Chapter 9 Simulation and Design

Objectives

- To understand the potential applications of simulation as a way to solve real-world problems.
- To understand pseudo-random numbers and their application in Monte Carlo simulations.
- To understand and be able to apply top-down and spiral design techniques in writing complex programs.
- To understand unit testing and be able to apply this technique in the implementation and debugging of complex programs.

9.1 Simulating Racquetball

You may not realize it, but you have reached a significant milestone in the journey to becoming a computer scientist. You now have all the tools to write programs that solve interesting problems. By interesting, I mean problems that would be difficult or impossible to solve without the ability to write and implement computer algorithms. You are probably not yet ready to write the next great killer application, but you can do some nontrivial computing.

One particularly powerful technique for solving real-world problems is *simulation*. Computers can model real-world processes to provide otherwise unobtainable information. Computer simulation is used every day to perform myriad tasks such as predicting the weather, designing aircraft, creating special effects for movies, and entertaining video game players, to name just a few. Most of these applications require extremely complex programs, but even relatively modest simulations can sometimes shed light on knotty problems.

In this chapter we are going to develop a simple simulation of the game of racquetball. Along the way, you will learn some important design and implementation strategies that will help you in tackling your own problems.

9.1.1 A Simulation Problem

Susan Computewell's friend, Denny Dibblebit, plays racquetball. Over years of playing, he has noticed a strange quirk in the game. He often competes with players who are just a little bit better than he is. In the process, he always seems to get thumped, losing the vast majority of matches. This has led him to question what is going on. On the surface, one would think that players who are *slightly* better should win *slightly* more often, but against Denny, they seem to win the lion's share.

One obvious possibility is that Denny Dibblebit's problem is in his head. Maybe his mental game isn't up to par with his physical skills. Or perhaps the other players are really *much* better than he is, and he just refuses to see it.

One day, Denny was discussing racquetball with Susan, when she suggested another possibility. Maybe it is the nature of the game itself that small differences in ability lead to lopsided matches on the court. Denny was intrigued by the idea; he didn't want to waste money on an expensive sports psychologist if it wasn't going to help. But how could he figure out if the problem was mental or just part of the game?

Susan suggested she could write a computer program to simulate certain aspects of racquetball. Using the simulation, they could let the computer model thousands of games between players of differing skill levels. Since there would not be any mental aspects involved, the simulation would show whether Denny is losing more than his share of matches.

Let's write our own racquetball simulation and see what Susan and Denny discovered.

9.1.2 Analysis and Specification

Racquetball is a sport played between two players using racquets to strike a ball in a four-walled court. It has aspects similar to many other ball and racquet games such as tennis, volleyball, badminton, squash and table tennis. We don't need to understand all the rules of racquetball to write the program, just the basic outline of the game.

To start the game, one of the players puts the ball into play—this is called *serving*. The players then alternate hitting the ball to keep it in play; this is a *rally*. The rally ends when one of the players fails to hit a legal shot. The player who misses the shot loses the rally. If the loser is the player who served, then service passes to the other player. If the server wins the rally, a point is awarded. Players can only score points during their own service. The first player to reach 15 points wins the game.

In our simulation, the ability level of the players will be represented by the probability that the player wins the rally when he or she serves. Thus, players with a 0.6 probability win a point on 60% of their serves. The program will prompt the user to enter the service probability for both players and then simulate multiple games of racquetball using those probabilities. The program will then print a summary of the results.

Here is a detailed specification:

Input The program first prompts for and gets the service probabilities of the two players (called "player A" and "player B"). Then the program prompts for and gets the number of games to be simulated.

Output The program will provide a series of initial prompts such as the following:

```
What is the prob. player A wins a serve?
What is the prob. player B wins a serve?
How many games to simulate?
```

The program will print out a nicely formatted report showing the number of games simulated and the number of wins and winning percentage for each player. Here is an example:

```
Games Simulated: 500
Wins for A: 268 (53.6%)
Wins for B: 232 (46.4%)
```

***Notes*:** All inputs are assumed to be legal numeric values; no error or validity checking is required.

In each simulated game, player A serves first.

9.2 Pseudo-random Numbers

Our simulation program will have to deal with uncertain events. When we say that a player wins 50% of the serves, that does not mean that every other serve is a winner. It's more like a coin toss. Overall, we expect that half the time the coin will come up heads and half the time it will come up tails, but there is nothing to prevent a run of five tails in a row. Similarly, our racquetball player should win or lose rallies randomly. The service probability provides a likelihood that a given serve will be won, but there is no set pattern.

Many simulations share this property of requiring events to occur with a certain likelihood. A driving simulation must model the unpredictability of other drivers; a bank simulation has to deal with the random arrival of customers. These sorts of simulations are sometimes called *Monte Carlo* algorithms because the results depend on "chance" probabilities.[1] Of course, you know that there is nothing random about computers; they are instruction-following machines. How can computer programs model seemingly random happenings?

Simulating randomness is a well-studied problem in computer science. Remember the chaos program from Chapter 1? The numbers produced by that program seemed to jump around randomly between zero and one. This apparent randomness came from repeatedly applying a function to generate a sequence of numbers. A similar approach can be used to generate random (actually *pseudo-random*) numbers.

A pseudo-random number generator works by starting with some *seed* value. This value is fed to a function to produce a "random" number. The next time a random number is needed, the current value is fed back into the function to produce a new number. With a carefully chosen function, the resulting sequence of values looks essentially random. Of course, if you start the process over again with the same seed value, you end up with exactly the same sequence of numbers. It's all determined by the generating function and the value of the seed.

Python provides a library module that contains a number of useful functions for generating pseudo-random numbers. The functions in this module derive an initial seed value from the date and time when the module is loaded, so you get a different seed value each time the program is run. This means that you will also get a unique sequence of pseudo-random values. The two functions of greatest interest to us are `randrange` and `random`.

[1]So probabilistic simulations written in Python *could* be called Monte Python programs (nudge, nudge; wink,wink).

The randrange function is used to select a pseudo-random int from a given range. It can be used with one, two, or three parameters to specify a range exactly as with the range function. For example, randrange(1,6) returns some number from the range [1,2,3,4,5], and randrange(5,105,5) returns a multiple of 5 between 5 and 100, inclusive. (Remember, ranges go up to, but do not include, the stopping value.)

