Chapter 10 Defining Classes

Objectives

- To appreciate how defining new classes can provide structure for a complex program.
- To be able to read and write Python class definitions.
- To understand the concept of encapsulation and how it contributes to building modular and maintainable programs.
- To be able to write programs involving simple class definitions.
- To be able to write interactive graphics programs involving novel (programmer-designed) widgets.

10.1 Quick Review of Objects

In the last three chapters, we have developed techniques for structuring the *computations* of a program. In the next few chapters, we will take a look at techniques for structuring the *data* that our programs use. You already know that objects are one important tool for managing complex data. So far, our programs have made use of objects created from pre-defined classes such as `Circle`. In this chapter, you will learn how to write new classes of your own.

Remember back in Chapter 4 I defined an *object* as an active data type that knows stuff and can do stuff. More precisely, an object consists of

1. A collection of related information.
2. A set of operations to manipulate that information.

The information is stored inside the object in *instance variables*. The operations, called *methods*, are functions that "live" inside the object. Collectively, the instance variables and methods are called the *attributes* of an object.

To take a now-familiar example, a `Circle` object will have instance variables such as `center`, which remembers the center point of the circle, and `radius`, which stores the length of the circle's radius. The methods of the circle will need this data to perform actions. The `draw` method examines the `center` and `radius` to decide which pixels in a window should be colored. The `move` method will change the value of `center` to reflect the new position of the circle.

Recall that every object is said to be an *instance* of some *class*. The class of the object determines what attributes the object will have. Basically a class is a description of what its instances will know and do. New objects are created from a class by invoking a *constructor*. You can think of the class itself as a sort of factory for creating new instances.

Consider making a new circle object:

```
myCircle = Circle(Point(0,0), 20)
```

`Circle`, the name of the class, is used to invoke the constructor. This statement creates a new `Circle` instance and stores a reference to it in the variable `myCircle`. The parameters to the constructor are used to initialize some of the instance variables (namely `center` and `radius`) inside `myCircle`. Once the instance has been created, it is manipulated by calling on its methods:

```
myCircle.draw(win)
myCircle.move(dx, dy)
...
```

10.2 Example Program: Cannonball

Before launching into a detailed discussion of how to write your own classes, let's take a short detour to see how useful new classes can be.

10.2.1 Program Specification

Suppose we want to write a program that simulates the flight of a cannonball (or any other projectile such as a bullet, baseball, or shot put). We are particularly interested in finding out how far the cannonball will travel when fired at various launch angles and initial velocities. The input to the program will be the launch

angle (in degrees), the initial velocity (in meters per second), and the initial height (in meters) of the cannonball. The output will be the distance that the projectile travels before striking the ground (in meters).

If we ignore the effects of wind resistance and assume that the cannonball stays close to earth's surface (i.e., we're not trying to put it into orbit), this is a relatively simple classical physics problem. The acceleration of gravity near the earth's surface is about 9.8 meters per second, per second. That means if an object is thrown upward at a speed of 20 meters per second, after one second has passed, its upward speed will have slowed to $20 - 9.8 = 10.2$ meters per second. After another second, the speed will be only 0.4 meters per second, and shortly thereafter it will start coming back down.

For those who know a little bit of calculus, it's not hard to derive a formula that gives the position of our cannonball at any given moment in its flight. Rather than take the calculus approach, however, our program will use simulation to track the cannonball moment by moment. Using just a bit of simple trigonometry to get started, along with the obvious relationship that the distance an object travels in a given amount of time is equal to its rate times the amount of time ($d = rt$), we can solve this problem algorithmically.

10.2.2 Designing the Program

Let's start by designing an algorithm. Given the problem statement, it's clear that we need to consider the flight of the cannonball in two dimensions: height, so we know when it hits the ground; and distance, to keep track of how far it goes. We can think of the position of the cannonball as a point (x, y) in a 2D graph where the value of y gives the height above the ground and the value of x gives the distance from the starting point.

Our simulation will have to update the position of the cannonball to account for its flight. Suppose the ball starts at position $(0, 0)$, and we want to check its position, say, every tenth of a second. In that interval, it will have moved some distance upward (positive y) and some distance forward (positive x). The exact distance in each dimension is determined by its velocity in that direction.

Separating out the x and y components of the velocity makes the problem easier. Since we are ignoring wind resistance, the x velocity remains constant for the entire flight. However, the y velocity changes over time due to the influence of gravity. In fact, the y velocity will start out being positive and then become negative as the cannonball starts back down.

