CHAPTER 5

# Forks in the Road

When you come to a fork in the road, take it.

Yogi Berra

So far, our algorithms have been entirely *deterministic*; they have done the same thing every time we executed them with the same inputs. However, the natural world and its inhabitants (including us) are usually not so predictable. Rather, we consider many natural processes to be, to various degrees, *random*. For example, the behaviors of crowds and markets often change in unpredictable ways. The genetic "mixing" that occurs in sexual reproduction can also be considered a random process because we cannot predict the characteristics of any particular offspring. And the unpredictable movements of tiny particles in the air are often modeled as random processes, with applications to studying airborne pollutants and viruses. To model these kinds of systems, our programs need to be able to both emulate randomness and change their behavior in response to stimuli.

More generally, most run-of-the-mill programs must also be able to conditionally change course, or select from among a variety of options, in response to input. Indeed, most common desktop applications do nothing unless prompted by a key press or a mouse click. Computer games like racing simulators react to a controller several times a second. The protocols that govern data traffic on the Internet adjust transmission rates continually in response to the perceived level of congestion on the network. In this chapter, we will discover how to design algorithms that can behave differently in response to input, both random and deterministic.

## 5.1 RANDOM WALKS

In 1827, British Botanist Robert Brown, while observing pollen grains suspended in water under his microscope, witnessed something curious. When the pollen grains burst, they emitted much smaller particles that proceeded to wiggle around randomly. This phenomenon, now called ***Brownian motion***, was caused by the particles' collisions with the moving water molecules. Brownian motion is now used to describe the motion of any sufficiently small particle (or molecule) in a fluid.

We can model the essence of Brownian motion with a single randomly moving particle in two dimensions. This process is known as a ***random walk***. Random walks are also used to model a wide variety of other phenomena such as markets and the foraging behavior of animals, and to sample large social networks. In this section, we will develop a Monte Carlo simulation of a random walk to discover how far away a randomly moving particle is likely to get in a fixed amount of time.

You may have already modeled a simple random walk in Exercise 2.3.23 by moving a turtle around the screen and choosing a random angle to turn at each step. We will now develop a more restricted version of a random walk in which the particle is forced to move on a two-dimensional grid. At each step, we want the particle to move in one of the four cardinal directions, each with equal probability.

To simulate random processes, we need an algorithm or device that produces random numbers, called a ***random number generator*** (RNG). A conventional computer processor cannot implement a true RNG because everything it does is entirely predictable. Therefore, a computer either needs to incorporate a specialized device that can detect and transmit truly random physical events (like subatomic quantum fluctuations) or simulate randomness with a clever algorithm called a ***pseudorandom number generator*** (PRNG). A PRNG generates a sequence of numbers that *appear* to be random although, in reality, they are not.

The Python module named `random` provides a PRNG in a function named `random`. The `random` function returns a pseudorandom number between zero and one, but not including one. For example:

```
>>> import random
>>> random.random()
0.9699738944412686
```

(Your output will differ.) It is convenient to refer to the range of real numbers produced by the `random` function as [0,1). The square bracket on the left means that 0 is included in the range, and the parenthesis on the right means that 1 is not included in the range. Tangent 5.1 explains a little more about this so-called *interval notation*, if it is unfamiliar to you.

**Tangent 5.1: Interval notation**

It is customary to represent the interval (i.e., set or range), of *real numbers* between $a$ and $b$, including $a$ and $b$, with the notation $[a,b]$. In contrast, the set of *integers* between the integers $a$ and $b$, including $a$ and $b$, is denoted $[a\,..\,b]$. For example, $[3,7]$ represents every real number greater than or equal to 3 and less than or equal to 7, while $[3\,..\,7]$ represents the integers $3,4,5,6,7$.

To denote an interval of real numbers between $a$ and $b$ that does not contain an endpoint $a$ or $b$, we replace the endpoint's square bracket with a parenthesis. So $[a,b)$ is the interval of real numbers between $a$ and $b$ that does contain $a$ but does not contain $b$. Similarly, $(a,b]$ contains $b$ but not $a$, and $(a,b)$ contains neither $a$ nor $b$.

### One small step

Now let's write an algorithm to take one step of a random walk. We will save the particle's $(x,y)$ coordinates in two variables, `x` and `y`. Then we will use the `random` function to assign a random value in [0,1) to a variable named `randi` (why not?).

```
x = 0                       # particle starts at (0, 0)
y = 0
randi = random.random()     # random number in the interval [0, 1)
```

To use this pseudorandom number to randomly move the particle, we need to think about the interval of possible random numbers as being divided into four equal subintervals, and associate a direction with each one:

1. If `randi` is in [0,0.25), then move east.
2. If `randi` is in [0.25,0.5), then move north.
3. If `randi` is in [0.5,0.75), then move west.
4. If `randi` is in [0.75,1.0), then move south.

We can implement this in Python with an `if` statement, which executes a particular sequence of statements only if some Boolean expression is true. The following statement increments `x` if `randi` is in the subinterval [0,0.25):

```
if randi < 0.25:    # if randi is in [0, 0.25), then
    x = x + 1       #   move east
```

**Reflection 5.1** *Why do we not need to also check whether* `randi` *is at least zero?*

An `if` statement is also called a ***conditional statement*** because, like the `while` loops we saw earlier, they make decisions that are conditioned on a Boolean expression. (Unlike `while` loops, however, an `if` statement is only executed once.) The Boolean expression in this case, `randi < 0.25`, is true if `randi` is less than 0.25 and false otherwise. If the Boolean expression is true, the statement(s) that are indented beneath the condition are executed. On the other hand, if the Boolean expression is false, the indented statement(s) are skipped, and the statement following the indented statement(s) is executed next.

| Math symbol | Python symbol |
|---|---|
| $<$ | `<` |
| $\leq$ | `<=` |
| $>$ | `>` |
| $\geq$ | `>=` |
| $=$ | `==` |
| $\neq$ | `!=` |

Table 5.1 Python's six comparison operators.

Let's move now to the second case. To check whether `randi` is in [0.25,0.5), we need to check whether `randi` is greater than or equal to 0.25 *and* `randi` is less than 0.5. The meaning of "and" in the previous sentence is identical to the Boolean operator from Section 3.2. In Python, this condition is represented just as you might expect:

```
randi >= 0.25 and randi < 0.5
```

The `>=` operator is Python's representation of "greater than or equal to" (≥). It is one of six ***comparison operators*** (or *relational operators*), listed in Table 5.1, some of which have two-character representations in Python. (Note especially that `==` is used to test for equality. We will discuss these operators further in Section 5.4.) Adding this case to the first case, we now have two `if` statements:

```
if randi < 0.25:                    # if randi is in [0, 0.25), then
    x = x + 1                       #   move east
if randi >= 0.25 and randi < 0.5:   # if randi is in [0.25, 0.5), then
    y = y + 1                       #   move north
```

Let's think about how these statements will behave in two different cases. First, if `randi` is assigned a value that is less than 0.25, the condition in the first `if` statement will be true and `x = x + 1` will be executed. Next, the condition in the second `if` statement will be checked. But since this condition is false, `y` will not be incremented.

On the other hand, if `randi` is assigned a value that is between 0.25 and 0.5, then the condition in the first `if` statement will be false, so the indented statement `x = x + 1` will be skipped and execution will continue with the second `if` statement. Since the condition in the second `if` statement is true, `y = y + 1` will be executed.

To complete our four-way decision, we can add two more `if` statements:

```
1 if randi < 0.25:                    # if randi is in [0, 0.25), then
2     x = x + 1                       #   move east
3 if randi >= 0.25 and randi < 0.5:   # if randi is in [0.25, 0.5), then
4     y = y + 1                       #   move north
5 if randi >= 0.5 and randi < 0.75:   # if randi is in [0.5, 0.75), then
6     x = x - 1                       #   move west
7 if randi >= 0.75 and randi < 1.0:   # if randi is in [0.75, 1.0), then
8     y = y - 1                       #   move south
9 print(x, y)                         # executed after all 4 cases
```

There are four possible ways these statements could execute, one for each interval in which `randi` can reside. To illustrate one of these cases, suppose `randi` was randomly assigned the value `0.4`.

**Trace arguments:** `randi = 0.4`

| Step | Line | x | y | Notes |
|---|---|---|---|---|
| — | — | 0 | 0 | assume `x = 0` and `y = 0` |
| 1 | 1 | ″ | ″ | `randi < 0.25` is false, so skip to line 3 |
| 2 | 3 | ″ | ″ | `randi >= 0.25 and randi < 0.5` is true, so do line 4 |
| 3 | 4 | ″ | 1 | `y ← y + 1` (move north) |
| 4 | 5 | ″ | ″ | `randi >= 0.5 and randi < 0.75` is false, so skip to line 7 |
| 5 | 7 | ″ | ″ | `randi >= 0.75 and randi < 1.0` is false, so skip to line 9 |
| 6 | 9 | ″ | ″ | prints `0 1` |

Since the condition of the first `if` statement is false, the trace table shows that the indented statement on line 2 is skipped. Next, we test the condition of the `if` statement on line 3. Since this condition is true ($0.25 < 0.4 < 0.5$), the indented statement on line 4, `y = y + 1`, is executed. We continue by testing the condition of the third `if` statement, on line 5. Since this condition is false, we skip the indented statement on line 6. Next, we continue to the fourth `if` statement on line 7, and test its condition, which is false. So line 8 is skipped and execution continues on line 9, which prints the values of `x` and `y`. Notice that, for any possible value of `randi`, only one of the four indented statements will be executed.

**Reflection 5.2** *Is this sequence of steps efficient? If not, what steps could be skipped and in what circumstances?*

The code behaves correctly, but it seems unnecessary to test subsequent conditions after we have already found the correct case. If there were many more than four cases, this extra work could be substantial. Here is a much more efficient structure:

```
1 if randi < 0.25:     # if randi is in [0, 0.25), then
2     x = x + 1        #   move east and finish
3 elif randi < 0.5:    # otherwise, if randi is in [0.25, 0.5), then
4     y = y + 1        #   move north and finish
5 elif randi < 0.75:   # otherwise, if randi is in [0.5, 0.75), then
6     x = x - 1        #   move west and finish
7 elif randi < 1.0:    # otherwise, if randi is in [0.75, 1.0), then
8     y = y - 1        #   move south and finish
9 print(x, y)          # executed after all 4 cases
```

The keyword `elif` is short for "else if," meaning that the condition that follows is checked *only* if no preceding condition was true. In other words, as we sequentially check each of the four conditions, if we find that one is true, then the associated indented statement(s) are executed, and we *skip* the remaining conditions in the group. We also eliminated the unnecessary `>=` checks from each condition (e.g., `randi >= 0.25`). These are redundant because, if we encounter an `elif` condition,

we know that the previous condition must have been false, i.e., we know that `randi` must be greater than all of the previously tested intervals.