Each call to randrange generates a new pseudo-random int. Here is an interactive session that shows randrange in action:

```
>>> from random import randrange
>>> randrange(1,6)
3
>>> randrange(1,6)
3
>>> randrange(1,6)
5
>>> randrange(1,6)
5
>>> randrange(1,6)
5
>>> randrange(1,6)
1
>>> randrange(1,6)
5
>>> randrange(1,6)
4
>>> randrange(1,6)
2
```

Notice it took nine calls to randrange to eventually generate every number in the range 1–5. The value 5 came up almost half of the time. This shows the probabilistic nature of random numbers. Over the long haul, this function produces a uniform distribution, which means that all values will appear an (approximately) equal number of times.

The random function can be used to generate pseudo-random floating-point values. It takes no parameters and returns values uniformly distributed between 0 and 1 (including 0, but excluding 1). Here are some interactive examples:

```
>>> from random import random
>>> random()
```

```
0.545146406725
>>> random()
0.221621655814
>>> random()
0.928877335157
>>> random()
0.258660828538
>>> random()
0.859346793436
```

The name of the module (`random`) is the same as the name of the function, which gives rise to the funny-looking import line.

Our racquetball simulation can make use of the `random` function to determine whether or not a player wins a serve. Let's look at a specific example. Suppose a player's service probability is 0.70. This means that they should win 70% of their serves. You can imagine a decision in the program something like this:

```
if <player wins serve>:
    score = score + 1
```

We need to insert a probabilistic condition that will succeed 70% of the time.

Suppose we generate a random value between 0 and 1. Exactly 70% of the interval 0. . .1 is to the left of 0.7. So 70% of the time the random number will be < 0.7, and it will be ≥ 0.7 the other 30% of the time. (The $=$ goes on the upper end, because the random generator can produce a 0, but never a 1.) In general, if `prob` represents the probability that the player wins a serve, the condition `random() < prob` will succeed with just the right probability. Here is how the decision will look:

```
if random() < prob:
    score = score + 1
```

9.3 Top-Down Design

Now you have the complete specification for our simulation and the necessary knowledge of random numbers to get the job done. Go ahead and take a few minutes to write up the program; I'll wait.

OK, seriously, this is a more complicated program than you've probably attempted so far. You may not even know where to begin. If you're going to make it through with minimal frustration, you'll need a systematic approach.

One proven technique for tackling complex problems is called *top-down design*. The basic idea is to start with the general problem and try to express a solution in terms of smaller problems. Then each of the smaller problems is attacked in turn using the same technique. Eventually the problems get so small that they are trivial to solve. Then you just put all the pieces back together and, voilà, you've got a program.

9.3.1 Top-Level Design

Top-down design is easier to illustrate than it is to define. Let's give it a try on our racquetball simulation and see where it takes us. As always, a good start is to study the program specification. In very broad brushstrokes, this program follows the basic input, process, output pattern. We need to get the simulation inputs from the user, simulate a bunch of games, and print out a report. Here is a basic algorithm:

```
Print an Introduction
Get the inputs: probA, probB, n
Simulate n games of racquetball using probA and probB
Print a report on the wins for playerA and playerB
```

Now that we've got an algorithm, we're ready to write a program. I know what you're thinking: this design is too high-level; you don't have any idea yet how it's all going to work. That's OK. Whatever we don't know how to do, we'll just ignore for now. Imagine that all of the components you need to implement the algorithm have already been written for you. Your job is to finish this top-level algorithm using those components.

First we have to print an introduction. I think I know how to do this. It just requires a few print statements, but I don't really want to bother with it right now. It seems an unimportant part of the algorithm. I'll procrastinate and pretend that someone else will do it for me. Here's the beginning of the program:

```
def main():
    printIntro()
```

Do you see how this works? I'm just assuming there is a `printIntro` function that takes care of printing the instructions. That step was easy! Let's move on.

Next, I need to get some inputs from the user. I also know how to do that—I just need a few input statements. Again, that doesn't seem very interesting, and I feel like putting off the details. Let's assume that a component already exists to solve that problem. We'll call the function `getInputs`. The point of this function is to get values for variables `probA`, `probB`, and `n`. The function must return these values for the main program to use. Here is our program so far:

```
def main():
    printIntro()
    probA, probB, n = getInputs()
```

We're making progress; let's move on to the next line.

Here we've hit the crux of the problem. We need to simulate `n` games of racquetball using the values of `probA` and `probB`. This time, I really don't have a very good idea how that will even be accomplished. Let's procrastinate again and push the details off into a function. (Maybe we can get someone else to write that part for us later.) But what should we put into `main`? Let's call our function `simNGames`. We need to figure out what the call of this function looks like.

Suppose you were asking a friend to actually carry out a simulation of `n` games. What information would you have to give him? Your friend would need to know how many games he was supposed to simulate and what the values of `probA` and `probB` should be for those simulations. These three values will, in a sense, be inputs to the function.

What information do you need to get back from your friend? Well, in order to finish out the program (print a report) you need to know how many games were won by player A and how many games were won by player B. These must be outputs from the `simNGames` function. Remember in the discussion of functions in Chapter 6, I said that parameters were used as function inputs, and return values serve as function outputs. Given this analysis, we now know how the next step of the algorithm can be coded:

```
def main():
    printIntro()
    probA, probB, n = getInputs()
    winsA, winsB = simNGames(n, probA, probB)
```

Are you getting the hang of this? The last step is to print a report. If you told your friend to type up the report, you would have to tell him how many wins there were for each player; these values are inputs to the function. Here's the complete program:

```
def main():
    printIntro()
    probA, probB, n = getInputs()
    winsA, winsB = simNGames(n, probA, probB)
    printSummary(winsA, winsB)
```

That wasn't very hard. The `main` function is only five lines long, and the program looks like a more precise formulation of the rough algorithm.

9.3.2 Separation of Concerns

Of course, the `main` function alone won't do very much; we've put off all of the interesting details. In fact, you may think that we have not yet accomplished anything at all, but that is far from true.

We have broken the original problem into four independent tasks: `printIntro`, `getInputs`, `simNGames` and `printSummary`. Further, we have specified the name, parameters, and expected return values of the functions that perform these tasks. This information is called the *interface* or *signature* of a function.

Having signatures allows us to tackle pieces independently. For the purposes of `main`, we don't care *how* `simNGames` does its job. The only concern is that, when given the number of games to simulate and the two probabilities, it must hand back the correct number of wins for each player. The `main` function only cares *what* each (sub-)function does.

