19. Inheritance

19.1. Pillars of OOP

19.2. Introduction to Inheritance

19.3. Extending

19.4. Reuse Through Composition

19.5. Class Diagrams

19.6. Composition vs. Inheritance

19.1. Pillars of OOP

Object-oriented programming involves four key ideas: encapsulation, information hiding, inheritance, and polymorphism. Encapsulation is the idea that a class can package some data together with the methods that manipulate the data. This is a powerful capability, and the chief idea that distinguishes OOP from structured programming. Information Hiding promotes quality code by allowing objects to protect their state against direct manipulation by code using the object. Python, like many languages, provides mechanisms to achieve information hiding, but we do not cover them in this book. Inheritance and polymorphism are mechanisms that help to enable code reuse and contract-based programming, and are the subject of this chapter.

19.2. Introduction to Inheritance¶

1 class Point:

2

​3 def \_\_init\_\_(self, initX, initY):

4 self.x = initX

5 self.y = initY

6

​7 def distanceFromOrigin(self):

8 return ((self.x \*\* 2) + (self.y \*\* 2)) \*\* 0.5

9

​10 def \_\_str\_\_(self):

11 return "x=" + str(self.x) + ", y=" + str(self.y)

12

​13 p = Point(7, 6)

14 print(p)

15

Now, suppose we want to create a class that works like Point in every respect, but also keeps track of a short description for the point. We could create a LabeledPoint class by copying and pasting the definition for Point, changing the name to LabeledPoint, and modifying the class to suit our purposes. However, any time you copy and paste code, keep in mind that you are copying and pasting bugs that may exist in the code. Inheritance provides a way to reuse the definition of Point without having to copy and paste.

We begin like this:

1 class LabeledPoint(Point):

2 pass

3

This example defines a class named LabeledPoint that inherits from the Point class. Putting the name Point in parenthesis tells Python that the new class, LabeledPoint, begins with all of the methods defined in its parent, Point. For example, we can instantiate LabeledPoint using the Point constructor, and invoke any Point methds we want to on it:

p = LabeledPoint(7,6)

dist = p.distanceFromOrigin()

Now, let’s refine LabeledPoint so that it holds a label, along with the x and y coordinates:

1 class LabeledPoint(Point):

2

​3 def \_\_init\_\_(self, initX, initY, label):

4 self.x = initX

5 self.y = initY

6 self.label = label

7

​8 def \_\_str\_\_(self):

9 r eturn "x=" + str(self.x) + ", y=" + str(self.y) + " (" + self.label + ")"

10

​11 labeledPt = LabeledPoint(7,6,"Here")

12 print(labeledPt)

13

Here, we have redefined two of the methods that LabeledPoint inherits from Point: \_\_init\_\_() and \_\_str\_\_(). This is called overriding. When a child class redefines methods that are defined in its parent, we say that the child overrides the functionality inherited from its parent. When both the parent class and child class have a method with the same name, an invocation of the method on an instance of the child class executes code in the child’s class; an invocation of the method on an instance of the parent class executes code in the parent’s class. For example, consider the following:

1 pt = Point(5,10)

2 labeledPt = LabeledPoint(7, 6, "Here")

3

​4 ptStr = str(pt)

5 labeledPtStr = str(labeledPt)

6

In Line 4, the call to str(pt) invokes the \_\_str\_\_() method in Point, because pt refers to an instance of Point. In Line 5, the call to str(labeledPt) invokes the \_\_str\_\_() method in LabeledPoint, because labeledPt refers to an instance of LabeledPoint.

19.3. Extending

If you compare the code in the \_\_init\_\_ methods of Point and LabeledPoint, you can see that there is some duplication–the initialization of x and y. We can eliminate the duplication by having LabeledPoint’s \_\_init\_\_() method invoke Point’s \_\_init\_\_() method. That way, each class will be responsible for initializing its own instance variables.

A method in a child class that overrides a method in the parent can invoke the overridden method using super(), like this:

class LabeledPoint(Point):

def \_\_init\_\_(self, initX, initY, label):

super().\_\_init\_\_(initX, initY)

self.label = label

In this example, line 4 invokes the \_\_init\_\_() method in Point, passing the values of initX and initY to be used in initializing the x and y instance variables.

Here is a complete code listing showing both classes, with a version of \_\_str\_\_() for LabeledPoint that invokes its parent’s implementation using super() to avoid duplicating the functionality provided in Point.

1 class Point:

2

​3 def \_\_init\_\_(self, initX, initY):

4 self.x = initX

5 self.y = initY

6

​7 def distanceFromOrigin(self):

8 return ((self.x \*\* 2) + (self.y \*\* 2)) \*\* 0.5

9

​10 def \_\_str\_\_(self):

11 return "x=" + str(self.x) + ", y=" + str(self.y)

12

​13 class LabeledPoint(Point):

14

​15 def \_\_init\_\_(self, initX, initY, label):

16 super().\_\_init\_\_(initX, initY)

17 s elf.label = label

18

​19 def \_\_str\_\_(self):

20 return super().\_\_str\_\_() + " (" + self.label + ")"

21

​22 p = LabeledPoint(7,6,"Here")

23 print(p)

24 print(p.distanceFromOrigin())

19.4. Reuse Through Composition

Inheritance is not the only way to reuse code. Composition occurs when an object stores a reference to one or more objects in one of its instance variables. The object is thus able to reuse code in the objects it embeds within itself.

