CHAPTER 9

Design with Classes

After completing this chapter, you will be able to

- Determine the attributes and behavior of a class of objects required by a program
- List the methods, including their parameters and return types, that realize the behavior of a class of objects
- Choose the appropriate data structures to represent the attributes of a class of objects
- Define a constructor, instance variables, and methods for a class of objects
- Recognize the need for a class variable and define it
- Define a method that returns the string representation of an object
- Define methods for object equality and comparisons
- Exploit inheritance and polymorphism when developing classes
- Transfer objects to and from files

This book has covered the use of many software tools in computational problem solving. The most important of these tools are the abstraction mechanisms for simplifying designs and controlling the complexity of solutions. Abstraction mechanisms include functions, modules, objects, and classes. In each case, we have begun with an external view of a resource, showing what it does and how it can be used. For example, to use a function in the built-in **`math`** module, you import it, run **`help`** to learn how to use the function correctly, and then include it appropriately in your code. You follow the same procedures for built-in data structures such as strings and lists, and for library resources such as the **`Turtle`** and **`Image`** classes covered in Chapter 7. From a user's perspective, you shouldn't be concerned with how a resource performs its task. The beauty and utility of an abstraction is that it frees you from the need to be concerned with such details.

Unfortunately, not all useful abstractions are built in. You will sometimes need to custom design an abstraction to suit the needs of a specialized application or suite of applications you are developing. You did exactly that while learning how to program GUIs in Chapter 8. There you learned how to customize an existing class by creating custom subclasses to represent windows for various applications. However, in defining a new subclass, you are still working within the helpful confines of already established abstractions, and merely extending them by adding new features and behavior.

The next step is to learn how to design and implement new classes from scratch. When designing your own abstraction, you must take a different view from that of users and concern yourself with the inner workings of a resource. In this chapter, we will take a more detailed internal view of objects and classes than we did in Chapter 8, showing how to design, implement, and test another useful abstraction mechanism—the class. You will learn how to take a real-world problem situation and model its structure and behavior with entirely new classes of objects.

Programming languages that allow the programmer to define new classes of objects are called **object-oriented languages**. These languages also support a style of programming called **object-oriented programming**. Unlike object-based programming, which simply uses ready-made objects and classes within a framework of functions and algorithmic code, object-oriented programming sustains an effort to conceive and build entire software systems from cooperating classes. We begin this chapter by exploring the definitions of a few classes. We then discuss how cooperating classes can be organized into complex software systems. This strategy is rather different from the strategy of procedural design with functions discussed in Chapter 6. The advantages and disadvantages of each design strategy will become clear as we proceed.

Getting Inside Objects and Classes

Programmers who use objects and classes know several things:

- The interface or set of methods that can be used with a class of objects
- The attributes of an object that describe its state from the user's point of view
- How to instantiate a class to obtain an object

Like functions, objects are abstractions. A function packages an algorithm in a single operation that can be called by name. An object packages a set of data values—its state—and a set of operations—its methods—in a single entity that can be referenced with a name. This makes an object a more complex abstraction than a function. To get inside a function, you must view the code contained in its definition. To get inside an object, you must view the code contained in its class. A class definition is like a blueprint for each of the objects of that class. This blueprint contains

- Definitions of all of the methods that its objects recognize
- Descriptions of the data structures used to maintain the state of an object, or its attributes, from the implementer's point of view

To illustrate these ideas, we now present a simple class definition for a course-management application, followed by a discussion of the basic concepts involved.

A First Example: The `Student` Class

A course-management application needs to represent information about students in a course. Each student has a name and a list of test scores. We can use these as the attributes of a class named **`Student`**. The **`Student`** class should allow the user to view a student's name, view a test score at a given position (counting from 1), reset a test score at a given position, view the highest test score, view the average test score, and obtain a string representation of the student's information. When a **`Student`** object is created, the user supplies the student's name and the number of test scores. Each score is initially presumed to be 0.

The interface or set of methods of the **`Student`** class is described in Table 9-1. Assuming that the **`Student`** class is defined in a file named **student.py**, the next session shows how it could be used:

```
>>> from student import Student
>>> s = Student("Maria", 5)
>>> print(s)
Name: Maria
Scores: 0 0 0 0 0
>>> s.setScore(1, 100)
>>> print(s)
Name: Maria
Scores: 100 0 0 0 0
>>> s.getHighScore()
100
>>> s.getAverage()
20
>>> s.getScore(1)
100
>>> s.getName()
'Maria'
```

Student Method	What It Does
`s = Student(name, number)`	Returns a **Student** object with the given **name** and **number** of scores. Each score is initially 0.
`s.getName()`	Returns the student's name.
`s.getScore(i)`	Returns the student's *i*th score, **i** must range from 1 through the number of scores.
`s.setScore(i, score)`	Resets the student's *i*th score to score, **i** must range from 1 through the number of scores.
`s.getAverage()`	Returns the student's average score.
`s.getHighScore()`	Returns the student's highest score.
`s.__str()__`	Same as **str(s)**. Returns a string representation of the student's information.

Table 9-1 The interface of the **Student** class

As you learned in Chapter 8, the syntax of a simple class definition is the following:

```
class <class name>(<parent class name>):
    <method definition-1>
    ...
    <method definition-n>
```

The class definition syntax has two parts: a class header and a set of method definitions that follow the class header. The class header consists of the class name and the parent class name.

The class name is a Python identifier. Although built-in type names are not capitalized, Python programmers typically capitalize their own class names to distinguish them from variable names.

The parent class name refers to another class. All Python classes, including the built-in ones, are organized in a tree-like **class hierarchy**. At the top, or root, of this tree is the most abstract class, named **object**, which is built in. Each class immediately below another class in the hierarchy is referred to as a **subclass**, whereas the class immediately above it, if there is one, is called its **parent class**. If the parenthesized parent class name is omitted from the class definition, the new class is automatically made a subclass of **object**. In the example class definitions shown in this book, we explicitly include the parent class names. More will be said about the relationships among classes in the hierarchy later in this chapter.

The code for the **Student** class follows, and its structure is explained in the next few subsections:

```
"""
File: student.py
Resources to manage a student's name and test scores.
"""
class Student(object):
    """Represents a student."""
```

```
def __init__(self, name, number):
    """Constructor creates a Student with the given
    name and number of scores and sets all scores
    to 0."""
    self.name = name
    self.scores = []
    for count in range(number):
        self.scores.append(0)

def getName(self):
    """Returns the student's name."""
    return self.name

def setScore(self, i, score):
    """Resets the ith score, counting from 1."""
    self.scores[i - 1] = score

def getScore(self, i):
    """Returns the ith score, counting from 1."""
    return self.scores[i - 1]

def getAverage(self):
    """Returns the average score."""
    return sum(self.scores) / len(self.scores)

def getHighScore(self):
    """Returns the highest score."""
    return max(self.scores)

def __str__(self):
    """Returns the string representation of the
    student."""
    return "Name: " + self.name + "\nScores: " + \
           " ".join(map(str, self.scores))
```

Docstrings

The first thing to note is the positioning of the docstrings in our code. They can occur at three levels. The first level is that of the module. Its purpose should be familiar to you by now. The second level is just after the class header. Because there might be more than one class defined in a module, each class can have a docstring that describes its purpose. The third level is located after each method header. Docstrings at this level serve the same role as they do for function definitions. When you enter **`help(Student)`** at a shell prompt, the interpreter prints the documentation for the class and all of its methods.

Method Definitions

All of the method definitions are indented below the class header. Because methods are a bit like functions, the syntax of their definitions is similar. As you learned in Chapter 8, each method definition must include a first parameter named **`self`**, even if that method seems to

expect no arguments when called. When a method is called with an object, the interpreter binds the parameter **self** to that object so that the method's code can refer to the object by name. Thus, for example, the code

```
s.getScore(4)
```


binds the parameter **self** in the method **getScore** to the **Student** object referenced by the variable **s**. The code for **getScore** can then use **self** to access that individual object's test scores.

Otherwise, methods behave just like functions. They can have required and/or optional arguments, and they can return values. They can create and use temporary variables. A method automatically returns the value **None** when it includes no **return** statement.

The __init__ Method and Instance Variables

Most classes include a special method named **__init__**. Here is the code for this method in the **Student** class:

```
def __init__(self, name, number):
    """All scores are initially 0."""
    self.name = name
    self.scores = []
    for count in range(number):
        self.scores.append(0)
```

Note that **__init__** must begin and end with two consecutive underscores. This method is also called the class's **constructor**, because it is run automatically when a user instantiates the class. Thus, when the code segment

```
s = Student("Juan", 5)
```

is run, Python automatically runs the constructor or **__init__** method of the **Student** class. The purpose of the constructor is to initialize an individual object's attributes. In addition to **self**, the **Student** constructor expects two arguments that provide the initial values for these attributes. From this point on, when we refer to a class's constructor, we mean its **__init__** method.

The attributes of an object are represented as **instance variables**. Each individual object has its own set of instance variables. These variables serve as storage for its state. The scope of an instance variable (including **self**) is the entire class definition. Thus, all of the class's methods are in a position to reference the instance variables. The lifetime of an instance variable is the lifetime of the enclosing object. An object's lifetime will be discussed in more detail later in this chapter.

Within the class definition, the names of instance variables must begin with **self**. For example, in the definition of the **Student** class, the instance variables **self.name** and **self.scores** are initialized to a string and a list, respectively.

The __str__ Method

Many built-in Python classes usually include an **__str__** method. This method builds and returns a string representation of an object's state. When the **str** function is called with an object, that object's **__str__** method is automatically invoked to obtain the string that **str** returns. For example, the function call **str(s)** is equivalent to the method call **s.__str__()**, and is simpler to write. The function call **print(s)** also automatically runs **str(s)** to obtain the object's string representation for output. Here is the code for the **__str__** method in the **Student** class:

```
def __str__(self) :
    """Returns the string representation of the student."""
    return "Name: " + self.name + "\nScores: " + \
           " ".join(map(str, self.scores))
```

The programmer can return any information that would be relevant to the users of a class. Perhaps the most important use of **__str__** is in debugging, when you often need to observe the state of an object after running another method.

Accessors and Mutators

Methods that allow a user to observe but not change the state of an object are called **accessors**. Methods that allow a user to modify an object's state are called **mutators**. The **Student** class has just one mutator method. It allows the user to reset a test score at a given position. The remaining methods are accessors. Here is the code for the mutator method **setScore**:

```
def setScore(self, i, score):
    """Resets the ith score, counting from 1."""
    self.scores[i - 1] = score
```

In general, the fewer the number of changes that can occur to an object, the easier it is to use it correctly. That is one reason Python strings are immutable. In the case of the **Student** class, if there is no need to modify an attribute, such as a student's name, we do not include a method to do that.

The Lifetime of Objects

Earlier, we said that the lifetime of an object's instance variables is the lifetime of that object. What determines the span of an object's life? We know that an object comes into being when its class is instantiated. When does an object die? In Python, an object

becomes a candidate for the graveyard when the program that created it can no longer refer to it. For example, the next session creates two references to the same **Student** object:

```
>>> s = Student("Sam", 10)
>>> cscilll = [s]
>>> cscilll
[<__main__.Student instance at 0x11ba2b0>]
>>> s
<__main__.Student instance at 0x11ba2b0>
```

The strange-looking code in angle brackets is what Python displays when it prints this type of object in the shell. As long as one of these references survives, the **Student** object can remain alive. Continuing this session, we now sever both references to the **Student** object:

```
>>> s = None
>>> cscilll.pop()
<__main__.Student instance at 0x11ba2b0>
>>> print(s)
None
>>> cscilll
[]
```

The **Student** object still exists, but the Python virtual machine will eventually recycle its storage during a process called **garbage collection**. For all intents and purposes, this object has expired, and its storage will eventually be used to create other objects.

Rules of Thumb for Defining a Simple Class

We conclude this section by listing several rules of thumb for designing and implementing a simple class:

1. Before writing a line of code, think about the behavior and attributes of the objects of the new class. What actions does an object perform, and how, from the external perspective of a user, do these actions access or modify the object's state?
2. Choose an appropriate class name, and develop a short list of the methods available to users. This interface should include appropriate method names and parameter names, as well as brief descriptions of what the methods do. Avoid describing how the methods perform their tasks.
3. Write a short script that appears to use the new class in an appropriate way. The script should instantiate the class and run all of its methods. Of course, you will not be able to execute this script until you have completed the next few steps, but it will help to clarify the interface of your class and serve as an initial test bed for it.
4. Choose the appropriate data structures to represent the attributes of the class. These will be either built-in types such as integers, strings, and lists, or other programmer-defined classes.

5. Fill in the class template with a constructor (an **__init__** method) and an **__str__** method. Remember that the constructor initializes an object's instance variables, whereas **__str__** builds a string from this information. As soon as you have defined these two methods, you can test your class by instantiating it and printing the resulting object.
6. Complete and test the remaining methods incrementally, working in a bottom-up manner. If one method depends on another, complete the second method first.
7. Remember to document your code. Include a docstring for the module, the class, and each method. Do not add docstrings as an afterthought. Write them as soon as you write a class header or a method header. Be sure to examine the results by running **help** with the class name.

Exercises

1. What are instance variables, and what role does the name **self** play in the context of a class definition?
2. Explain what a constructor does.
3. Explain what the **__str__** method does and why it is a useful method to include in a class.
4. The **Student** class has no mutator method that allows a user to change a student's name. Define a method **setName** that allows a user to change a student's name.
5. The method **getAge** expects no arguments and returns the value of an instance variable named **self.age**. Write the code for the definition of this method.
6. How is the lifetime of an object determined? What happens to an object when it dies?

CASE STUDY: Playing the Game of Craps

Some college students are known to study hard and play hard. In this case study, we develop some classes that cooperate to allow students to play and study the behavior of the game of craps.

Request

Write a program that allows the user to play and study the game of craps.

(continues)

(continued)

Analysis

A player in the game of craps rolls a pair of dice. If the sum of the values on this initial roll is 2, 3, or 12, the player loses. If the sum is 7 or 11, the player wins. Otherwise, the player continues to roll until the sum is 7, indicating a loss, or the sum equals the initial sum, indicating a win.

During analysis, you decide which classes of objects will be used to model the behavior of the objects in the problem domain. The classes often become evident when you consider the nouns used in the problem description. In this case, the two most significant nouns in our description of a game of craps are "player" and "dice." Thus, the classes will be named **`Player`** and **`Die`** (the singular of "dice").

Analysis also specifies the roles and responsibilities of each class. You can describe these in terms of the behavior of each object in the program. A **`Die`** object can be rolled and its value examined. That's about it. A **`Player`** object can play a complete game of craps. During the course of this game, the player keeps track of the rolls of the dice. After a game is over, the player can be asked for a history of the rolls and for the game's outcome. The player can then play another game, and so on.

A terminal-based user interface for this program prompts the user for the number of games to play. The program plays that number of games and generates and displays statistics about the results for that round of games. These results, our "study" of the game, include the number of wins, losses, rolls per win, rolls per loss, and winning percentage, for the given number of games played.

The program includes two functions, **`playOneGame`** and **`playManyGames`**, for convenient testing in the IDLE shell. Here is a sample session with these functions:

```
>>> playOneGame()
(2, 2) 4
(2, 1) 3
(4, 6) 10
(6, 5) 11
(4, 1) 5
(5, 6) 11
(3, 5) 8
(3, 1) 4
You win!
>>> playManyGames(100)
The total number of wins is 49
The total number of losses is 51
The average number of rolls per win is 3.37
The average number of rolls per loss is 4.20
The winning percentage is 0.490
```

(continues)

(continued)

Design

During design, you choose the appropriate data structures for the instance variables of each class and develop its methods using pseudocode, if necessary. You can work from class interfaces provided by analysis or develop the interfaces as the first step of design. The interfaces of the **Die** and **Player** classes are listed in Table 9-2.