Given this analysis, it's pretty clear what our simulation will have to do. Here

is a rough outline:

```
input the simulation parameters: angle, velocity, height, interval
calculate the initial position of the cannonball: xpos, ypos
calculate the initial velocities of the cannonball: xvel, yvel
while the cannonball is still flying:
    update the values of xpos, ypos, and yvel for interval seconds
        further into the flight
output the distance traveled as xpos
```

Let's turn this into a program using stepwise refinement.

The first line of the algorithm is straightforward. We just need an appropriate sequence of input statements. Here's a start:

```
def main():
    angle = float(input("Enter the launch angle (in degrees): "))
    vel = float(input("Enter the initial velocity (in meters/sec): "))
    h0 = float(input("Enter the initial height (in meters): "))
    time = float(input(
           "Enter the time interval between position calculations: "))
```

Calculating the initial position for the cannonball is also easy. It will start at distance 0 and height `h0`. We just need a couple of assignment statements:

```
xpos = 0.0
ypos = h0
```

Next we need to calculate the x and y components of the initial velocity. We'll need a little high-school trigonometry. (See, they told you you'd use that some day.) If we consider the initial velocity as consisting of some amount of change in y and some amount of change in x, then these three components (velocity, x velocity and y velocity) form a right triangle. Figure 10.1 illustrates the situation. If we know the magnitude of the velocity and the launch angle (labeled *theta*, because the Greek letter θ is often used as the measure of angles), we can easily calculate the magnitude of $xvel$ by the equation $xvel = velocity \cos theta$. A similar formula (using $\sin theta$) provides $yvel$.

Even if you don't completely understand the trigonometry, the important thing is that we can translate these formulas into Python code. There's still one subtle issue to consider. Our input angle is in degrees, and the Python `math` library uses radian measures. We'll have to convert our angle before applying the formulas. There are 2π radians in a circle (360 degrees), so $theta = \frac{\pi * angle}{180}$.

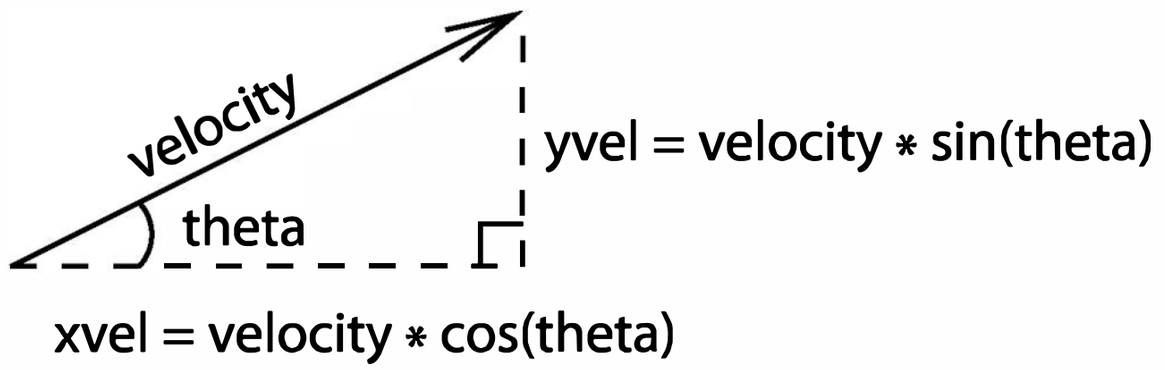

Figure 10.1: Finding the x and y components of velocity

This is such a common conversion that the math library provides a convenient function called `radians` that does this computation. These three formulas give us the code for computing the initial velocities:

```
theta = math.radians(angle)
xvel = velocity * math.cos(theta)
yvel = velocity * math.sin(theta)
```

That brings us to the main loop in our program. We want to keep updating the position and velocity of the cannonball until it reaches the ground. We can do this by examining the value of `ypos`:

```
while ypos >= 0.0:
```

I used >= as the relationship so that we can start with the cannonball on the ground (= 0) and still get the loop going. The loop will quit as soon as the value of `ypos` dips just below 0, indicating the cannonball has embedded itself slightly in the ground.

Now we arrive at the crux of the simulation. Each time we go through the loop, we want to update the state of the cannonball to move it `time` seconds farther in its flight. Let's start by considering movement in the horizontal direction. Since our specification says that we can ignore wind resistance, the horizontal speed of the cannonball will remain constant and is given by the value of `xvel`.

