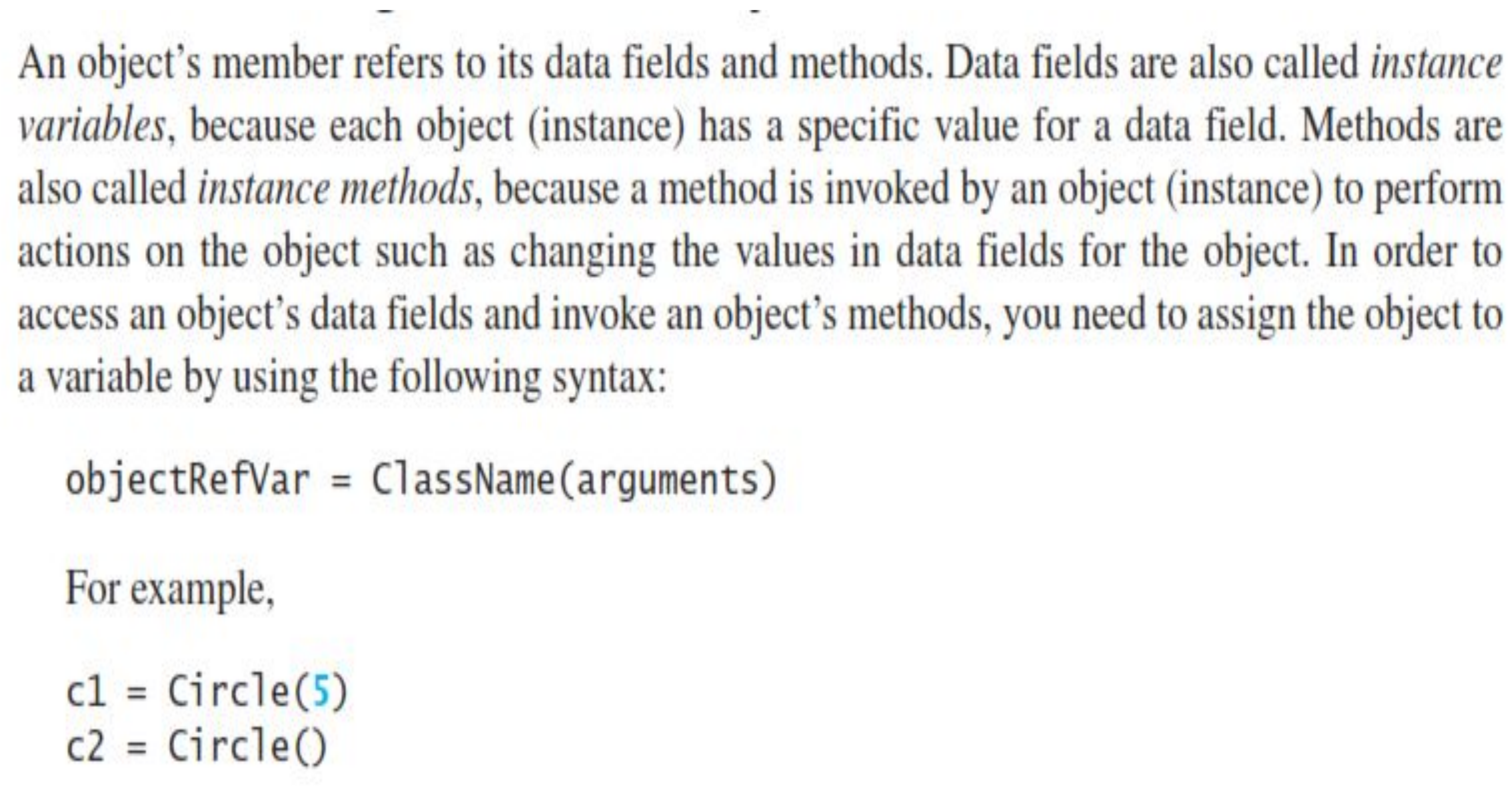

1. Creates a Circle object.
Circle object
2. Invokes __init__(self, radius)
Circle object
radius: 5

FIGURE 7.4 A circle object is constructed using `Circle(5)`.

The initializer in the `Circle` class has a default `radius` value of `1`. The following constructor creates a `Circle` object with default radius `1`:

```
Circle()
```

Accessing members of objects



An object's member refers to its data fields and methods. Data fields are also called *instance variables*, because each object (instance) has a specific value for a data field. Methods are also called *instance methods*, because a method is invoked by an object (instance) to perform actions on the object such as changing the values in data fields for the object. In order to access an object's data fields and invoke an object's methods, you need to assign the object to a variable by using the following syntax:

```
objectRefVar = ClassName(arguments)
```

For example,

```
c1 = Circle(5)
c2 = Circle()
```

You can access the object's data fields and invoke its methods by using the *dot operator* (`.`), also known as the *object member access operator*. The syntax for using the dot operator is:

```
objectRefVar.datafield
objectRefVar.method(args)
```

For example, the following code accesses the **`radius`** data field (line 3), and then invokes the **`getPerimeter`** method (line 5) and the **`getArea`** method (line 7). Note that line 1 imports the **`Circle`** class defined in the Circle module in Listing 7.1, Circle.py.

```
1  >>> from Circle import Circle
2  >>> c = Circle(5)
3  >>> c.radius
4  5
5  >>> c.getPerimeter()
6  31.41592653589793
7  >>> c.getArea()
8  78.53981633974483
9  >>>
```

Note

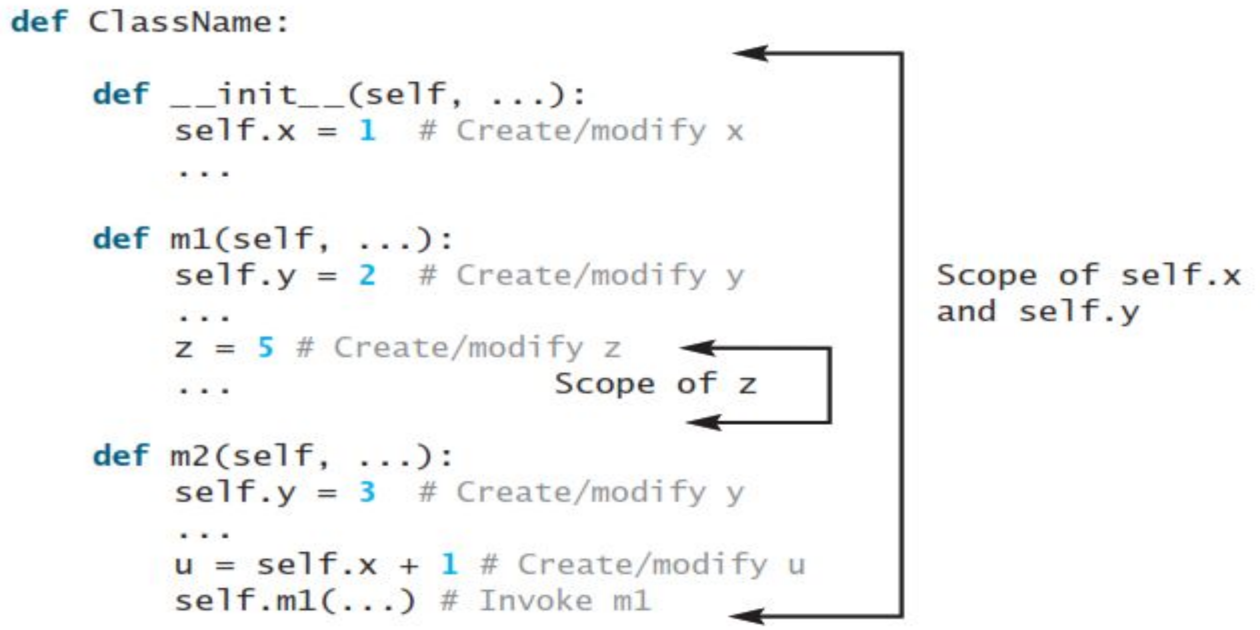
Usually you create an object and assign it to a variable. Later you can use the variable to reference the object. Occasionally an object does not need to be referenced later. In this case, you can create an object without explicitly assigning it to a variable, as shown below:

```
print("Area is", Circle(5).getArea())
```

The statement creates a `Circle` object and invokes its `getArea` method to return its area. An object created in this way is known as an *anonymous object*.