Player Method	**What It Does**
`p = Player()`	Returns a new player object.
`p.play()`	Plays the game and returns **True** if there is a win, **False** otherwise.
`p.getNumberOfRolls()`	Returns the number of rolls.
`p.__str__()`	Same as **str(p)**. Returns a formatted string representation of the rolls.
Die METHOD	**What It Does**
`d = Die()`	Returns a new die object whose initial value is 1.
`d.roll()`	Resets the die's value to a random number between 1 and 6.
`d.getValue()`	Returns the die's value.
`d.__str__()`	Same as **str(d)**. Returns the string representation of the die's value.

Table 9-2 The interfaces of the **Die** and **Player** classes

A **Die** object has a single attribute, an integer ranging in value from 1 through 6. At instantiation, the instance variable **self.value** is initialized to 1. The method **roll** modifies this value by resetting it to a random number from 1 to 6. The method **getValue** returns this value. The method **__str__** returns its string representation. The **Die** class can be coded immediately without further design work.

A **Player** object has three attributes, a pair of dice and a history of rolls in its most recent game. We represent each roll as a tuple of two integers and the set of rolls as a list of these tuples. At instantiation, the instance variable **self.rolls** is set to an empty list.

The method **__str__** converts the list of rolls to a formatted string that contains a roll and the sum from that roll on each line.

(continues)

(continued)

The **play** method implements the logic of playing a game and tracking its results. Here is the pseudocode:

```
Create a new list of rolls
Roll the dice and add their values to the rolls list
If sum of the initial roll is 2, 3, or 12
    return false
If the sum of the initial roll is 7 or 11
    return true
While true
    Roll the dice and add their values to the rolls list
    If the sum of the roll is 7
        return false
    Else if the sum of the roll equals the initial sum,
        return true
```

Note that the rolls list, which is an instance variable, is reset to an empty list on each play. That allows the same player to play multiple games.

The script that defines the **Player** and **Die** classes also includes two functions. The role of these functions is to interact with the human user by playing the games and displaying their results. The **playManyGames** function expects the number of games as an argument, creates a single **Player** object, plays the games and gathers data on the results, processes these data, and displays the required information. We also include a simpler function **playOneGame** that plays just one game and displays the results.

Implementation (Coding)

The **Die** class is defined in a file named **die.py**. The **Player** class and the top-level functions are defined in a file named **craps.py**. Here is the code for the two modules:

```
"""
File: die.py
This module defines the Die class.
"""

from random import randint

class Die(object):
    """This class represents a six-sided die."""

    def __init__(self):
        """Sets the initial face of the die."""
        self.value = 1
```

(continues)

(continued)

```
    def roll(self):
        """Resets the die's value to a random number
        between 1 and 6."""
        self.value = randint(1, 6)

    def getvalue(self):
        """Returns the current face of the die."""
        return self.value

    def __str__(self):
        """Returns the string rep of the die."""
        return str(self.value)
"""
File: craps.py
This module studies and plays the game of craps.
"""
from die import Die

class Player(object):

    def __init__(self):
        """Has a pair of dice and an empty rolls list."""
        self.die1 = Die()
        self.die2 = Die()
        self.rolls = []

    def __str__(self):
        """Returns the string rep of the history of
        rolls."""
        result = ""
        for (v1, v2) in self.rolls:
            result = result + str((v1, v2)) + " " + \
            str(v1 + v2) + "\n"
        return result

    def getNumberOfRolls(self):
        """Returns the number of the rolls in one game."""
        return len(self.rolls)

    def play(self):
        """Plays a game, saves the rolls for that game,
        and returns True for a win and False for a loss."""
        self.rolls = []
        self.die1.roll()
        self.die2.roll()
```

(continues)

(continued)

```
            (v1, v2) = (self.die1.getvalue(),
                        self.die2.getvalue())
            self.rolls.append((v1, v2) )
            initialSum = v1 + v2
            if initialSum in (2, 3, 12):
                return False
            elif initialSum in (7, 11):
                return True
            while True:
                self.die1.roll()
                self.die2.roll()
                (v1, v2) = (self.die1.getvalue(),
                            self.die2.getvalue())
                self.rolls.append((v1, v2))
                laterSum = v1 + v2
                if laterSum == 7:
                    return False
                elif laterSum == initialSum:
                    return True

    # Functions that interact with the user to play the games
    def playOneGame():
        """Plays a single game and prints the results."""
        player = Player()
        youWin = player.play()
        print(player)
        if youWin:
            print("You win!")
        else:
            print("You lose!")

    def playManyGames(number):
        """Plays a number of games and prints statistics."""
        wins = 0
        losses = 0
        winRolls = 0
        lossRolls = 0
        player = Player()
        for count in range(number):
            hasWon = player.play()
            rolls = player.getNumberOfRolls()
            if hasWon:
                wins += 1
                winRolls += rolls
```

(continues)

(continued)

```
            else:
                losses += 1
                lossRolls += rolls
        print("The total number of wins is", wins)
        print("The total number of losses is", losses)
        print("The average number of rolls per win is %0.2f" % \
              (winRolls / wins))
        print("The average number of rolls per loss is %0.2f" % \
              (lossRolls / losses))
        print("The winning percentage is %0.3f" % \
              (wins / number))

    def main():
        """Plays a number of games and prints statistics."""
        number = int(input("Enter the number of games: "))
        playManyGames(number)

    if __name__ == "__main__":
        main()
```

A GUI for Dice Games

Gambling is gambling, but it's more fun on a computer if you can visualize the dice. You can deploy the skills you picked up in Chapter 8 to create the graphical user interface shown in Figure 9-1.

Figure 9-1 Displaying images of dice

(continues)

(continued)

The code for this version of the interface loads and displays images of dice from a folder of GIF files. It is a good idea to rough out the GUI before incorporating the game logic. Here is the code for laying out the window and rolling two dice:


```
"""
File: dicedemo.py
Pops up a window that allows the user to roll the dice.
"""

from breezypythongui import EasyFrame
from tkinter import PhotoImage
from die import Die

class DiceDemo(EasyFrame):

    def __init__(self):
        """Creates the dice, and sets up the Images and
        labels for the two dice to be displayed,
        the state label, and the two command buttons."""
        EasyFrame.__init__(self, title = "Dice Demo")
        self.setSize(220, 200)
        self.die1 = Die()
        self.die2 = Die()
        self.dieLabel1 = self.addLabel("", row = 0,
                                       column = 0,
                                       sticky = "NSEW")
        self.dieLabel2 = self.addLabel("", row = 0,
                                       column = 1,
                                       sticky = "NSEW",
                                            columnspan = 2)
        self.stateLabel = self.addLabel("", row = 1,
                                        column = 0,
                                        sticky = "NSEW",
                                        columnspan = 2)
        self.addButton(row = 2,column = 0, text = "Roll",
                       command = self.nextRoll)
        self.addButton(text = "New game", row = 2,
                       column = 1,
                       command = self.newGame)
        self.refreshImages()

    def nextRoll(self):
        """Rolls the dice and updates the view with
        the results."""
        self.die1.roll()
        self.die2.roll()
```

(continues)

(continued)

```
        total = self.die1.getValue() + self.die2.getValue()
        self.stateLabel["text"] = "Total = " + str(total)
        self.refreshImages()

    def newGame(self):
        """Create new dice and updates the view."""
        self.die1 = Die()
        self.die2 = Die()
        self.stateLabel["text"] = ""
        self.refreshImages()

    def refreshImages(self):
        """Updates the images in the window."""
        fileName1 = "DICE/" + str(self.die1) + ".gif"
        fileName2 = "DICE/" + str(self.die2) + ".gif"
        self.image1 = PhotoImage(file = fileName1)
        self.dieImageLabel1["image"] = self.image1
        self.image2 = PhotoImage(file = fileName2)
        self.dieImageLabel2["image"] = self.image2

def main():
    """Instantiate and pop up the window."""
    DiceDemo().mainloop()

if __name__ == "__main__":
    main()
```

The completion of a GUI-based craps game is left as an exercise.

Data-Modeling Examples

As you have seen, objects and classes are useful for modeling objects in the real world. In this section, we explore several other examples.

Rational Numbers

We begin with numbers. A **rational number** consists of two integer parts, a numerator and a denominator, and is written using the format *numerator* / *denominator*. Examples are 1/2, 1/3, and so forth. Operations on rational numbers include arithmetic and comparisons. Python has no built-in type for rational numbers. Let us develop a new class named **`Rational`** to support this type of data.

The interface of the **`Rational`** class includes a constructor for creating a rational number, an **`str`** function for obtaining a string representation, and accessors for the numerator and

denominator. We will also show how to include methods for arithmetic and comparisons. Here is a sample session to illustrate the use of the new class:

```
>>> oneHalf = Rational(1, 2)
>>> oneSixth = Rational(1, 6)
>>> print(oneHalf)
1/2
>>> print(oneHalf + oneSixth)
2/3
>>> oneHalf == oneSixth
False
>>> oneHalf > oneSixth
True
```

Note that this session uses the built-in operators +, ==, and < with objects of the new class, **Rational**. Python allows the programmer to **overload** many of the built-in operators for use with new data types.

We develop this class in two steps. First, we take care of the internal representation of a rational number and also its string representation. The constructor expects the numerator and denominator as arguments and sets two instance variables to this information. This method then reduces the rational number to its lowest terms. To reduce a rational number to its lowest terms, you first compute the greatest common divisor (GCD) of the numerator and the denominator, using Euclid's algorithm, as described in Programming Project 8 of Chapter 3. You then divide the numerator and the denominator by this GCD. These tasks are assigned to two other **Rational** methods, **_reduce** and **_gcd**. Because these methods are not intended to be in the class's interface, their names begin with the _ symbol. Performing the reduction step in the constructor guarantees that it will not have to be done in any other operation. Here is the code for the first step:

```
"""
File: rational.py
Resources to manipulate rational numbers.
"""

class Rational(object):
    """Represents a rational number."""

    def __init__(self, numer, denom) :
        """Constructor creates a number with the given
        numerator and denominator and reduces it to lowest
        terms."""
        self.numer = numer
        self.denom = denom
        self._reduce()

    def numerator(self):
        """Returns the numerator."""
        return self.numer
```

```
    def denominator(self):
        """Returns the denominator."""
        return self.denom

    def __str__(self):
        """Returns the string representation of the
        number."""
        return str(self.numer) + "/" + str(self.denom)

    def _reduce(self):
        """Helper to reduce the number to lowest terms."""
        divisor = self._gcd(self.numer, self.denom)
        self.numer = self.numer // divisor
        self.denom = self.denom // divisor

    def _gcd(self, a, b):
        """Euclid's algorithm for greatest common
        divisor (hacker's version)."""
        (a, b) = (max(a, b), min(a, b))
        while b > 0:
            (a, b) = (b, a % b)
        return a

    # Methods for arithmetic and comparisons go here
```

You can now test the class by instantiating numbers and printing them. Note that this class only supports positive rational numbers. When you are satisfied that the data are being represented correctly, you can move on to the next step.

Rational Number Arithmetic and Operator Overloading

We now add methods to perform arithmetic with rational numbers. Recall that the earlier session used the built-in operators for arithmetic. For a built-in type such as **`int`** or **`float`**, each arithmetic operator corresponds to a special method name. You will see many of these methods by entering **`dir(int)`** or **`dir(str)`** at a shell prompt, and they are listed in Table 9-3. The object on which the method is called corresponds to the left operand,

Operator	Method Name
+	`__add__`
-	`__sub__`
*	`__mul__`
/	`__div__`
%	`__mod__`

Table 9-3 Built-in arithmetic operators and their corresponding methods

whereas the method's second parameter corresponds to the right operand. Thus, for example, the code **`x + y`** is actually shorthand for the code **`x.__add__(y)`**.

To overload an arithmetic operator, you just define a new method using the appropriate method name. The code for each method applies a rule of rational number arithmetic. The rules are listed in Table 9-4.

Type of Operation	Rule
Addition	$n_1/d_1 + n_2/d_2 = (n_1d_2 + n_2d_1) / d_1d_2$
Subtraction	$n_1/d_1 - n_2/d_2 = (n_1d_2 - n_2d_1) / d_1d_2$
Multiplication	$n_1/d_1 * n_2/d_2 = n_1n_2 / d_1d_2$
Division	$n_1/d_1 / n_2/d_2 = n_1d_2 / d_1n_2$

Table 9-4 Rules for rational number arithmetic

Each method builds and returns a new rational number that represents the result of the operation. Here is the code for the addition operation:

```
def __add__(self, other):
    """Returns the sum of the numbers.
    self is the left operand and other is
    the right operand."""
    newNumer = self.numer * other.denom + \
               other.numer * self.denom
    newDenom = self.denom * other.denom
    return Rational(newNumer, newDenom)
```

Note that the parameter **`self`** is viewed as the left operand of the operator, whereas the parameter **`other`** is viewed as the right operand. The instance variables of the rational number named **`other`** are accessed in the same manner as the instance variables of the rational number named **`self`**. Note also that this method, like the other methods for rational arithmetic, returns a rational number. Arithmetic operations on numbers are said to be **closed under combination**, meaning that these operations usually return values of the same types as their arguments, allowing the user to combine the operations in arbitrarily complex expressions.

Operator overloading is another example of an abstraction mechanism. In this case, programmers can use operators with single, standard meanings even though the underlying operations vary from data type to data type.

Comparison Methods

You can compare integers and floating-point numbers using the operators **`==`**, **`!=`**, **`<`**, **`>`**, **`<=`**, and **`>=`**. When the Python interpreter encounters one of these operators, it uses a corresponding method defined in the **`float`** or **`int`** class. Each of these methods expects two

arguments. The first argument, **self**, represents the operand to the left of the operator, and the second argument represents the other operand. Table 9-5 lists the comparison operators and the corresponding methods.

Operator	Meaning	Method
==	Equals	__eq__
!=	Not equals	__ne__
<	Less than	__lt__
<=	Less than or equal	__le__
>	Greater than	__gt__
>=	Greater than or equal	__ge__

Table 9-5 The comparison operators and methods

To use the comparison operators with a new class of objects, such as rational numbers, the class must include these methods with the appropriate comparison logic. However, once the implementer of the class has defined methods for ==, <, and >=, the remaining methods are automatically provided.

Let's implement < here and wait on == until the next section. The simplest way to compare two rational numbers is to compare the product of the extremes and the product of the means. The extremes are the first numerator and the second denominator, whereas the means are the second numerator and the first denominator. Thus, the comparison $1/6 < 2/3$ translates to $1 * 3 < 2 * 6$. The implementation of the **__lt__** method for rational numbers uses this strategy, as follows:

```
def __lt__(self, other):
    """Compares two rational numbers, self and other,
    using <."""
    extremes = self.numer * other.denom
    means = other.numer * self.denom
    return extremes < means
```

When objects of a new class are comparable, it's a good idea to include the comparison methods in that class. Then, other built-in methods, such as the **sort** method for lists, will be able to use your objects appropriately.

Equality and the __eq__ Method

Equality is a different kind of relationship from the other types of comparisons. Not all objects are comparable using less than or greater than, but any two objects can be compared for equality or inequality. For example, when the variable **twoThirds** refers to a

rational number, it does not make sense to say **twoThirds < "hi there"**, but it does make sense to say **twoThirds != "hi there"** (true, they aren't the same). Put another way, the first expression should generate a semantic error, whereas the second expression should return **True**.