Our work so far can be represented as a *structure chart* (also called a *module hierarchy chart*). Figure 9.1 illustrates this. Each component in the design is a rectangle. A line connecting two rectangles indicates that the one above uses the one below. The arrows and annotations show the interfaces between the components in terms of information flow.

At each level of a design, the interface tells us which details of the lower level are important. Anything else can be ignored (for the moment). The general process of determining the important characteristics of something and ignoring other details is called *abstraction*. Abstraction is *the* fundamental tool of design. You might view the entire process of top-down design as a systematic method for discovering useful abstractions.

9.3.3 Second-Level Design

Now all we need to do is repeat the design process for each of the remaining components. Let's take them in order. The `printIntro` function should print

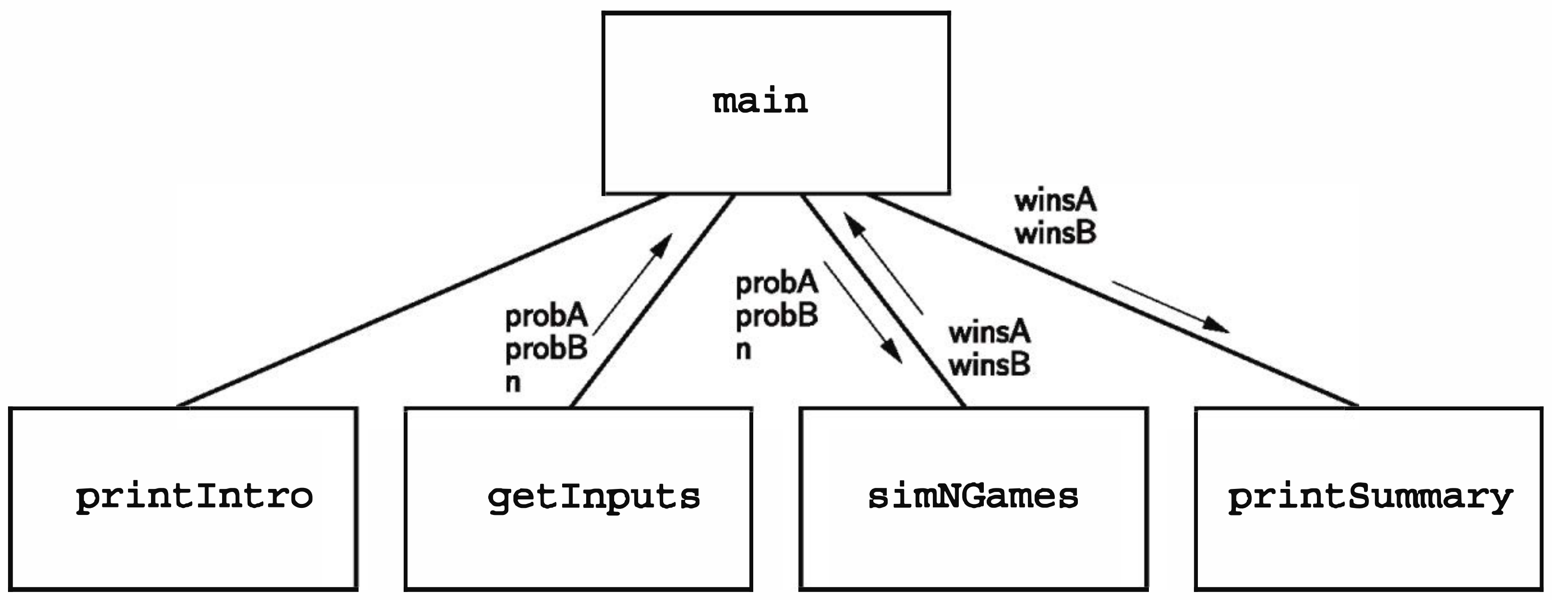

Figure 9.1: First-level structure chart for racquetball simulation

an introduction to the program. Let's compose a suitable sequence of `print` statements:

```
def printIntro():
    print("This program simulates a game of racquetball between two")
    print('players called "A" and "B".  The ability of each player is')
    print("indicated by a probability (a number between 0 and 1) that")
    print("the player wins the point when serving. Player A always")
    print("has the first serve.")
```

Notice the second line. I put double quotes around "A" and "B" so that the entire string is enclosed in apostrophes. This function comprises only primitive Python instructions. Since we didn't introduce any new functions, there is no change to our structure chart.

Now let's tackle `getInputs`. We need to prompt for and get three values, which are returned to the main program. Again, this is simple to code:

```
def getInputs():
    # Returns the three simulation parameters probA, probB and n
    a = float(input("What is the prob. player A wins a serve? "))
    b = float(input("What is the prob. player B wins a serve? "))
    n = int(input("How many games to simulate? "))
    return a, b, n
```

Notice that I have taken some shortcuts with the variable names. Remember, variables inside a function are local to that function. This function is so short, it's very easy to see what the three values represent. The main concern here is to make sure the values are returned in the correct order to match with the interface we established between getInputs and main.

9.3.4 Designing simNGames

Now that we are getting some experience with the top-down design technique, we are ready to try our hand at the real problem, simNGames. This one requires a bit more thought. The basic idea is to simulate n games and keep track of how many wins there are for each player. Well, "simulate n games" sounds like a counted loop, and tracking wins sounds like the job for a couple of accumulators. Using our familiar patterns, we can piece together an algorithm:

```
Initialize winsA and winsB to 0
loop n times
    simulate a game
    if playerA wins
        Add one to winsA
    else
        Add one to winsB
```

It's a pretty rough design, but then so was our top-level algorithm. We'll fill in the details by turning it into Python code.

Remember, we already have the signature for our function:

```
def simNGames(n, probA, probB):
    # Simulates n games and returns winsA and winsB
```

We'll add to this by initializing the two accumulator variables and adding the counted loop heading:

```
def simNGames(n, probA, probB):
    # Simulates n games and returns winsA and winsB
    winsA = 0
    winsB = 0
    for i in range(n):
```

The next step in the algorithm calls for simulating a game of racquetball. I'm not quite sure how to do that, so as usual, I'll put off the details. Let's just assume there's a function called simOneGame to take care of this.