As a concrete example, suppose the ball is traveling at 30 meters per second and is currently 50 meters from the firing point. In another second, it will go 30 more meters and be 80 meters from the firing point. If the interval is only 0.1 second (rather than a full second), then the cannonball will only fly another $0.1(30) = 3$ meters and be at a distance of 53 meters. You can see that the distance traveled is always given by `time * xvel`. To update the horizontal position, we need just one statement:

```
xpos = xpos + time * xvel
```

The situation for the vertical component is slightly more complicated, since gravity causes the y velocity to change over time. Each second, `yvel` must decrease by 9.8 meters per second, the acceleration of gravity. In 0.1 seconds the velocity will decrease by $0.1(9.8) = 0.98$ meters per second. The new velocity at the *end* of the interval is calculated as

```
yvel1 = yvel - time * 9.8
```

To calculate how far the cannonball travels during this interval, we need to know its *average* vertical velocity. Since the acceleration due to gravity is constant, the average velocity will just be the average of the starting and ending velocities: `(yvel+yvel1)/2.0`. Multiplying this average velocity by the amount of time in the interval gives us the change in height.

Here is the completed loop:

```
while ypos >= 0.0:
    xpos = xpos + time * xvel
    yvel1 = yvel - time * 9.8
    ypos = ypos + time * (yvel + yvel1)/2.0
    yvel = yvel1
```

Notice how the velocity at the end of the time interval is first stored in the temporary variable `yvel1`. This is done to preserve the initial `yvel` so that the average velocity can be computed from the two values. Finally, the value of `yvel` is assigned its new value at the end of the loop. This represents the correct vertical velocity of the cannonball at the end of the interval.

The last step of our program simply outputs the distance traveled. Adding this step gives us the complete program:

```
# cball1.py
from math import sin, cos, radians

def main():
    angle = float(input("Enter the launch angle (in degrees): "))
    vel = float(input("Enter the initial velocity (in meters/sec): "))
    h0 = float(input("Enter the initial height (in meters): "))
    time = float(input(
           "Enter the time interval between position calculations: "))
```

```
# convert angle to radians
theta = radians(angle)

# set the initial position and velocities in x and y directions
xpos = 0
ypos = h0
xvel = vel * cos(theta)
yvel = vel * sin(theta)

# loop until the ball hits the ground
while ypos >= 0.0:
    # calculate position and velocity in time seconds
    xpos = xpos + time * xvel
    yvel1 = yvel - time * 9.8
    ypos = ypos + time * (yvel + yvel1)/2.0
    yvel = yvel1

print("\nDistance traveled: {0:0.1f} meters.".format(xpos))
```

10.2.3 Modularizing the Program

You may have noticed during the design discussion that I employed stepwise refinement (top-down design) to develop the program, but I did not divide the program into separate functions. We are going to modularize the program in two different ways. First, we'll use functions (à la top-down design).

While the final program is not too long, it is fairly complex for its length. One cause of the complexity is that it uses ten variables, and that is a lot for the reader to keep track of. Let's try dividing the program into functional pieces to see if that helps. Here's a version of the main algorithm using helper functions:

```
def main():
    angle, vel, h0, time = getInputs()
    xpos, ypos = 0, h0
    xvel, yvel = getXYComponents(vel, angle)
    while ypos >= 0:
        xpos, ypos, yvel = updateCannonBall(time, xpos, ypos, xvel, yvel)
    print("\nDistance traveled: {0:0.1f} meters.".format(xpos))
```

It should be obvious what each of these functions does based on their names and the original program code. You might take a couple of minutes to code the three helper functions.

This second version of the main algorithm is certainly more concise. The number of variables has been reduced to eight, since `theta` and `yvel1` have been eliminated from the main algorithm. Do you see where they went? The value of `theta` is only needed locally inside of `getXYComponents`. Similarly, `yvel1` is now local to `updateCannonBall`. Being able to hide some of the intermediate variables is a major benefit of the separation of concerns provided by top-down design.

Even this version seems overly complicated. Look especially at the loop. Keeping track of the state of the cannonball requires four pieces of information, three of which must change from moment to moment. All four variables along with the value of `time` are needed to compute the new values of the three that change. That results in an ugly function call having five parameters and three return values. An explosion of parameters is often an indication that there might be a better way to organize a program. Let's try another approach.

The original problem specification itself suggests a better way to look at the variables in our program. There is a single real-world cannonball object, but describing it in the current program requires four pieces of information: `xpos`, `ypos`, `xvel`, and `yvel`. Suppose we had a `Projectile` class that "understood" the physics of objects like cannonballs. Using such a class, we could express the main algorithm in terms of creating and updating a suitable object using a single variable. With this *object-based* approach, we might write `main` like this:

```
def main():
    angle, vel, h0, time = getInputs()
    cball = Projectile(angle, vel, h0)
    while cball.getY() >= 0:
        cball.update(time)
    print("\nDistance traveled: {0:0.1f} meters.".format(cball.getX()))
```

Obviously, this is a much simpler and more direct expression of the algorithm. The initial values of `angle`, `vel`, and `h0` are used as parameters to create a `Projectile` called `cball`. Each time through the loop, `cball` is asked to update its state to account for `time`. We can get the position of `cball` at any moment by using its `getX` and `getY` methods. To make this work, we just need to define a suitable `Projectile` class that implements the methods `update`, `getX`, and `getY`.