The next trace table, with `randi` again randomly assigned `0.4`, illustrates the advantage of this alternative structure.

**Trace arguments:** `randi = 0.4`

| Step | Line | x | y | Notes |
|---|---|---|---|---|
| — | — | 0 | 0 | assume `x = 0` and `y = 0` |
| 1 | 1 | ″ | ″ | `randi < 0.25` is false, so skip to line 3 |
| 2 | 3 | ″ | ″ | `randi < 0.5` is true, so do line 4 |
| 3 | 4 | ″ | 1 | `y ← y + 1` (move north) and skip to line 9 |
| 4 | 9 | ″ | ″ | prints `1 0` |

Everything up through step 3 is identical, but now, after the condition on line 3 is found to be true and line 4 is executed, the remaining `elif` statements are skipped, and execution continues on line 9, after the last `elif`.

**Reflection 5.3** *For each of the four possible intervals to which* `randi` *could belong, how many* `elif` *conditions are checked?*

**Reflection 5.4** *Suppose you replace every* `elif` *with* `if` *in the most recent version above. What would then happen if* `randi` *had the value* `0.4`*?*

This code can be streamlined a bit more. Since `randi` must be in [0,1), there is no point in checking the last condition, `randi < 1.0`. If execution has proceeded that far, `randi` *must* be in [0.75,1). So we can safely execute the last statement, `y = y - 1`, without checking anything. This is accomplished by replacing the last `elif` with an `else` statement:

```
1 if randi < 0.25:     # if randi is in [0, 0.25), then
2     x = x + 1        #   move east and finish
3 elif randi < 0.5:    # otherwise, if randi is in [0.25, 0.5), then
4     y = y + 1        #   move north and finish
5 elif randi < 0.75:   # otherwise, if randi is in [0.5, 0.75), then
6     x = x - 1        #   move west and finish
7 else:                # otherwise,
8     y = y - 1        #   move south and finish
9 print(x, y)          # executed after all 4 cases
```

The `else` signals that, if no previous condition is true, the statement(s) indented under the `else` should be executed.

**Reflection 5.5** *Again, suppose you replace every* `elif` *with* `if` *in this newest version. What would happen now if* `randi` *had the value* `0.4`*?*

If we (erroneously) replaced the two instances of `elif` above with `if`, then the final `else` would be associated *only* with the last `if`. So if `randi` had the value `0.4`, the second and third `if` conditions would both be true and the second and third indented statements would both be executed. The last indented statement would not be executed because the last `if` condition was true.

In situations where there are only two choices, an `else` can just accompany an `if`. For example, if wanted to randomly move a particle on a line instead of in two dimensions, our conditional would look like:

```
if randi < 0.5:    # if randi is in [0, 0.5), then
    x = x + 1      #   move east and finish
else:              # otherwise,
    x = x - 1      #   move west and finish
print(x)           # executed after the if/else
```

### Monte Carlo simulation

The `randomWalk` function below uses our `if/elif/else` conditional structure in a loop to simulate a random walk, and then returns the distance the particle has moved from the origin. To make the grid movement easier to see, we multiply the turtle's movement by a variable `moveLength`. Figure 5.1 shows an example of what you will see when you call `randomWalk` from a `main` function.

```
def randomWalk(steps, tortoise):
    """Displays a random walk on a grid.

    Parameters:
        steps:    the number of steps in the random walk
        tortoise: a Turtle object

    Return value: the final distance from the origin
    """

    x = 0                       # initialize (x, y) = (0, 0)
    y = 0
    moveLength = 10             # length of a turtle step
    for step in range(steps):
        randi = random.random()    # randomly choose a direction
        if randi < 0.25:     # if randi is in [0, 0.25), then
            x = x + 1        #   move east and finish
        elif randi < 0.5:    # otherwise, if randi is in [0.25, 0.5), then
            y = y + 1        #   move north and finish
        elif randi < 0.75:   # otherwise, if randi is in [0.5, 0.75), then
            x = x - 1        #   move west and finish
        else:                # otherwise,
            y = y - 1        #   move south and finish
        tortoise.goto(x * moveLength, y * moveLength)  # draw one step

    return math.sqrt(x * x + y * y)  # return distance from (0, 0)
```

How far, on average, does a randomly walking particle move from its origin in a given number of steps? The answer to this question can, for example, provide insight into the rate at which a fluid spreads or the extent of an animal's foraging territory.

The distance traveled in any one particular random walk is meaningless; the particle

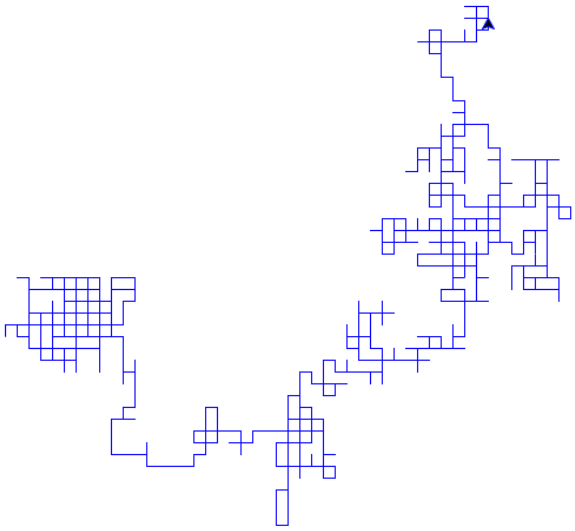


Figure 5.1 A 1000-step random walk produced by the `randomWalk` function.

may return the origin, walk away from the origin at every step, or do something in between. None of these outcomes tells us anything about the *expected*, or average, behavior of the system. To model the expected behavior, we need to compute the average distance over many, many random walks. This kind of simulation is called a ***Monte Carlo simulation***, after the famous casino in Monaco.

As we will be calling `randomWalk` many times, we would like to speed things up by skipping the turtle visualization of the random walks. We can prevent drawing by incorporating a ***flag variable*** as a parameter to `randomWalk`. A flag variable has a Boolean value, and is used to switch some behavior on or off. In Python, the two possible Boolean values are named `True` and `False` (note the capitalization). In the `randomWalk` function, we will call the flag variable `draw`, and cause its value to influence the drawing behavior with another `if` statement:

```
def randomWalk(steps, tortoise, draw):
    """Displays a random walk on a grid.

    Parameters:
        steps:    the number of steps in the random walk
        tortoise: a Turtle object
        draw:     a Boolean indicating whether to draw the random walk

    Return value: the final distance from the origin
    """

    ⋮
        if draw:
            tortoise.goto(x * moveLength, y * moveLength)
    ⋮
```

Now, when we call `randomWalk`, we pass in either `True` or `False` for our third argument. If `draw` is `True`, then `tortoise.goto(···)` will be executed but, if `draw` is `False`, it will be skipped.

**Reflection 5.6** *Incorporate* `draw` *into your* `randomWalk` *function and try calling it with both* `True` *and* `False` *passed in for the third parameter.*

To find the average over many trials, we will call our `randomWalk` function repeatedly in a loop, and use an accumulator variable to sum up all the distances.

```
def rwMonteCarlo(steps, trials):
    """A Monte Carlo simulation to find the expected distance
       that a random walk finishes from the origin.

    Parameters:
        steps:  the number of steps in the random walk
        trials: the number of random walks

    Return value: the average distance from the origin
    """

    totalDistance = 0
    for trial in range(trials):
        distance = randomWalk(steps, None, False)
        totalDistance = totalDistance + distance
    return totalDistance / trials
```

The parameters `steps` and `trials` are the number of steps in each random walk and the number of times to call the `randomWalk` function, respectively. Notice that we have passed in `None` as the argument for the second parameter (`tortoise`) of `randomWalk`. With `False` being passed in for the parameter `draw`, the value assigned to `tortoise` is never used, so we pass in `None` as a placeholder.

**Reflection 5.7** *Call* `rwMonteCarlo(500, 5)` *ten times in a loop, printing the result each time. What do you notice? Do you think five trials is enough? Now perform the same experiment with 10, 100, 1000, and 10000 trials. How many trials do you think are sufficient to get a reliable result?*

As we increase the number of trials in a Monte Carlo simulation, our average results become more consistent, but what do the individual trials look like? To find out, we can generate a *histogram* of the individual random walk distances. A histogram for a data set is a bar graph that shows how the items in the data set are distributed across some number of intervals, which are usually called "bins" or "buckets." To make a histogram of the distances, we need to create a list of these values in the loop, and then call the `pyplot.hist` function to display them as a histogram. A modified version of `rwMonteCarlo` that also displays a histogram is shown below.

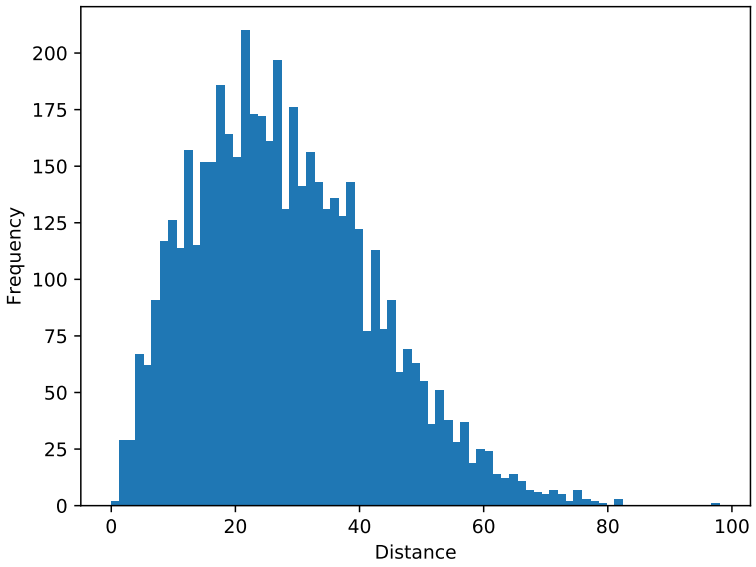



Figure 5.2 A histogram showing the distribution of distances in 5000 trials of a 1000-step random walk.

```
def rwMonteCarlo(steps, trials):
    """ (docstring omitted) """

    totalDistance = 0
    distances = [ ]
    for trial in range(trials):
        distance = randomWalk(steps, None, False)
        totalDistance = totalDistance + distance
        distances.append(distance)

    pyplot.hist(distances, 75)
    pyplot.xlabel('Distance')
    pyplot.ylabel('Frequency')
    pyplot.show()

    return totalDistance / trials
```

The first argument to `pyplot.hist` is the list of values, and the second is the number of bins to use. A histogram produced by calling the modified function with `rwMonteCarlo(1000, 5000)` is shown in Figure 5.2. The $x$-axis represents the distance moved from the origin in the random walks, grouped into 75 bins, and the $y$-axis is the number of times each bin of distances appeared among the 5000 trials. The mean distance returned by the function was about 28, and you can see

that the histogram shows that most of the trials were near that value. The overall shape of the histogram resembles a "bell curve," more formally known as a normal distribution. (To learn more about why this phenomenon occurs, see Section 5.3.)