Self parameter

self is a parameter that references the object itself. Using self, you can access object's members in a class definition. For example, you can use the syntax self.x to access the instance variable x and syntax self.m1() to invoke the instance method m1 for the object self in a class, as illustrated in Figure 7.5.



```
def ClassName:

    def __init__(self, ...):
        self.x = 1  # Create/modify x
        ...

    def m1(self, ...):
        self.y = 2  # Create/modify y
        ...
        z = 5 # Create/modify z
        ...

    def m2(self, ...):
        self.y = 3  # Create/modify y
        ...
        u = self.x + 1 # Create/modify u
        self.m1(...) # Invoke m1
```

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 Figure 7.5, `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`. Note that 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`.

LISTING 7.2 TestCircle.py

import Circle

main function

create object

invoke methods

create object

create object

```python
from Circle import Circle

def main():
    # Create a circle with radius 1
    circle1 = Circle()
    print("The area of the circle of radius",
        circle1.radius , "is", circle1.getArea())

    # Create a circle with radius 25
    circle2 = Circle(25)
    print("The area of the circle of radius",
        circle2.radius, "is", circle2.getArea() )

    # Create a circle with radius 125
    circle3 = Circle(125)
    print("The area of the circle of radius",
        circle3.radius , "is", circle3.getArea() )

    # Modify circle radius
    circle2.radius = 100 # or circle2.setRadius(100)
    print("The area of the circle of radius",
        circle2.radius, "is", circle2.getArea())

main() # Call the main function
```

```
The area of the circle of radius 1.0 is 3.141592653589793
The area of the circle of radius 25.0 is 1963.4954084936207
The area of the circle of radius 125.0 is 49087.385212340516
The area of the circle of radius 100.0 is 31415.926535897932
```

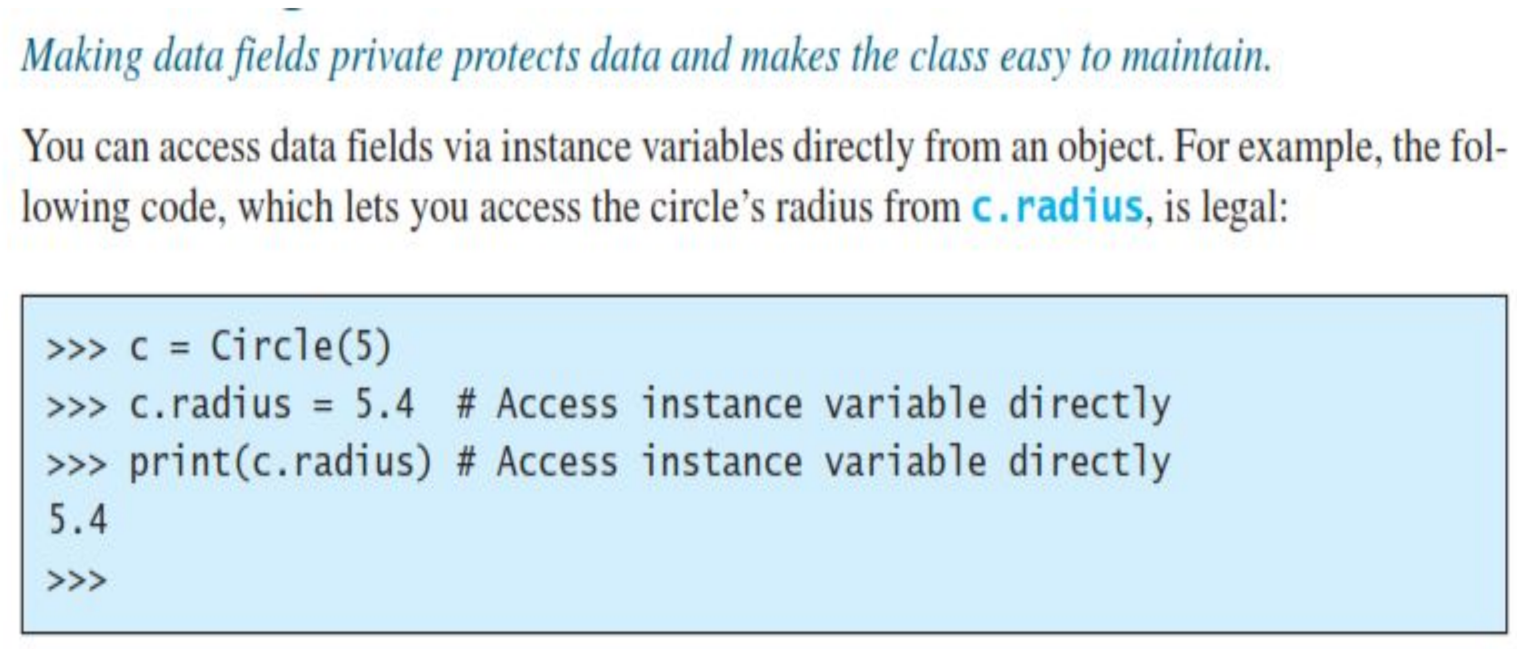
The program uses the `Circle` class to create `Circle` objects. Such a program that uses the class (such as `Circle`) is often referred to as a *client* of the class.

The `Circle` class is defined in Listing 7.1, Circle.py, and this program imports it in line 1 using the syntax `from Circle import Circle`. The program creates a `Circle` object with a default radius `1` (line 5) and creates two `Circle` objects with the specified radii (lines 10, 15), and then retrieves the `radius` property and invokes the `getArea()` method on the objects to obtain the area (lines 7, 12, and 17). The program sets a new `radius` property on `circle2` (line 20). This can also be done by using `circle2.setRadius(100)`.

Note

An variable that appears to hold an object actually contains a reference to that object. Strictly speaking, a variable and an object are different, but most of the time the distinction can be ignored. So it is fine, for simplicity, to say that "`circle1` is a `Circle` object" rather than use the longer-winded description that "`circle1` is a variable that contains a reference to a `Circle` object."

Hiding data fields



Making data fields private protects data and makes the class easy to maintain.

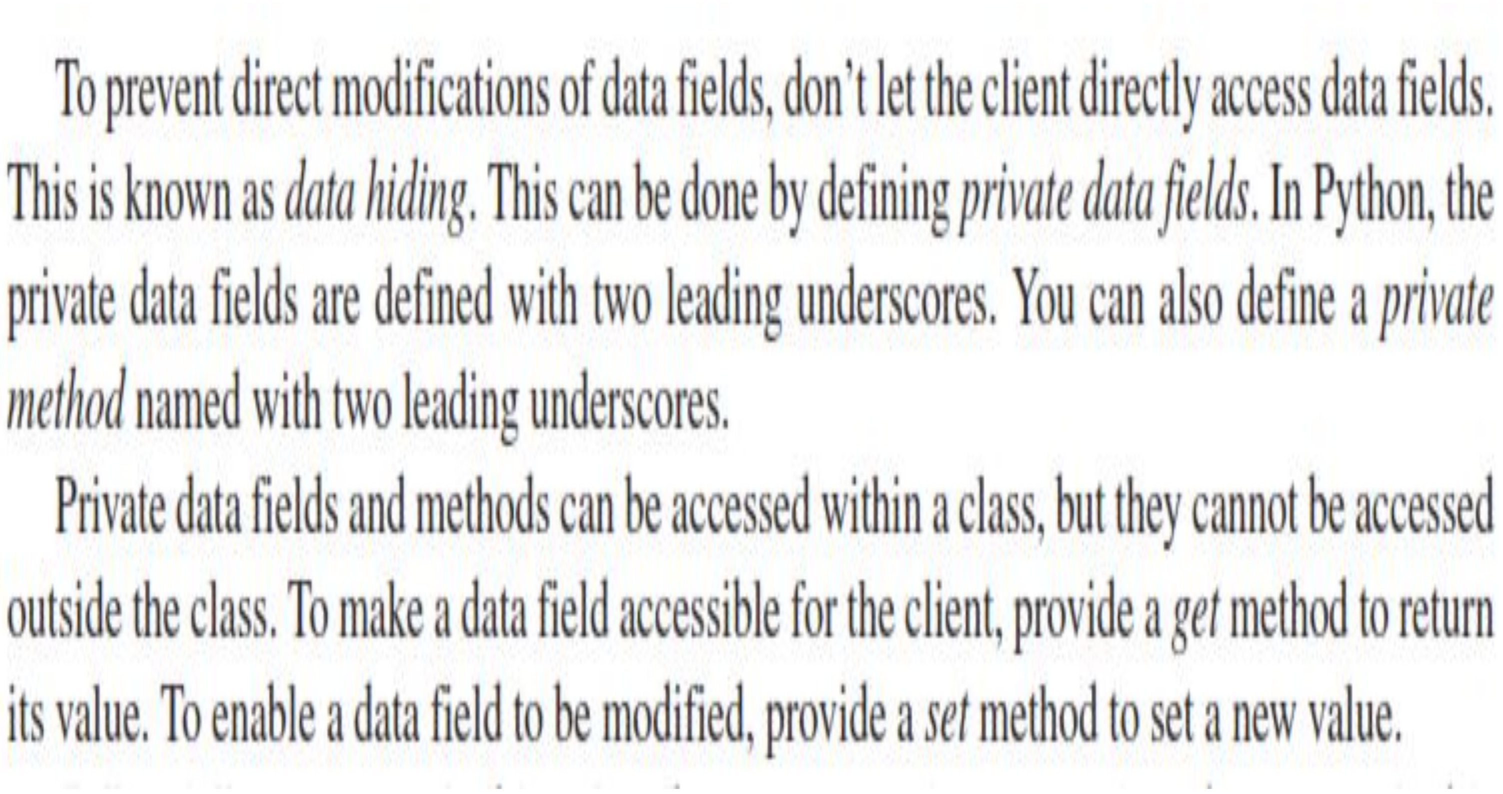
You can access data fields via instance variables directly from an object. For example, the following code, which lets you access the circle's radius from `c.radius`, is legal:

```
>>> c = Circle(5)
>>> c.radius = 5.4  # Access instance variable directly
>>> print(c.radius) # Access instance variable directly
5.4
>>>
```