10.3 Defining New Classes

Before designing a `Projectile` class, let's take an even simpler example to examine the basic ideas.

10.3.1 Example: Multi-sided Dice

You know that a normal die (the singular of dice) is a cube, and each face shows a number from one to six. Some games employ nonstandard dice that may have fewer (e.g., four) or more (e.g., thirteen) sides. Let's design a general class `MSDie` to model multi-sided dice. We could use such an object in any number of simulation or game programs.

Each `MSDie` object will know two things:

1. How many sides it has.

2. Its current value.

When a new `MSDie` is created, we specify how many sides it will have, n. We can then operate on the die through three provided methods: `roll`, to set the die to a random value between 1 and n, inclusive; `setValue`, to set the die to a specific value (i.e., cheat); and `getValue`, to see what the current value is.

Here is an interactive example showing what our class will do:

```
>>> die1 = MSDie(6)
>>> die1.getValue()
1
>>> die1.roll()
>>> die1.getValue()
4
>>> die2 = MSDie(13)
>>> die2.getValue()
1
>>> die2.roll()
>>> die2.getValue()
12
>>> die2.setValue(8)
>>> die2.getValue()
8
```

Do you see how this might be useful? I can define any number of dice having arbitrary numbers of sides. Each die can be rolled independently and will always produce a random value in the proper range determined by the number of sides.

Using our object-oriented terminology, we create a die by invoking the `MSDie` constructor and providing the number of sides as a parameter. Our die object will keep track of this number internally using an instance variable. Another instance variable will be used to store the current value of the die. Initially, the value of the die will be set to be 1, since that is a legal value for any die. The value can be changed by the `roll` and `setValue` methods and returned from the `getValue` method.

Writing a definition for the `MSDie` class is really quite simple. A class is a collection of methods, and methods are just functions. Here is the class definition for `MSDie`:

```
# msdie.py
#    Class definition for an n-sided die.

from random import randrange

class MSDie:

    def __init__(self, sides):
        self.sides = sides
        self.value = 1

    def roll(self):
        self.value = randrange(1,self.sides+1)

    def getValue(self):
        return self.value

    def setValue(self, value):
        self.value = value
```

As you can see, a class definition has a simple form:

```
class <class-name>:
    <method-definitions>
```

Each method definition looks like a normal function definition. Placing the

function inside a class makes it a method of that class, rather than a stand-alone function.

Let's take a look at the three methods defined in this class. You'll notice that each method has a first parameter named `self`. The first parameter of a method is special—it always contains a reference to the object on which the method is acting. As usual, you can use any name you want for this parameter, but the traditional name is `self`, so that is what I will always use.

An example might be helpful in making sense of `self`. Suppose we have a `main` function that executes `die1.setValue(8)`. A method invocation is a function call. Just as in normal function calls, Python executes a four-step sequence:

1. The calling program (`main`) suspends at the point of the method application. Python locates the appropriate method definition inside the class of the object to which the method is being applied. In this case, control is transferring to the `setValue` method in the `MSDie` class, since `die1` is an instance of `MSDie`.

2. The formal parameters of the method get assigned the values supplied by the actual parameters of the call. In the case of a method call, the first formal parameter corresponds to the object. In our example, it is as if the following assignments are done before executing the method body:

   ```
   self = die1
   value = 8
   ```

3. The body of the method is executed.

4. Control returns to the point just after where the method was called, in this case, the statement immediately following `die1.setValue(8)`.

Figure 10.2 illustrates the method-calling sequence for this example. Notice how the method is called with one parameter (the value), but the method definition has two parameters, due to `self`. Generally speaking, we would say `setValue` requires one parameter. The `self` parameter in the definition is a bookkeeping detail. Some languages do this implicitly; Python requires us to add the extra parameter. To avoid confusion, I will always refer to the first formal parameter of a method as the *self* parameter and any others as *normal* parameters. So I would say `setValue` uses one normal parameter.