Ultimately, we want to understand the distance traveled as a function of the number of steps. In other words, if the particle moves $n$ steps, does it travel an average distance of $n/2$ or $n/25$ or $\sqrt{n}$ or $\log_2 n$ or . . . ? To empirically discover the answer, we need to run the Monte Carlo simulation for many different numbers of steps, and try to infer a pattern from a plot of the results. We leave this as an exercise.

## Exercises

5.1.1* What is printed by the following sequence of statements in each of the cases below? Explain your answers.

```
if votes1 >= votes2:
    print('Candidate one wins!')
elif votes1 <= votes2:
    print('Candidate two wins!')
else:
    print('There was a tie.')
```

(a) `votes1 = 184` and `votes2 = 206`

(b) `votes1 = 255` and `votes2 = 135`

(c) `votes1 = 195` and `votes2 = 195`

5.1.2* There is a problem with the statements in the previous exercise. Fix them so that they correctly fulfill their intended purpose.

5.1.3. What is printed by the following sequence of statements in each of the cases below? Explain your answers.

```
majority = (votes1 + votes2 + votes3) / 2
if votes1 > majority:
    print('Candidate one wins!')
if votes2 > majority:
    print('Candidate two wins!')
if votes3 > majority:
    print('Candidate three wins!')
else:
    print('A runoff is required.')
```

(a) `votes1 = 5` and `votes2 = 5` and `votes3 = 5`

(b) `votes1 = 9` and `votes2 = 2` and `votes3 = 4`

(c) `votes1 = 0` and `votes2 = 15` and `votes3 = 0`

5.1.4. Make the statements in the previous problem more efficient and fix them so that they fulfill their intended purpose.

5.1.5. What is syntactically wrong with the following sequence of statements?

```
if x < 1:
    print('Something.')
else:
    print('Something else.')
elif x > 3:
    print('Another something.')
```

5.1.6. What is the final value assigned to `result` after each of the following code segments?

(a)
```
n = 13
result = n
if n > 12:
    result = result + 12
if n < 5:
    result = result + 5
else:
    result = result + 2
```

(b)
```
n = 13
result = n
if n > 12:
    result = result + 12
elif n < 5:
    result = result + 5
else:
    result = result + 2
```

5.1.7* Suppose the weather forecast calls for a 70% chance of rain. Write a function

`weather()`

that prints `'RAIN'` with probability 0.7, and `'SUN!'` otherwise. Then write another version that snows with probability 0.66, produces a sunny day with probability 0.33, and rains cats and dogs with probability 0.01.

5.1.8* Write a function

`roll()`

that simulates the rolling of a single fair die by returning each of the integers 1 through 6 with equal probability. Use `random.random()`.

5.1.9. Write a function

`loaded()`

that simulates the rolling of a single "loaded die" that rolls more 1's and 6's than it should. The probability of rolling each of 1 or 6 should be 0.25. The function should use the `random.random` function and an `if/elif/else` conditional construct to assign a roll value to a variable named `roll`, and then return the value of `roll`.

5.1.10* Write a function that chooses a random number between 1 and 100, prompts for a guess, and prints whether the guess is correct, too high, or too low. Use the function `random.randrange(1, 101)` to get your random number. For example, your function might display:

```
Guess my number: 50
Your guess was too high.  My number was 5.
```

5.1.11. Write a function to implement a simple calculator. The function should prompt for an operation (addition, subtraction, multiplication, or division), the two numbers to operate on, and then print the result. If an unknown operation is entered, the program should say so. For example, your function might display:

```
Operation: *
First number: 67
Second number: 34.1
The answer is 2284.7.
```

or

```
Operation: &
First number: 5
Second number: 3.2
I don't know how to do that!
```

5.1.12. Write a function

```
factors(number)
```

that prints all of the factors of the given number. For example, `factors(66)` should print `2 3 6 11 22 33 66` (one per line).

5.1.13* Write a function

```
countVotes(votes)
```

that implements the algorithm from Exercise 1.2.7, but also accounts for the possibility of a tie. The parameter is a list of strings like `['John', 'Laura', 'Laura', 'John', 'Laura']`. If Laura receives the most votes, return the string `'Laura'`; if John receives the most votes, return `'John'`; if it is a tie, return `'Tie'`.

5.1.14. Write a function

```
assignGrades(scores)
```

that returns a list of letter grades corresponding to the numerical scores in the list `scores`. For example, `assignGrades([78, 91, 85])` should return the list `['C', 'A', 'B']`.

5.1.15. Sometimes we want a random walk to reflect circumstances that bias the probability of a particle moving in some direction (i.e., gravity, water current, or wind). For example, suppose that we need to incorporate gravity, so a movement to the north is modeling a real movement up, away from the force of gravity. Then we might want to decrease the probability of moving north to 0.15, increase the probability of moving south to 0.35, and leave the other directions as they were. Show how to modify the `randomWalk` function to implement this situation.

5.1.16* To discover the distance traveled by a randomly walking particle as a function of the number of its steps, we can call the `rwMonteCarlo` function for many different numbers of steps, and try to infer a pattern from a plot of the results.

(a) Write a function

```
plotDistances(maxSteps, trials)
```

that does this with `steps` equal to 100, 200, ..., `maxSteps`, and then plots the results with `matplotlib.pyplot`. Include properly labeled axes and a legend.

(b) Call `plotDistances(1000, 5000)` to view the relationship between the number of steps taken and the distance moved from the origin. What is your hypothesis for the function approximated by the plot?

(c) The function we are seeking has actually been mathematically determined, and is approximately $\sqrt{n}$. Confirm this empirically by plotting this function alongside the simulated results. To do so, initialize a new list of $y$ values before your loop, and append `math.sqrt(steps)` to this list inside your loop (`steps` is your `for` loop index variable). This creates a list of values of the $\sqrt{n}$ function. Plot these values alongside your random walk results.

(d) As you discovered in Reflection 5.7, the quality of any Monte Carlo approximation depends on the number of trials. Call `plotDistances` a few more times with smaller and larger numbers of trials. What do you notice in your plot?

5.1.17. Determining the number of bins to use in a histogram is part science, part art. If you use too few bins, you might miss the shape of the distribution. If you use too many bins, there may be many empty bins and the shape of the distribution will be too jagged. Experiment with the correct number of bins for 10,000 trials in the `rwHistogram` function you wrote in the previous exercise. At the extremes, create a histogram with only 3 bins and another with 1,000 bins. Then try numbers in between. What seems to be a good number of bins? (You may also want to do some research on this question.)

5.1.18. The Monty Hall problem is a famous puzzle based on the game show "Let's Make a Deal," hosted, in its heyday, by Monty Hall. You are given the choice of three doors. Behind one is a car, and behind the other two are goats. You pick a door, and then Monty, who knows what's behind all three doors, opens a different one, which always reveals a goat. You can then stick with your original door or switch. What do you do (assuming you would prefer a car)?

We can write a Monte Carlo simulation to find out. First, write a function

```
montyHall(choice, switch)
```

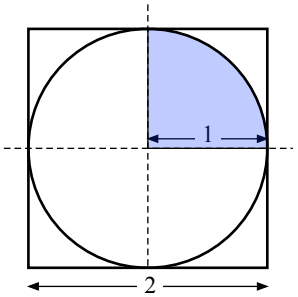
that decides whether we win or lose, based on our original door choice and whether we decide to switch doors. Assume that the doors are numbered 0, 1, and 2, and that the car is always behind door number 2. If we originally chose the car, then we lose if we switch but we win if we don't. Otherwise, if we did not originally choose the car, then we win if we switch and lose if we don't. The function should return `True` if we win and `False` if we lose.

Now write a function

```
monteMonty(trials)
```

that performs a Monte Carlo simulation with the given number of trials to find the probability of winning if we decide to switch doors. For each trial, choose a random door number (between 0 and 2), and call the `montyHall` function with your choice and `switch = True`. Count the number of times we win, and return this number divided by the number of trials. Can you explain the result?

5.1.19. The value of $\pi$ can be estimated with Monte Carlo simulation. Suppose you draw a circle on the wall with radius 1, inscribed inside a square with side length 2, as shown to the right. You then close your eyes and throw darts at the circle. Assuming every dart lands inside the square, the fraction of the darts that land in the circle estimates the ratio between the area of the circle and the area of the square. We know that the area of the circle is $C = \pi r^2 = \pi 1^2 = \pi$ and the area of the square is $S = 2^2 = 4$. So the exact ratio is $\pi/4$. With enough darts, $f$, the fraction (between 0 and 1) that lands in the circle will approximate this ratio: $f \approx \pi/4$, which means that $\pi \approx 4f$.




To make matters a little simpler, we can just throw darts in the upper right quarter of the circle (shaded above). The ratio here is the same: $(\pi/4)/1 = \pi/4$. If we place this quarter circle on $x$ and $y$ axes, with the center of the circle at (0,0), our darts will now all land at points with $x$ and $y$ coordinates between 0 and 1. Use this idea to write a function

```
montePi(darts)
```

that approximates the value $\pi$ by repeatedly throwing random virtual darts that land at points with $x$ and $y$ coordinates in [0,1). Count the number that land at points within distance 1 of the origin, and return this fraction.

5.1.20. The Good, The Bad, and The Ugly are in a three-way gun fight (sometimes called a "truel"). The Good always hits his target with probability 0.8, The Bad always hits his target with probability 0.7, and The Ugly always hits his target with probability 0.6. Initially, The Good aims at The Bad, The Bad aims at The Good, and The Ugly aims at The Bad. The gunmen shoot simultaneously. In the next round, each gunman, if he is still standing, aims at his same target, if that target is alive, or at the other gunman, if there is one, otherwise. This continues until only one gunman is standing or all are dead. What is the probability that they all die? What is the probability that The Good survives? What about The Bad? The Ugly? On average, how many rounds are there? Write a function

```
goodBadUgly()
```

that simulates one instance of this three-way gun fight. Your function should return 1, 2, 3, or 0 depending upon whether The Good, The Bad, The Ugly, or nobody is left standing, respectively. Next, write a function

```
monteGBU(trials)
```

that calls your `goodBadUgly` function repeatedly in a Monte Carlo simulation to answer the questions above.