The Python interpreter picks out equality from the other comparisons by looking for an **__eq__** method when it encounters the **==** and **!=** operators. As you'll recall from Chapter 5, Python includes an implementation of this method for objects like lists and dictionaries as well as the numeric types. However, unless you include an implementation of this method for a new class, Python relies upon the implementation of **__eq__** in the **object** class, which uses the **is** operator. This implementation returns **True** only if the two operands refer to the exact same object (object identity). This criterion of equality is too narrow for many objects, such as rational numbers, where you might want two distinct objects that both represent the same number to be considered equal.

To remedy this problem, you must include an **__eq__** method in a new class. This method supports equality tests with any types of objects. Here is the code for this method in the **Rational** class:

```
def __eq__(self, other):
    """Tests self and other for equality."""
    if self is other:                    # Object identity?
        return True
    elif type(self) != type(other):      # Types match?
        return False
    else:
        return self.numer == other.numer and \
               self.denom == other.denom
```

Note that the method first tests the two operands for object identity using Python's **is** operator. The **is** operator returns **True** if **self** and **other** refer to the exact same object. If the two objects are distinct, the method then uses Python's **type** function to determine whether or not they are of the same type. If they are not of the same type, they cannot be equal. Finally, if the two operands are of the same type, the second one must be a rational number, so it is safe to access the components of both operands to compare them for equality in the last alternative.

As a rule of thumb, you should include an **__eq__** method in any class where a comparison for equality uses a criterion other than object identity. You should also include comparison methods for **<** and **>=** when the objects are comparable using less than or greater than.

Savings Accounts and Class Variables

Turning to the world of finance, banking systems are easily modeled with classes. For example, a savings account allows owners to make deposits and withdrawals. These accounts also compute interest periodically. A simplified version of a savings account includes an

owner's name, PIN, and balance as attributes. The interface for a **SavingsAccount** class is listed in Table 9-6.

SavingsAccount Method	What It Does
`a = SavingsAccount(name, pin, balance = 0.0)`	Returns a new account with the given name, PIN, and balance.
`a.deposit(amount)`	Deposits the given amount to the account's balance.
`a.withdraw(amount)`	Withdraws the given amount from the account's balance.
`a.getBalance()`	Returns the account's balance.
`a.getName()`	Returns the account's name.
`a.getPin()`	Returns the account's PIN.
`a.computeInterest()`	Computes the account's interest and deposits it.
`a.__str__()`	Same as `str(a)`. Returns the string representation of the account.

Table 9-6 The interface for **SavingsAccount**

When the interest is computed, a rate is applied to the balance. If you assume that the rate is the same for all accounts, then it does not have to be an instance variable. Instead, you can use a **class variable**. A class variable is visible to all instances of a class and does not vary from instance to instance. While it normally behaves like a constant, in some situations a class variable can be modified. But when it is, the change takes effect for the entire class.

To introduce a class variable, we place the assignment statement that initializes it between the class header and the first method definition. For clarity, class variables are written in uppercase only. The code for **SavingsAccount** shows the definition and use of the class variable **RATE**:

```
"""
File: savingsaccount.py
This module defines the SavingsAccount class.
"""

class SavingsAccount(object):
    """This class represents a savings account
    with the owner's name, PIN, and balance."""
```

```
    RATE = 0.02    # Single rate for all accounts

    def __init__(self, name, pin, balance = 0.0):
        self.name = name
        self.pin = pin
        self.balance = balance

    def __str__(self) :
        """Returns the string rep."""
        result = 'Name:        ' + self.name + '\n'
        result += 'PIN:         ' + self.pin + '\n'
        result += 'Balance:    ' + str(self.balance)
        return result

    def getBalance(self):
        """Returns the current balance."""
        return self.balance

    def getName(self):
        """Returns the current name."""
        return self.name

    def getPin(self):
        """Returns the current PIN."""
        return self.pin

    def deposit(self, amount):
        """Deposits the given amount and returns None."""
        self.balance += amount
        return None

    def withdraw(self, amount):
        """Withdraws the given amount.
        Returns None if successful, or an
        error message if unsuccessful."""
        if amount < 0:
            return "Amount must be >= 0"
        elif self.balance < amount:
            return "Insufficient funds"
        else:
            self.balance -= amount
            return None

    def computeInterest(self):
        """Computes, deposits, and returns the interest."""
        interest = self.balance * SavingsAccount.RATE
        self.deposit(interest)
        return interest
```

When you reference a class variable, you must prefix it with the class name and a dot, as in **SavingsAccount.RATE**. Class variables are visible both inside a class definition and to external users of the class.

In general, you should use class variables only for symbolic constants or to maintain data held in common by all objects of a class. For data that are owned by individual objects, you must use instance variables instead.

Putting the Accounts into a Bank

Savings accounts most often make sense in the context of a bank. A very simple bank allows a user to add new accounts, remove accounts, get existing accounts, and compute interest on all accounts. A **Bank** class thus has these four basic operations (**add**, **remove**, **get**, and **computeInterest**) and a constructor. This class, of course, also includes the usual **str** function for development and debugging. We assume that **Bank** is defined in a file named **bank.py**. Here is a sample session that uses a **Bank** object and some **SavingsAccount** objects. The interface for **Bank** is listed in Table 9-7.

```
>>> from bank import Bank
>>> from savingsaccount import SavingsAccount
>>> bank = Bank()
>>> bank.add(SavingsAccount("Wilma", "1001", 4000.00))
>>> bank.add(SavingsAccount("Fred", "1002", 1000.00))
>>> print(bank)
Name:    Fred
PIN:     1002
Balance: 1000.00
Name:    Wilma
PIN:     1001
Balance: 4000.00
>>> account = bank.get("Wilma", "1000")
>>> print(account)
None
>>> account = bank.get("Wilma", "1001")
>>> print (account)
Name:    Wilma
PIN:     1001
Balance: 4000.00
>>> account.deposit(25.00)
>>> print(account)
Name:    Wilma
PIN:     1001
Balance: 4025.00
>>> print(bank)
Name:    Fred
PIN:     1002
```

```
Balance: 1000.00
Name:    Wilma
PIN:     1001
Balance: 4025.00
```


Bank Method	What It Does
`b = Bank()`	Returns a bank.
`b.add(account)`	Adds the given account to the bank.
`b.remove(name, pin)`	Removes the account with the given **name** and **pin** from the bank and returns the account. If the account is not in the bank, returns **None**.
`b.get(name, pin)`	Returns the account associated with the **name** and **pin** if it's in the bank. Otherwise, returns **None**.
`b.computeInterest()`	Computes the interest on each account, deposits it in that account, and returns the total interest.
`b.__str__()`	Same as **str(b)**. Returns a string representation of the bank (all the accounts).

Table 9-7 The interface for the **Bank** class

To keep the design simple, the bank maintains the accounts in no particular order. Thus, you can choose a dictionary keyed by owners' credentials to represent the collection of accounts. Access and removal then depend on an owner's credentials. Here is the code for the **Bank** class:

```
"""
File: bank.py
This module defines the Bank class.
"""

from savingsaccount import SavingsAccount

class Bank(object):

    def __init__(self):
        self.accounts = {}

    def __str__(self) :
        """Return the string rep of the entire bank."""
        return '\n'.join(map(str, self.accounts.values()))

    def makeKey(self, name, pin):
        """Makes and returns a key from name and pin."""
        return name + "/" + pin
```

```
    def add(self, account):
        """Inserts an account with name and pin as a key."""
        key = self.makeKey(account.getName(),
                           account.getPin())
        self.accounts[key] = account

    def remove(self, name, pin):
        """Removes an account with name and pin as a key."""
        key = self.makeKey(name, pin)
        return self.accounts.pop(key, None)

    def get(self, name, pin):
        """Returns an account with name and pin as a key
        or None if not found."""
        key = self.makeKey(name, pin)
        return self.accounts.get(key, None)

    def computeInterest(self):
        """Computes interest for each account and
        returns the total."""
        total = 0.0
        for account in self.accounts.values():
            total += account.computeInterest()
        return total
```

Note the use of the value **None** in the methods **remove** and **get**. In this context, **None** indicates to the user that the given account is not in the bank. Note also that the module names for the **Bank** and **SavingsAccount** classes are **bank** and **savingsaccount**, respectively. This naming convention is standard practice among Python programmers and helps them to remember where classes are located for import.

Using `pickle` for Permanent Storage of Objects

Chapter 4 discussed saving data in permanent storage with text files. Now suppose you want to save new types of objects to files. For example, it would be a wise idea to back up the information for a savings account to a file whenever that account is modified. You can convert any object to text for storage, but the mapping of complex objects to text and back again can be tedious and cause maintenance headaches. Fortunately, Python includes a module that allows the programmer to save and load objects using a process called **pickling**. The term comes from the process of converting cucumbers to pickles for preservation in jars. However, in the case of computational objects, you can get the cucumbers back from the pickle jar again. You can pickle an object before it is saved to a file, and then unpickle it as it is loaded from a file into a program. Python takes care of all of the conversion details automatically. You start by importing the **pickle** module. Files are opened for input and output and closed in the usual manner, except that the flags **"rb"** and **"wb"** are used instead of **'r'** and **'w'**, respectively. To save an object, you use the function **pickle.dump**. Its first argument is the object to be "dumped," or saved to a file, and its second argument is the file object.

You can use the **pickle** module to save the accounts in a bank to a file. You start by defining a **Bank** method named **save**. The method includes an optional argument for the filename. You assume that the **Bank** object also has an instance variable for the filename. For a new, empty bank, this variable's value is initially **None**. Whenever the bank is saved to a file, this variable becomes the current filename. When the method's filename argument is not provided, the method uses the bank's current filename if there is one. This is similar to using the **Save** option in a **File** menu. When the filename argument is provided, it is used to save the bank to a different file. This is similar to the **Save As** option in a **File** menu. Here is the code:

```
import pickle

def save(self, fileName = None):
    """Saves pickled accounts to a file. The parameter
    allows the user to change filenames."""
    if fileName != None:
        self.fileName = fileName
    elif self.fileName == None:
        return
    fileObj = open(self. fileName, "wb")
    for account in self.accounts.values():
        pickle.dump(account, fileObj)
    fileObj.close()
```

Input of Objects and the `try-except` Statement

You can load pickled objects into a program from a file using the function **pickle.load**. If the end of the file has been reached, this function raises an exception. This complicates the input process, because we have no apparent way to detect the end of the file before the exception is raised. However, Python's **try-except** statement comes to our rescue. As you learned in Chapter 8, this statement allows an exception to be caught and the program to recover. The syntax of a simple **try-except** statement is the following:

```
try:
    <statements>
except <exception type>:
    <statements>
```

When this statement is run, the statements within the **try** clause are executed. If one of these statements raises an exception, control is immediately transferred to the **except** clause. If the type of exception raised matches the type in this clause, its statements are executed. Otherwise, control is transferred to the caller of the **try-except** statement and further up the chain of calls, until the exception is successfully handled or the program halts with an error message. If the statements in the **try** clause raise no exceptions, the **except** clause is skipped, and control proceeds to the end of the **try-except** statement.

We can now construct an input file loop that continues to load objects until the end of the file is encountered. When this happens, an **EOFError** is raised. The **except** clause then closes the file and breaks out of the loop. We also add a new instance variable to

track the bank's filename for saving the bank to a file. Here is the code for a **Bank** method **__init__** that can take some initial accounts from an input file. This method now either creates a new, empty bank if the filename is not present, or loads accounts from a file into a **Bank** object.

```
def __init__(self, fileName = None):
    """Creates a new dictionary to hold the accounts.
    If a filename is provided, loads the accounts from
    a file of pickled accounts."""
    self.accounts = {}
    self.fileName = fileName
    if fileName != None:
        fileObj = open(fileName, "rb")
        while True:
            try:
                account = pickle.load(fileObj)
                self.add(account)
            except EOFError:
                fileObj.close()
                break
```

Playing Cards

Many games, such as poker, blackjack, and solitaire, use playing cards. Modeling playing cards provides a nice illustration of the design of cooperating classes. A standard deck of cards has 52 cards. There are four suits: spades, hearts, diamonds, and clubs. Each suit contains 13 cards. Each card also has a rank, which is a number used to sort the cards and determine the count in a hand. The literal numbers are 2 through 10. An Ace counts as the number 1 or some other number, depending on the game being played. The face cards, Jack, Queen, and King, often count as 11, 12, and 13, respectively.

A **Card** class and a **Deck** class would be useful resources for game-playing programs. A **Card** object has two instance attributes, a rank and a suit. The **Card** class has two class attributes, the set of all suits and the set of all ranks. You can represent these two sets of attributes as instance variables and class variables in the **Card** class.

Because the attributes are only accessed and never modified, we do not include any methods other than an **__str__** method for the string representation. The **__init__** method expects an integer rank and a string suit as arguments and returns a new card with that rank and suit. The next session shows the use of the **Card** class:

```
>>> threeOfSpades = Card(3, "Spades")
>>> jackOfSpades = Card(11, "Spades")
>>> print(jackOfSpades)
Jack of Spades
>>> threeOfSpades.rank < jackOfSpades.rank
True
>>> print(jackOfSpades.rank, jackOfSpades.suit)
11 Spades
```

Note that you can directly access the rank and suit of a **Card** object by using a dot followed by the instance variable names. A card is little more than a container of two data values. Here is the code for the **Card** class:

```
class Card(object):
    """ A card object with a suit and rank."""

    RANKS = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13)
    SUITS = ("Spades", "Diamonds", "Hearts", "Clubs")

    def __init__(self, rank, suit):
        """Creates a card with the given rank and suit."""
        self.rank = rank
        self.suit = suit

    def __str__(self) :
        """Returns the string representation of a card."""
        if self.rank == 1:
            rank = "Ace"
        elif self.rank == 11:
            rank = "Jack"
        elif self.rank == 12:
            rank = "Queen"
        elif self.rank == 13:
            rank = "King"
        else:
            rank = self.rank
        return str(rank) + " of " + self.suit
```

Unlike an individual card, a deck has significant behavior that can be specified in an interface. One can shuffle the deck, deal a card, and determine the number of cards left in it. Table 9-8 lists the methods of a **Deck** class and what they do. Here is a sample session that tries out a deck:

```
>>> deck = Deck()
>>> print(deck)
--- the print reps of 52 cards, in order of suit and rank
>>> deck.shuffle()
>>> len(deck)
52
>>> while len(deck) > 0:
        card = deck.deal()
        print(card)
--- the print reps of 52 randomly ordered cards
>>> len(deck)
0
```

During instantiation, all 52 unique cards are created and inserted in sorted order into a deck's internal list of cards. The **Deck** constructor makes use of the class variables **RANKS** and **SUITS** in the **Card** class to order the new cards appropriately. The **shuffle** method simply passes the list of cards to **random.shuffle**. The **deal** method removes and returns the first card in the list, if there is one, or returns the value **None** otherwise. The **len** function,

Deck Method	What It Does
`d = Deck()`	Returns a deck.
`d.__len__()`	Same as `len(d)`. Returns the number of cards currently in the deck.
`d.shuffle()`	Shuffles the cards in the deck.
`d.deal()`	If the deck is not empty, removes and returns the topmost card. Otherwise, returns **None**.
`d.__str__()`	Same as `str(d)`. Returns a string representation of the deck (all the cards in it).