However, direct access of a data field in an object is not a good practice—for two reasons:

- First, data may be tampered with. For example, `channel` in the `TV` class has a value between `1` and `120`, but it may be mistakenly set to an arbitrary value (e.g., `tv1.channel = 125`).
- Second, the class becomes difficult to maintain and vulnerable to bugs. Suppose you want to modify the `Circle` class to ensure that the radius is nonnegative after other programs have already used the class. You have to change not only the `Circle` class but also the programs that use it, because the clients may have modified the radius directly (e.g., `myCircle.radius = -5`).

Data hiding-private



To prevent direct modifications of data fields, don't let the client directly access data fields. This is known as *data hiding*. This can be done by defining *private data fields*. In Python, the private data fields are defined with two leading underscores. You can also define a *private method* named with two leading underscores.

Private data fields and methods can be accessed within a class, but they cannot be accessed outside the class. To make a data field accessible for the client, provide a *get* method to return its value. To enable a data field to be modified, provide a *set* method to set a new value.

[illegible]

Colloquially, a **get** method is referred to as a *getter* (or *accessor*), and a **set** method is referred to as a *setter* (or *mutator*).

A **get** method has the following header:

```
def getPropertyName(self):
```

If the return type is Boolean, the **get** method is defined as follows by convention:

```
def isPropertyName(self):
```

A **set** method has the following header:

```
def setPropertyName(self, propertyValue):
```

Listing 7.6 revises the **Circle** class in Listing 7.1 by defining the **radius** property as p vate by placing two underscores in front of the property name (line 6).

Listing 7.6 CircleWithPrivateRadius.py

```
 1  import math
 2  
 3  class Circle:
 4      # Construct a circle object
 5      def __init__(self, radius = 1):
 6          self.__radius = radius
 7  

 8      def getRadius(self):
 9          return self.__radius
10  
11      def getPerimeter(self):
12          return 2 * self.__radius * math.pi
13  
14      def getArea(self):
15          return self.__radius * self.__radius * math.pi
```

The **radius** property cannot be directly accessed in this new **Circle** class. However, you can read it by using the **getRadius()** method. For example:

```
1  >>> from CircleWithPrivateRadius import Circle
2  >>> c = Circle(5)
3  >>> c.__radius
4  AttributeError: no attribute '__radius'
5  >>> c.getRadius()
6  5
7  >>>
```

Line 1 imports the **Circle** class, which is defined in the **CircleWithPrivateRadius** module in Listing 7.6. Line 2 creates a **Circle** object. Line 3 attempts to access the property **__radius**. This causes an error, because **__radius** is private. However, you can use the **getRadius()** method to return the **radius** (line 5).

Tip

If a class is designed for other programs to use, to prevent data from being tampered with and to make the class easy to maintain, define data fields as private. If a class is only used internally by your own program, there is no need to hide the data fields.

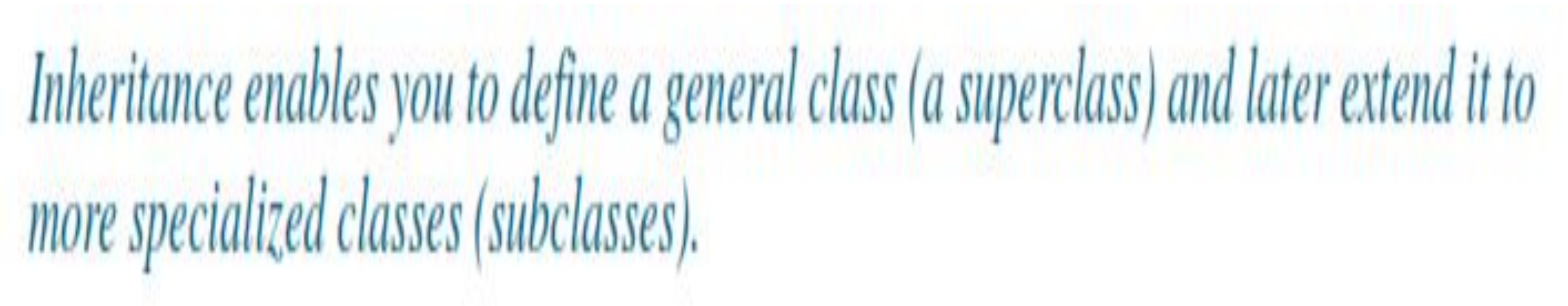
Note

Name private data fields and methods with two leading underscores, but don't end the name with more than one underscores. The names with two leading underscores and two ending underscores have special meaning in Python. For example, `__radius` is a private data field, but, `__radius__` is not a private data field.

inheritance

- ★ Object-oriented programming (OOP) allows you to define new classes from existing classes. This is called inheritance.
- ★ Inheritance extends the power of the object-oriented paradigm by adding an important and powerful feature <u>for reusing software.</u>
- ★ Suppose that you want to define classes to model circles, rectangles, and triangles. These classes have many common features.
- ★ What is the best way to design these classes to avoid redundancy and make the system easy to comprehend and maintain? The answer is to use inheritance.