For example, our LabeledPoint example could have been implemented as follows:

1 class Point:

2

​3 def \_\_init\_\_(self, initX, initY):

4 self.x = initX

5 self.y = initY

6

​7 def distanceFromOrigin(self):

8 return ((self.x \*\* 2) + (self.y \*\* 2)) \*\* 0.5

9

​10 def \_\_str\_\_(self):

11 return "x=" + str(self.x) + ", y=" + str(self.y)

12

​13 class LabeledPoint:

14

​15 def \_\_init\_\_(self, initX, initY, label):

16 self.point = Point(initX, initY)

17 self.label = label

18

​19 def distanceFromOrigin(self):

20 return self.point.distanceFromOrigin()

21

​22 def \_\_str\_\_(self):

23 return str(self.point) + " (" + self.label + ")"

24

The first thing to notice about this version of LabeledPoint does not inherit from Point. Instead, its constructor instantiates a Point and stores a reference to it in its point instance variable so that it can be used by the other methods.

Next, notice how the distanceFromOrigin() method reuses the code in Point` by invoking it. We say that ``LabeledPoint’s distanceFromOrigin() delegates its implementation to Point’s implementation.

Finally, notice how the \_\_str\_\_() method also reuses the code in Point by calling str(self.point).

19.5. Class Diagrams

When two classes are involved in an inheritance or composition relationship, we say that an association exists between them. Often it is helpful to depict the associations between classes on a graphical diagram, so that developers working on the code can see at a glance how the classes are related to each other. The Unified Modeling Language (UML) is a graphical notation that provides a standard for depicting classes and their associations on various types of diagrams. It gives software engineers a way to record both their proposed designs, as well as the design of the final product. Put another way, UML provides the notation for constructing blueprints for software, and since it is widely used in both industry and academia, it is important for you to have some familiarity with it.

UML defines notations for several different kinds of diagrams. Here, we will introduce class diagrams, one of the most common UML diagrams. Class diagrams can contain a number of different elements, but we will focus on the basics: depicting classes with their instance variables and methods, and relationships to other classes.

On a UML class diagram, a class is depicted as a rectangle with three sections. The top section contains the class name; the middle section, the instance variables; and the bottom section, the methods. The following diagram shows the classes Point and LabeledPoint in their original inheritance relationship.

19.6. Composition vs. Inheritance

Now you have seen two ways for a class to reuse code in another class. So, is one better than the other? When do you use inheritance, and when is composition the better choice?

Although the subject of this chapter is inheritance, the truth is that composition is usually a better choice than inheritance to reuse code. Perhaps 95% of cases where you are debating about choosing inheritance or composition, you should choose composition. It’s hard to go wrong with composition, but you can get into all kinds of trouble if you go with inheritance and inheritance is not an appropriate choice.

So, it’s easier to address the question of which technique to use by defining when inheritance is an appropriate choice. Inheritance is appropriate when the proposed child class (the one reusing the functionality in its parent) represents a specialization of its parent. Class A is a specialization of Class B if class A represents a specific type of class B. This is generally the case if you can fill in the following sentence with the names of the proposed child and parent classes:

(child class) is a type of (parent class).

Let’s try some examples. Using the LabeledPoint example from the previous section: “LabeledPoint is a type of Point.” Since a LabeledPoint is a specific type of Point–a Point that has a label–that sentence makes sense. LabeledPoint is a specialization of Point, and inheritance is an appropriate choice.

Now, suppose you wanted to define a class that represents a rectangle. Like a Point, a Rectangle would need to keep track of an x and y location to determine its position, and might also have a width and a height. You’re thinking about defining Rectangle to inherit from Point, so that it reuses all of Point’s functionality (like knowing its position and calculating its distance from origin), and adding just the two new instance variables it needs for its width and height. From a pure code reuse standpoint, inheritance seems plausible. But wait–let’s apply the “is-a” linguistic test. Filling in the blanks in the sentence template above, we get: “Rectangle is a type of Point.” Most people would feel there is something wrong with that statement. A rectangle is not a more specific type of a point. A rectangle contains points and consists of points, but is not itself a point. Thus, it fails the linguistic test; composition is the better choice here.

So what happens if you decide to ignore the linguistic test and go ahead and make Rectangle inherit from Point? In some cases, you won’t run into trouble right away. Often, the difficulties don’t start to crop up until later, when you decide to add more methods to Point (the parent) that aren’t appropriate for Rectangle (the child). This leads to a program that is confusing to understand and contains bugs that occur when methods intended for Point are invoked on Rectangle instances by mistake. Also, since inheritance is the strongest form of relationship between classes, changes to code in a parent class have a stronger likelihood of breaking code in its children than would tend to occur if composition were used.

Inheritance is a powerful feature and, when used appropriately, a terrific way to reuse code. But, like most power tools, it can cut you up pretty badly if you don’t know what you are doing. Use it with caution and respect.