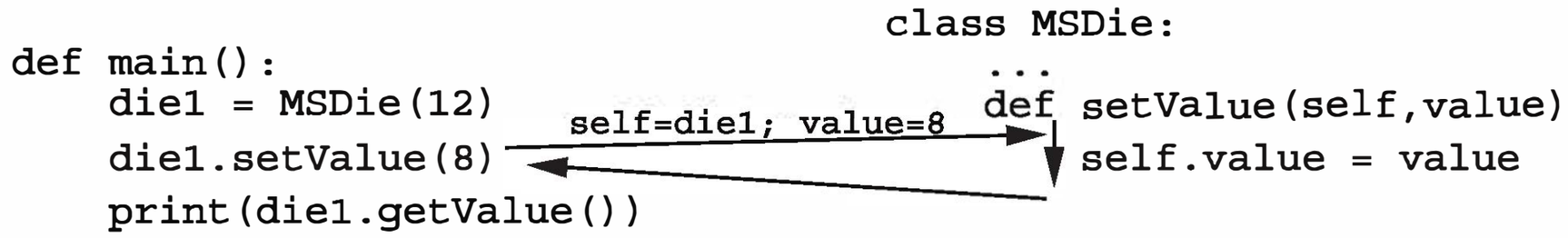

Figure 10.2: Flow of control in call: `die1.setValue(8)`

OK, so `self` is a parameter that represents an object. But what exactly can we do with it? The main thing to remember is that objects contain their own data. Conceptually, instance variables provide a way to remember data inside an object. Just as with regular variables, instance variables are accessed by name. We can use our familiar dot notation: <object>.<instance-var>. Look at the definition of `setValue`; `self.value` refers to the instance variable `value` that is associated with the object. Each instance of a class has its own instance variables, so each `MSDie` object has its very own `value`.

Certain methods in a class have special meaning to Python. These methods have names that begin and end with two underscores. The special method `__init__` is the object constructor. Python calls this method to initialize a new `MSDie`. The role of `__init__` is to provide initial values for the instance variables of an object. From outside the class, the constructor is called by the class name.

```
die1 = MSDie(6)
```

This statement causes Python to create a new `MSDie` and execute `__init__` on that object. The net result is that `die1.sides` is 6 and `die1.value` is 1.

The power of instance variables is that we can use them to remember the state of a particular object, and this information then gets passed around the program as part of the object. The values of instance variables can be referred to again in other methods or even in successive calls to the same method. This is different from regular local function variables, whose values disappear once the function terminates.

Here is a simple illustration:

```
>>> die1 = Die(13)
>>> print(die1.getValue())
1
>>> die1.setValue(8)
>>> print(die1.getValue())
8
```

The call to the constructor sets the instance variable `die1.value` to 1. The next line prints out this value. The value set by the constructor persists as part of the object, even though the constructor is over and done with. Similarly, executing `die1.setValue(8)` changes the object by setting its value to 8. When the object is asked for its value the next time, it responds with 8.

That's just about all there is to know about defining new classes in Python. Now it's time to put this new knowledge to use.

10.3.2 Example: The Projectile Class

Returning to the cannonball example, we want a class that can represent projectiles. This class will need a constructor to initialize instance variables, an `update` method to change the state of the projectile, and `getX` and `getY` methods so that we can find the current position.

Let's start with the constructor. In the main program, we will create a cannonball from the initial angle, velocity and height:

```
cball = Projectile(angle, vel, h0)
```

The `Projectile` class must have an `__init__` method that uses these values to initialize the instance variables of `cball`. But what should the instance variables be? Of course, they will be the four pieces of information that characterize the flight of the cannonball: `xpos`, `ypos`, `xvel`, and `yvel`. We will calculate these values using the same formulas that were in the original program.

Here is how our class looks with the constructor:

```
class Projectile:

    def __init__(self, angle, velocity, height):
        self.xpos = 0.0
        self.ypos = height
        theta = math.radians(angle)
        self.xvel = velocity * math.cos(theta)
        self.yvel = velocity * math.sin(theta)
```

Notice how we have created four instance variables inside the object using the `self` dot notation. The value of `theta` is not needed after `__init__` terminates, so it is just a normal (local) function variable.

The methods for accessing the position of our projectiles are straightforward; the current position is given by the instance variables `xpos` and `ypos`. We just need a couple of methods that return these values.

```
    def getX(self):
        return self.xpos

    def getY(self):
        return self.ypos
```

Finally, we come to the `update` method. This method takes a single normal parameter that represents an interval of time. We need to update the state of the projectile to account for the passage of that much time. Here's the code:

```
    def update(self, time):
        self.xpos = self.xpos + time * self.xvel
        yvel1 = self.yvel - time * 9.8
        self.ypos = self.ypos + time * (self.yvel + yvel1)/2.0
        self.yvel = yvel1
```

Basically, this is the same code that we used in the original program updated to use and modify instance variables. Notice the use of `yvel1` as a temporary (ordinary) variable. This new value is saved by storing it into the object in the last line of the method.