Superclasses and sub classes



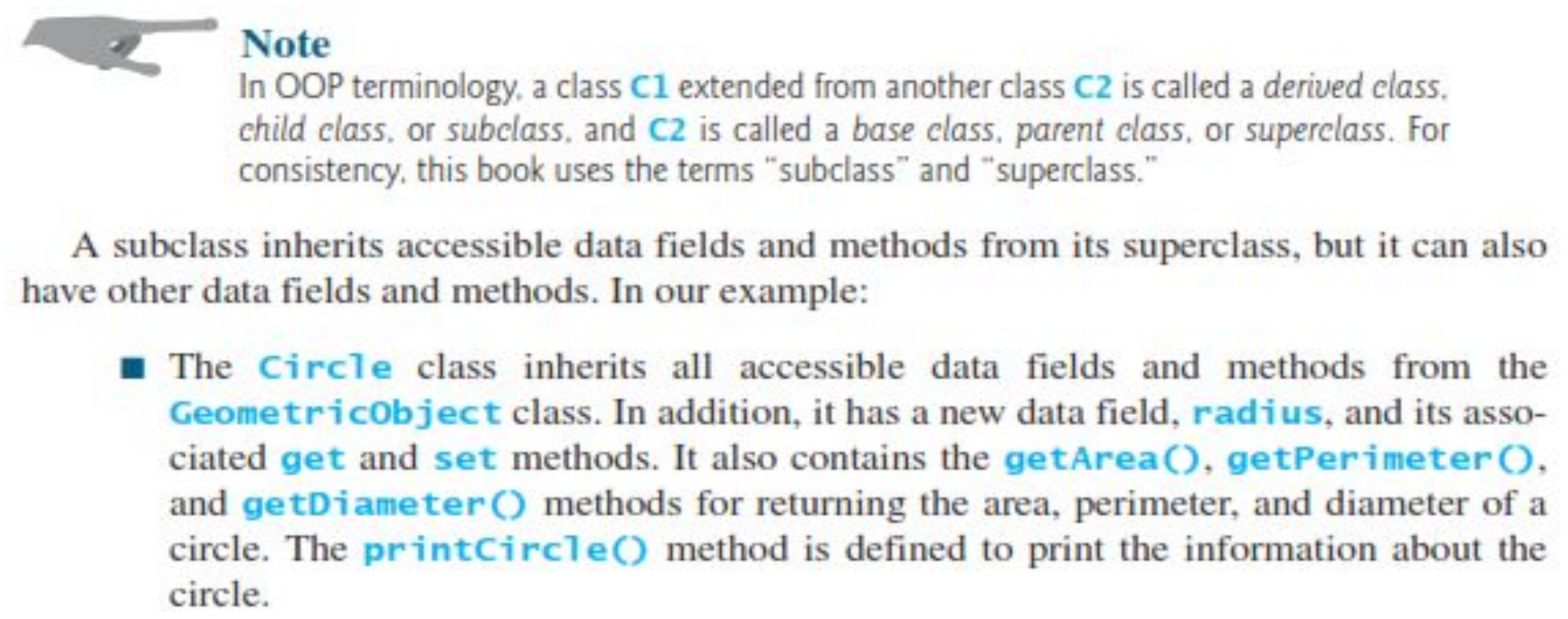
Inheritance enables you to define a general class (a superclass) and later extend it to more specialized classes (subclasses).

- ★ You use a class to model objects of the same type.
- ★ Different classes may have some common properties and behaviors that you can generalize in a class, which can then be shared by other classes.
- ★ Inheritance enables you to define a general class and later extend it to define more specialized classes.
- ★ The specialized classes inherit the properties and methods from the general class.

Consider geometric objects. Suppose you want to design classes to model geometric objects such as circles and rectangles. Geometric objects have many common properties and behaviors; for example, they can be drawn in a certain color, and they can be either filled or unfilled. Thus, a general class `GeometricObject` can be used to model all geometric objects. This class contains the properties `color` and `filled` and their appropriate `get` and `set` methods. Assume that this class also contains the `dateCreated` property and the `getDateCreated()` and `__str__()` methods. The `__str__()` method returns a string description for the object.

Because a circle is a special type of geometric object, it shares common properties and methods with other geometric objects. For this reason, it makes sense to define a `Circle` class that extends the `GeometricObject` class. Similarly, you can define `Rectangle` as a subclass of `GeometricObject`. Figure 12.1 shows the relationship among these classes. A *triangular arrow* pointing to the superclass is used to denote the inheritance relationship between the two classes involved.

Sub class & super class



Note
In OOP terminology, a class `C1` extended from another class `C2` is called a *derived class*, *child class*, or *subclass*, and `C2` is called a *base class*, *parent class*, or *superclass*. For consistency, this book uses the terms "subclass" and "superclass."

A subclass inherits accessible data fields and methods from its superclass, but it can also have other data fields and methods. In our example:

- The `Circle` class inherits all accessible data fields and methods from the `GeometricObject` class. In addition, it has a new data field, `radius`, and its associated `get` and `set` methods. It also contains the `getArea()`, `getPerimeter()`, and `getDiameter()` methods for returning the area, perimeter, and diameter of a circle. The `printCircle()` method is defined to print the information about the circle.

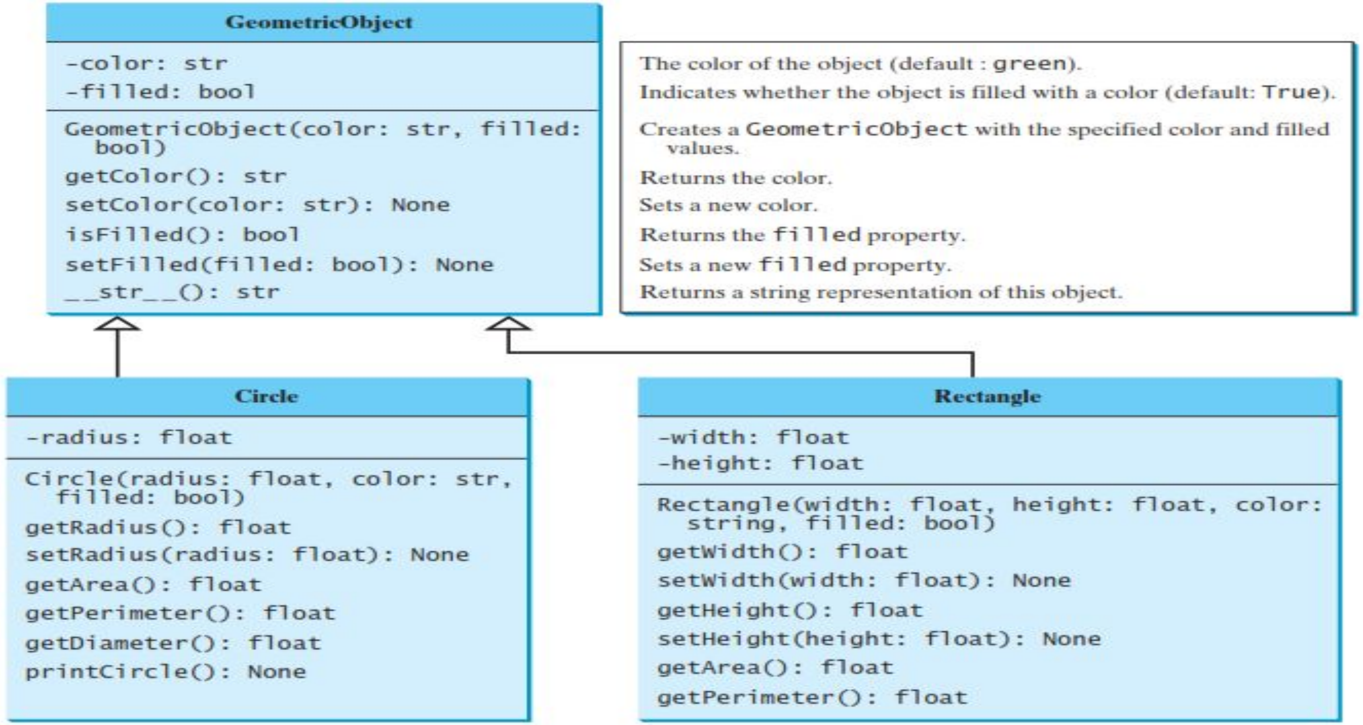

GeometricObject
-color: str
-filled: bool
GeometricObject(color: str, filled: bool)
getColor(): str
setColor(color: str): None
isFilled(): bool
setFilled(filled: bool): None
__str__(): str
The color of the object (default : green).
Indicates whether the object is filled with a color (default: True).
Creates a GeometricObject with the specified color and filled values.
Returns the color.
Sets a new color.
Returns the filled property.
Sets a new filled property.
Returns a string representation of this object.
Circle
-radius: float
Circle(radius: float, color: str, filled: bool)
getRadius(): float
setRadius(radius: float): None
getArea(): float
getPerimeter(): float
getDiameter(): float
printCircle(): None
Rectangle
-width: float
-height: float
Rectangle(width: float, height: float, color: string, filled: bool)
getWidth(): float
setWidth(width: float): None
getHeight(): float
setHeight(height: float): None
getArea(): float
getPerimeter(): float

FIGURE 12.1 The GeometricObject class is the superclass for Circle and Rectangle.