We need to figure out what the interface for this function will be. The inputs for the function seem straightforward. In order to accurately simulate a game, we need to know what the probabilities are for each player. But what should the output be? In the next step of the algorithm, we will need to know who won the game. How do you know who won? Generally, you look at the final score.

Let's have simOneGame return the final scores for the two players. We can update our structure chart to reflect these decisions. The result is shown in Figure 9.2. Translating this structure into code yields this nearly completed function:

```
def simNGames(n, probA, probB):
    # Simulates n games and returns winsA and winsB
    winsA = 0
    winsB = 0
    for i in range(n):
        scoreA, scoreB = simOneGame(probA, probB)
```

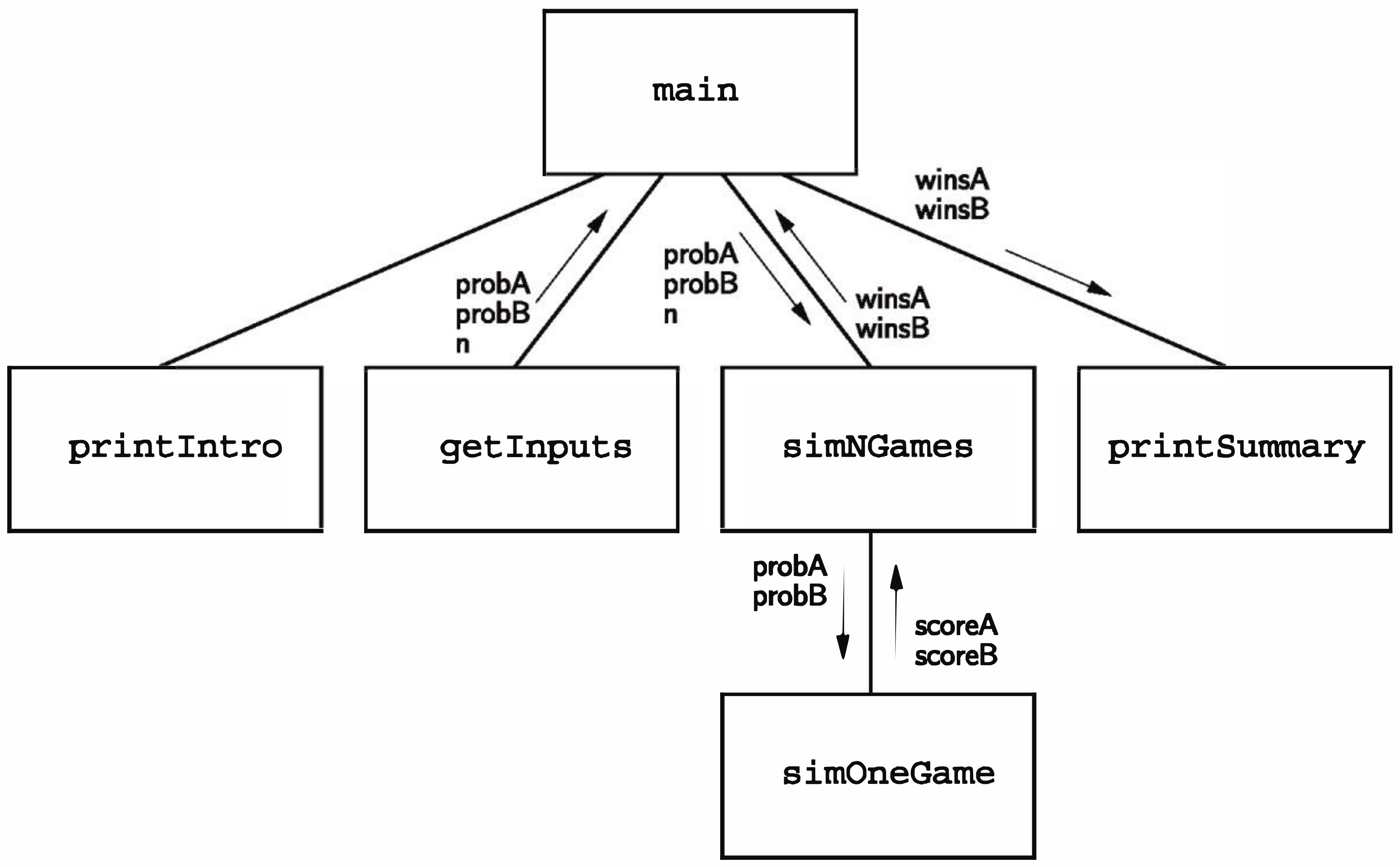

Figure 9.2: Second-level structure chart for racquetball simulation

Finally, we need to check the scores to see who won and update the appropriate accumulator. Here is the result:

```
def simNGames(n, probA, probB):
    winsA = winsB = 0
    for i in range(n):
        scoreA, scoreB = simOneGame(probA, probB)
        if scoreA > scoreB:
            winsA = winsA + 1
        else:
            winsB = winsB + 1
    return winsA, winsB
```

9.3.5 Third-Level Design

Everything seems to be coming together nicely. Let's keep working on the guts of the simulation. The next obvious point of attack is `simOneGame`. Here's where we actually have to code up the logic of the racquetball rules. Players keep doing rallies until the game is over. That suggests some kind of indefinite loop structure; we don't know how many rallies it will take before one of the players gets to 15. The loop just keeps going until the game is over.

Along the way, we need to keep track of the score(s), and we also need to know who is currently serving. The scores will probably just be a couple of int-valued accumulators, but how do we keep track of who's serving? It's either player A or player B. One approach is to use a string variable that stores either `"A"` or `"B"`. It's also an accumulator of sorts, but to update its value we just switch it from one value to the other.

That's enough analysis to put together a rough algorithm. Let's try this:

```
Initialize scores to 0
Set serving to "A"
Loop while game is not over:
    Simulate one serve of whichever player is serving
    update the status of the game
Return scores
```

It's a start, at least. Clearly there's still some work to be done on this one.

We can quickly fill in the first couple of steps of the algorithm to get the following:

```
def simOneGame(probA, probB):
    scoreA = 0
    scoreB = 0
    serving = "A"
    while <condition>:
```

The question at this point is exactly what the condition will be. We need to keep looping as long as the game is not over. We should be able to tell if the game is over by looking at the scores. We discussed a number of possibilities for this condition in the previous chapter, some of which were fairly complex. Let's hide the details in another function, `gameOver`, that looks at the scores and returns `True` if the game is over, and `False` if it is not. That gets us on to the rest of the loop for now.