That completes our projectile class. We now have a complete object-based solution to the cannonball problem:

```
# cball3.py
from math import sin, cos, radians

class Projectile:

    def __init__(self, angle, velocity, height):
        self.xpos = 0.0
        self.ypos = height
        theta = radians(angle)
        self.xvel = velocity * cos(theta)
        self.yvel = velocity * sin(theta)

    def update(self, time):
        self.xpos = self.xpos + time * self.xvel
        yvel1 = self.yvel - 9.8 * time
        self.ypos = self.ypos + time * (self.yvel + yvel1) / 2.0
        self.yvel = yvel1
```

```
    def getY(self):
        return self.ypos

    def getX(self):
        return self.xpos

def getInputs():
    a = float(input("Enter the launch angle (in degrees): "))
    v = float(input("Enter the initial velocity (in meters/sec): "))
    h = float(input("Enter the initial height (in meters): "))
    t = float(input(
            "Enter the time interval between position calculations: "))
    return a,v,h,t

def main():
    angle, vel, h0, time = getInputs()
    cball = Projectile(angle, vel, h0)
    while cball.getY() >= 0:
        cball.update(time)
    print("\nDistance traveled: {0:0.1f} meters.".format(cball.getX()))
```

10.4 Data Processing with Class

The projectile example shows how useful a class can be for modeling a real-world object that has complex behavior. Another common use for objects is simply to group together a set of information that describes a person or thing. For example, a company needs to keep track of information about all of its employees. Their personnel system might make use of an `Employee` object that contains data such as the employee's name, Social Security number, address, salary, department, etc. A grouping of information of this sort is often called a *record*.

Let's try our hand at some simple data processing involving university students. In a typical university, courses are measured in terms of credit hours, and grade point averages are calculated on a 4-point scale where an "A" is 4 points, a "B" is 3 points, etc. Grade point averages are generally computed using quality points. If a class is worth 3 credit hours and the student gets an "A," then he or she earns $3(4) = 12$ quality points. To calculate a student's grade

point average (GPA), we divide the total quality points by the number of credit hours completed.

Suppose we have a data file that contains student grade information. Each line of the file consists of a student's name, credit hours, and quality points. These three values are separated by a tab character. For example, the contents of the file might look something like this:

```
Adams, Henry    127    228
Computewell, Susan      100     400
DibbleBit, Denny        18      41.5
Jones, Jim      48.5    155
Smith, Frank    37      125.33
```

Our job is to write a program that reads through this file to find the student with the best GPA and print out his/her name, credits hours, and GPA. We can begin by creating a `Student` class. An object of type `Student` will be a record of information for a single student. In this case, we have three pieces of information: name, credit hours, and quality points. We can save this information as instance variables that are initialized in the constructor:

```
class Student:
    def __init__(self, name, hours, qpoints):
        self.name = name
        self.hours = float(hours)
        self.qpoints = float(qpoints)
```

Notice that I have used parameter names that match the instance variable names. This looks a bit strange at first, but it is a very common style for this sort of class. I have also `floated` the values of `hours` and `qpoints`. This makes the constructor a bit more versatile by allowing it to accept parameters that may be floats, ints, or even strings.

Now that we have a constructor, it's easy to create student records. For example, we can make a record for Henry Adams like this:

```
aStudent = Student("Adams, Henry", 127, 228)
```

Using objects allows us to collect all of the information about an individual in a single variable.

Next we must decide what methods a student object should have. Obviously, we would like to be able to access the student's information, so we should define a set of accessor methods.

```
def getName(self):
    return self.name

def getHours(self):
    return self.hours

def getQPoints(self):
    return self.qpoints
```

These methods allow us to get information back out of a student record. For example, to print a student's name we could write:

```
print(aStudent.getName())
```

One method that we have not yet included in our class is a way of computing GPA. We *could* compute it separately using the `getHours` and `getQPoints` methods, but GPA is so handy that it probably warrants its own method.

```
def gpa(self):
    return self.qpoints/self.hours
```

With this class in hand, we are ready to attack the problem of finding the best student. Our algorithm will be similar to the one used for finding the max of n numbers. We'll look through the file of students one by one, keeping track of the best student seen so far. Here's the algorithm for our program:

```
Get the file name from the user
Open the file for reading
Set best to be the first student
For each student s in the file
    if s.gpa() > best.gpa()
        set best to s
print out information about best
```