- The Rectangle class inherits all accessible data fields and methods from the GeometricObject class. In addition, it has the data fields width and height and the associated get and set methods. It also contains the getArea() and getPerimeter() methods for returning the area and perimeter of the rectangle.

The GeometricObject, Circle, and Rectangle classes are shown in Listings 12.1, 12.2, and 12.3.

LISTING 12.1 GeometricObject.py

- The `Rectangle` class inherits all accessible data fields and methods from the `GeometricObject` class. In addition, it has the data fields `width` and `height` and the associated `get` and `set` methods. It also contains the `getArea()` and `getPerimeter()` methods for returning the area and perimeter of the rectangle.

The `GeometricObject`, `Circle`, and `Rectangle` classes are shown in Listings 12.1, 12.2, and 12.3.

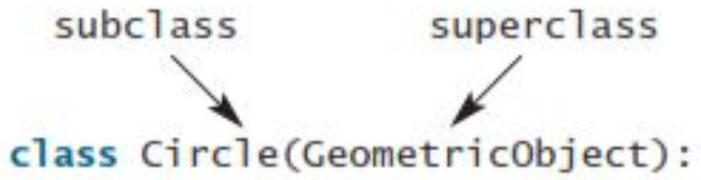
LISTING 12.1 GeometricObject.py

```python
class GeometricObject:
    def __init__(self, color = "green", filled = True):
        self.__color = color
        self.__filled = filled

    def getColor(self):
        return self.__color

    def setColor(self, color):
        self.__color = color

    def isFilled(self):
        return self.__filled

    def setFilled(self, filled):
        self.__filled = filled

    def __str__(self):
        return "color: " + self.__color + \
            " and filled: " + str(self.__filled)
```

Listing 12.2 CircleFromGeometricObject.py

```python
from GeometricObject import GeometricObject
import math # math.pi is used in the class

class Circle(GeometricObject):
    def __init__(self, radius):
        super().__init__()
        self.__radius = radius

    def getRadius(self):
        return self.__radius

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

    def getArea(self):
        return self.__radius * self.__radius * math.pi

    def getDiameter(self):
        return 2 * self.__radius

    def getPerimeter(self):
        return 2 * self.__radius * math.pi

    def printCircle(self):
        print(self.__str__() + " radius: " + str(self.__radius))
```

The `Circle` class is derived from the `GeometricObject` class (Listing 12.1), based on the following syntax:



```
class Circle(GeometricObject):
```

This tells Python that the `Circle` class inherits the `GeometricObject` class, thus inheriting the methods `getColor`, `setColor`, `isFilled`, `setFilled`, and `__str__`. The `printCircle` method invokes the `__str__()` method defined to obtain properties defined in the superclass (line 25).

`super().__init__()` calls the superclass's `__init__` method (line 6). This is necessary to create data fields defined in the superclass.

Note

Alternatively, you can invoke the superclass's `__init__` method by using:

```
GeometricObject.__init__(self)
```

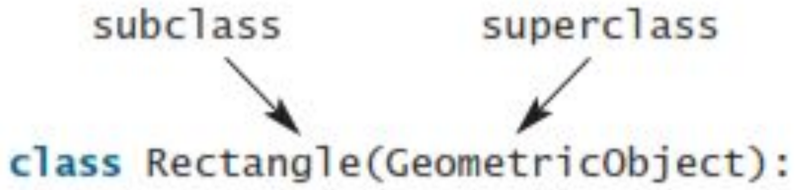
This is an old style of syntax that is still supported in Python, but it isn't the preferred style. **super()** refers to the superclass. Using **super()** lets you avoid referring the superclass explicitly. When invoking a method using **super()**, don't pass **self** in the argument. For example, you should use

```
super().__init__()
```

rather than

```
super().__init__(self)
```

The **Rectangle** class, derived from the **GeometricObject** class (Listing 12.1), is defined similarly in Listing 12.3.



```
class Rectangle(GeometricObject):
```

LISTING 12.3 RectangleFromGeometricObject.py

```python
from GeometricObject import GeometricObject

class Rectangle(GeometricObject):
    def __init__(self, width = 1, height = 1):
        super().__init__()
        self.__width = width
        self.__height = height

    def getWidth(self):
        return self.__width

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

    def getHeight(self):
        return self.__height

    def setHeight(self, height):
        self.__height = self.__height

    def getArea(self):
        return self.__width * self.__height

    def getPerimeter(self):
        return 2 * (self.__width + self.__height)
```

extend superclass
initializer
superclass initializer
methods

The code in Listing 12.4 creates **Circle** and **Rectangle** objects and invokes the **getArea()** and **getPerimeter()** methods on these objects. The **__str__()** method is inherited from the **GeometricObject** class and is invoked from a **Circle** object (line 5) and a **Rectangle** object (line 11).

LISTING 12.4 TestCircleRectangle.py

```python
from CircleFromGeometricObject import Circle
from RectangleFromGeometricObject import Rectangle

def main():
    circle = Circle(1.5)
    print("A circle", circle)
    print("The radius is", circle.getRadius())
    print("The area is", circle.getArea())
    print("The diameter is", circle.getDiameter())

    rectangle = Rectangle(2, 4)
    print("\nA rectangle", rectangle)
    print("The area is", rectangle.getArea())
    print("The perimeter is", rectangle.getPerimeter())

main() # Call the main function
```

```
A circle color: green and filled: True
The radius is 1.5
The area is 7.06858347058
The diameter is 3.0

A rectangle color: green and filled: True
The area is 8
The perimeter is 12
```

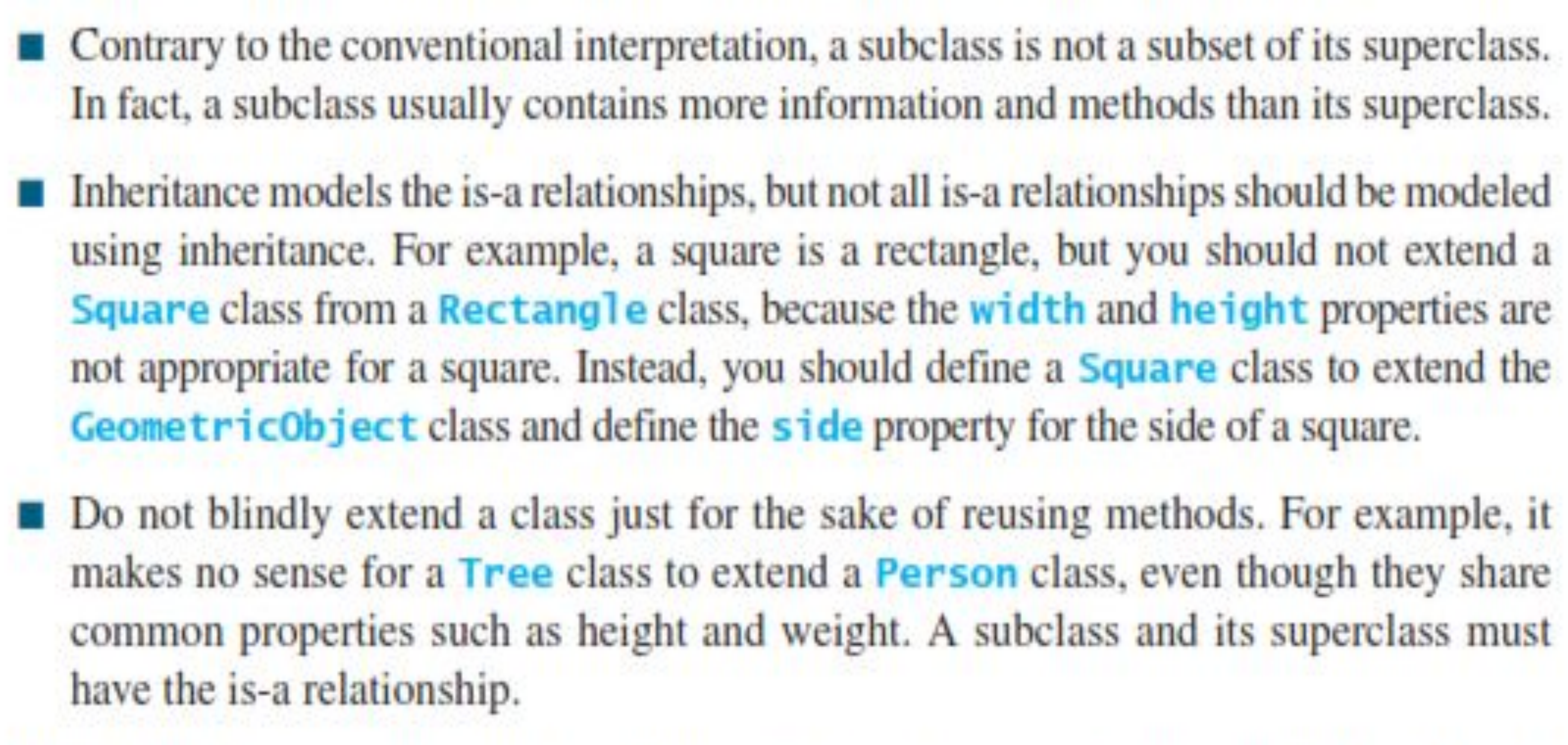
Line 6 invokes the print function to print a circle. Recall from Section 8.5, this is the same as

```
print("A circle", circle.__str__())
```

The __str__() method is not defined in the Circle class, but is defined in the GeometricObject class. Since Circle is a subclass of GeometricObject, __str__() can be invoked from a Circle object.