Figure 9.3 shows the structure chart with our new function. The code for `simOneGame` now looks like this:

```
def simOneGame(probA, probB):
    scoreA = 0
    scoreB = 0
    serving = "A"
    while not gameOver(scoreA, scoreB):
```

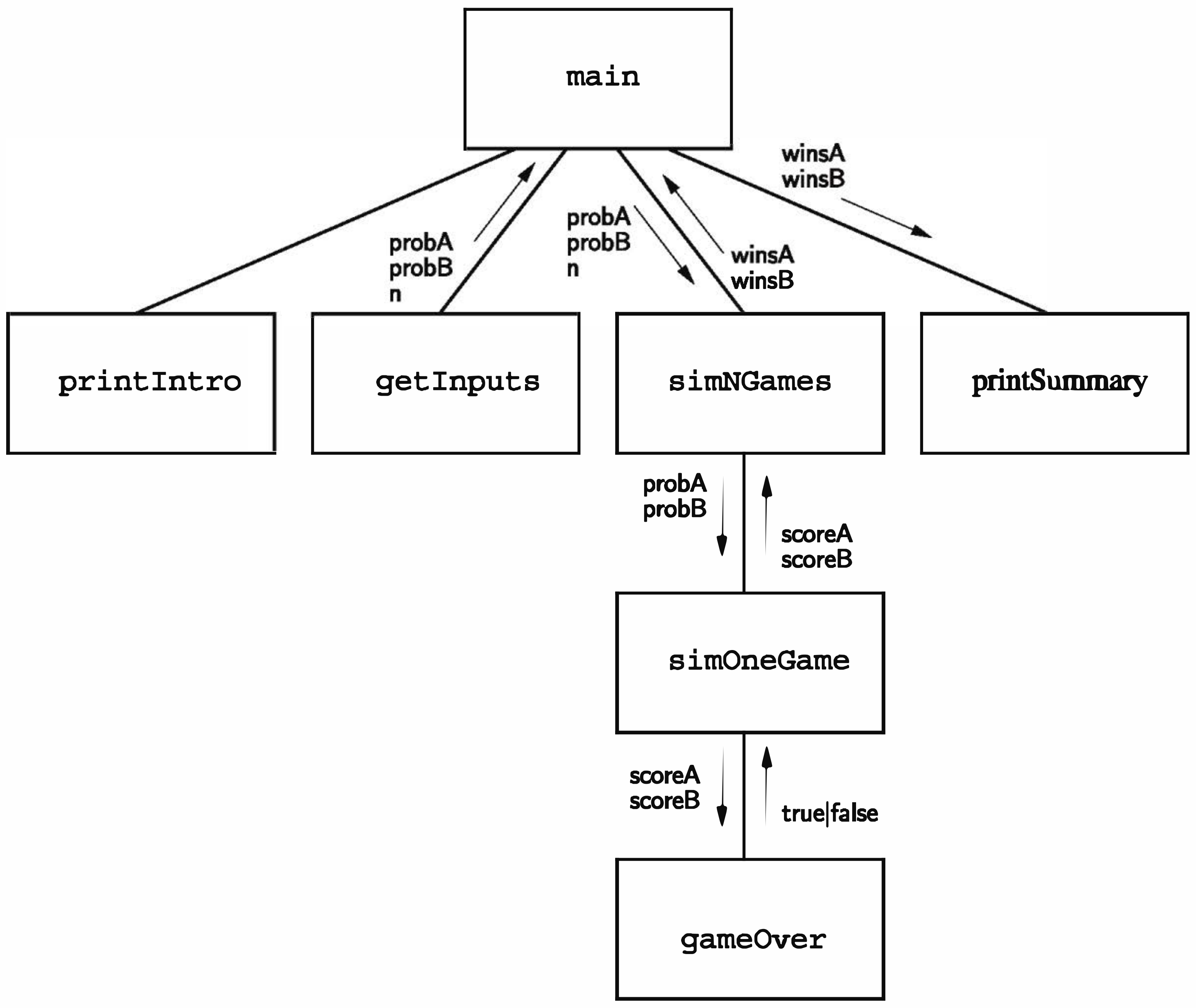

Figure 9.3: Third-level structure chart for racquetball simulation

Inside the loop, we need to do a single serve. Remember, we are going to compare a random number to a probability in order to determine whether the server wins the point (`random() < prob`). The correct probability to use is determined by the value of `serving`. We will need a decision based on this value. If A is serving, then we need to use A's probability, and, based on the result of the serve, either update A's score or change the service to B. Here is the code:

```
if serving == "A":
    if random() < probA:  # A wins the serve
        scoreA = scoreA + 1
    else:                 # A loses the serve
        serving = "B"
```

Of course, if A is not serving, we need to do the same thing, only for B. We just need to attach a mirror image `else` clause.

```
if serving == "A":
    if random() < probA:     # A wins the serve
        scoreA = scoreA + 1
    else:                    # A loses serve
        serving = "B"
else:
    if random() < probB:     # B wins the serve
        scoreB = scoreB + 1
    else:                    # B loses the serve
        serving = "A"
```

That pretty much completes the function. It got a bit complicated, but seems to reflect the rules of the simulation as they were laid out. Putting the function together, here is the result:

```
def simOneGame(probA, probB):
    scoreA = 0
    scoreB = 0
    serving = "A"
    while not gameOver(scoreA, scoreB):
        if serving == "A":
            if random() < probA:
                scoreA = scoreA + 1
            else:
                serving = "B"
        else:
            if random() < probB:
                scoreB = scoreB + 1
            else:
                serving = "A"
    return scoreA, scoreB
```

9.3.6 Finishing Up

Whew! We have just one more troublesome function left, `gameOver`. Here is what we know about it so far:

```
def gameOver(a,b):
    # a and b represent scores for a racquetball game
    # Returns True if the game is over, False otherwise.
```

According to the rules for our simulation, a game is over when either player reaches a total of 15. We can check this with a simple Boolean condition.

```
def gameOver(a,b):
    # a and b represent scores for a racquetball game
    # Returns True if the game is over, False otherwise.
    return a==15 or b==15
```

Notice how this function directly computes and returns the Boolean result all in one step.