## *5.2 PSEUDORANDOM NUMBER GENERATORS

This section is available on the book website.

## *5.3 SIMULATING PROBABILITY DISTRIBUTIONS

This section is available on the book website.

## 5.4 BACK TO BOOLEANS

In this section, we will further develop your facility with conditional expressions and Boolean logic. (If your Boolean logic is rusty, you may want to review Section 3.2 first.)

In Python, Boolean expressions evaluate to either the value `True` or the value `False`, which correspond to the binary values `1` and `0`, respectively, that we worked with in Section 3.2. The values `True` and `False` can be printed, assigned to variable names, and manipulated just like numeric values. For example, try the following examples and make sure you understand each result.

```
>>> print(0 < 1)
True
>>> result = 0 > 1
>>> print(result)
False
>>> name = 'Kermit'
>>> print(name == 'Gonzo')
False
```

The "double equals" (`==`) operator tests for equality; it has nothing to do with assignment. The Python interpreter will remind you if you mistakenly use a single equals in an `if` statement. For example, try this:

```
>>> if value = 0:
    if value = 0:
             ^
SyntaxError: invalid syntax
```

However, the interpreter will *not* catch the error if you mistakenly use `==` in an assignment statement. For example, try this:

```
>>> value = 1
>>> value == value + 1   # increment value?
False
```

In a program, nothing will be printed as a result of the second statement, and `value` will not be incremented as expected. So be careful!

As we saw in Section 3.2, Boolean expressions can be combined with the ***Boolean operators*** (or *logical operators*) **and**, **or**, and **not**. As a reminder, Figure 5.3 contains the truth tables for the three Boolean operators, expressed in Python notation. In the tables, the variable names `a` and `b` represent arbitrary Boolean variables or expressions.

For example, suppose we wanted to determine whether a household has an annual income within some range, say $40,000 to $65,000. We can use an `and` operator, as we initially did to check if `randi` was within an interval earlier.

| a | b | a and b | a or b | not a |
|---|---|---|---|---|
| False | False | False | False | True |
| False | True | False | True | True |
| True | False | False | True | False |
| True | True | True | True | False |

Figure 5.3 Combined truth table for the three Python Boolean operators.

```
>>> pay = 53000
>>> (pay >= 40000) and (pay <= 65000)
True
>>> pay = 12000
>>> (pay >= 40000) and (pay <= 65000)
False
>>> pay = 78000
>>> (pay >= 40000) and (pay <= 65000)
False
```

When 53000 is assigned to pay, the two Boolean expressions pay >= 40000 and pay <= 65000 are both True, so (pay >= 40000) and (pay <= 65000) is also True, as summarized in the table below.

| pay | pay >= 40000 | pay <= 65000 | (pay >= 40000) and (pay <= 65000) |
|---|---|---|---|
| 53000 | True | True | True |
| 12000 | False | True | False |
| 78000 | True | False | False |

However, as shown in the second and third rows, when 12000 or 78000 is assigned to pay, one of the components in the and expression is False, so the entire and expression is also False.

Using an or operator in this situation would be incorrect, as you can see below.

| pay | pay >= 40000 | pay <= 65000 | (pay >= 40000) or (pay <= 65000) |
|---|---|---|---|
| 53000 | True | True | True |
| 12000 | False | True | True |
| 78000 | True | False | True |

Since an or expression is True if at least one of its operands is True, the expression (pay >= 40000) or (pay <= 65000) will be True for *every* possible value of pay! (Think about it.)

### Predicate functions

We can incorporate this income test into a function like this:

```
def middleClass(pay):
    """Decide whether an income is classified as "middle class."

    Parameter:
        pay: annual household income

    Return: Boolean value indicating whether pay is a middle class income
    """

    if (pay >= 40000) and (pay <= 65000):
        result = True
    else:
        result = False
    return result
```

Functions that return Boolean values are called ***predicate functions***. Calling `middleClass(53000)` will return the value `True` while `middleClass(12000)` will return `False`. But this function is equivalent to one that just returns the value of the Boolean expression:

```
def middleClass(pay):
    """ (docstring omitted) """

    return (pay >= 40000) and (pay <= 65000)
```

To see that these two functions return identical values, notice that the return value in the first version is always the same as the value of the expression `(pay >= 40000) and (pay <= 65000)`. If the expression is `True`, the function returns `True`. If the expression is `False`, the function returns `False`. So simply returning the value of the expression, as in the second version, accomplishes the same thing.

As another example, suppose we wanted to write a function to decide, in some for-profit company, whether the CEO's compensation divided by the average employees' is at most some "fair" ratio. A simple function that returns the result of this test looks like this:

```
def fair(employee, ceo, ratio):
    """Decide whether the ratio of CEO to employee pay is fair.

    Parameters:
        employee: average employee pay
        ceo:      CEO pay
        ratio:    the fair ratio

    Return:
       a Boolean indicating whether ceo / employee is fair
    """

    return (ceo / employee) <= ratio
```

**Reflection 5.8** *There is a subtle problem though with this function. What is it?*

This function will not always work properly because, if the average employees' compensation equals zero (or zero is mistakenly passed in), the division operation will result in an error. Therefore, we have to test whether `employee == 0` before attempting the division and, if so, return `False` (because not paying employees is obviously not fair). Otherwise, we want to return the result of the fairness test. The following function implements this algorithm.

```
def fair(employee, ceo, ratio):
    """ (docstring omitted) """

    if employee == 0:
        result = False
    else:
        result = (ceo / employee) <= ratio
    return result
```

### Short circuit evaluation

The `fair` function can be simplified by making use of a feature that Python applies to both the `and` and `or` operators called ***short circuit evaluation***. Since only one operand of an `and` expression must be `False` for the expression to be `False`, the Python interpreter does not bother to evaluate the second operand in an `and` expression if the first is `False`. Likewise, since only one operand of an `or` expression must be `True` for the expression to be `True`, the Python interpreter does not bother to evaluate the second operand in an `or` expression if the first is `True`.

This means that our `fair` function can be simplified to:

```
def fair(employee, ceo, ratio):
    """ (docstring omitted) """

    return (employee != 0) and ((ceo / employee) <= ratio)
```

If `employee` is `0`, then `(employee != 0)` is `False`, and the function returns `False` *without* evaluating `(ceo / employee <= ratio)`. On the other hand, if `(employee != 0)` is `True`, then `(ceo / employee <= ratio)` is evaluated, and the return value depends on this outcome. Notice that this would not work if the `and` operator did not use short circuit evaluation because, if `(employee != 0)` were `False` and then `(ceo / employee <= ratio)` was evaluated, the division would result in a "divide by zero" error!

To illustrate the analogous mechanism with the `or` operator, suppose we wanted to write the function in the opposite way, instead returning `True` if the ratio is unfair. The first version of the function would look like this:

```
def unfair(employee, ceo, ratio):
    """ (docstring omitted) """

    if employee == 0:
        result = True
    else:
        result = (ceo / employee) > ratio
    return result
```

However, taking advantage of short circuit evaluation with the `or` operator, we can simplify the whole function to:

```
def unfair(employee, ceo, ratio):
    """ (docstring omitted) """

    return (employee == 0) or ((ceo / employee) > ratio)
```

In this case, if `(employee == 0)` is `True`, the whole expression returns `True` *without* evaluating the division test, thus avoiding an error. On the other hand, if `(employee == 0)` is `False`, the division test is evaluated, and the final result is equal to the outcome of this test.

### DeMorgan's laws

Let's now create a new function that uses a `while` loop to repeatedly prompt for employee and CEO salaries, and decide whether the ratio is fair. The function will ask at the end of each iteration whether it should continue.

```
def fairnessChecker():
    """ (docstring omitted) """

    ratio = float(input('Maximum fair ceo:employee pay ratio: '))
    answer = 'y'
    while answer != 'n':
        employee = float(input('Employee pay: '))
        ceo = float(input('CEO pay: '))
        if fair(employee, ceo, ratio):
            print("That's fair.")
        else:
            print("That's not fair.")
        answer = input('Continue (y/n)? ')
```

**Reflection 5.9** *Why is* `answer` *initialized to* `'y'` *before the loop?*

The function repeatedly prompts for salaries while `answer`, which is obtained at the end of the loop body, is not `'n'`. If `answer` is not initialized before the loop, then the `while` loop condition will not make sense before the first iteration. If it is not

initialized to something other than `'n'`, then the loop will not iterate the first time. It could be intialized to anything other than `'n'`, but `'y'` makes the most sense. Notice also that, because `fair` is a predicate function, we simply use its return value as the Boolean condition in the `if` statement.

How can we modify the `while` loop condition to also allow an uppercase `'N'` to exit the loop? Should the `while` loop condition be changed to

```
(answer != 'n') and (answer != 'N')
```

or to

```
(answer != 'n') or (answer != 'N') ?
```

To make the correct choice, it sometimes helps to think about the opposite situation: when we want the loop to *stop*. This is rather easy in this case: when `answer` is either `'n'` or `'N'`. In other words, we want the loop to stop when `(answer == 'n') or (answer == 'N')`. So the `while` loop condition needs to be the negation of this, or:

```
not((answer == 'n') or (answer == 'N')) .
```

We can use ***De Morgan's laws***, named after 19th century British mathematician Augustus De Morgan, to express this condition in a different way. They are:

1. **not** ($a$ **and** $b$) is equivalent to **not** $a$ **or not** $b$.
2. **not** ($a$ **or** $b$) is equivalent to **not** $a$ **and not** $b$.