The completed program looks like this:

```
# gpa.py
#    Program to find student with highest GPA

class Student:
```

```
    def __init__(self, name, hours, qpoints):
        self.name = name
        self.hours = float(hours)
        self.qpoints = float(qpoints)

    def getName(self):
        return self.name

    def getHours(self):
        return self.hours

    def getQPoints(self):
        return self.qpoints

    def gpa(self):
        return self.qpoints/self.hours

def makeStudent(infoStr):
    # infoStr is a tab-separated line: name hours qpoints
    # returns a corresponding Student object
    name, hours, qpoints = infoStr.split("\t")
    return Student(name, hours, qpoints)

def main():
    # open the input file for reading
    filename = input("Enter the name of the grade file: ")
    infile = open(filename, 'r')

    # set best to the record for the first student in the file
    best = makeStudent(infile.readline())

    # process subsequent lines of the file
    for line in infile:
        # turn the line into a student record
        s = makeStudent(line)
        # if this student is best so far, remember it.
        if s.gpa() > best.gpa():
            best = s
```

```
    infile.close()

    # print information about the best student
    print("The best student is:", best.getName())
    print("hours:", best.getHours())
    print("GPA:", best.gpa())

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

You will notice that I added a helper function called `makeStudent`. This function takes a single line of the file, splits it into its three tab-separated fields, and returns a corresponding `Student` object. Right before the loop, this function is used to create a record for the first student in the file:

```
best = makeStudent(infile.readline())
```

It is called again inside the loop to process each subsequent line of the file:

```
s = makeStudent(line)
```

Here's how it looks running the program on the sample data:

```
Enter name the grade file: students.dat
The best student is: Computewell, Susan
hours: 100.0
GPA: 4.0
```

One unresolved issue with this program is that it only reports back a single student. If multiple students are tied for the best GPA, only the first one found is reported. I leave it as an interesting design issue for you to modify the program so that it reports all students having the highest GPA.

10.5 Objects and Encapsulation

10.5.1 Encapsulating Useful Abstractions

Hopefully, you are seeing how defining new classes like `Projectile` and `Student` can be a good way to modularize a program. Once we identify some useful

objects, we can write an algorithm using those objects and push the implementation details into a suitable class definition. This gives us the same kind of separation of concerns that we had using functions in top-down design. The main program only has to worry about what objects can do, not about how they are implemented.

Computer scientists call this separation of concerns *encapsulation*. The implementation details of an object are encapsulated in the class definition, which insulates the rest of the program from having to deal with them. This is another application of abstraction (ignoring irrelevant details), which is the essence of good design.

For completeness, I should mention that encapsulation is only a programming convention in Python. It is not enforced by the language, per se. In our `Projectile` class we included two short methods, `getX` and `getY`, that simply returned the values of instance variables `xpos` and `ypos`, respectively. Our `Student` class has similar accessor methods for its instance variables. Strictly speaking, these methods are not absolutely necessary. In Python, you can access the instance variables of any object with the regular dot notation. For example, we could test the constructor for the `Projectile` class interactively by creating an object and then directly inspecting the values of the instance variables:

```
>>> c = Projectile(60, 50, 20)
>>> c.xpos
0.0
>>> c.ypos
20
>>> c.xvel
25.0
>>> c.yvel
43.301270
```

Accessing the instance variables of an object is very handy for testing purposes, but it is generally considered poor practice to do this in programs. One of the main reasons for using objects is to hide the internal complexities of those objects from the programs that use them. References to instance variables should generally remain inside the class definition with the rest of the implementation details. From outside the class, all interaction with an object should generally be done using the interface provided by its methods. However, this is not a hard-and-fast rule, and Python program designers often specify that

certain instance variables are accessible as part of the interface. [1]

One immediate advantage of encapsulation is that it allows us to modify and improve classes independently, without worrying about "breaking" other parts of the program. As long as the interface provided by a class stays the same, the rest of the program can't even tell that a class has changed. As you begin to design classes of your own, you should strive to provide each with a complete set of methods to make it useful.

10.5.2 Putting Classes in Modules

Often a well-defined class or set of classes provides useful abstractions that can be leveraged in many different programs. For example, we might want to turn our projectile class into its own module file so that it can be used in other programs. In doing so, it would be a good idea to add documentation that describes how the class can be used so that programmers who want to use the module don't have to study the code to figure out (or remember) what the class and its methods do.

10.5.3 Module Documentation

You are already familiar with one way of documenting programs, namely comments. It's always a good idea to provide comments explaining the contents of a module and its uses. In fact, comments of this sort are so important that Python incorporates a special kind of commenting convention called a *docstring*. You can insert a plain string literal as the first line of a module, class, or function to document that component. The advantage of docstrings is that, while ordinary comments are simply ignored by Python, docstrings are actually carried along during execution in a special attribute called `__doc__`. These strings can be examined dynamically.

Most of the Python library modules have extensive docstrings that you can use to get help on using the module or its contents. For example, if you can't remember how to use the `random` function, you can print its docstring directly like this:

```
>>> import random
>>> print(random.random.__doc__)
```

[1] In fact, Python provides an interesting mechanism called "properties" that makes providing access to instance variables safe and elegant. You can consult the Python documentation for the details.