Table 9-8 The interface for the **Deck** class

like the `str` function, calls a method (in this case, `__len__`) that returns the length of the list of cards. Here is the code for **Deck**:

```
import random

# The definition of the Card class goes here

class Deck(object):
    """ A deck containing 52 cards."""

    def __init__(self):
        """Creates a full deck of cards."""
        self.cards = []
        for suit in Card.SUITS:
            for rank in Card.RANKS:
                c = Card(rank, suit)
                self.cards.append(c)

    def shuffle(self):
        """Shuffles the cards."""
        random.shuffle(self.cards)

    def deal(self):
        """Removes and returns the top card or None
        if the deck is empty."""
        if len(self) == 0:
            return None
        else:
            return self.cards.pop(0)

    def __len__(self):
        """Returns the number of cards left in the deck."""
        return len(self.cards)
```

```
def __str__(self) :
    """Returns the string representation of a deck."""
    result = ""
    for c in self.cards:
        result = result + str(c) + '\n'
    return result
```


Exercises

1. Although the use of a PIN to identify a person's bank account is simple, it's not very realistic. Real banks typically assign a unique 12-digit number to each account and use this as well as the customer's PIN during a login at an ATM. Suggest how to rework the banking system discussed in this section to use this information.
2. What is a class variable? When should the programmer define a class variable rather than an instance variable?
3. Describe how the arithmetic operators can be overloaded to work with a new class of numbers.
4. Define a method for the **Bank** class that returns the total assets in the bank (the sum of all account balances).
5. Describe the benefits of pickling objects for file storage.
6. Why would you use a **try-except** statement in a program?
7. Two playing cards can be compared by rank. For example, an Ace is less than a 2. When **c1** and **c2** are cards, **c1.rank < c2.rank** expresses this relationship. Explain how a method could be added to the **Card** class to simplify this expression to **c1 < c2**.

CASE STUDY: An ATM

In this case study, we develop a simple ATM program that uses the **Bank** and **SavingsAccount** classes discussed in the previous section.

Request

Write a program that simulates a simple ATM.

Analysis

Our ATM user logs in with a name and a personal identification number, or PIN. If either string is unrecognized, an error message is displayed. Otherwise, the user can repeatedly

(continues)

(continued)

select options to get the balance, make a deposit, and make a withdrawal. A final option allows the user to log out. Figure 9-2 shows the sample interface for this application.

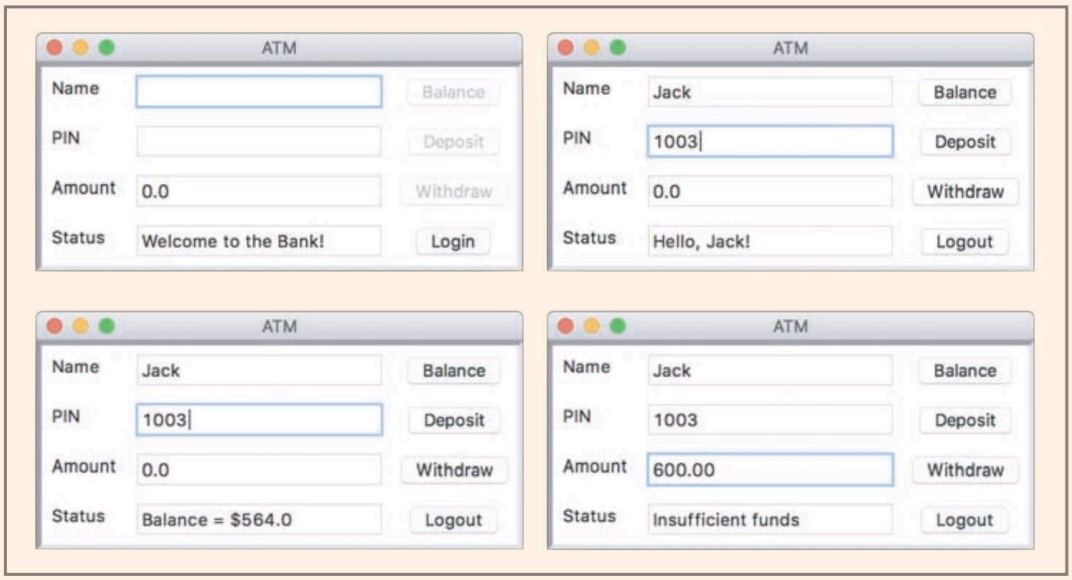

Figure 9-2 The user interface for the ATM program

The data model classes for the program are the **Bank** and **SavingsAccount** classes developed earlier in this chapter. To support user interaction, we also develop a new class called **ATM**. The **class diagram** in Figure 9-3 shows the relationships among these classes.

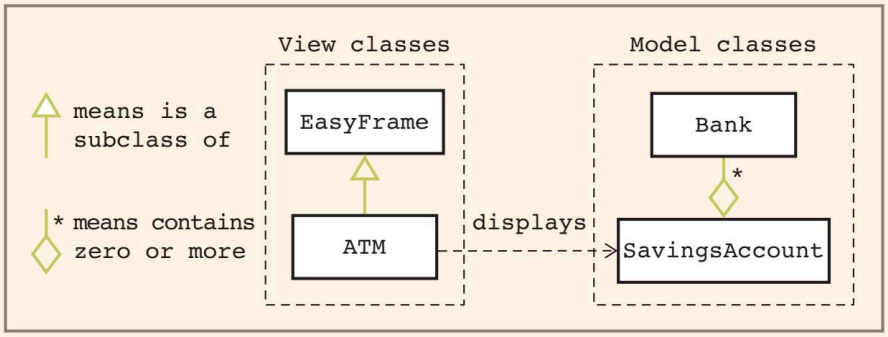

Figure 9-3 A UML diagram for the ATM program showing the program's classes

As you learned in Chapter 8, in a class diagram the name of each class appears in a box. The lines or edges connecting the boxes show the relationships. Note that these edges are labeled or contain arrows. This information describes the number of

(continues)

(continued)

accounts in a bank (zero or more) and the dependency of one class on another (the direction of an arrow). Class diagrams of this type are part of a graphical notation called the **Unified Modeling Language**, or UML. UML is used to describe and document the analysis and design of complex software systems.

In general, it is a good idea to divide the code for most interactive applications into at least two sets of classes. One set of classes, which we call the **view**, handles the program's interactions with human users, including the input and output operations. The other set of classes, called the **model**, represents and manages the data used by the application. In the current case study, the `Bank` and `SavingsAccount` classes belong to the model, whereas the `ATM` class belongs to the view. One of the benefits of this separation of responsibilities is that you can write different views for the same data model, such as a terminal-based view and a GUI-based view, without changing a line of code in the data model. Alternatively, you can write different representations of the data model without altering a line of code in the views. In some of the case studies that follow, we apply this framework, called the **model/view pattern**, to structure the code.

Design

The `ATM` class maintains two instance variables. Their values are the following:

- A `Bank` object
- The `SavingsAccount` of the currently logged-in user

At program start-up, a `Bank` object is loaded from a file. An `ATM` object is then created for this bank. The ATM's `mainloop` method is then called. This method enters an event-driven loop that waits for user events. If a user's name and PIN match those of an account, the ATM's `account` variable is set to the user's account, and the buttons for manipulating the account are enabled. The selection of an option triggers an event-handling method to process that option. Table 9-9 lists the methods in the `ATM` class.

ATM Method	What It Does
`ATM(bank)`	Returns a new `ATM` object based on the data model **bank**.
`login()`	Allows the user to log in.
`logout()`	Allows the user to log out.
`getBalance()`	Displays the user's balance.
`deposit()`	Allows the user to make a deposit.
`withdraw()`	Allows the user to make a withdrawal and displays any error messages.

Table 9-9 The interface for the `ATM` class

(continues)

(continued)

The **ATM** constructor receives a **Bank** object as an argument and saves a reference to it in an instance variable. It also sets its **account** variable to **None**.

Implementation (Coding)

The data model classes **Bank** and **SavingsAccount** are already available in **bank.py** and **savingsaccount.py**. The code for the GUI, in **atm.py**, includes definitions of a main window class named **ATM** and a **main** function. We discuss this function and several of the **ATM** methods, without presenting the complete implementation here.

Before you can run this program, you need to create a bank. For testing purposes, we include in the **Bank** class a simple function named **createBank** that creates and returns a **Bank** object with a number of dummy accounts. Alternatively, the program can load a bank object that has been saved in a file, as discussed earlier.

The **main** function creates a bank and passes this object to the constructor of the **ATM** class. The **ATM** object's **mainloop** method is then run to pop up the window. Here is the code for the imports and the **main** function:

```
"""
File: atm.py
This module defines the ATM class, which provides a window
for bank customers to perform deposits, withdrawals, and
check balances.
"""

from breezypythongui import EasyFrame
from bank import Bank, createBank

# Code for the ATM class goes here (in atm.py)

def main(fileName = None):
    """Creates the bank with the optional file name,
    wraps the window around it, and opens the window.
    Saves the bank when the window closes."""
    if not fileName:
        bank = createBank(5)
    else:
        bank = Bank(fileName)
    print(bank)              # For testing only
    atm = ATM(bank)
    atm.mainloop()
    # Could save the bank to a file here.

if __name__ == "__main__":
     main()
```

(continues)

(continued)

Note that when you launch this as a standalone program, you open the ATM on a bank with 5 dummy accounts; but if you run **main** with a filename argument in the IDLE shell, you open the ATM on a bank created from a saved bank file.

The **__init__** method of **ATM** receives a **Bank** object as an argument and saves a reference to it in an instance variable. This step connects the view (**ATM**) to the model (**Bank**) for the application. The **ATM** object also keeps a reference to the currently open account, which has an initial value of **None**. Here is the code for this method, which omits the straightforward, but rather lengthy and tedious, step of adding the widgets to the window:

```
class ATM(EasyFrame):
    """Represents an ATM window.
    The window tracks the bank and the current account.
    The current account is None at startup and logout.
    """

    def __init__(self, bank):
        """Initialize the window and establish
        the data model."""
        EasyFrame.__init__(self, title = "ATM")
        # Create references to the data model.
        self.bank = bank
        self.account = None
        # Create and add the widgets to the window.
        # Detailed code available in atm.py

    # Event handling methods go here
```

The event handling method to log the user in takes the username and pin from the input fields and attempts to retrieve an account with these credentials from the bank. If this step is successful, the **account** variable will refer to this account, a greeting will be displayed in the status area, and the buttons to manipulate the account will be enabled. Otherwise, the program displays an error message in the status area. Here is the code for the method **login**:

```
def login(self):
    """Attempts to login the customer.  If successful,
    enables the buttons, including logout."""
    name = self.nameField.getText()
    pin = self.pinField.getText()
    self.account = self.bank.get(name, pin)
    if self.account:
        self.statusField.setText("Hello, " + name + "!")
        self.balanceButton["state"] = "normal"
```

(continues)

(continued)

```
            self.depositButton["state"] = "normal"
            self.withdrawButton["state"] = "normal"
            self.loginButton["text"] = "Logout"
            self.loginButton["command"] = self.logout
        else:
            self.statusField.setText("Name and pin not found!")
```

Note that if a login succeeds, the **text** and **command** attributes of the button named **loginButton** are set to the information for logging out. This allows the login and logout functions to be assigned to a single button, as if it were an on/off switch, thereby simplifying the user interface.

The **logout** method clears the view and restores it to its initial state, where it can await another customer, as follows:

```
def logout(self):
    """Logs the customer out, clears the fields,
    disables the buttons, and enables login."""
    self.account = None
    self.nameField.setText("")
    self.pinField.setText("")
    self.amountField.setNumber(0.0)
    self.statusField.setText("Welcome to the Bank!")
    self.balanceButton["state"] = "disabled"
    self.depositButton["state"] = "disabled"
    self.withdrawButton["state"] = "disabled"
    self.loginButton["text"] = "Login"
    self.loginButton["command"] = self.login
```

The remaining three methods cannot be run unless a user has logged in and the account object is currently available. Each method operates on the **ATM** object's **account** variable. The **getBalance** method asks the account for its balance and displays it in the status field:

```
def getBalance(self):
    """Displays the current balance in the
    status field."""
    balance = self.account.getBalance()
    self.statusField.setText("Balance: $" + str(balance))
```

Here you can clearly see the model/view design pattern in action: the user's button click triggers the **getBalance** method, which obtains data from the **SavingsAccount** object (the model), and updates the **TextField** object (the view) with those data.

(continues)

The **withdraw** method exhibits a similar pattern, but it obtains input from the view and handles possible error conditions as well:

```
def withdraw(self):
    """Attempts a withdrawal. If not successful,
    displays error message in statusfield;
    otherwise, announces success."""
    amount = ammountField.getNumber()
    message = self.account.withdraw(amount)
    if message:              # Check for an error message
        self.statusField.setText(message)
    else:
        self.statusField.setText("Withdrawal successful!")
```

Note that the logic of error checking (an amount greater than the funds available) and the logic of the withdrawal itself are the responsibilities of the **SavingsAccount** object (the model), not of the **ATM** object (the view).

Building a New Data Structure: The Two-Dimensional Grid

Like most programming languages, Python includes several basic types of data structures, such as strings, lists, tuples, and dictionaries. Each type of data structure has a specific way of organizing the data contained therein: strings, lists, and tuples are sequences of items ordered by position, whereas dictionaries are sets of key/value pairs in no particular order. Another useful data structure is a **two-dimensional grid**. A grid organizes items by position in rows and columns. You have worked with grids to organize

- pixels in images (Chapter 7)
- widgets in window layouts (Chapter 8)

In Chapter 4, we mentioned that a sophisticated data encryption algorithm uses an invertible matrix, which is also a type of grid. In this section, we develop a new class called **Grid** for applications that require grids.

The Interface of the `Grid` Class

The first step in building a new class is to describe the kind of object it models. You focus on the object's attributes and behavior. A grid is basically a container where you organize items by row and column. You can visualize a grid as a rectangular structure with rows and columns. The rows are numbered from 0 to the number of rows minus 1. The columns are numbered from 0 to the number of columns minus 1. Unlike a list, a grid has a height (number of rows) and a width (number of columns), rather than a length.

The constructor or operation to create a grid allows you to specify the width, the height, and an optional initial fill value for all of the positions. The default fill value is **None**. You

access or replace an item at a given position by specifying the row and column of that position, using the notation

```
grid[<row>][<column>]
```

To assist in operations such as traversals, a grid provides operations to obtain its height and its width. A search operation returns the position, expressed as **(<row>, <column>)** of a given item, or the value **None** if the item is not present in the grid. Finally, an operation builds and returns a two-dimensional string representation of the grid. A sample session shows how these operations might be used:

```
>>> from grid import Grid
>>> grid = Grid(rows = 3, columns = 4, fillValue = 0)
>>> print(grid)
0 0 0 0
0 0 0 0
0 0 0 0
>>> grid[1][2] = 5
>>> print(grid)
0 0 0 0
0 0 5 0
0 0 0 0
>>> print(grid.find(5))
(1,2)
>>> print(grid.find(6))
None
>>> for row in range(grid.getHeight()):
        for column in range(grid.getWidth()):
            grid[row][column] = (row, column)
>>> print(grid)
(0,0) (0,1) (0,2) (0,3)
(1,0) (1,1) (1,2) (1,3)
(2,0) (2,1) (2,2) (2,3)
```

Using these requirements, we can provide the interface for the **Grid** class shown in Table 9-10.

Grid Method	What It Does
`g = Grid(rows, columns, fillValue = None)`	Returns a new **Grid** object.
`g.getHeight()`	Returns the number of rows.
`g.getWidth()`	Returns the number of columns.
`g.__str__()`	Same as **str(g)**. Returns the string representation.
`g.__getitem__(row)[column]`	Same as **g.[row][column]**.
`g.find(value)`	Returns **(row, column)** if **value** is found, or **None** otherwise.

Table 9-10 The interface for the **GRID** class

The Implementation of the `Grid` Class: Instance Variables for the Data

The implementation of a class provides the code for the methods in its interface, as well as the instance variables needed to track the data contained in objects of that class. Because none of these resources can be inherited from a parent class, the **`Grid`** class will be a subclass of **`object`**.