You may recognize these from Exercises 3.2.12 and 3.2.13, which asked you to construct the following truth tables to prove the laws' veracity!

| $a$ | $b$ | $a$ **and** $b$ | **not**($a$ **and** $b$) | **not** $a$ | **not** $b$ | **not** $a$ **or not** $b$ |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 |

| $a$ | $b$ | $a$ **or** $b$ | **not**($a$ **or** $b$) | **not** $a$ | **not** $b$ | **not** $a$ **and not** $b$ |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 |

The first law says that $a$ **and** $b$ is false if either $a$ is false or $b$ is false. The second law says that $a$ **or** $b$ is false if both $a$ is false and $b$ is false. Applying the first law to our condition, we find that `not((answer == 'n') or (answer == 'N'))` is equivalent to

```
not (answer == 'n') and not (answer == 'N')
```

or

```
(answer != 'n') and (answer != 'N') .
```

| | Operators | Description |
|---|---|---|
| 1. | `**` | exponentiation (power) |
| 2. | `+, -` | unary positive and negative |
| 3. | `*, /, //, %` | multiplication and division |
| 4. | `+, -` | addition and subtraction |
| 5. | `<, <=, >, >=, !=, ==, in, not in` | comparison operators |
| 6. | `not` | Boolean **not** |
| 7. | `and` | Boolean **and** |
| 8. | `or` | Boolean **or** |

Table 5.2 Operator precedence, listed from highest to lowest. This is an expanded version of Table 1.1.

Incorporating this new condition into the `while` loop in `fairnessChecker` is easy.

Let's take the complexity one step further by changing two things. First, let's stop the loop when a fair pay ratio is obtained. Second, instead of iterating while `answer` is anything but `'n'` or `'N'`, let's iterate only while `answer` is either `'y'` or `'Y'`. These changes are incorporated into the following revised function.

```
1 def fairnessChecker():
2     """ (docstring omitted) """

3     ratio = float(input('Maximum fair ceo:employee pay ratio: '))
4     answer = 'y'
5     isFair = False
6     while (answer == 'y' or answer == 'Y') and not isFair:
7         employee = float(input('Employee pay: '))
8         ceo = float(input('CEO pay: '))
9         isFair = fair(employee, ceo, ratio)
10        if isFair:
11            print("That's fair.")
12        else:
13            print("That's not fair.")
14            answer = input('Continue (y/n)? ')
```

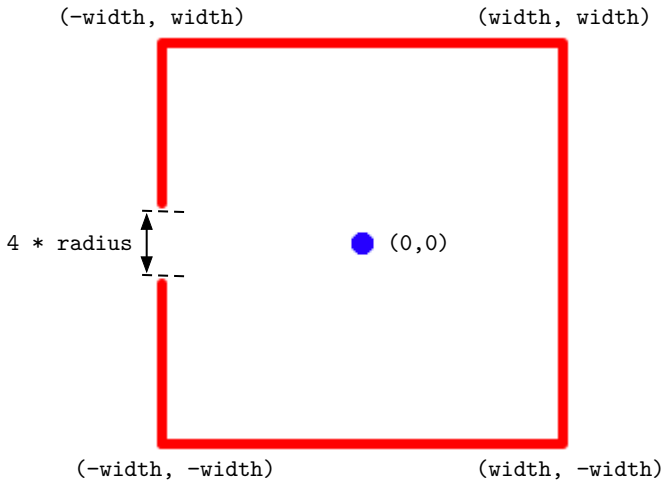
We have changed the test involving `answer` on line 6 and introduced a new Boolean flag variable `isFair` to keep track of whether the current pay ratio is fair. Notice that, to keep the `while` loop going, we need *both* `answer` to be `'y'` or `'Y'` *and* for the pay ratio to be unfair, that is, `not isFair`. We have been using parentheses around certain expressions in this section mostly for clarity, but the parentheses in this `while` loop condition are actually necessary. As shown in Table 5.2, the `and` operator has precedence over the `or` operator. Therefore, without the parentheses,

the expression `answer == 'Y' and not isFair` would be evaluated first, and that would not make sense.

The variable `isFair` is initialized to `False` before the loop so that it iterates at least once, and assigned within the loop on line 9 to be the value returned by the `fair` function. To avoid unnecessarily calling `fair` twice, we also use `isFair` as the `if` condition. We also now only prompt to continue if the pay ratio is not fair.

### Thinking inside the box

Let's next use a random walk simulation to analyze how long it takes for a particle (or a person) to escape an enclosed space. This sort of scenario can also simulate how likely it is for a virus to escape quarantine if there is a slight chance of a break out. We will use turtle graphics to visualize a simple square room with a door, as shown below:




The variables `width` and `radius` will be set to half the width of the room and the radius of the particle, respectively. We want to move the particle randomly in the room while also respecting the walls. For the particle movement, we will use a slightly different random walk in which the amount added to the particle's position is dictated by a normal probability distribution with mean zero. (Also see Exercise 5.3.1.) Here is the overall idea of the algorithm in pseudocode.

**Algorithm** ESCAPE

**Input:** *width* of the room

```
1  draw the room with the given width and create a turtle
2  x ← 0
3  y ← 0
4  number of steps ← 0
5  repeat while the particle has not escaped:
6      number of steps ← number of steps + 1
7      make one normally distributed step, modifying x and y
8      if the particle hits a wall, then:
9          bounce back to the previous position
10     else if the particle finds the door, then:
11         escape and exit the loop
12     move the turtle to position (x, y)
```

**Output:** *number of steps*

The easier parts of this implementation (essentially lines 2–7), including how to perform the random walk, look like this:

```
x = 0      # position of the particle
y = 0
escaped = False
numSteps = 0
while not escaped:
    numSteps = numSteps + 1
    dx = random.gauss(0, step)  # normal with mean 0 and std dev step
    dy = random.gauss(0, step)
    x = x + dx
    y = y + dy

    # bounce off the walls and set escaped to True if
    # the particle finds the door

    particle.goto(x, y)
```

Since the normal (i.e., Gaussian) distribution is centered at zero, `dx` and `dy` can be either positive or negative, meaning that the particle can move in any direction. The random walk will continue until the Boolean variable `escaped` is set to `True`, which we will implement next.

When the particle hits a wall, we want it to "bounce" back to its previous position. We can tell that the left edge of the particle is touching the west wall if `(x - radius <= -width)` is true. Similarly, if `(x + radius >= width)` is true, then the right edge of the particle must be touching the east wall. By combining these with analogous expressions for the north and south walls, we can make the particle bounce back like this:

```
if (x - radius <= -width) or (x + radius >= width) \
  or (y - radius <= -width) or (y + radius >= width):
    x = x - dx   # return to previous position
    y = y - dy
```

**Reflection 5.10** *Why is* `or` *the correct Boolean operator here? When would the expression be true if we used* `and` *instead?*

To incorporate the door, we need to modify the condition so that it excludes the opening in the west wall. The door opening extends from $y$ coordinate `-2 * radius` at the bottom to `2 * radius` at the top. So we will know if the particle is in the doorway if it is touching the west wall and

`(y - radius >= -2 * radius) and (y + radius <= 2 * radius)`,

or equivalently, `(y >= -radius) and (y <= radius)`. But we need the negation of this expression so we can *exclude* the opening in the "bounce" condition.

**Reflection 5.11** *What is the negation of* `(y >= -radius) and (y <= radius)`*?*

Using DeMorgan's laws, the negation of this expression is

`not(y >= -radius) or not(y <= radius).`

Or equivalently,

`(y < -radius) or (y > radius).`

Therefore, the modified `if` condition, causing the particle to bounce back when it touches a wall, is:

```
if ((x <= -width + radius) and ((y < -radius) or (y > radius))) \
  or (x >= width - radius) or (y <= -width + radius) \
  or (y >= width - radius):
    x = x - dx
    y = y - dy
elif (x <= -width + radius) and ((y >= -radius) and (y <= radius)):
    escaped = True
```

Note that the outer parentheses around `((y < -radius) or (y > radius))` are necessary to ensure that the expression is evaluated correctly. We added an `elif` clause to catch when the particle has found the open doorway and set the flag variable to `True` to end the `while` loop. The finished function, with drawing statements omitted, follows. The complete function can be found on the book website.

```
import random
import turtle

def escape(width):
    """Compute the number of steps required for a randomly moving
       particle to escape a square room with a door.

    Parameter:
        width: the width of the room

    Return value: the number of steps needed to escape
    """

    step = 15       # standard deviation of one particle step
    radius = 10     # radius of the particle

    # draw the room and create a turtle named particle here (omitted)

    x = 0          # position of the particle
    y = 0
    escaped = False
    numSteps = 0
    while not escaped:
        numSteps = numSteps + 1
        dx = random.gauss(0, step)  # normal with mean 0 and std dev step
        dy = random.gauss(0, step)
        x = x + dx
        y = y + dy

        if ((x <= -width + radius) and ((y < -radius) or (y > radius))) \
          or (x >= width - radius) or (y <= -width + radius) \
          or (y >= width - radius):
            x = x - dx
            y = y - dy
        elif (x <= -width + radius) and ((y >= -radius) and (y <= radius)):
            escaped = True

        particle.goto(x, y)

    return numSteps
```

**Reflection 5.12** *How can we confirm that this expression is really correct?*

To confirm that any Boolean expression is correct, we can create a truth table for it, and then confirm that every case matches what we intended. This is often not really necessary in practice but, at times, with really complex situations, it can reassuring. We will illustrate by just considering the expression testing whether the particle is touching the west wall but is not in the doorway:

```
(x <= -width + radius) and ((y < -radius) or (y > radius))
```

In this expression, there are three separate Boolean "inputs," one for each expression containing a comparison operator. In the truth table, we will represent each of these with a letter to save space:

- `(x <= -width + radius)` will be represented by $a$,
- `(y < -radius)` will be represented by $b$, and
- `(y > radius)` will be represented by $c$.

So the final expression we want to evaluate is $a$ `and (`$b$ `or` $c$`)`.

In the truth table below, the first three columns represent our three inputs. With three inputs, we need $2^3 = 8$ rows, one for each possible assignment of truth values. There is a trick to quickly writing down all the truth value combinations; see if you can spot it in the first three columns. (We are using `T` and `F` as abbreviations for `True` and `False`.)

| $a$ | $b$ | $c$ | $b$ `or` $c$ | $a$ `and (`$b$ `or` $c$`)` |
|---|---|---|---|---|
| F | F | F | F | F |
| F | F | T | T | F |
| F | T | F | T | F |
| F | T | T | T | F |
| T | F | F | F | F |
| T | F | T | T | T |
| T | T | F | T | T |
| T | T | T | T | T |

We need to first evaluate `(y < -radius) or (y > radius)`. The result, shown in the fourth column, is the `or` of the second and third columns. Then, in the fifth column, we `and` the first column with the fourth to get our final result. This column says that the expression is true in the three highlighted cases. For our expression to be correct, these need to be exactly the situations in which we want the particle to bounce off the west wall. (We can assume that the particle is also within the proper $y$ bounds because that is checked elsewhere in the original expression.) Let's examine each highlighted row to make sure that this result is correct:

- row 6: $a$ is true, $b$ is false, and $c$ is true

  In this case, `(x <= -width + radius)` is true, `(y < -radius)` is false, and `(y > radius)` is true. So the particle is touching the wall and is not below the door and is above the door. Check.