We've done it! Except for `printSummary`, the program is complete. Let's fill in the missing details and call it a wrap. Here is the complete program from start to finish:

```
# rball.py
from random import random

def main():
    printIntro()
    probA, probB, n = getInputs()
    winsA, winsB = simNGames(n, probA, probB)
    printSummary(winsA, winsB)
```

```
def printIntro():
    print("This program simulates a game of racquetball between two")
    print('players called "A" and "B".  The ability of each player is')
    print("indicated by a probability (a number between 0 and 1) that")
    print("the player wins the point when serving. Player A always")
    print("has the first serve.")

def getInputs():
    # Returns the three simulation parameters
    a = float(input("What is the prob. player A wins a serve? "))
    b = float(input("What is the prob. player B wins a serve? "))
    n = int(input("How many games to simulate? "))
    return a, b, n

def simNGames(n, probA, probB):
    # Simulates n games of racquetball between players whose
    #    abilities are represented by the probability of winning a serve.
    # Returns number of wins for A and B
    winsA = winsB = 0
    for i in range(n):
        scoreA, scoreB = simOneGame(probA, probB)
        if scoreA > scoreB:
            winsA = winsA + 1
        else:
            winsB = winsB + 1
    return winsA, winsB

def simOneGame(probA, probB):
    # Simulates a single game or racquetball between players whose
    #    abilities are represented by the probability of winning a serve.
    # Returns final scores for A and B
    serving = "A"
    scoreA = 0
    scoreB = 0
    while not gameOver(scoreA, scoreB):
        if serving == "A":
            if random() < probA:
                scoreA = scoreA + 1
            else:
                serving = "B"
        else:
            if random() < probB:
                scoreB = scoreB + 1
            else:
                serving = "A"
    return scoreA, scoreB

def gameOver(a, b):
    # a and b represent scores for a racquetball game
    # Returns True if the game is over, False otherwise.
    return a==15 or b==15

def printSummary(winsA, winsB):
    # Prints a summary of wins for each player.
    n = winsA + winsB
    print("\nGames simulated:", n)
    print("Wins for A: {0} ({1:0.1%})".format(winsA, winsA/n))
    print("Wins for B: {0} ({1:0.1%})".format(winsB, winsB/n))

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

You might take notice of the string formatting in `printSummary`. The type specifier `%` is useful for printing percentages. Python automatically multiplies the number by 100 and adds a trailing percent sign.

9.3.7 Summary of the Design Process

You have just seen an example of top-down design in action. Now you can really see why it's called top-down design. We started at the highest level of our structure chart and worked our way down. At each level, we began with a general algorithm and then gradually refined it into precise code. This approach is sometimes called *step-wise refinement*. The whole process can be summarized in four steps:

1. Express the algorithm as a series of smaller problems.

2. Develop an interface for each of the small problems.

3. Detail the algorithm by expressing it in terms of its interfaces with the smaller problems.

4. Repeat the process for each smaller problem.

Top-down design is an invaluable tool for developing complex algorithms. The process may seem easy, since I've walked you through it step by step. When you first try it out for yourself, though, things probably won't go quite so smoothly. Stay with it—the more you do it, the easier it will get. Initially, you may think writing all of those functions is a lot of trouble. The truth is, developing any sophisticated system is virtually impossible without a modular approach. Keep at it, and soon expressing your own programs in terms of cooperating functions will become second nature.

9.4 Bottom-Up Implementation

Now that we've got a program in hand, your inclination might be to run off, type the whole thing in, and give it a try. If you do that, the result will probably be disappointment and frustration. Even though we have been very careful in our design, there is no guarantee that we haven't introduced some silly errors. Even if the code is flawless, you'll probably make some mistakes when you enter it. Just as designing a program one piece at a time is easier than trying to tackle the whole problem at once, implementation is best approached in small doses.

9.4.1 Unit Testing

A good way to approach the implementation of a modest-sized program is to start at the lowest levels of the structure chart and work your way up, testing each component as you complete it. Looking back at the structure chart for our simulation, we could start with the `gameOver` function. Once this function is typed into a module file, we can immediately import the file and test it. Here is a sample session testing out just this function:

```
>>> gameOver(0,0)
False
>>> gameOver(5,10)
False
>>> gameOver(15,3)
True
```

```
>>> gameOver(3,15)
True
```

I have selected test data that tries all the important cases for the function. The first time it is called, the score will be 0 to 0. The function correctly responds with `False`; the game is not over. As the game progresses, the function will be called with intermediate scores. The second example shows that the function again responded that the game is still in progress. The last two examples show that the function correctly identifies that the game is over when either player reaches 15.

Having confidence that `gameOver` is functioning correctly, now we can go back and implement the `simOneGame` function. This function has some probabilistic behavior, so I'm not sure exactly what the output will be. The best we can do in testing it is to see that it behaves reasonably. Here is a sample session:

```
>>> simOneGame(.5,.5)
(13, 15)
>>> simOneGame(.5,.5)
(15, 11)
>>> simOneGame(.3,.3)
(15, 11)
>>> simOneGame(.3,.3)
(11, 15)
>>> simOneGame(.4,.9)
(4, 15)
>>> simOneGame(.4,.9)
(1, 15)
>>> simOneGame(.9,.4)
(15, 3)
>>> simOneGame(.9,.4)
(15, 0)
>>> simOneGame(.4,.6)
(9, 15)
>>> simOneGame(.4,.6)
(6, 15)
```

Notice that when the probabilities are equal, the scores are close. When the probabilities are farther apart, the game is a rout. That squares with how we think this function should behave.

We can continue this piecewise implementation, testing out each component as we add it into the code. Software engineers call this process *unit testing*. Testing each function independently makes it easier to spot errors. By the time you get around to testing the entire program, chances are that everything will work smoothly.

Separating concerns through a modular design makes it possible to design sophisticated programs. Separating concerns through unit testing makes it possible to implement and debug sophisticated programs. Try these techniques for yourself, and you'll see that you are getting your programs working with less overall effort and far less frustration.

9.4.2 Simulation Results

Finally, we can take a look at Denny Dibblebit's question. Is it the nature of racquetball that small differences in ability lead to large differences in the outcome? Suppose Denny wins about 60% of his serves and his opponent is 5% better. How often should Denny win the game? Here's an example run where Denny's opponent always serves first:

```
This program simulates a game of racquetball between two
players called "A" and "B".  The ability of each player is
indicated by a probability (a number between 0 and 1) that
the player wins the point when serving. Player A always
has the first serve.