The next step is to choose the data structures that will represent the two-dimensional structure within a **`Grid`** object. A list of lists seems like a wise choice, because most of the grid's operations can easily map to list operations. A single instance variable named **`self.data`** holds the top-level list of rows, and each item within this list will be a list of the columns in that row. The method **`getHeight`** returns the length of the top-level list, while the method **`getWidth`** returns the length of the list at position 0 within the top-level list. Note that because the grid is rectangular, all of the nested lists are of the same length. The expression **`self.data[row][column]`** drills into the list at position **`row`** within the top-level list, and then accesses the item at position **`column`** in the nested list.

The other two methods to treat in this step are **`__init__`**, which initializes the instance variables, and **`__str__`**, which allows you to view the data during testing. Here is the code for a working prototype of the **`Grid`** class with the four methods discussed thus far:

```
class Grid(object):
    """Represents a two-dimensional grid."""

    def __init__(self, rows, columns, fillValue = None):
        """Sets up the data."""
        self.data = []
        for row in range(rows):
            dataInRow = []
            for column in range(columns):
                dataInRow.append(fillValue)
            self.data.append(dataInRow)

    def getHeight(self):
        """Returns the number of rows."""
        return len(self.data)

    def getWidth(self):
        """Returns the number of columns."""
        return len(self.data[0])

    def __str__(self):
        """Returns a string representation of the grid."""
        result = ""
        for row in range(self.getHeight()):
            for col in range(self.getWidth()):
                result += str(self.data[row][col]) + " "
            result += "\n"
        return result
```

The Implementation of the `Grid` Class: Subscript and Search

The remaining methods implement the subscript and the search operations on a grid.

The subscript operator is used to access an item at a grid position or to replace it there. In the case of access, the subscript appears within an expression, as in **`grid[1][2]`**. In this case, when Python sees the **`[]`** following an object, it looks for a method named **`__getitem__`** in the object's class. This method expects an index as an argument, and returns the item at that index in the underlying data structure. In the case of the **`Grid`** class, this method returns a nested list at the given index in the top-level list. This list represents a row of data in the grid. Python then uses the second subscript on this list to obtain the item at the given column in this row. Here is the code for this method:

```
def __getitem__(self, index):
    """Supports two-dimensional indexing with [][].
    Index represents a row number."""
    return self.data[index]
```

This method also handles the case when the subscript appears on the left side of an assignment statement, during a replacement of at item at a given position in a grid, as in

```
grid[1][2] = 5
```

The search operation named **`find`** must loop through the grid's list of lists, until it finds the target item or runs out of items to examine. The code for the implementation uses the familiar grid traversal pattern that you learned in Chapters 7 and 8:

```
def find(self, value):
    """Returns (row, column) if value is found,
    or None otherwise."""
    for row in range(self.getHeight()):
        for column in range(self.getWidth()):
            if self[row][column] == value:
                return (row, column)
    return None
```

Note how this method uses the subscripts with **`self`** rather than **`self.data`**. Here we take advantage of the fact that the subscripts now work with grids as well as lists.

CASE STUDY: Data Encryption with a Block Cipher

In Chapter 4, we developed code to encrypt text with a Caesar cipher. We mentioned that a linear encryption method like this one is easy to crack, but that a method that uses a block cipher is harder to crack. In this case study, we use the **`Grid`** class to develop an encryption program that employs a block cipher.

Request

Write a program that uses a block cipher to encrypt text.

(continues)

(continued)

Analysis

A block cipher encryption method uses a two-dimensional grid of the characters, also called a **matrix**, to convert the plaintext to the ciphertext. The algorithm converts consecutive pairs of characters in the plaintext to pairs of characters in the ciphertext. For each pair of characters, it locates the positions of those characters in the matrix. If the two characters are in the same row or column, it simply swaps the positions of these characters and adds them to the ciphertext. Otherwise, it locates the two characters at the opposite corners of a rectangle in the matrix, and adds these two characters to the ciphertext. Figure 9-4 shows a snapshot of this process.

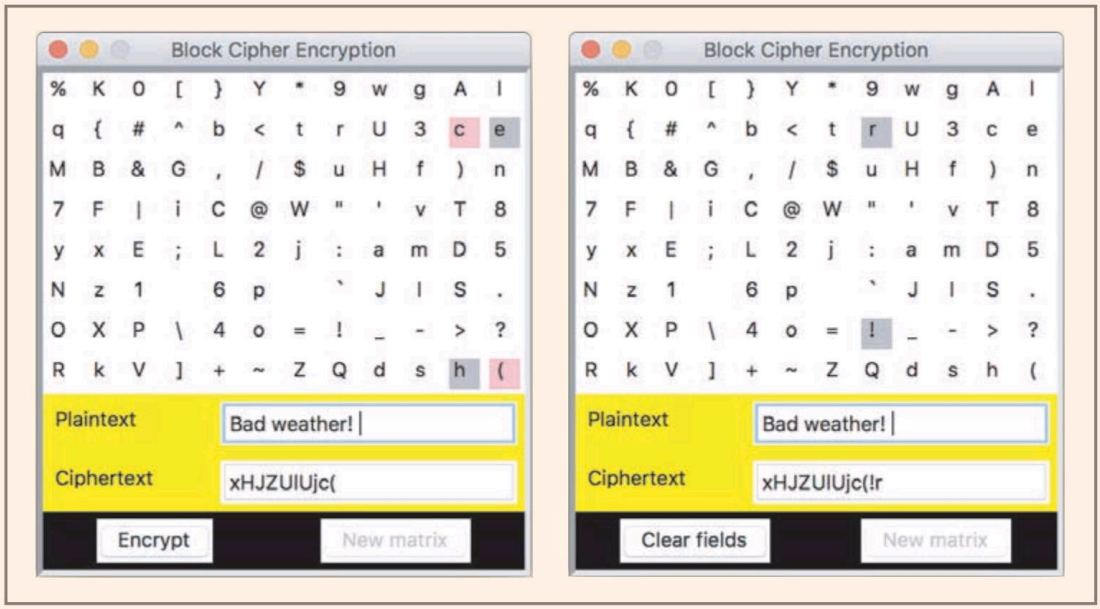

Figure 9-4 Encrypting text with a block cipher

This interface allows the user to step through the encryption process. When the user clicks the **Encrypt** button, the program locates the next pair of plaintext characters in the matrix and marks them with gray boxes. It then marks the ciphertext characters at the opposite corners, if they exist, with pink boxes. The window on the left in Figure 9-4 shows two sets of marked characters, whereas the window on the right shows gray marks only. In the first case, the characters in pink are added to the ciphertext; in the second case, the characters in gray are reversed before this addition.

Note that the characters are in random order in the matrix, and that the program allows the user to reset the grid to a new randomly ordered set of characters when the encryption is finished.

(continues)

(continued)

Although the GUI shown in Figure 9-4 is available in the example programs for this book, we develop a simpler terminal-based version here. The following session illustrates its features:

```
>>> main()
Enter the plaintext: Ken Lambert
Encrypting . . .
 Plain text: Ken Lambert
Cipher text: .n1UM@8Gs/t
Decrypting . . .
 Cipher text: .n1UM@8Gs/t
Plain text: Ken Lambert
>>> main("Weather: cloudy tomorrow")
Encrypting . . .
 Plain text: Weather: cloudy tomorrow
Cipher text: q-taPgfQ@solWQgUTa];rr;T
Decrypting . . .
 Cipher text: q-taPgfQ@solWQgUTa];rr;T
Plain text: Weather: cloudy tomorrow
```

Note that the **main** function defaults to a prompt for user input if an argument is not supplied. Otherwise, **main** uses its argument as the plaintext.

Design and Implementation

The first step in this design is to define a function that builds the matrix for the block cipher. This function, **makeMatrix**, fills a list with the characters from ASCII 32 through ASCII 127. These are the printable characters in this set (except for the new-line and tab characters). The function then shuffles the list to randomize the characters. It next creates a new 8-by-12 **Grid** object and copies the characters from the list to the grid. Finally, the grid is returned. Here is the code for the **makeMatrix** function:

```
from grid import Grid
import random

def makeMatrix():
    """Builds and returns an encryption matrix."""
    listOfChars = []
    for ascii in range(32, 128):
        listOfChars.append(chr(ascii))
    random.shuffle(listOfChars)
    matrix = Grid(8, 12)
    i = 0
    for row in range(matrix.getHeight()):
        for column in range(matrix.getWidth()):
            matrix[row][column] = listOfChars[i]
            i += 1
    return matrix
```

(continues)

(continued)

The next step is the design of the **encrypt** function. This function expects the plaintext and a matrix as arguments and returns the ciphertext. The function guides the process of moving through consecutive pairs of characters in the plaintext and adding the corresponding pairs of characters to the ciphertext under construction. Because the process of converting a pair of characters is rather complicated, we delegate it to a helper function named **encryptPair**. The **encrypt** function also handles the oddball case of a plaintext with an odd number of characters. In that case, the function simply adds the last plaintext character to the ciphertext. Here is the code for the **encrypt** function:

```
def encrypt(plainText, matrix):
    """Uses matrix to encrypt plainText,
    and returns cipherText."""
    cypherText = ""
    limit = len(plainText)
    # Adjust for an odd number of characters
    if limit % 2 == 1:
        limit -= 1
    # Use the matrix to encrypt pairs of characters
    i = 0
    while i < limit:
        cypherText += encryptPair(plainText, i, matrix)
        i += 2
    # Add the last character if length was odd
    if limit < len(plainText):
        cypherText += plainText[limit]
    return cypherText
```

The **encryptPair** function expects the plaintext, the current character position, and the matrix as arguments and returns a string containing a two-character ciphertext. The function first searches the matrix for the characters at the current and next positions in the plaintext. The function then uses the results, two pairs of grid coordinates, to generate the pair of characters in the ciphertext. In one case, where the two rows or the two columns in the coordinates are the same, the function just swaps the positions of the plaintext characters. In the other case, it retrieves the ciphertext characters from the opposite corners of the rectangle formed by the positions of the plaintext characters in the matrix. Here is the code for the **encryptPair** function:

```
def encryptPair(plainText, i, matrix):
    """Returns the cipherText of the pair of
    characters at i and i + 1 in plainText."""
    # Locate the characters in the matrix
    (row1, col1) = matrix.find(plainText[i])
    (row2, col2) = matrix.find(plainText[i + 1])
```

(continues)

(continued)

```
    # Swap them if they are in the same row or column
    if row1 == row2 or col1 == col2:
        return plainText[i + 1] + plainText[i]
    # Otherwise, use the characters at the opposite
    # corners of the rectangle in the matrix
    else:
        ch1 = matrix[row2][col1]
        ch2 = matrix[row1][col2]
        return ch1 + ch2
```

The good news is that the algorithm to decrypt a ciphertext with a block cipher is the same as the algorithm to encrypt a plaintext with the same block cipher. Therefore, the **decrypt** function simply calls **encrypt** with the ciphertext and matrix as arguments:

```
def decrypt(cipherText, matrix):
    """Uses matrix to decrypt cipherText,
    and returns plainText."""
    return encrypt(cipherText, matrix)
```

One limitation of our design is that it works only for one-line strings of text. A more general method would add all 128 ASCII values, including the newline and tab characters, to the matrix. Then you would be able to encrypt and decrypt entire text files.

Structuring Classes with Inheritance and Polymorphism

Object-based programming involves the use of objects, classes, and methods to solve problems. Object-oriented programming requires the programmer to master the following additional concepts:

1. **Data encapsulation**. Restricting the manipulation of an object's state by external users to a set of method calls.
2. **Inheritance**. Allowing a class to automatically reuse and extend the code of similar but more general classes.
3. **Polymorphism**. Allowing several different classes to use the same general method names.

Although Python is considered an object-oriented language, its syntax does not enforce data encapsulation. As you have seen, in the case of simple container objects, like playing cards, with little special behavior, it is handy to be able to access the objects' data without a method call.

Unlike data encapsulation, inheritance and polymorphism are built into Python's syntax. In this section we examine how they can be exploited to structure code.

Inheritance Hierarchies and Modeling

Objects in the natural world and objects in the world of artifacts can be classified using **inheritance hierarchies**. A simplified hierarchy of natural objects is depicted in Figure 9-5.

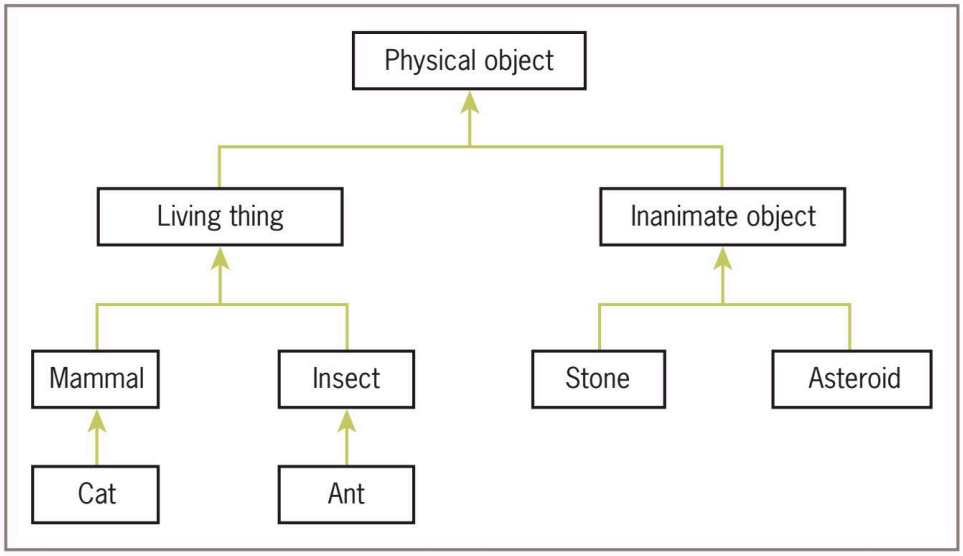

Figure 9-5 A simplified hierarchy of objects in the natural world

At the top of a hierarchy is the most general class of objects. This class defines features that are common to every object in the hierarchy. For example, every physical object has a mass. Classes just below this one have these features as well as additional ones. Thus, a living thing has a mass and can also grow and die. The path from a given class back up to the topmost one goes through all of that given class's ancestors. Each class below the topmost one inherits attributes and behaviors from its ancestors and extends these with additional attributes and behavior.

An object-oriented software system models this pattern of inheritance and extension in real-world systems by defining classes that extend other classes. In Python, all classes automatically extend the built-in **`object`** class, which is the most general class possible. However, as you learned in Chapter 8, it is possible to extend any existing class using the syntax:

```
class <new class name>(<existing parent class name>):
```

Thus, for example, **`PhysicalObject`** would extend **`object`**, **`LivingThing`** would extend **`PhysicalObject`**, and so on.

The real advantage of inheritance in a software system is that each new subclass acquires all of the instance variables and methods of its ancestor classes for free. Like function definitions and class definitions, inheritance hierarchies provide an abstraction mechanism that allows the programmer to avoid reinventing the wheel or writing redundant code, as you clearly saw in Chapter 8. To review how inheritance works in Python, we explore two more examples.

Example 1: A Restricted Savings Account

So far, our examples have focused on ordinary savings accounts. Banks also provide customers with restricted savings accounts. These are like ordinary savings accounts in most ways, but with some special features, such as allowing only a certain number of deposits or withdrawals a month. Let's assume that a savings account has a name, a PIN, and a balance. You can make deposits and withdrawals and access the account's attributes. Let's also assume that this restricted savings account permits only three withdrawals per month. The next session shows an interaction with a **RestrictedSavingsAccount** that permits up to three withdrawals:

```
>>> account = RestrictedSavingsAccount("Ken", "1001", 500.00)
>>> print(account)
Name:    Ken
PIN:     1001
Balance: 500.0
>>> account.getBalance()
500.0
>>> for count in range(3):
        account.withdraw(100)
>>> account.withdraw(50)
'No more withdrawals this month'
>>> account.resetCounter()
>>> account.withdraw(50)
```

The fourth withdrawal has no effect on the account, and it returns an error message. A new method named **resetCounter** is called to enable withdrawals for the next month.