- row 7: $a$ is true, $b$ is true, and $c$ is false

  In the second case, `(x <= -width + radius)` is true, `(y < -radius)` is true, and `(y > radius)` is false. So the particle is touching the wall and is below the door and is not above the door. Check.

- row 8: $a$ is true, $b$ is true, and $c$ is true

  Finally, this case is when all three are true. So the particle is touching the wall and is below the door and is above the door. This cannot possibly happen, so it doesn't matter that the expression is (oddly) true. Check.

### Many happy returns

We often come across situations in which we want a function to return different values depending on the outcome of a condition. The simplest example is finding the maximum of two numbers `a` and `b`. If `a` is at least as large as `b`, we want to return `a`; otherwise, `b` must be larger, so we return `b`.

```
def max(a, b):
    """ (docstring omitted) """

    if a >= b:
        result = a
    else:
        result = b
    return result
```

We can simplify this function a bit by returning the appropriate value right in the `if/else` statement:

```
def max(a, b):
    """ (docstring omitted) """

    if a >= b:
        return a
    else:
        return b
```

It may look strange at first to see two `return` statements in one function, but it all makes perfect sense. Recall from Section 2.5 that `return` *both* ends the function *and* assigns the function's return value. So this means that at most one `return` statement can ever be executed in a function. In this case, if `a >= b` is `True`, the function ends and returns the value of `a`. Otherwise, the function executes the `else` clause, which returns the value of `b`.

The fact that the function ends if `a >= b` is `True` means that we can simplify it even further: if execution continues past the `if` part of the `if/else`, it *must* be the case that `a >= b` is `False`. So the `else` is extraneous; the function can be simplified to:

```
def max(a, b):
    """ (docstring omitted) """

    if a >= b:
        return a
    return b
```

This same principle can be applied to situations with more than two cases. Suppose we wanted to use `if` statements to convert a percentage grade to a grade point (i.e., GPA) on a 0–4 scale. A natural implementation of this might look like the following:

```
def assignGP(score):
    """Returns the grade point equivalent of score.

    Parameter:
        score: a score between 0 and 100

    Return value: the equivalent grade point value
    """

    if score >= 90:
        return 4
    elif score >= 80:
        return 3
    elif score >= 70:
        return 2
    elif score >= 60:
        return 1
    else:
        return 0
```

**Reflection 5.13** *Why do we not need to check upper bounds on the scores in each case? In other words, why does the second condition not need to be* `score >= 80 and score < 90`*?*

Suppose `score` was `92`. Then the first condition is `True`, so the function returns the value `4` and ends. Execution never proceeds past the statement `return 4`. For this reason, the "`el`" in the next `elif` is extraneous. In other words, because execution would never have made it there if the previous condition was `True`, there is no need to tell the interpreter to skip testing this condition if that was the case.

Now suppose `score` was `82`. In this case, the first condition would be `False`, so we continue on to the first `elif` condition. Because we got to this point, we already know that `score < 90` (hence the omission of that check). The first `elif` condition is `True`, so we immediately return the value `3`. So there is no need for the "`el`" in the second `elif` either because there is no need to skip testing this condition if either of the previous conditions were `True`. In fact, we can remove the "`el`"s from all of the `elif`s, and the final `else`, with no loss in efficiency at all.

```
def assignGP(score):
    """ (docstring omitted) """

    if score >= 90:
        return 4
    if score >= 80:
        return 3
    if score >= 70:
        return 2
    if score >= 60:
        return 1
    return 0
```

Some programmers find it clearer to leave the `elif` statements in, and that is fine too. We will do it both ways in the coming chapters. But, as you begin to see more algorithms, you will probably see code like this, and so it is important to understand why it is correct.

## Exercises

*Write a function for each of the following exercises. Test each one with both common and boundary case arguments, as described on page 38, and document your test cases.*

5.4.1* Write a function

`password()`

that asks for a username and a password. It should return `True` if the username is entered as `alan.turing` and the password is entered as `notTouring`, and return `False` otherwise.

5.4.2. Suppose that in a game that you are making, the player wins if her score is at least 100. Write a function

`hasWon(score)`

that returns `True` if she has won, and `False` otherwise.

5.4.3. Suppose you have designed a sensor that people can wear to monitor their health. One task of this sensor is to monitor body temperature: if it falls outside the range 97.9° F to 99.3° F, the person may be getting sick. Write a function

`monitor(temperature)`

that takes a temperature (in Fahrenheit) as a parameter, and returns `True` if `temperature` falls in the healthy range and `False` otherwise.

5.4.4* A year is a leap year if it is divisible by four, unless it is a century year in which case it must be divisible by 400. For example, 2028 and 1600 are leap years, but 2027 and 1800 are not. Write a function

`leapYear(year)`

that returns the value of a single Boolean expression to indicate whether the year is a leap year. Also write a function

`leapYears(beginYear, endYear)`

that uses your `leapYear` function to return a list of all the leap years between (and including) the two years given as parameters.

5.4.5* Write a function

```
nextLeapYear(afterYear)
```

that uses your `leapYear` function from the previous exercise to return the closest leap year after the year given as a parameter.

5.4.6. Write a function

```
even(number)
```

that returns `True` if `number` is even, and `False` otherwise.

5.4.7* Write a function

```
between(number, low, high)
```

that returns `True` if `number` is in the interval [`low`,`high`] (between `low` and `high`, including both `low` and `high`), and `False` otherwise.

5.4.8. Write a function

```
justOne(a, b)
```

that returns `True` if exactly one (but not both) of the numbers a or b is 10, and `False` otherwise.

5.4.9. Write a function

```
roll()
```

that simulates rolling two of the loaded dice implemented in Exercise 5.1.9 (by calling the function `loaded`), and returns `True` if the sum of the dice is 7 or 11, or `False` otherwise.

5.4.10. The following function returns a Boolean value indicating whether an integer `number` is a perfect square. Rewrite the function in one line, taking advantage of the short-circuit evaluation of `and` expressions.

```
def perfectSquare(number):
    if number < 0:
        return False
    else:
        return math.sqrt(number) == int(math.sqrt(number))
```

5.4.11. Write a function

```
previousSquare(before)
```

that uses your `perfectSquare` function from the previous exercise to return the maximum perfect square smaller than the number given as a parameter. If the parameter is not positive, return 0.

5.4.12* Write a function

```
winner(score1, score2)
```

that returns `1` or `2`, indicating whether the winner of a game is Player 1 or Player 2. The higher score wins and you can assume that there are no ties.

5.4.13. Repeat the previous exercise, but also return `0` to indicate a tie.

5.4.14* Your firm is looking to buy computers from a distributor for $1500 per machine. The distributor will give you a 5% discount if you purchase more than 20 computers. Write a function

```
cost(quantity)
```

that takes as a parameter the `quantity` of computers you wish to buy, and returns the cost of buying them from this distributor.

5.4.15. Repeat the previous exercise, but add three more parameters: the cost per machine, the number of computers necessary to get a discount, and the discount.

5.4.16. The speeding ticket fine in a nearby town is $50 plus $5 for each mph over the posted speed limit. In addition, there is an extra penalty of $200 for all speeds above 90 mph. Write a function

```
fine(speedLimit, clockedSpeed)
```

that returns the fine amount (or 0 if `clockedSpeed` ≤ `speedLimit`).

5.4.17. Write a function

```
gradeRemark()
```

that prompts for a grade, and then returns the corresponding remark (as a string) from the table to the right.

| Grade | Remark |
|---|---|
| 96–100 | Outstanding |
| 90–95 | Exceeds expectations |
| 80–89 | Acceptable |
| 1–79 | Trollish |

5.4.18. Write a function that takes two integer values as parameters and returns their sum if they are not equal and their product if they are.

5.4.19. Write a function

```
amIRich(amount, rate, years)
```

that accumulates interest on `amount` dollars at an annual rate of `rate` percent for a number of `years`. If your final investment is at least double your original amount, return `True`; otherwise, return `False`.

5.4.20* Write a function

```
maxOfThree(a, b, c)
```

the returns the maximum value of the parameters `a`, `b`, and `c`. Be sure to test it with many different numbers, including some that are equal.

5.4.21. Write a function

```
shipping(amount)
```

that returns the shipping charge for an online retailer based on a purchase of `amount` dollars. The company charges a flat rate of $6.95 for purchases up to $100, plus 5% of the amount over $100.

5.4.22* Write a function

```
oddFactors(number)
```

that returns a list of all of the odd factors of the given number. For example, `oddFactors(66)` should return the list `[3, 11, 33]`.

5.4.23. Write a function

```
commonFactors(number1, number2)
```

that returns a list containing all of the common factors of two numbers. For example, `commonFactors(18, 36)` should return the list `[2, 3, 6, 9, 18]`.

5.4.24. Write a function

```
isPrime(number)
```

that checks whether a given number is prime. The function should return `True` if the number is prime and `False` otherwise. (Use a `for` loop.) Then write a function

```
primes(begin, end)
```

that uses your `isPrime` function to return a list of all prime numbers between (and including) the two numbers given as parameters.

5.4.25. Write a function

```
nextPrime(after)
```

that uses your `isPrime` function from the previous exercise to return the smallest prime number after the number given as a parameter.

5.4.26. Starting with two positive integers $a$ and $b$, consider the sequence in which the next number is the digit in the ones place of the sum of the previous two numbers. For example, if $a = 1$ and $b = 1$, the sequence is $1, 1, 2, 3, 5, 8, 3, 1, 4, 5, 9, 4, 3, 7, 0, \ldots$ Write a function

```
mystery(a, b)
```

that returns the length of the sequence when the last two numbers repeat the values of $a$ and $b$ for the first time. (When $a = 1$ and $b = 1$, the function should return 62.)