```
random() -> x in the interval [0, 1).
```

Docstrings are also used by the Python online help system and by a utility called pydoc that automatically builds documentation for Python modules. You could get the same information using interactive help like this:

```
>>> import random
>>> help(random.random)
Help on built-in function random:

random(...)
    random() -> x in the interval [0, 1).
```

If you want to see a whole bunch of information about the entire random module, try typing `help(random)`.

Here is a version of our `Projectile` class as a module file with docstrings included:

```
# projectile.py

"""projectile.py
Provides a simple class for modeling the
flight of projectiles."""

from math import sin, cos, radians

class Projectile:

    """Simulates the flight of simple projectiles near the earth's
    surface, ignoring wind resistance. Tracking is done in two
    dimensions, height (y) and distance (x)."""

    def __init__(self, angle, velocity, height):
        """Create a projectile with given launch angle, initial
        velocity and height."""
        self.xpos = 0.0
        self.ypos = height
        theta = radians(angle)
        self.xvel = velocity * cos(theta)
        self.yvel = velocity * sin(theta)
```

```
def update(self, time):
    """Update the state of this projectile to move it time seconds
    farther into its flight"""
    self.xpos = self.xpos + time * self.xvel
    yvel1 = self.yvel - 9.8 * time
    self.ypos = self.ypos + time * (self.yvel + yvel1) / 2.0
    self.yvel = yvel1

def getY(self):
    "Returns the y position (height) of this projectile."
    return self.ypos

def getX(self):
    "Returns the x position (distance) of this projectile."
    return self.xpos
```

You might notice that many of the docstrings in this code are enclosed in triple quotes (`"""`). This is a third way that Python allows string literals to be delimited. Triple quoting allows us to directly type multi-line strings. Here is an example of how the docstrings appear when they are printed:

```
>>> print(projectile.Projectile.__doc__)
Simulates the flight of simple projectiles near the earth's
    surface, ignoring wind resistance. Tracking is done in two
    dimensions, height (y) and distance (x).
```

You might try `help(projectile)` to see how the complete documentation looks for this module.

10.5.4 Working with Multiple Modules

Our main program can now simply import from the `projectile` module in order to solve the original problem:

```
# cball4.py
from projectile import Projectile

def getInputs():
    a = float(input("Enter the launch angle (in degrees): "))
```

```
    v = float(input("Enter the initial velocity (in meters/sec): "))
    h = float(input("Enter the initial height (in meters): "))
    t = float(input("Enter the time interval between position calculations:
    return a,v,h,t

def main():
    angle, vel, h0, time = getInputs()
    cball = Projectile(angle, vel, h0)
    while cball.getY() >= 0:
        cball.update(time)
    print("\nDistance traveled: {0:0.1f} meters.".format(cball.getX()))
```

In this version, details of projectile motion are now hidden in the `projectile` module file.

If you are testing multi-module Python projects interactively (a good thing to do), you need to be aware of a subtlety in the Python module-importing mechanism. When Python first imports a given module, it creates a module object that contains all of the things defined in the module (technically, this is called a *namespace*). If a module imports successfully (it has no syntax errors), subsequent imports do not reload the module; they just create additional references to the existing module object. Even if a module has been changed (its source file edited), re-`import`ing it into an ongoing interactive session will not get you an updated version.

It *is* possible to interactively replace a module object using the function `reload(<module>)` in the `imp` module of the standard library (consult the Python documentation for details). But often this won't give you the results you want. That's because `reload`ing a module doesn't change the values of any identifiers in the current session that already refer to objects from the old version of the module. In fact, it's pretty easy to create a situation where objects from both the old and new version of a module are active at the same time, which is confusing to say the least.

The simplest way to avoid this confusion is to make sure you start a new interactive session for testing each time any of the modules involved in your tests is modified. That way you are guaranteed to get a fresh (updated) import of all the modules that you are using. If you are using IDLE, you will notice that it takes care of this for you by doing a shell restart when you select "run module."