If **RestrictedSavingsAccount** is defined as a subclass of **SavingsAccount**, every method but **withdraw** can simply be inherited and used without changes. The **withdraw** method is redefined in **RestrictedSavingsAccount** to return an error message if the number of withdrawals has exceeded the maximum. The maximum will be maintained in a new class variable, and the monthly count of withdrawals will be tracked in a new instance variable. Finally, a new method, **resetCounter**, is included to reset the number of withdrawals to 0 at the end of each month. Here is the code for the **RestrictedSavingsAccount** class, followed by a brief explanation:

```
"""
File: restrictedsavingsaccount.py
This module defines the RestrictedSavingsAccount class.
"""

from savingsaccount import SavingsAccount

class RestrictedSavingsAccount(SavingsAccount):
    """This class represents a restricted
    savings account."""

    MAX_WITHDRAWALS = 3
```

```
    def __init__(self, name, pin, balance = 0.0):
        """Same attributes as SavingsAccount, but with
        a counter for withdrawals."""
        SavingsAccount.__init__(self, name, pin, balance)
        self.counter = 0

    def withdraw(self, amount):
        """Restricts number of withdrawals to MAX_WITHDRAWALS."""
        if self.counter == RestrictedSavingsAccount.MAX_WITHDRAWALS:
            return "No more withdrawals this month"
        else:
            message = SavingsAccount.withdraw(self, amount)
            if message == None:
                self.counter += 1
            return message

    def resetCounter(self):
        """Resets the withdrawal count."""
        self.counter = 0
```


The **RestrictedSavingsAccount** class includes a new class variable not found in **SavingsAccount**. This variable, called **MAX_WITHDRAWALS**, is used to restrict the number of withdrawals that are permitted per month.

The **RestrictedSavingsAccount** constructor first calls the constructor in the **SavingsAccount** class to initialize the instance variables for the name, PIN, and balance defined there. The syntax uses the class name before the dot, and explicitly includes **self** as the first argument. The general form of the syntax for calling a method in the parent class from within a method with the same name in a subclass follows:

```
<parent class name>.<method name>(self, <other arguments>)
```

Continuing in **RestrictedSavingsAccount**'s constructor, the new instance variable **counter** is then set to 0. The rule of thumb to remember when writing the constructor for a subclass is that each class is responsible for initializing its own instance variables. Thus, the constructor of the parent class should always be called to do this.

The **withdraw** method is redefined in **RestrictedSavingsAccount** to override the definition of the same method in **SavingsAccount**. You allow a withdrawal only when the counter's value is less than the maximum, and you increment the counter only after a withdrawal is successful. Note that this version of the method calls the same method in the parent or superclass to perform the actual withdrawal. The syntax for this is the same as is used in the constructor.

Finally, the new method **resetCounter** is included to allow the user to continue withdrawals in the next month.

Example 2: The Dealer and a Player in the Game of Blackjack

The card game of blackjack is played with at least two players, one of whom is also a dealer. The object of the game is to receive cards from the deck and play to a count of 21 without going over 21. A card's point equals its rank, but all face cards are 10 points, and an Ace

can count as either 1 or 11 points as needed. At the beginning of the game, the dealer and the player each receive two cards from the deck. The player can see both of her cards and just one of the dealer's cards initially. The player then "hits" or takes one card at a time until her total exceeds 21 (a "bust" or loss), or she "passes" (stops taking cards). When the player passes, the dealer reveals his other card and must keep taking cards until his total is greater than or equal to 17. If the dealer's final total is greater than 21, he also loses. Otherwise, the player with the higher point total wins, or else there is a tie.

A computer program that plays this game can use a **Dealer** object and a **Player** object. The dealer's moves are completely automatic, whereas the player's moves (decisions to pass or hit) are partly controlled by a human user. A third object belonging to the **Blackjack** class sets up the game and manages the interactions with the user. The **Deck** and **Card** classes developed earlier are also included. A class diagram of the system is shown in Figure 9-6.

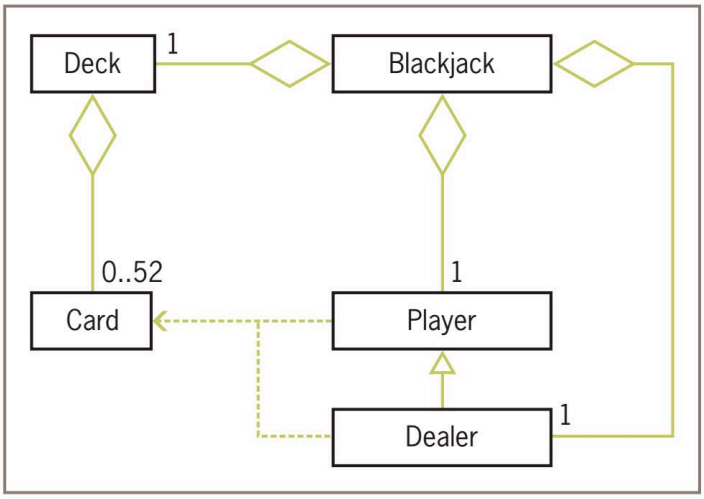

Figure 9-6 The classes in the blackjack game application

Here is a sample run of the program:

```
>>> from blackjack import Blackjack
>>> game = Blackjack()
>>> game.play()
Player:
2 of Spades, 5 of Spades
7 points Dealer:
5 of Hearts
Do you want a hit? [y/n]: y
Player:
2 of Spades, 5 of Spades, King of Hearts
17 points
Do you want a hit? [y/n]: n
Dealer:
5 of Hearts, Queen of Hearts, 7 of Diamonds
22 points
Dealer busts and you win
```

When a **Player** object is created, it receives two cards. A **Player** object can be hit with another card, can be asked for the points in its hand, and can be asked for its string representation. Here is the code for the **Player** class, followed by a brief explanation:


```
from cards import Deck, Card

class Player(object):
    """This class represents a player in
    a blackjack game."""

    def __init__(self, cards):
        self.cards = cards

    def __str__(self):
        """Returns string rep of cards and points."""
        result = ", ".join(map(str, self.cards))
        result += "\n " + str(self.getPoints()) + " points"
        return result

    def hit(self, card):
        self.cards.append(card)

    def getPoints(self ) :
        """Returns the number of points in the hand."""
        count = 0
        for card in self.cards:
            if card.rank > 9:
               count += 10
        elif card.rank == 1:
            count += 11
        else:
            count += card.rank
        # Deduct 10 if Ace is available and needed as 1
        for card in self.cards:
            if count <= 21:
                break
            elif card.rank == 1:
                count -= 10
        return count

    def hasBlackjack(self):
        """Dealt 21 or not."""
        return len(self.cards) == 2 and self.getPoints() == 21
```

The problem of computing the points in a player's hand is complicated by the fact that an Ace can count as either 1 or 11. The **getPoints** method solves this problem by first totaling the points using an Ace as 11. If this initial count is greater than 21, then there is a need to count an Ace, if there is one, as a 1. The second loop accomplishes this by counting such Aces as long as they are available and needed. The other methods require no comment.

A **Dealer** object also maintains a hand of cards and recognizes the same methods as a **Player** object. However, the dealer's behavior is a bit more specialized. For example, the dealer at first shows just one card, and the dealer repeatedly hits until 17 points are reached or exceeded. Thus, as Figure 9-6 shows, **Dealer** is best defined as a subclass of **Player**. Here is the code for the **Dealer** class, followed by a brief explanation:

```
class Dealer(Player) :
    """Like a Player, but with some restrictions."""

    def __init__(self, cards):
        """Initial state: show one card only."""
        Player.__init__(self, cards)
        self.showOneCard = True

    def __str__(self) :
        """Return just one card if not hit yet."""
        if self.showOneCard:
            return str(self.cards[0])
        else:
            return Player.__str__(self)

    def hit(self, deck):
        """Add cards while points < 17,
        then allow all to be shown."""
        self.showOneCard = False
        while self.getPoints() < 17:
            self.cards.append(deck.deal())
```

Dealer maintains an extra instance variable, **showOneCard**, which restricts the number of cards in the string representation to one card at start-up. As soon as the dealer hits, this variable is set to **False**, so all of the cards will be included in the string from then on. The **hit** method actually receives a deck rather than a single card as an argument, so cards may repeatedly be dealt and added to the dealer's list at the close of the game.

The **Blackjack** class coordinates the interactions among the **Deck** object, the **Player** object, the **Dealer** object, and the human user. Here is the code:

```
class Blackjack(object):

    def __init__(self):
        self.deck = Deck()
        self.deck.shuffle()
        # Pass the player and the dealer two cards each
        self.player = Player([self.deck.deal(),
                              self.deck.deal()])
        self.dealer = Dealer([self.deck.deal(),
                              self.deck.deal()])
    def play(self):
        print("Player:\n", self.player)
        print("Dealer:\n", self.dealer)
```

```
# Player hits until user says NO
while True:
    choice = input("Do you want a hit? [y/n]: ")
    if choice in ("Y", "y"):
        self.player.hit(self.deck.deal())
        points = self.player.getPoints()
        print("Player:\n", self.player)
        if points >= 21:
            break
    else:
        break
    playerPoints = self.player.getPoints()
    if playerPoints > 21:
        print("You bust and lose")
    else:
        # Dealer's turn to hit
        self.dealer.hit(self.deck)
        print("Dealer:\n", self.dealer)
        dealerPoints = self.dealer.getPoints()
        # Determine the outcome
        if dealerPoints > 21:
            print("Dealer busts and you win")
        elif dealerPoints > playerPoints:
            print("Dealer wins")
        elif dealerPoints < playerPoints and  \
             playerPoints <= 21:
            print("You win")
        elif dealerPoints == playerPoints:
            if self. player.hasBlackjack() and \
               not self.dealer.hasBlackjack():
                print("You win")
            elif not self.player.hasBlackjack() and \
                 self.dealer.hasBlackjack():
                print("Dealer wins")
            else:
                print("There is a tie")
```

Polymorphic Methods

As we have seen in our two examples, a subclass inherits data and methods from its parent class. We would not bother subclassing unless the two classes shared a substantial amount of **abstract behavior**. By this term, we mean that the classes have similar sets of methods or operations. A subclass usually adds something extra, such as a new method or a data attribute, to the ensemble provided by its superclass. A new data attribute is included in both of our examples, and a new method is included in the first one.

In some cases, the two classes have the same interface, or set of methods available to external users. In these cases, one or more methods in a subclass override the definitions of the

same methods in the superclass to provide specialized versions of the abstract behavior. Like any object-oriented language, Python supports this capability with **polymorphic methods**. The term *polymorphic* means "many bodies," and it applies to two methods that have the same header but have different definitions in different classes. Two examples are the `withdraw` method in the bank account hierarchy and the `hit` method in the blackjack player hierarchy. The `__str__` method is a good example of a polymorphic method that appears throughout Python's system of classes.

Like other abstraction mechanisms, polymorphic methods make code easier to understand and use, because the programmer does not have to remember so many different names.

The Costs and Benefits of Object-Oriented Programming

Whenever you learn a new style of programming, you sooner or later become acquainted with its costs and benefits. To hasten this process, we conclude this section by comparing several programming styles, all of which have been used in this book.

The approach with which this book began is called **imperative programming**. Code in this style consists of input and output statements, assignment statements, and control statements for selection and iteration. The name derives from the idea that a program consists of a set of commands to the computer, which responds by performing such actions as manipulating data values in memory. This style is appropriate for writing very short code sequences that accomplish simple tasks, such as solving the problems that were introduced in Chapters 1 through 5 of this book.

However, as problems become more complex, the imperative programming style does not scale well. In particular, the number of interactions among statements that manipulate the same data variables quickly grows beyond the point of comprehension of a human programmer who is trying to verify or maintain the code.

As we saw in Chapter 6, you can mitigate some of this complexity by embedding sequences of imperative code in function definitions or subprograms. It then becomes possible to decompose complex problems into simpler subproblems that can be solved by these subprograms. In other words, the use of subprograms reduces the number of program components that one must keep track of. Moreover, when each subprogram has its own temporary variables and receives data from the surrounding program by means of explicit parameters, the number of possible dependencies and interactions among program components also decreases. The use of cooperating subprograms to solve problems is called **procedural programming**.

Although procedural programming takes a step in the direction of controlling program complexity, it simply masks and ultimately recapitulates the problems of imperative programming at a higher level of abstraction. When many subprograms share and modify a common data pool, as they did in some of our early examples, it becomes difficult once again for the programmer to keep track of all of the interactions among the subprograms during verification and maintenance.

One cause of this problem is the use of the assignment statement to modify data. Some computer scientists have developed a style of programming that dispenses with assignment altogether. This radically different approach, called **functional programming**, views a program as a set of cooperating functions. A function in this sense is a highly restricted subprogram. Its sole purpose is to transform the data in its arguments into other data, its returned value. Because assignment does not exist, functions perform computations by either evaluating expressions or calling other functions. Selection is handled by a conditional expression, which is like an `if-else` statement that returns a value, and iteration is implemented by recursion. By restricting how functions can use data, this very simple model of computation dramatically reduces the conceptual complexity of programs. However, some argue that this style of programming does not conveniently model situations where data objects must change their state.

Object-oriented programming attempts to control the complexity of a program while still modeling data that change their state. This style divides up the data into relatively small units called objects. Each object is then responsible for managing its own data. If an object needs help with its own tasks, it can call upon another object or relies on methods defined in its superclass. The main goal is to divide responsibilities among small, relatively independent or loosely coupled components. Cooperating objects, when they are well designed, decrease the likelihood that a system will break when changes are made within a component.

Although object-oriented programming has become quite popular, it can be overused and abused. Many small and medium-sized problems can still be solved effectively, simply, and—most important—quickly using any of the other three styles of programming mentioned here, either individually or in combination. The solutions of problems, such as numerical computations, often seem contrived when they are cast in terms of objects and classes. For other problems, the use of objects is easy to grasp, but their implementation in the form of classes reflects a complex model of computation with daunting syntax and semantics. Finally, hidden and unpleasant interactions can lurk in poorly designed inheritance hierarchies that resemble those afflicting the most brittle procedural programs.

To conclude, whatever programming style or combination of styles you choose to solve a problem, good design and common sense are essential.

Exercises

1. What are the benefits of having class `B` extend or inherit from class `A`?
2. Describe what the `__init__` method should do in a class that extends another class.
3. Class `B` extends class `A`. Class `B` defines an `__str__` method that returns the string representation of its instance variables. Class `B` defines a single instance variable named `age`, which is an integer. Write the code to define the `__str__` method for class `B`. This method should return the combined string information from both classes. Label the data for `age` with the string `"Age: "`.

CHAPTER 8

Graphical User Interfaces

After completing this chapter, you will be able to:

- Design and code a GUI-based program
- Define a new class using subclassing and inheritance
- Instantiate and lay out different types of window components, such as labels, entry fields, and command buttons, in a window's frame
- Define methods that handle events associated with window components
- Organize sets of window components in nested frames

Most people do not judge a book by its cover. They are interested in its contents, not its appearance. However, users judge a software product by its user interface because they have no other way to access its functionality. With the exception of Chapter 7, in which we explored graphics and image processing, this book has focused on programs that present a terminal-based user interface. This type of user interface is perfectly adequate for some applications, and it is the simplest and easiest for beginning programmers to code. However, 99% of the world's computer users never see such a user interface. Instead, most interactive computer software employs a **graphical user interface** or **GUI** (or its close relative, the touchscreen interface). A GUI displays text as well as small images (called icons) that represent objects such as folders, files of different types, command buttons, and drop-down menus. In addition to entering text at the keyboard, the user of a GUI can select some of these icons with a pointing device, such as a mouse, and move them around on the display. Commands can be activated by pressing the enter key or control keys, by pressing a command button, by selecting a drop-down menu item, or by double-clicking on some icons with the mouse. Put more simply, a GUI displays all information, including text, graphically to its users and allows them to manipulate this information directly with a pointing device.