5.4.27. The Chinese zodiac relates each year to an animal in a twelve-year cycle. The animals for one particular cycle are given in the table to the right. Write a function

```
zodiac(year)
```

that takes as a parameter a `year` (this could be any year in the past or future) and returns the corresponding animal as a string. Then write another function

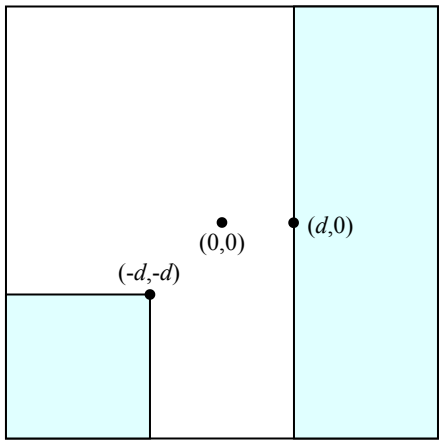
```
zodiacTable(beginYear, endYear)
```

that uses your `zodiac` function to print a table of the zodiac animals for all years between (and including) the two parameters.

| Year | Animal | Year | Animal |
|---|---|---|---|
| 2004 | monkey | 2010 | tiger |
| 2005 | rooster | 2011 | rabbit |
| 2006 | dog | 2012 | dragon |
| 2007 | pig | 2013 | snake |
| 2008 | rat | 2014 | horse |
| 2009 | ox | 2015 | goat |

5.4.28. Consider the `rwMonteCarlo` function on page 173. What will the function return if `trials` equals 0? What if `trials` is negative? Propose a way to deal with these issues by adding statements to the function.

5.4.29. Write a Boolean expression that is true if a point $(x,y)$ resides in either of the shaded boxes below (including their boundaries), and false otherwise. Assume that the particle is not able to ever roam outside the outermost square. The shaded rectangle on the right represents all points with $x$ coordinates at least $d$.




5.4.30. Use a truth table to show that the expression you derived in the previous exercise is correct.

5.4.31. Use a truth table to show that the Boolean expressions

```
(x > d and y > d) or (x < -d and y > d)
```

and

```
(y > d) and (x > d or x < -d)
```

are equivalent.

5.4.32. Write a function

```
drawRow(tortoise, row)
```

that uses turtle graphics to draw one row of an $8 \times 8$ red/black checkerboard. If the value of `row` is even, the row should start with a red square; otherwise, it should start with a black square. You may want to use the `drawSquare` function you wrote in Exercise 2.3.11. Your function should only need one `for` loop and only need to draw one square in each iteration.

5.4.33. Write a function

```
checkerBoard(tortoise)
```

that draws an $8 \times 8$ red/black checkerboard, using the function you wrote in Exercise 5.4.32.

## 5.5 DEFENSIVE PROGRAMMING

At the end of the first chapter, we previewed the coming attractions by pointing out that once your toolbox contained the four types of statements you now know—assignments, arithmetic, loops, and conditionals—the possibilities were endless. But with great power comes great responsibility.[1] On the one hand, you can let your creativity flourish; on the other hand, you know enough to get yourself into convoluted situations that are harder to debug. So it is time to start taking a more deliberate approach to writing correct programs. Our strategy will be twofold. First, we will think more carefully and rigorously about what the inputs to a problem should look like and anticipate what might go awry if they don't. Second, we will start testing each function more formally with carefully chosen inputs to make sure it really does what we think it does.

### Checking parameters

When we are identifying the input to a problem, we need to also think about constraints on the input. What class(es) should the input belong to? What range of values make sense for the problem? How should the algorithm or function behave if it receives input that doesn't make sense? Thinking about these issues proactively in advance, *defensively*, tends to result in more robust and correct programs. This is especially important if other people are going to use your programs because you cannot control what (foolish) things they might do. Taken further, defensive programming is a key component of ***computer security***. A security analyst's job is to anticipate how a criminal might exploit errors in a program to gain access to files and resources that should be off-limits.

To illustrate these strategies, we will revisit the `leapYear` and `nextLeapYear` functions from Exercises 5.4.4–5.4.5. Here are possible implementations of these functions.

```
def leapYear(year):
    """Determine whether a year is a leap year.

    Parameter:
        year: an integer year

    Return value: a Boolean value indicating whether year is a leap year
    """

    if year % 100 != 0:          # if year is not a century year
        return year % 4 == 0     #   it is a leap year if divisible by 4
    else:                        # else if year is a century year
        return year % 400 == 0   #   it is a leap year if divisible by 400
```

[1] Also known as the Peter Parker principle.

```
def nextLeapYear(afterYear):
    """Determine the next leap year after a given year.

    Parameter:
        afterYear: an integer year

    Return value: the next leap year
    """

    year = afterYear + 1        # start looking in the next year
    while not leapYear(year):   # while we haven't found a leap year
        year = year + 1         #   keep looking ...
    return year
```

**Reflection 5.14** *In the* `leapYear` *function, are there any values of* `year` *that should be disallowed?*

To answer this question, it helps to know something about the Gregorian calendar, which is divided into two eras: the Common Era (CE) and Before the Common Era (BCE) (or AD and BC in the Christian tradition). Both CE and BCE start at year one; there is no year zero. Therefore, since the function does not allow a way to specify CE or BCE, we should assume that only CE years, starting at year one, are allowed. If any value of `year` less than one is given, we will simply return `False` by inserting an `if` statement at the beginning of the function.

```
def leapYear(year):
    """Determine whether a year is a leap year.

    Parameter:
        year: an integer year greater than zero in the Common Era

    Return value: a Boolean value indicating whether year is a leap year
    """

    if year < 1:
        return False

    if year % 100 != 0:         # if year is not a century year
        return year % 4 == 0    #   it is a leap year if divisible by 4
    else:                       # else if year is a century year
        return year % 400 == 0  #   it is a leap year if divisible by 400
```

The highlighted requirement that `year` must be an integer greater than zero is called a ***precondition***. A precondition for a function is something that must be true when the function is called for the function to behave correctly. But our `leapYear` function currently only enforces part of the stated precondition.

**Reflection 5.15** *What happens if the argument passed in for* `year` *is not an integer? For example, try calling* `leapYear('cookies')`.

Calling `leapYear('cookies')` results in the following error:

```
TypeError: not all arguments converted during string formatting
```

This happens when the `%` operator is applied to the value of `year`, which is `'cookies'`, and `100` in the first `if` statement. Because `year` is a string, Python is interpreting this expression as an old kind of string formatting operation that we don't cover. To avoid this rather opaque error message, we need to make sure that `year` is an integer at the beginning of the function. We can do this with the `isinstance` function, which takes two arguments: a value and the name of a class. The function returns `True` if the variable or value refers to an object (i.e., instance) of the class, and `False` otherwise. For example,

```
>>> isinstance(5.0, int)
False
>>> isinstance(5.0, float)
True
>>> word = 'cookies'
>>> isinstance(word, str)
True
>>> isinstance(word, int)
False
```

To use this in the `leapYear` function, we replace the first `if` statement with

```
if not isinstance(year, int) or (year < 1):
    return False
```

**Reflection 5.16** *The order of the two conditions in the* `or` *expression is important. Why?*

Similar to a precondition, the ***postcondition*** for a function is a statement of what must be true when the function finishes. A postcondition usually specifies what the function returns and what, if any, side effects it has. Recall that a side effect occurs when a global variable is modified or some other event in a function modifies a global resource. For example, calls to `print` and modifications of files are considered to be side effects because they have impacts outside of the function itself.

**Reflection 5.17** *What is the postcondition of the* `leapYear` *function?*

The postcondition for the `leapYear` function is that the function returns a Boolean indicating whether `year` is a leap year, which is already stated in the docstring. Because they describe the input and output of a function, and therefore how the function can be used, preconditions and postconditions are often included explicitly in the docstring. For now, we will retain our docstring format and just make sure that preconditions and postconditions are implicitly stated in our parameter and return value sections.

The use of preconditions and postconditions is called ***design by contract*** because the precondition and postcondition establish a contract between the function designer and the function caller. The function caller understands that the precondition must be met before the function is called and the function designer guarantees that, if the precondition is met, the postcondition will also be met.

Now let's look at the `nextLeapYear` function.

**Reflection 5.18** *In the* `nextLeapYear` *function, what happens if* `afterYear` *is a* `float`*? Or something else? Or a negative integer?*

The precondition for the `nextLeapYear` function should be the same as the `leapYear` function since the parameter is also a year in CE. Indeed, if `afterYear` is anything but an `int`, then the `while` loop will be infinite because we modified `leapYear` to always return `False` in these cases! So we definitely need to check that `afterYear` is an `int` but making sure it is positive is a judgment call. If `afterYear` is a negative integer, one could argue that the function behaves correctly because it will eventually return `4`, the smallest leap year in CE. On the other hand, passing in any year less than one is technically an error because there are no valid years less than one in CE. When in doubt, it is best to take the more cautious approach to prevent the most errors.

**Reflection 5.19** *What should the* `nextLeapYear` *function return if the precondition is not met?*

Knowing what to return in this case is trickier. One option would be to return `None`. But this could be inconvenient for a calling function to handle because the possible return values are of two different types. An alternative would be to return the same type as the normal return value (an `int`) but a value that the function would not otherwise return, like zero in this case. This behavior needs to be clearly stated in the docstring as part of the function's postcondition.

```
def nextLeapYear(afterYear):
    """Determine the next leap year after a given year.

    Parameter:
        afterYear: an integer year greater than zero in the Common Era

    Return value: the next leap year or zero if afterYear is not
                  a positive integer
    """

    if not isinstance(afterYear, int) or (afterYear < 1):
        return 0

    year = afterYear + 1        # start looking in the next year
    while not leapYear(year):   # while we haven't found a leap year
        year = year + 1         #   keep looking ...
    return year
```

### Assertions

An alternative way to enforce preconditions is to raise an ***exception*** in lieu of letting the function return normally. This is precisely what happens when the Python interpreter displays a `TypeError` or a `ValueError` and aborts a program. A built-in function raises an exception when something goes wrong and the function

cannot continue. When a function raises an exception, it does not return normally. Instead, execution of the function ends at the moment the exception is raised and execution instead continues in part of the Python interpreter called an *exception handler*. By default, the exception handler prints an error message and aborts the entire program.

It is possible for our functions to also raise `TypeError` and `ValueError` exceptions, but we will not address this option until Chapter 12. We will instead consider just one particularly simple type of exception called an `AssertionError`, which may be raised by an `assert` statement. An `assert` statement tests a Boolean condition, and raises an `AssertionError` if the condition is `False`. If the condition is `True`, the `assert` statement does nothing. For example,

```
>>> year = 2021
>>> assert year > 0  # does nothing
>>> year = -1
>>> assert year > 0
AssertionError
>>> assert year > 0, 'year must be a positive'
AssertionError: year must be a positive
```

The first `assert` statement above does nothing because the condition being asserted is `True`. But the condition in the second `assert` statement is `False`, so an `AssertionError` exception is raised. The third `assert` statement demonstrates that we can also include an informative message to accompany an `AssertionError`.