What is the prob. player A wins a serve? .65
What is the prob. player B wins a serve? .6
How many games to simulate? 5000

Games simulated: 5000
Wins for A: 3360 (67.2%)
Wins for B: 1640 (32.8%)
```

Even though there is only a small difference in ability, Denny should win only about one in three games. His chances of winning a three- or five-game match are pretty slim. Apparently, Denny is winning his share. He should skip the shrink and work harder on his game.

Speaking of matches, expanding this program to compute the probability of winning multi-game matches would be a great exercise. Why don't you give it a try?

9.5 Other Design Techniques

Top-down design is a very powerful technique for program design, but it is not the only way to go about creating a program. Sometimes you may get stuck at a step and not know how to go about refining it. Or the original specification might be so complicated that refining it level by level is just too daunting.

9.5.1 Prototyping and Spiral Development

Another approach to design is to start with a simple version of a program or program component and then try to gradually add features until it meets the full specification. The initial stripped-down version is called a *prototype*. Prototyping often leads to a sort of *spiral* development process. Rather than taking the entire problem and proceeding through specification, design, implementation, and testing, we first design, implement, and test a prototype. Then new features are designed, implemented, and tested. We make many mini-cycles through the development process as the prototype is incrementally expanded into the final program.

As an example, consider how we might have approached the racquetball simulation. The very essence of the problem is simulating a game of racquetball. We might have started with just the `simOneGame` function. Simplifying even further, our prototype could assume that each player has a 50-50 chance of winning any given point and just play a series of 30 rallies. That leaves the crux of the problem, which is handling the awarding of points and change of service. Here is an example prototype:

```
from random import random

def simOneGame():
    scoreA = 0
    scoreB = 0
    serving = "A"
    for i in range(30):
        if serving == "A":
            if random() < .5:
                scoreA = scoreA + 1
            else:
                serving = "B"
        else:
            if random() < .5:
                scoreB = scoreB + 1
            else:
                serving = "A"
        print(scoreA, scoreB)

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

You can see that I have added a print statement at the bottom of the loop. Printing out the scores as we go along allows us to see that the prototype is playing a game. Here is some example output:

```
1 0
1 0
2 0
...
7 7
7 8
```

It's not pretty, but it shows that we have gotten the scoring and change of service working.

We could then work on augmenting the program in phases. Here's a project plan:

Phase 1 Initial prototype. Play 30 rallies where the server always has a 50% chance of winning. Print out the scores after each serve.

Phase 2 Add two parameters to represent different probabilities for the two players.

Phase 3 Play the game until one of the players reaches 15 points. At this point, we have a working simulation of a single game.

Phase 4 Expand to play multiple games. The output is the count of games won by each player.

Phase 5 Build the complete program. Add interactive inputs and a nicely formatted report of the results.

Spiral development is particularly useful when dealing with new or unfamiliar features or technologies. It's helpful to "get your hands dirty" with a quick prototype just to see what you can do. As a novice programmer, everything may seem new to you, so prototyping might prove useful. If full-blown top-down design does not seem to be working for you, try some spiral development.

9.5.2 The Art of Design

It is important to note that spiral development is not an alternative to top-down design. Rather, they are complementary approaches. When designing the prototype, you will still use top-down techniques. In Chapter 12, you will see yet another approach called object-oriented design.

There is no "one true way" of design. The truth is that good design is as much a creative process as a science. Designs can be meticulously analyzed after the fact, but there are no hard-and-fast rules for producing a design. The best software designers seem to employ a variety of techniques. You can learn about techniques by reading books like this one, but books can't teach how and when to apply them. That you have to learn for yourself through experience. In design, as in almost anything, the key to success is *practice*.

9.6 Chapter Summary

- Computer simulation is a powerful technique for answering questions about real-world processes. Simulation techniques that rely on probabilistic or chance events are known as Monte Carlo simulations. Computers use pseudo-random numbers to perform Monte Carlo simulations.
- Top-down design is a technique for designing complex programs. The basic steps are:
  1. Express an algorithm in terms of smaller problems.
  2. Develop an interface for each of the smaller problems.
  3. Express the algorithm in terms of its interfaces with the smaller problems.
  4. Repeat the process for each of the smaller problems.
- Top-down design was illustrated by the development of a program to simulate the game of racquetball.
- Unit-testing is the process of trying out each component of a larger program independently. Unit-testing and bottom-up implementation are useful in coding complex programs.
- Spiral development is the process of first creating a simple version (prototype) of a complex program and gradually adding features. Prototyping

and spiral development are often useful in conjunction with top-down design.

- Design is a combination of art and science. Practice is the best way to become a better designer.

9.7 Exercises

Review Questions

True/False

1. Computers can generate truly random numbers.
2. The Python `random` function returns a pseudo-random int.
3. Top-down design is also called stepwise refinement.
4. In top-down design, the main algorithm is written in terms of functions that don't yet exist.
5. The `main` function is at the top of a functional structure chart.
6. A top-down design is best implemented from the top down.
7. Unit-testing is the process of trying out a component of a larger program in isolation.
8. A developer should use either top-down or spiral design, but not both.
9. Reading design books alone will make you a great designer.
10. A simplified version of a program is called a simulation.

Multiple Choice

1. Which expression is true approximately 66% of the time?
   a) `random() >= 66` b) `random() < 66`
   c) `random() < 0.66` d) `random() >= 0.66`

2. Which of the following is *not* a step in pure top-down design?
   a) Repeat the process on smaller problems.
   b) Detail the algorithm in terms of its interfaces with smaller problems.
   c) Construct a simplified prototype of the system.
   d) Express the algorithm in terms of smaller problems.