In this chapter, you will learn how to develop GUIs. Much GUI-based programming requires you to use existing classes, objects, and their methods, as you did in previous chapters. Along the way, you will also learn to how to develop new classes of objects, such as application windows, by extending or repurposing existing classes. Rather than defining a new class of objects from scratch, you will create a customized version of an existing class by the mechanisms of **subclassing** and **inheritance**. GUI programming provides an engaging area for learning these techniques, which play a prominent role in modern software development.

The Behavior of Terminal-Based Programs and GUI-Based Programs

The transition to GUIs involves making a significant adjustment to your thinking. A GUI program is event driven, meaning that it is inactive until the user clicks a button or selects a menu option. In contrast, a terminal-based program maintains constant control over the interactions with the user. Put differently, a terminal-based program prompts users to enter successive inputs, whereas a GUI program puts users in change, allowing them to enter inputs in any order and waiting for them to press a command button or select a menu option.

To make this difference clear, we begin by examining the look and behavior of two different versions of the same program from a user's point of view. This program, first introduced in Chapter 2, computes and displays a person's income tax, given two inputs—the gross income and the number of dependents. The first version of the program includes a terminal-based user interface, whereas the second version uses a graphical user interface. Although both programs perform the same function, their behavior, look, and feel from a user's perspective are quite different.

The Terminal-Based Version

The terminal-based version of the program prompts the user for his gross income and number of dependents. After he enters his inputs, the program responds by computing and displaying his income tax. The program then terminates execution. A sample session with this program is shown in Figure 8-1.

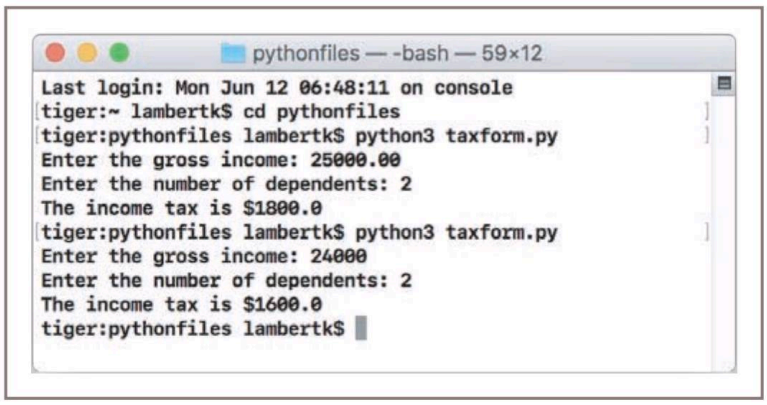

Figure 8-1 A session with the terminal-based tax calculator program

This terminal-based user interface has several obvious effects on its users:

- The user is constrained to reply to a definite sequence of prompts for inputs. Once an input is entered, there is no way to back up and change it.
- To obtain results for a different set of input data, the user must run the program again. At that point, all of the inputs must be re-entered.

Each of these effects poses a problem for users that can be solved by converting the interface to a GUI.

The GUI-Based Version

The GUI-based version of the program displays a **window** that contains various components, also called **widgets**. Some of these components look like text, while others provide visual cues as to their use. Figure 8-2 shows snapshots of a sample session with this version of the program. The snapshot on the left shows the interface at program start-up, whereas the snapshot on the right shows the interface after the user has entered inputs and clicked the **Compute** button. This program was run on a Macintosh; on a Windows- or Linux-based PC, the windows look slightly different.

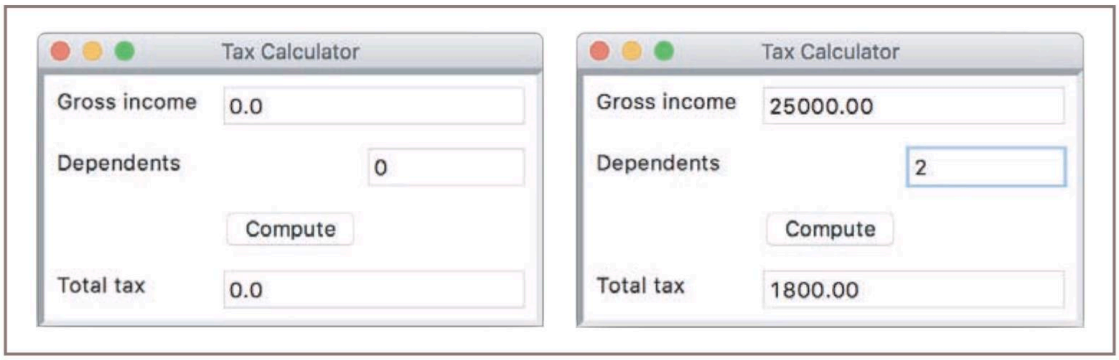

Figure 8-2 A GUI-based tax calculator program

The window in Figure 8-2 contains the following components:

- A **title bar** at the top of the window. This bar contains the title of the program, "Tax Calculator." It also contains three colored disks. Each disk is a **command button**. The user can use the mouse to click the left disk to quit the program, the middle disk to minimize the window, or the right disk to zoom the window. The user can also move the window around the screen by holding the left mouse button on the title bar and dragging the mouse.
- A set of **labels** along the left side of the window. These are text elements that describe the inputs and outputs. For example, "Gross income" is one label.
- A set of **entry fields** along the right side of the window. These are boxes within which the program can output text or receive it as input from the user. The first two entry fields will be used for inputs, while the last field will be used for the output. At program start-up, the fields contain default values, as shown in the window on the left side of Figure 8-2.
- A single command button labeled **Compute**. When the user uses the mouse to press this button, the program responds by using the data in the two input fields to compute the income tax. This result is then displayed in the output field. Sample input data and the corresponding output are shown in the window on the right side of Figure 8-2.
- The user can also alter the size of the window by holding the mouse on its lower-right corner and dragging in any direction.

Although this review of features might seem tedious to anyone who regularly uses GUI-based programs, a careful inventory is necessary for the programmer who builds them. Also, a close study of these features reveals the following effects on users:

- The user is not constrained to enter inputs in a particular order. Before she presses the **Compute** button, she can edit any of the data in the two input fields.
- Running different data sets does not require re-entering all of the data. The user can edit just one value and press the **Compute** button to observe different results.

When we compare the effects of the two interfaces on users, the GUI seems to be a definite improvement on the terminal-based user interface. The improvement is even more noticeable as the number of command options increases and the information to be presented grows in quantity and complexity.

Event-Driven Programming

Rather than guide the user through a series of prompts, a GUI-based program opens a window and waits for the user to manipulate window components with the mouse. These user-generated events, such as mouse clicks, trigger operations in the program to respond by pulling in inputs, processing them, and displaying results. This type of software system is event-driven, and the type of programming used to create it is called **event-driven programming**.

Like any complex program, an event-driven program is developed in several steps. In the analysis step, the types of window components and their arrangement in the window are determined. Because GUI-based programs are almost always object based, this becomes a matter of choosing among GUI component classes available in the programming language or inventing new ones if needed. Graphic designers and cognitive psychologists might be called in to assist in this phase, if the analysts do not already possess this type of expertise. To a certain extent, the number, types, and arrangement of the window components depend on the nature of the information to be displayed and also depend on the set of commands that will be available to the user for manipulating that information.

Let us return to the example of the tax calculator program to see how it might be structured as an event-driven program. The GUI in this program consists of the window and its components, including the labeled entry fields and the **Compute** button. The action triggered when this button is clicked is a method call. This method fetches the input values from the input fields and performs the computation. The result is then sent to the output field to be displayed.

Once the interactions among these resources have been determined, their coding can begin. This phase consists of several steps:

1. Define a new class to represent the main application window.
2. Instantiate the classes of window components needed for this application, such as labels, fields, and command buttons.
3. Position these components in the window.
4. Register a method with each window component in which an event relevant to the application might occur.
5. Define these methods to handle the events.
6. Define a `main` function that instantiates the window class and runs the appropriate method to launch the GUI.

In coding the program, you could initially skip steps 4 and 5, which concern responding to user events. This would allow you to preview and refine the window and its layout, even though the command buttons and other GUI elements lack functionality.

In the sections that follow, we explore these elements of GUI-based, event-driven programming with examples in Python.

Exercises

1. Describe two fundamental differences between terminal-based user interfaces and GUIs.
2. Give an example of one application for which a terminal-based user interface is adequate and one example that lends itself best to a GUI.

Coding Simple GUI-Based Programs

In this section, we show some examples of simple GUI-based programs in Python. Python's standard **`tkinter`** module includes classes for windows and numerous types of window components, but its use can be challenging for beginners. Therefore, this book uses a custom, open-source module called **`breezypythongui`**, while occasionally relying upon some of the simpler resources of **`tkinter`**. You will find the code, documentation, and installation instructions for the **`breezypythongui`** module at *http://home.wlu.edu/~lambertk/breezypythongui/*. We start with some short demo programs that illustrate some basic GUI components, and, in later sections, we develop some examples with more significant functionality.

A Simple "Hello World" Program

Our first demo program defines a class for a main window that displays a greeting. Figure 8-3 shows a screenshot of the window.

Figure 8-3 Displaying a label with text in a window

As in all of our GUI-based programs, a new window class **extends** the **`EasyFrame`** class. By "extends," we mean "repurposes" or "provides extra functionality for." The **`EasyFrame`** class

provides the basic functionality for any window, such as the command buttons in the title bar. Our new class, named **LabelDemo**, provides additional functionality to the **EasyFrame** class. Here is the code for the program:

```
"""
File: labeldemo.py
"""

from breezypythongui import EasyFrame

class LabelDemo(EasyFrame):
    """Displays a greeting in a window."""

    def __init__(self):
        """Sets up the window and the label."""
        EasyFrame.__init__(self)
        self.addLabel(text = "Hello world!", row = 0, column = 0)

def main():
    """Instantiates and pops up the window."""
    LabelDemo().mainloop()

if __name__ == "__main__":
    main()
```

We will speak more generally about class definitions shortly. For now, note that this program performs the following steps:

1. Import the **EasyFrame** class from the **breezypythongui** module. This class is a subclass of **tkinter**'s **Frame** class, which represents a top-level window. In many GUI programs, this is the only import that you will need.
2. Define the **LabelDemo** class as a subclass of **EasyFrame**. The **LabelDemo** class describes the window's layout and functionality for this application.
3. Define an **__init__** method in the **LabelDemo** class. This method is automatically run when the window is created. The **__init__** method runs a method with the same name on the **EasyFrame** class and then sets up any window components to display in the window. In this case, the **addLabel** method is run on the window itself. The **addLabel** method creates a window component, a **label object** with the text "Hello world!," and adds it to the window at the grid position (0, 0).
4. The last five lines of code define a **main** function and check to see if the Python code file is being run as a program. If this is true, the **main** function is called to create an instance of the **LabelDemo** class. The **mainloop** method is then run on this object. At this point, the window pops up for viewing. Note that **mainloop**, as the name implies, enters a loop. The Python Virtual Machine runs this loop behind the scenes. Its purpose is to wait for user events, as mentioned earlier. The loop terminates when the user clicks the window's close box.

Because steps 1 and 4 typically have the same format in each program, they will be omitted from the text of many of the program examples that follow.

A Template for All GUI Programs

Writing the code to pop up a window that says "Hello world!" might seem like a lot of work. However, the good news is that the structure of a GUI program is always the same, no matter how complex the application becomes. Here is the template for this structure:

```
from breezypythongui import EasyFrame

Other imports

class ApplicationName(EasyFrame):

    The __init__ method definition

    Definitions of event handling methods

def main():
    ApplicationName().mainloop()

if __name__ == "__main__":
    main()
```

A GUI application window is always represented as a class that extends **EasyFrame**. The **__init__** method initializes the window by setting its attributes and populating it with the appropriate GUI components. In our example, Python runs this method automatically when the constructor function **LabelDemo** is called. The event handling methods provide the responses of the application to user events (not relevant in this example program). The last lines of code, beginning with the definition of the **main** function, create an instance of the application window class and run the **mainloop** method on this instance. The window then pops up and waits for user events. Pressing the window's close button will quit the program normally. If you have launched the program from an IDLE window, you can run it again after quitting by entering **main()** at the shell prompt.

The Syntax of Class and Method Definitions

Note that the syntax of class and method definitions is a bit like the syntax of function definitions. Each definition has a one-line header that begins with a keyword (**class** or **def**), followed by a body of code indented one level in the text.

A class header contains the name of the class, conventionally capitalized in Python, followed by a parenthesized list of one or more parent classes. The body of a class definition, nested one tab under the header, consists of one or more method definitions, which may appear in any order.

A method header looks very much like a function header, but a method always has at least one parameter, in the first position, named **self**. At call time, the PVM automatically assigns to this parameter a reference to the object on which the method is called; thus, you do not pass this object as an explicit argument at call time. For example, given the method header


```
def someMethod(self):
```

the method call

```
anObject.someMethod()
```

automatically assigns the object **anObject** to the **self** parameter for this method. The parameter **self** is used within class and method definitions to call other methods on the same object, or to access that object's instance variables or data, as will be explained shortly.

Subclassing and Inheritance as Abstraction Mechanisms

Our first example program defined a new class named **LabelDemo**. This class was defined as a **subclass** of the class **breezypythongui.EasyFrame**, which in turn is a subclass of the class **tkinter.Frame**. The subclass relationships among these classes are shown in the **class diagram** of Figure 8-4.

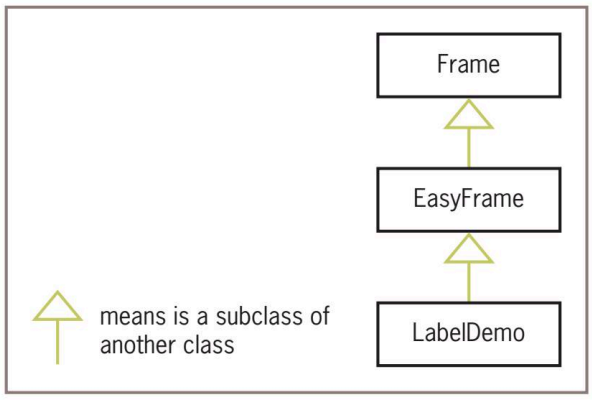

Figure 8-4 A class diagram for the label demo program

Note that the **EasyFrame** class is the **parent** of the **LabelDemo** class, and the **Frame** class is the parent of the **EasyFrame** class. This makes the **Frame** class the **ancestor** of the **LabelDemo** class. When you make a new class a subclass of another class, your new class inherits and thereby acquires the attributes and behavior defined by its parent class, and any of its ancestor classes, for free. Subclassing and inheritance are thus useful abstraction mechanisms, in that you do not have to reinvent the entire wheel when defining a new class of objects, but only customize it a bit. For example, the **EasyFrame** class customizes the **Frame** class with methods to add window components to a window; the **LabelDemo** class customizes the **EasyFrame** method **__init__** to set up a window with a specific window component.

As a rule of thumb, when you are defining a new class of objects, you should look around for a class that already supports some of the structure and behavior of such objects, and then subclass that class to provide exactly the service that you need.

Exercises

1. Describe what usually happens in the `__init__` method of a main window class.
2. Explain why it's a good idea to make a new class a subclass of an existing class.

Windows and Window Components

In this section, you will explore the details of windows and window components. In the process, you will learn how to choose appropriate classes of GUI objects, to access and modify their attributes, and to organize them to cooperate to perform the task at hand.