The __str__() method displays the color and filled properties of a GeometricObject (lines 18–20 in Listing 12.1). The default color for a GeometricObject object is green and filled is True (line 2 Listing 12.1). Since a Circle inherits from GeometricObject, the default color for a Circle object is green and the default value for filled is True.

Inheritance-some points



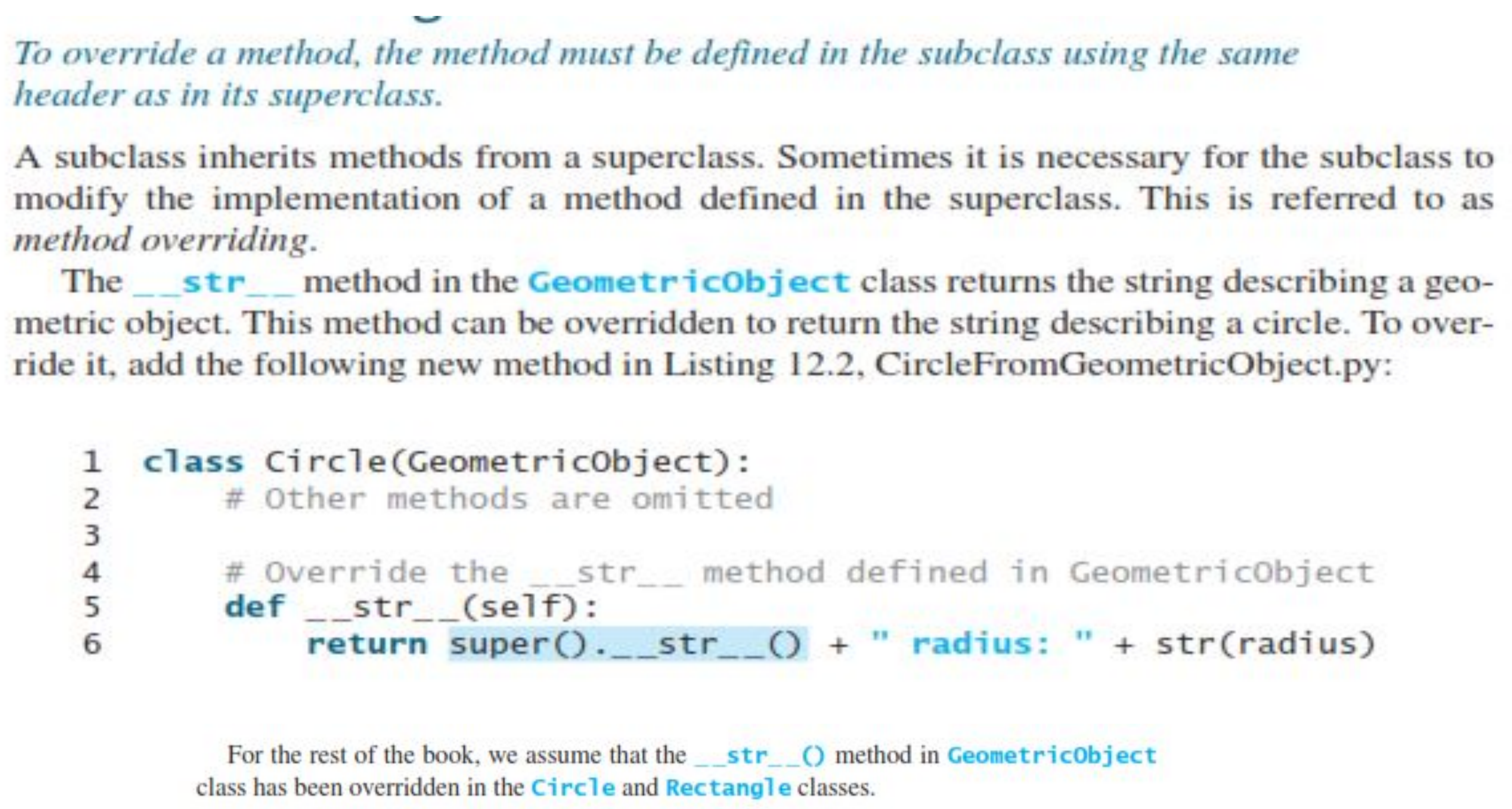
- Contrary to the conventional interpretation, a subclass is not a subset of its superclass. In fact, a subclass usually contains more information and methods than its superclass.
- Inheritance models the is-a relationships, but not all is-a relationships should be modeled using inheritance. For example, a square is a rectangle, but you should not extend a `Square` class from a `Rectangle` class, because the `width` and `height` properties are not appropriate for a square. Instead, you should define a `Square` class to extend the `GeometricObject` class and define the `side` property for the side of a square.
- Do not blindly extend a class just for the sake of reusing methods. For example, it makes no sense for a `Tree` class to extend a `Person` class, even though they share common properties such as height and weight. A subclass and its superclass must have the is-a relationship.

have the is-a relationship.

- Python allows you to derive a subclass from several classes. This capability is known as *multiple inheritance*. To define a class derived from multiple classes, use the following syntax:

```
class Subclass(SuperClass1, SuperClass2, ...):
    initializer
    methods
```

Overriding methods



To override a method, the method must be defined in the subclass using the same header as in its superclass.

A subclass inherits methods from a superclass. Sometimes it is necessary for the subclass to modify the implementation of a method defined in the superclass. This is referred to as *method overriding*.

The `__str__` method in the `GeometricObject` class returns the string describing a geometric object. This method can be overridden to return the string describing a circle. To override it, add the following new method in Listing 12.2, CircleFromGeometricObject.py:

```
class Circle(GeometricObject):
    # Other methods are omitted

    # Override the __str__ method defined in GeometricObject
    def __str__(self):
        return super().__str__() + " radius: " + str(radius)
```

For the rest of the book, we assume that the `__str__()` method in `GeometricObject` class has been overridden in the `Circle` and `Rectangle` classes.

The `__str__()` method is defined in the `GeometricObject` class and modified in the `Circle` class. Both methods can be used in the `Circle` class. To invoke the `__str__` method defined in the `GeometricObject` class from the `Circle` class, use `super().__str__()` (line 6).