3. A graphical view of the dependencies among components of a design is called a(n)
   a) flowchart b) prototype c) interface d) structure chart

4. The arrows in a module hierarchy chart depict
   a) information flow b) control flow
   c) sticky-note attachment d) one-way streets

5. In top-down design, the subcomponents of the design are
   a) objects b) loops c) functions d) programs

6. A simulation that uses probabilistic events is called
   a) Monte Carlo b) pseudo-random c) Monty Python d) chaotic

7. The initial version of a system used in spiral development is called a
   a) starter kit b) prototype c) mock-up d) beta-version

8. In the racquetball simulation, what data type is returned by the `gameOver` function?
   a) bool b) int c) string d) float

9. How is a percent sign indicated in a string-formatting template?
   a) `%` b) `\%` c) `%%` d) `\%%`

10. The easiest place in a system structure to start unit-testing is
   a) the top b) the bottom c) the middle d) the `main` function

Discussion

1. Draw the top levels of a structure chart for a program having the following `main` function:

```
def main():
    printIntro()
    length, width = getDimensions()
    amtNeeded = computeAmount(length,width)
    printReport(length, width, amtNeeded)
```

2. Write an expression using either `random` or `randrange` to calculate the following:

   a) A random int in the range 0–10
   b) A random float in the range -0.5–0.5
   c) A random number representing the roll of a six-sided die
   d) A random number representing the *sum* resulting from rolling two six-sided dice
   e) A random float in the range -10.0–10.0

3. In your own words, describe what factors might lead a designer to choose spiral development over a top-down approach.

Programming Exercises

1. Revise the racquetball simulation so that it computes the results for best of n game matches. First service alternates, so player A serves first in the odd games of the match, and player B serves first in the even games.

2. Revise the racquetball simulation to take shutouts into account. Your updated version should report for both players the number of wins, percentage of wins, number of shutouts, and percentage of wins that are shutouts.

3. Design and implement a simulation of the game of volleyball. Normal volleyball is played like racquetball in that a team can only score points when it is serving. Games are played to 15, but must be won by at least two points.

4. Most sanctioned volleyball is now played using rally scoring. In this system, the team that wins a rally is awarded a point, even if they were not the serving team. Games are played to a score of 25. Design and implement a simulation of volleyball using rally scoring.

5. Design and implement a system that compares regular volleyball games to those using rally scoring. Your program should be able to investigate whether rally scoring magnifies, reduces, or has no effect on the relative advantage enjoyed by the better team.

6. Design and implement a simulation of some other racquet sport (e.g., tennis or table tennis).

7. Craps is a dice game played at many casinos. A player rolls a pair of normal six-sided dice. If the initial roll is 2, 3, or 12, the player loses. If the roll is 7 or 11, the player wins. Any other initial roll causes the player to "roll for point." That is, the player keeps rolling the dice until either rolling a 7 or re-rolling the value of the initial roll. If the player re-rolls the initial value before rolling a 7, it's a win. Rolling a 7 first is a loss.

   Write a program to simulate multiple games of craps and estimate the probability that the player wins. For example, if the player wins 249 out of 500 games, then the estimated probability of winning is $249/500 = 0.498$.

8. Blackjack (twenty-one) is a casino game played with cards. The goal of the game is to draw cards that total as close to 21 points as possible without going over. All face cards count as 10 points, aces count as 1 or 11, and all other cards count their numeric value.

   The game is played against a dealer. The player tries to get closer to 21 (without going over) than the dealer. If the dealer busts (goes over 21), the player automatically wins (provided the player had not already busted). The dealer must always take cards according to a fixed set of rules. The dealer takes cards until he or she achieves a total of at least 17. If the dealer's hand contains an ace, it will be counted as 11 when that results in a total between 17 and 21 inclusive; otherwise, the ace is counted as 1.

   Write a program that simulates multiple games of blackjack and estimates the probability that the dealer will bust. *Hints*: Treat the deck of cards as infinite (casinos use a "shoe" containing many decks). You do not need to keep track of the cards in the hand, just the total so far (treating an ace as 1) and a bool variable `hasAce` that tells whether or not the hand contains an ace. A hand containing an ace should have 10 points added to the total exactly when doing so would produce a stopping total (something between 17 and 21 inclusive).

9. A blackjack dealer always starts with one card showing. It would be useful for a player to know the dealer's bust probability (see previous problem) for each possible starting value. Write a simulation program that runs multiple hands of blackjack for each possible starting value (ace–10) and estimates the probability that the dealer busts for each starting value.

10. Monte Carlo techniques can be used to estimate the value of pi. Suppose you have a round dartboard that just fits inside of a square cabinet. If

you throw darts randomly, the proportion that hit the dartboard vs. those that hit the cabinet (in the corners not covered by the board) will be determined by the relative area of the dartboard and the cabinet. If n is the total number of darts randomly thrown (that land within the confines of the cabinet), and h is the number that hit the board, it is easy to show that

$$\pi \approx 4(\frac{h}{n})$$

Write a program that accepts the "number of darts" as an input and then performs a simulation to estimate π. *Hint*: You can use `2*random() - 1` to generate the x and y coordinates of a random point inside a 2x2 square centered at $(0, 0)$. The point lies inside the inscribed circle if $x^2 + y^2 \leq 1$.

11. Write a program that performs a simulation to estimate the probability of rolling five of a kind in a single roll of five six-sided dice.

12. A *random walk* is a particular kind of probabilistic simulation that models certain statistical systems such as the Brownian motion of molecules. You can think of a one-dimensional random walk in terms of coin flipping. Suppose you are standing on a very long straight sidewalk that extends both in front of and behind you. You flip a coin. If it comes up heads, you take a step forward; tails means to take a step backward.

    Suppose you take a random walk of n steps. On average, how many steps away from the starting point will you end up? Write a program to help you investigate this question.

13. Suppose you are doing a random walk (see previous problem) on the blocks of a city street. At each "step" you choose to walk one block (at random) either forward, backward, left or right. In n steps, how far do you expect to be from your starting point? Write a program to help answer this question.

14. Write a graphical program to trace a random walk (see previous two problems) in two dimensions. In this simulation you should allow the step to be taken in *any* direction. You can generate a random direction as an angle off of the x axis.

```
angle = random() * 2 * math.pi
```

The new x and y positions are then given by these formulas:

```
x = x + cos(angle)
y = y + sin(angle)
```

The program should take the number of steps as an input. Start your walker at the center of a 100x100 grid and draw a line that traces the walk as it progresses.

15. (Advanced) Here is a puzzle problem that can be solved with either some fancy analytic geometry (calculus) or a (relatively) simple simulation.

    Suppose you are located at the exact center of a cube. If you could look all around you in every direction, each wall of the cube would occupy $\frac{1}{6}$ of your field of vision. Suppose you move toward one of the walls so that you are now halfway between it and the center of the cube. What fraction of your field of vision is now taken up by the closest wall? *Hint*: Use a Monte Carlo simulation that repeatedly "looks" in a random direction and counts how many times it sees the wall.