Windows and Their Attributes

A window has several attributes. The most important ones are its

- title (an empty string by default)
- width and height in pixels
- resizability (true by default)
- background color (white by default)

With the exception of the window's title, the attributes of our label demo program's window have the default values. The background color is white and the window is resizable. The window's initial dimensions are automatically established by shrink-wrapping the window around the label contained in it. We can override the window's default title, an empty string, by supplying another string as an optional **`title`** argument to the **`EasyFrame`** method **`__init__`**. Other options are to provide a custom initial width and height in pixels. Note that whenever we supply arguments to a method call, we use the corresponding keywords for clarity in the code. For example, you might override the dimensions and title of our first program's window as follows:

```
EasyFrame.__init__(self, width = 300, height = 200,
                   title = "Label Demo")
```

Another way to change a window's attributes is to reset them in the window's **attribute dictionary**. Each window or window component maintains a dictionary of its attributes and their values. To access or modify an attribute, the programmer uses the standard subscript notation with the attribute name as a dictionary key. For example, later in the label demo's

__init__ method, the window's background color can be set to yellow with the following statement:

```
self["background"] = "yellow"
```

Note that **self** in this case refers to the window itself.

The final way to change a window's attributes is to run a method included in the **EasyFrame** class. This class includes the four methods listed in Table 8-1.

EasyFrame Method	What It Does
setBackground(color)	Sets the window's background color to **color**.
setResizable(aBoolean)	Makes the window resizable (**True**) or not (**False**).
setSize(width, height)	Sets the window's width and height in pixels.
setTitle(title)	Sets the window's title to **title**.

Table 8-1 Methods to change a window's attributes

For example, later in the **LabelDemo** class's **__init__** method, the window's size can be permanently frozen with the following statement:

```
self.setResizable(False)
```

Window Layout

Window components are laid out in the window's two-dimensional **grid**. The grid's rows and columns are numbered from the position (0, 0) in the upper left corner of the window. A window component's row and column position in the grid is specified when the component is added to the window. For example, the next program (**layoutdemo.py**) labels the four quadrants of the window shown in Figure 8-5:

```
class LayoutDemo(EasyFrame):
    """Displays labels in the quadrants."""

    def __init__(self):
        """Sets up the window and the labels."""
        EasyFrame.__init__(self)
        self.addLabel(text = "(0, 0)", row = 0, column = 0)
        self.addLabel(text = "(0, 1)", row = 0, column = 1)
        self.addLabel(text = "(1, 0)", row = 1, column = 0)
        self.addLabel(text = "(1, 1)", row = 1, column = 1)
```

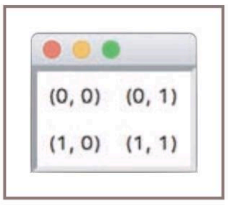

Figure 8-5 Laying out labels in the window's grid

Because the window is shrink-wrapped around the four labels, they appear to be centered in their rows and columns. However, when the user stretches this window, the labels stick to the upper left or northwest corners of their grid positions.

Each type of window component has a default alignment within its grid position. Because labels frequently appear to the left of data entry fields, their default alignment is northwest. The programmer can override the default alignment by including the **sticky** attribute as a keyword argument when the label is added to the window. The values of **sticky** are the strings "N," "S," "E," and "W," or any combination thereof. The next code segment centers the four labels in their grid positions:

```
self.addLabel(text = "(0, 0)", row = 0, column = 0,
              sticky = "NSEW")
self.addLabel(text = "(0, 1)", row = 0, column = 1,
              sticky = "NSEW")
self.addLabel(text = "(1, 0)", row = 1, column = 0,
              sticky = "NSEW")
self.addLabel(text = "(1, 1)", row = 1, column = 1,
              sticky = "NSEW")
```

Now, when the user expands the window, the labels retain their alignments in the exact center of their grid positions.

One final aspect of window layout involves the spanning of a window component across several grid positions. For example, when a window has two components in the first row and only one component in the second row, the latter component might be centered in its row, thus occupying two grid positions. The programmer can force a horizontal and/or vertical spanning of grid positions by supplying the **rowspan** and **columnspan** keyword arguments when adding a component (like merging cells in a table or spreadsheet). The spanning does not take effect unless the alignment of the component is centered along that dimension, however. The next code segment adds the three labels shown in Figure 8-6. The window's grid cells are outlined in the figure.

```
self.addLabel(text = "(0, 0)", row = 0, column = 0,
              sticky = "NSEW")
self.addLabel(text = "(0, 1)", row = 0, column = 1,
              sticky = "NSEW")
self.addLabel(text = "(1, 0 and 1)", row = 1, column = 0,
              sticky = "NSEW", columnspan = 2)
```

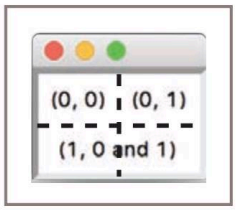

Figure 8-6 Labels with center alignment and a column span of 2

Types of Window Components and Their Attributes

GUI programs use several types of window components, or widgets as they are commonly called. These include labels, entry fields, text areas, command buttons, drop-down menus, sliding scales, scrolling list boxes, canvases, and many others. The **`breezypythongui`** module includes methods for adding each type of window component to a window. Each such method uses the form

`self.add`*`ComponentType`*`(<arguments>)`

When this method is called, **`breeypythongui`**

- Creates an instance of the requested type of window component
- Initializes the component's attributes with default values or any values provided by the programmer
- Places the component in its grid position (the row and column are required arguments)
- Returns a reference to the component

The window components supported by **`breezypythongui`** are either of the standard **`tkinter`** types, such as **`Label`**, **`Button`**, and **`Scale`**, or subclasses thereof, such as **`FloatField`**, **`TextArea`**, and **`EasyCanvas`**. A complete list is shown in Table 8-2. Parent classes are shown in parentheses.

Type of Window Component	Purpose
`Label`	Displays text or an image in the window.
`IntegerField(Entry)`	A box for input or output of integers.
`FloatField(Entry)`	A box for input or output of floating-point numbers.
`TextField(Entry)`	A box for input or output of a single line of text.
`TextArea(Text)`	A scrollable box for input or output of multiple lines of text.
`EasyListbox(Listbox)`	A scrollable box for the display and selection of a list of items.

Type of Window Component	Purpose
`Button`	A clickable command area.
`EasyCheckbutton(Checkbutton)`	A labeled checkbox.
`Radiobutton`	A labeled disc that, when selected, deselects related radio buttons.
`EasyRadiobuttonGroup(Frame)`	Organizes a set of radio buttons, allowing only one at a time to be selected.
`EasyMenuBar(Frame)`	Organizes a set of menus.
`EasyMenubutton(Menubutton)`	A menu of drop-down command options.
`EasyMenuItem`	An option in a drop-down menu.
`Scale`	A labeled slider bar for selecting a value from a range of values.
`EasyCanvas(Canvas)`	A rectangular area for drawing shapes or images.
`EasyPanel(Frame)`	A rectangular area with its own grid for organizing window components.
`EasyDialog(simpleDialog.Dialog)`	A resource for defining special-purpose popup windows.

Table 8-2 Window components in **`breezypythongui`**

As with windows, some of a window component's attributes can be set when the component is created, or can be reset by accessing its attribute dictionary at a later time.

Displaying Images

To illustrate the use of attribute options for a label component, let's examine a program (**imagedemo.py**) that displays an image with a caption. The program's window is shown in Figure 8-7.

This program adds two labels to the window. One label displays the image and the other label displays the caption. Unlike earlier examples, the program now keeps variable references to both labels for further processing.

The image label is first added to the window with an empty text string. The program then creates a **`PhotoImage`** object from an image file and sets the **`image`** attribute of the image label to this object. Note that the variable used to hold the reference to the image must be an instance variable (prefixed by **`self`**), rather than a temporary variable. The image file must be

Figure 8-7 Displaying a captioned image

in GIF format. Lastly, the program creates a **Font** object with a non-standard font and resets the text label's **font** and **foreground** attributes to obtain the caption shown in Figure 8-7. The window is shrink-wrapped around the two labels and its dimensions are fixed.

Here is the code for the program:

```
from breezypythongui import EasyFrame
from tkinter import PhotoImage
from tkinter.font import Font

class ImageDemo(EasyFrame):
    """Displays an image and a caption."""

    def __init__(self):
        """Sets up the window and the widgets."""
        EasyFrame.__init__(self, title = "Image Demo")
        self.setResizable(False);
        imageLabel = self.addLabel(text = "",
                                   row = 0, column = 0,
                                   sticky = "NSEW")
        textLabel = self.addLabel(text = "Smokey the cat",
                                  row = 1, column = 0,
                                  sticky = "NSEW")

    # Load the image and associate it with the image label.
    self.image = PhotoImage(file = "smokey.gif")
    imageLabel["image"] = self.image

    # Set the font and color of the caption.
    font = Font(family = "Verdana", size = 20,
                slant = "italic")
    textLabel["font"] = font
    textLabel["foreground"] = "blue"
```

Table 8-3 summarizes the **tkinter.Label** attributes used in this book.

Label Attribute	Type of Value
`image`	A **`PhotoImage`** object (imported from **`tkinter.font`**). Must be loaded from a GIF file.
`text`	A string.
`background`	A color. A label's background is the color of the rectangular area enclosing the text of the label.
`foreground`	A color. A label's foreground is the color of its text.
`font`	A **`Font`** object (imported from **`tkinter.font`**).

Table 8-3 The **`tkinter.Label`** attributes

You are encouraged to browse the **`breezypythongui`** documentation for information on the different types of window components and their attributes. Python also has excellent documentation on the window components at *https://docs.python.org/3/library/tkinter.html#module-tkinter*. For an overview of fonts, see *https://en.wikipedia.org/wiki/Font*. Learning which fonts are available on your system requires some geekery with **`tkinter`**. A demo program, **fontdemo.py**, that lets you view these fonts is available in the example programs for this book.

In the next section, we show how to make GUI programs interactive by responding to user events.

Exercises

1. Write a code segment that centers the labels RED, WHITE, and BLUE vertically in a GUI window. The text of each label should have the color that it names, and the window's background color should be green. The background color of each label should also be green.
2. Run the demo program **fontdemo.py** to explore the font families available on your system. Then write a code segment that centers the labels COURIER, HELVETICA, and TIMES horizontally in a GUI window. The text of each label should be the name of a font family. Substitute a different font family if necessary.
3. Write a code segment that uses a loop to create and place nine labels into a 3-by-3 grid. The text of each label should be its coordinates in the grid, starting with (0, 0) in the upper left corner. Each label should be centered in its grid cell. You should use a nested **`for`** loop in your code.
4. Jill has a plan for a window layout with two rows of widgets. The first row contains two widgets, and the second row contains four widgets. Describe how she can align the widgets so that they are evenly spaced in each row.
5. Describe the procedure for setting up the display of an image in a window.

Command Buttons and Responding to Events

A command button is added to a window just like a label, by specifying its text and position in the grid. A button is centered in its grid position by default. The method **addButton** accomplishes all this and returns an object of type **tkinter.Button**. Like a label, a button can display an image, usually a small icon, instead of a string. A button also has a **state** attribute, which can be set to "normal" to enable the button (its default state) or "disabled" to disable it.

GUI programmers often lay out a window and run the application to check its look and feel, before adding the code to respond to user events. Let's adopt this strategy for our next example. This fanciful program **(buttondemo.py)** displays a single label and two command buttons. The buttons allow the user to clear or restore the label. When the user clicks **Clear**, the label is erased, the **Clear** button is disabled, and the **Restore** button is enabled. When the user clicks **Restore**, the label is redisplayed, the **Restore** button is disabled, and the **Clear** button is enabled.

Figure 8-8 shows these two states of the window, followed by the code for the initial version of the program.

Figure 8-8 Using command buttons

```
class ButtonDemo(EasyFrame):
    """Illustrates command buttons and user events."""

    def __init__(self):
        """Sets up the window, label, and buttons."""
        EasyFrame.__init__(self)

        # A single label in the first row.
        self.label = self.addLabel(text = "Hello world!",
                                   row = 0, column = 0,
                                   columnspan = 2,
                                   sticky = "NSEW")

        # Two command buttons in the second row.
        self.clearBtn = self.addButton(text = "Clear",
                                       row = 1, column = 0)
        self.restoreBtn = self.addButton(text = "Restore",
                                         row = 1, column = 1,
                                         state = "disabled")
```

Note that the **Restore** button, which appears in gray in the window on the left, is initially disabled. When running the first version of the program, the user can click the **Clear** button, but to no effect.

To allow a program to respond to a button click, the programmer must set the button's **command** attribute. There are two ways to do this: either by supplying a keyword argument when the button is added to the window or, later, by assignment to the button's attribute dictionary. The value of the **command** attribute should be a method of no arguments, defined in the program's window class. The default value of this attribute is a method that does nothing.

The completed version of the example program supplies two methods, which are commonly called **event handlers**, for the program's two buttons. Each of these methods resets the label to the appropriate string and then enables and disables the relevant buttons.

```
class ButtonDemo(EasyFrame):
    """Illustrates command buttons and user events."""

    def __init__(self):
        """Sets up the window, label, and buttons."""
        EasyFrame.__init__(self)

        # A single label in the first row.
        self.label = self.addLabel(text = "Hello world!",
                                   row = 0, column = 0,
                                   columnspan = 2,
                                   sticky = "NSEW")

        # Two command buttons in the second row, with event
        # handler methods supplied.
        self.clearBtn = self.addButton(text = "Clear",
                                       row = 1, column = 0,
                                       command = self.clear)
        self.restoreBtn = self.addButton(text = "Restore",
                                         row = 1, column = 1,
                                         state = "disabled",
                                         command = self.restore)

    # Methods to handle user events.
    def clear(self):
        """Resets the label to the empty string and updates
        the button states."""
        self.label["text"] = ""
        self.clearBtn["state"] = "disabled"
        self.restoreBtn["state"] = "normal"

    def restore(self):
        """Resets the label to 'Hello world!' and updates
        the button states."""
        self.label["text"] = "Hello world!"
        self.clearBtn["state"] = "normal"
        self.restoreBtn["state"] = "disabled"
```

Now, when the user clicks the **Clear** button, Python automatically runs the **`clear`** method on the window. Likewise, when the programmer clicks the **Restore** button, Python automatically runs the **`restore`** method on the window.

Exercises

1. Explain what happens when a user clicks a command button in a fully functioning GUI program.
2. Why is it a good idea to write and test the code for laying out a window's components before you add the methods that perform computations in response to events?

Input and Output with Entry Fields

An entry field is a box in which the user can position the mouse cursor and enter a number or a single line of text. This section explores the use of entry fields to allow a GUI program to take input text or numbers from a user and display text or numbers as output.

Text Fields

A text field is appropriate for entering or displaying a single-line string of characters. The programmer uses the method **`addTextField`** to add a text field to a window. The method returns an object of type **`TextField`**, which is a subclass of **`tkinter.Entry`**. Required arguments to **`addTextField`** are **`text`** (the string to be initially displayed), **`row`**, and **`column`**. Optional arguments are **`rowspan`**, **`columnspan`**, **`sticky`**, **`width`**, and **`state`**.

A text field is aligned by default to the northeast of its grid cell. A text field has a default width of 20 characters. This represents the maximum number of characters viewable in the box, but the user can continue typing or viewing them by moving the cursor key to the right.

The programmer can set a text field's **`state`** attribute to "readonly" to prevent the user from editing an output field.

The **`TextField`** method **`getText`** returns the string currently contained in a text field. Thus, it serves as an input operation. The method **`setText`** outputs its string argument to a text field.

Our example program (**textfielddemo.py**) converts a string to uppercase. The user enters text into the input field, clicks the **Convert** button, and views the result in the output field. The output field is read only, to prevent editing the result. Figure 8-9 shows an interaction with the program's window, and the code follows.