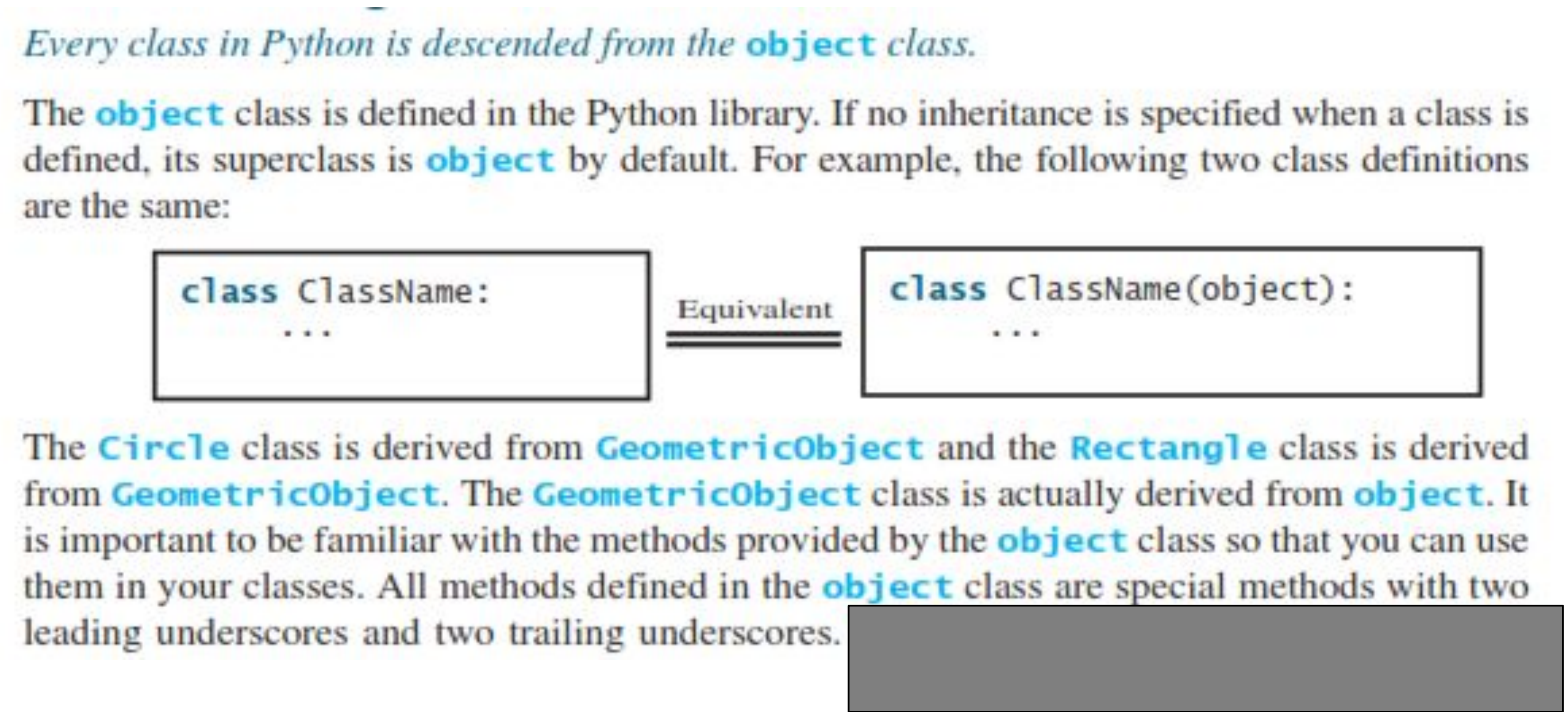
Similarly, you can override the `__str__` method in the `Rectangle` class as follows:

```
def __str__(self):
    return super().__str__() + " width: " + \
        str(self.__width) + " height: " + str(self.__height)
```

Note

Recall that you can define a private method in Python by adding two underscores in front of a method name (see Chapter 7). A private method cannot be overridden. If a method defined in a subclass is private in its superclass, the two methods are completely unrelated, even though they have the same name.

The object class



Every class in Python is descended from the `object` *class.*

The `object` class is defined in the Python library. If no inheritance is specified when a class is defined, its superclass is `object` by default. For example, the following two class definitions are the same:

```
class ClassName:
    ...
```

Equivalent

```
class ClassName(object):
    ...
```

The `Circle` class is derived from `GeometricObject` and the `Rectangle` class is derived from `GeometricObject`. The `GeometricObject` class is actually derived from `object`. It is important to be familiar with the methods provided by the `object` class so that you can use them in your classes. All methods defined in the `object` class are special methods with two leading underscores and two trailing underscores. [redacted]

The `__new__()` method is automatically invoked when an object is constructed. This method then invokes the `__init__()` method to initialize the object. Normally you should only override the `__init__()` method to initialize the data fields defined in the new class.

The `__str__()` method returns a string description for the object. By default, it returns a string consisting of a class name of which the object is an instance and the object's memory address in hexadecimal format. For example, consider the following code for the `Loan` class, which was defined in Listing 7.8:

```
loan = Loan(1, 1, 1, "Smith")
print(loan) # Same as print(loan.__str__())
```

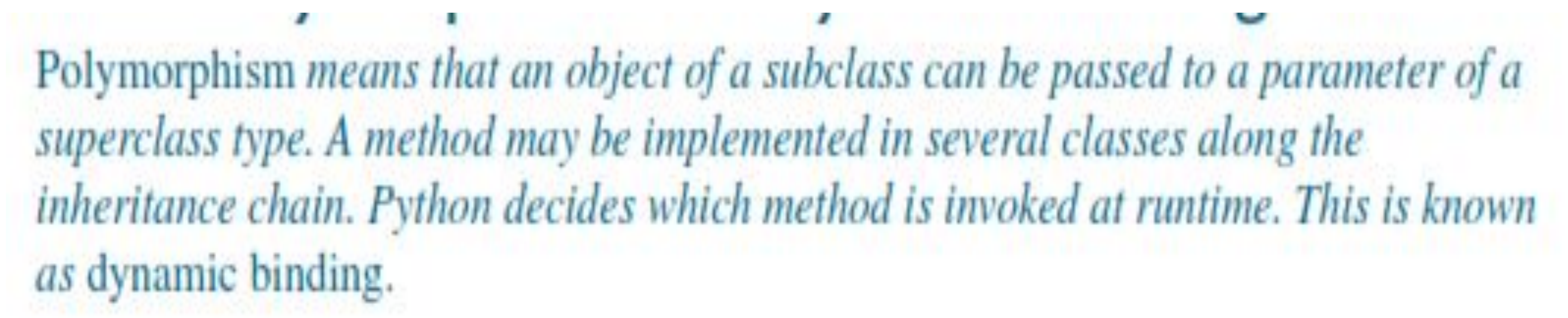
The code displays something like <Loan.Loan object at 0x01B99C10>. This message is not very helpful or informative. Usually you should override the __str__() method so that it returns an informative description for the object. For example, the __str__() method in the object class was overridden in the GeometricObject class in lines 18–20 in Listing 12.1 as follows:

```
def __str__(self):
    return "color: " + self.__color + \
        " and filled: " + str(self.__filled)
```

The __eq__(other) method returns True if two objects are the same. So, x.__eq__(x) is True, but x.__eq__(y) returns False, because x and y are two different objects even though they may have the same contents. Recall that x.__eq__(y) is same as x == y (see Section 8.5).

You can override this method to return True if two objects have the same contents. The __eq__ method is overridden in many Python built-in classes such as int, float, bool, string, and list to return True if two objects have the same contents.

Polymorphism   and dynamic binding



Polymorphism *means that an object of a subclass can be passed to a parameter of a superclass type. A method may be implemented in several classes along the inheritance chain. Python decides which method is invoked at runtime. This is known as* dynamic binding.

The three pillars of object-oriented programming are encapsulation, inheritance, and polymorphism.

The inheritance relationship enables a subclass to inherit features from its superclass with additional new features. A subclass is a specialization of its superclass; every instance of a subclass is also an instance of its superclass, but not vice versa. For example, every circle is a geometric object, but not every geometric object is a circle. Therefore, you can always pass an instance of a subclass to a parameter of its superclass type. Consider the code in Listing 12.5.

LISTING 12.5 PolymorphismDemo.py

```python
from CircleFromGeometricObject import Circle
from RectangleFromGeometricObject import Rectangle

def main():
    # Display circle and rectangle properties
    c = Circle(4)
    r = Rectangle(1, 3)
    displayObject(c)
    displayObject(r)
    print("Are the circle and rectangle the same size?",
        isSameArea(c, r))

# Display geometric object properties
def displayObject(g) :
    print(g.__str__())

# Compare the areas of two geometric objects
def isSameArea(g1, g2) :
    return g1.getArea() == g2.getArea()

main() # Call the main function
```

```
color: green and filled: True radius: 4
color: green and filled: True width: 1 height: 3
Are the circle and rectangle the same size? False
```

The `displayObject` method (line 14) takes a parameter of the `GeometricObject` type. You can invoke `displayObject` by passing any instance of `GeometricObject` (for example, `Circle(4)` and `Rectangle(1, 3)` in lines 8–9). An object of a subclass can be used wherever its superclass object is used. This is commonly known as *polymorphism* (from a Greek word meaning "many forms").

As seen in this example, `c` is an object of the `Circle` class. `Circle` is a subclass of `GeometricObject`. The `__str__()` method is defined in both classes. So, which `__str__()` method is invoked by `g` in the `displayObject` method (line 15)? The `__str__()` method invoked by `g` is determined using *dynamic binding*.

Dynamic binding works as follows: Suppose an object `o` is an instance of classes C_1, C_2, ..., C_{n-1}, and C_n, where C_1 is a subclass of C_2, C_2 is a subclass of C_3, ..., and C_{n-1} is a subclass of C_n, as shown in Figure 12.2. That is, C_n is the most general class, and C_1 is the most specific class. In Python, C_n is the `object` class. If `o` invokes a method `p`, Python searches the implementation for the method `p` in C_1, C_2, ..., C_{n-1}, and C_n, in this order, until it is found. Once an implementation is found, the search stops and the first-found implementation is invoked.

Listing 12.6 provides an example that demonstrates dynamic binding.

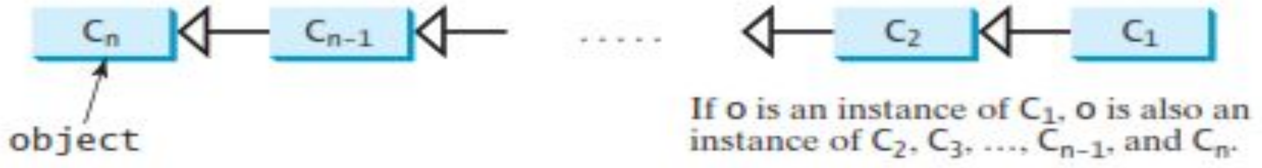

C_n
C_{n-1}
C_2
C_1
object
If o is an instance of C_1, o is also an instance of C_2, C_3, ..., C_{n-1}, and C_n.

FIGURE 12.2 The method to be invoked is dynamically bound at runtime.

LISTING 12.6 DynamicBindingDemo.py

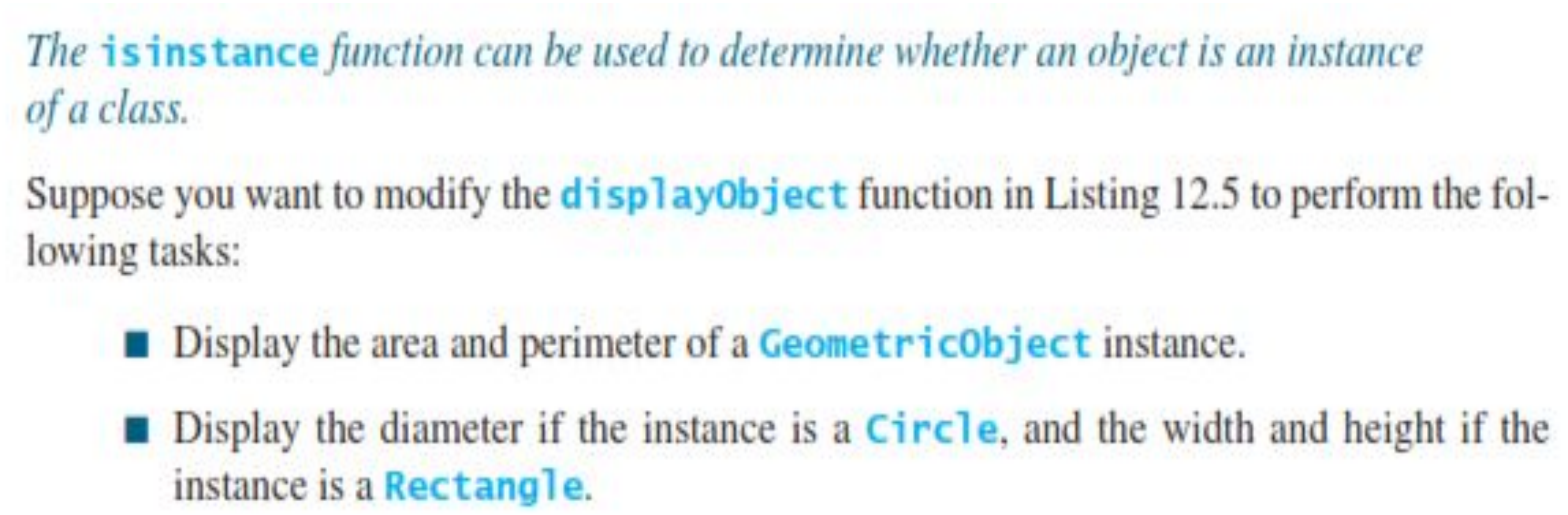
```python
class Student:
    def __str__(self):
        return "Student"

    def printStudent(self):
        print(self.__str__())

class GraduateStudent(Student):
    def __str__(self):
        return "Graduate Student"

a = Student()
b = GraduateStudent()
a.printStudent()
b.printStudent()
```

```
Student
Graduate Student
```

Since `a` is an instance of `Student`, the `printStudent` method in the `Student` class is invoked for `a.printStudent()` (line 14), which invokes the `Student` class's `__str__()` method to return `Student`.

No `printStudent` method is defined in `GraduateStudent`. However, since it is defined in the `Student` class and `GraduateStudent` is a subclass of `Student`, the `printStudent` method in the `Student` class is invoked for `b.printStudent()` (line 15). The `printStudent` method invokes `GraduateStudent`'s `__str__()` method to display `Graduate Student`, since the object `b` that invokes `printStudent` is `GraduateStudent` (lines 6 and 10).

Isinstance function



The `isinstance` *function can be used to determine whether an object is an instance of a class.*

Suppose you want to modify the `displayObject` function in Listing 12.5 to perform the following tasks:

- Display the area and perimeter of a `GeometricObject` instance.
- Display the diameter if the instance is a `Circle`, and the width and height if the instance is a `Rectangle`.

How can this be done? You might be tempted to write the function as:

```
def displayObject(g):
    print("Area is", g.getArea())
    print("Perimeter is", g.getPerimeter())
    print("Diameter is", g.getDiameter())
    print("Width is", g.getWidth())
    print("Height is", g.getHeight())
```

This won't work, however, because not all `GeometricObject` instances have the `getDiameter()`, `getWidth()`, or `getHeight()` methods. For example, invoking `display(Circle(5))` will cause a runtime error because `Circle` does not have the `getWidth()` and `getHeight()` methods, and invoking `display(Rectangle(2, 3))` will cause a runtime error because `Rectangle` does not have the `getDiameter()` method.

You can fix this problem by using Python's built-in `isinstance` function. This function determines whether an object is an instance of a class by using the following syntax:

```
isinstance(object, ClassName)
```

For example, `isinstance("abc", str)` returns `True` because `"abc"` is an instance of the `str` class, but `isinstance(12, str)` returns `False` because `12` is not an instance of the `str` class.

LISTING 12.7 IsinstanceDemo.py

```python
from CircleFromGeometricObject import Circle
from RectangleFromGeometricObject import Rectangle

def main():
    # Display circle and rectangle properties
    c = Circle(4)
    r = Rectangle(1, 3)
    print("Circle...")
    displayObject(c)
    print("Rectangle...")
    displayObject(r)

# Display geometric object properties
def displayObject(g):
    print("Area is", g.getArea())
    print("Perimeter is", g.getPerimeter())

    if isinstance(g, Circle):
        print("Diameter is", g.getDiameter())
    elif isinstance(g, Rectangle):
        print("Width is", g.getWidth())
        print("Height is", g.getHeight())

main() # Call the main function
```

```
Circle...
Area is 50.26548245743669
Perimeter is 25.132741228718345
Diameter is 8
Rectangle...
Area is 3
Perimeter is 8
Width is 1
Height is 3
```

Invoking displayObject(c) passes c to g (line 9). g is now an instance of Circle (line 18). The program displays the circle's diameter (line 19).

Invoking displayObject(r) passes r to g (line 11). g is now an instance of Rectangle (line 20). The program displays the rectangle's width and height (lines 21–22).