We can replace the first `if` statement in our `leapYear` function with one or more assertions to catch both types of errors that we discussed previously:

```
assert isinstance(year, int), 'year must be an integer'
assert year > 0, 'year must be positive'
```

Or, we could combine them into one `assert` statement:

```
assert isinstance(year, int) and (year > 0),
       'year must be a positive integer'
```

This statement is saying that `year` must both be an integer and positive. If either of these conditions is `False`, then the `assert` statement will display

```
AssertionError: year must be a positive integer
```

and abort the program.

**Reflection 5.20** *Call this modified version of* `leapYear` *from the* `nextLeapYear` *function. What happens now when you call* `nextLeapYear(2020.4)`*?*

Note that, since the `assert` statement aborts the entire program, it should only be used in circumstances in which there is no other reasonable course of action. But the definition of "reasonable" usually depends on the circumstances.

### Unit testing

The only way to really ensure that a function is correct is to either mathematically prove it is correct or test it with every possible input. But since both of these strategies are virtually impossible in all but the most trivial situations, the best we can do is to test our functions with a variety of carefully chosen inputs that are representative of the entire range of possibilities. In large software companies, there are dedicated teams whose sole jobs are to design and carry out tests of individual functions, the interplay between functions, and the overall software project.

In the context of a program with many functions, it is very important to test each function *before* you move on to other functions. The process of writing a program consists of multiple iterations of Polya's four-step process, as we outlined in Chapter 1.

1. Understand the problem.
2. Design an algorithm.
3. Write a program.
4. Look back.

Once you come up with an overall design and identify what functions you need in your program, you should follow the "Design–Program–Test" process in steps 2–4 for *each function individually*. If you do not follow this advice and instead test everything for the first time when you think you are done, it will be very hard to discern where your errors are and you are guaranteed to waste all sorts of time. As you test each function, you are likely to discover situations that you had not thought of previously, sending you back to the drawing board, so to speak. If you discover these issues too late, they may have adverse effects on everything else and throw your whole project into disarray.

We will group our tests for each function in what is known as a ***unit test***. The "unit" in our case will be an individual function, but in general it may be any block of code with a specific purpose. Each unit test will itself be a function, named `test_` followed by the name of the function that we are testing. For example, our unit test function for the `leapYear` function will be named `test_leapYear`. Each unit test function will contain several individual tests, each of which will `assert` that calling the function with a particular set of parameters returns the correct answer.

Let's design a unit test for the `leapYear` function. We will assume that the functions we are testing are checking preconditions with `if` statements rather than assertions. We will start with a few easy tests:

```
def test_leapYear():
    assert leapYear(2024) == True
    assert leapYear(2025) == False

    print('Passed all tests of leapYear!')
```

(Since `leapYear` returns a Boolean value, these could also be written as

**Tangent 5.2: Unit testing frameworks**

Python also provides two modules to specifically facilitate unit testing. The `doctest` module provides a way to incorporate tests directly into function docstrings. The `unittest` module provides a fully featured unit testing framework that automates the unit and regression testing process, and is more similar to the approach we are taking in this section. If you would like to learn more about these tools, visit

`http://docs.python.org/3/library/development.html`

`assert leapYear(2024)` and `assert not leapYear(2025)`.) Add this function to the same file as your `leapYear` and `nextLeapYear` functions. Also include another function that will call all of your unit tests:

```
def test():
    test_leapYear()
```

**Reflection 5.21** *What is printed by the* `assert` *statements in the* `test_leapYear` *function when you call the* `test` *function?*

If the `leapYear` function is working correctly, *nothing* should be printed by the `assert` statements. On the other hand, if the `leapYear` function were to fail one of the tests, the program would abort at that point with an `AssertionError` exception. To see this, change the first assertion to (incorrectly) read `assert leapYear(2024) == False`. Then rerun the program. You should see

```
Traceback (most recent call last):

    ⋮  (order of function calls displayed here)

    assert leapYear(2024) == False
AssertionError
```

This error tells you which assertion failed so that you can track down the problem. (In this case, there is no problem; change the `assert` statement back to the correct version.)

On their own, the results of these two tests do not provide enough evidence to show that the `leapYear` function is correct. As we first discussed back in Section 1.4, we need to choose a variety of tests that are representative of the entire range of possibilities. The input that we use for a particular test is called a ***test case***. As we first saw back on page 38, we can generally divide test cases into *common case*, *boundary cases*, and *corner cases*.

### *Common cases*

First, test the function on several straightforward inputs to make sure that its basic functionality is intact, as we started to do above. Be sure to choose test cases that cover the range of possible inputs and possible outputs.

```
assert leapYear(2024) == True
assert leapYear(2025) == False
assert leapYear(1969) == False
assert leapYear(1900) == False
assert leapYear(40) == True
assert leapYear(11111) == False
```

### *Boundary and corner cases*

A ***boundary case*** is an input that rests on a boundary of the range of legal inputs or on the boundary between different outputs. In the case of the `leapYear` function, we want to test the function on the smallest allowable years. (If there were a maximum year, we would want to test that too.) We also want to test on inputs that are next to each other but result in different answers.

```
assert leapYear(1) == False
assert leapYear(0) == False
assert leapYear(-1) == False
assert leapYear(4) == True

assert leapYear(2019) == False
assert leapYear(2020) == True
assert leapYear(2021) == False
```

**Reflection 5.22** *Does the* `leapYear` *function pass these tests?*

A ***corner case*** is any other kind of rare input that might cause the function to break. For the `leapYear` function, our boundary cases took care of most of these. Thinking up pathological corner cases is an acquired skill that comes with experience. Many companies pay top dollar for programmers whose sole job is to discover corner cases that break their software!

After adding test cases for the `nextLeapYear` function, our program looks like this:

```
# test_leapyear.py

def test_leapYear():
    # omitted here...

def test_nextLeapYear():
    assert nextLeapYear(2020) == 2024     # common cases
    assert nextLeapYear(2025) == 2028
    assert nextLeapYear(1899) == 1904
    assert nextLeapYear(100) == 104
    assert nextLeapYear(11111) == 11112

    assert nextLeapYear(1) == 4           # boundary cases
    assert nextLeapYear(0) == 0
    assert nextLeapYear(-100) == 0
    assert nextLeapYear(2023) == 2024
    assert nextLeapYear(2024) == 2028
```

```
    print('Passed all tests of nextLeapYear!')

def test():
    test_leapYear()
    test_nextLeapYear()

test()
```

Notice that we always call all of the individual test functions from `test()`. This is called ***regression testing***; we will revisit this in the next chapter.

### Testing floats

Special care must be taken when testing functions that return floating point numbers. To see why, consider the following small function.

```
def addFloats(steps):
    total = 0
    for count in range(steps):
        total = total + 0.0001
    return total
```

If we call this function with `addFloats(1000000)`, the loop adds one ten-thousandth one million times, so the answer should be one hundred. However, if we try to test this with `assert addFloats(1000000) == 100.0`, the `assert` will fail because rounding errors caused the value of `total` to be slightly greater than 100. To deal with this inconvenience, we need to always test floating point values within a range instead. In this case, the following `assert` statement is much more appropriate:

```
assert (addFloats(1000000) > 99.9999) and (addFloats(1000000) < 100.0001)
```

The size of the range that you test will depend on the accuracy that is necessary in your particular application.

### Catching exceptions

As you know, when exceptions like `ValueError` or `TypeError` are raised, the default behavior is for the program to abort. However, it is possible to change this behavior by "catching" exceptions. A good application of this is testing whether numeric values are received by `input` functions. For example, consider the first prompt in the `fairnessChecker` function on page 186.

```
ratio = float(input('Maximum fair ceo:employee pay ratio: '))
```

If one mistakenly enters a non-numeric value at this prompt, the `float` function will generate a `ValueError` exception like this:

```
Maximum fair ceo:employee pay ratio: cookies

ValueError: could not convert string to float: 'cookies'
```

To avoid this behavior and print a more helpful error message, we can use a `try/except` statement:

```
try:
    ratio = float(input('Maximum fair ceo:employee pay ratio: '))
except ValueError:
    print('The ratio must be a number.')
    return
```

If an exception is raised while executing the statement(s) in the `try` clause, execution immediately jumps to an `except` clause for that exception. In this case, if a value is entered that cannot be converted by the `float` function, triggering a `ValueError` exception, the `print` statement in the `except` clause will be executed followed by a `return` to end the function since it cannot go forward without a value for `ratio`. If an exception is generated but no matching `except` clause is found, the default behavior (usually aborting the program) is followed. After executing the `except` clause, execution continues normally. If no exception occurs in the `try` clause, then the `except` clause is skipped.

An even nicer solution would be to place the `input` function in a loop to prompt again if an exception is raised.

```
good = False
while not good:
    try:
        ratio = float(input('Maximum fair ceo:employee pay ratio: '))
    except ValueError:
        print('Please enter a number.')
    else:                                   # no exception was raised
        good = True                         # so end the loop
```

The optional `else` clause is executed if no exception is raised in the `try` clause. In this case, if a `ValueError` exception is raised, the `print` in the `except` clause is executed and the prompt is issued again. If no exception is raised, `good` is set to `True`, causing the loop to end.

## Exercises

*For each of the following functions from earlier chapters, (a) write a suitable precondition and postcondition, and (b) add* `assert` *statement(s) to enforce your precondition.*

5.5.1* 
```
import math

def volumeSphere(radius):
    return (4 / 3) * math.pi * (radius ** 3)
```

5.5.2. 
```
def fair(employee, ceo, ratio):
    return (ceo / employee <= ratio)
```

5.5.3. 
```
def middleClass(pay):
    return (pay >= 40000) and (pay <= 65000)
```

5.5.4.
```
def windChill(temperature, windSpeed):
    # Note: only valid for temperatures <= 10 degrees Celsius
    #       and wind speeds above 4.8 km/h.
    chill = 13.12 + 0.6215 * temperature \
                  + (0.3965 * temperature - 11.37) \
                  * windSpeed ** 0.16
    temperature = round(chill)
    return temperature
```

5.5.5*
```
import turtle

def plot(tortoise, n):
    for x in range(-n, n + 1):
        tortoise.goto(x, x * x)
```

5.5.6.
```
def pond(years):
    population = 12000
    for year in range(1, years + 1):
        population = 1.08 * population - 1500
    return population
```

5.5.7.
```
def decayC14(originalAmount, years, dt):
    amount = originalAmount
    k = -0.00012096809434
    numIterations = int(years / dt) + 1
    for i in range(1, numIterations):
        amount = amount + k * amount * dt
    return amount
```

5.5.8.
```
def argh(n):
    return 'A' + ('r' * n) + 'gh!'
```

*Design a thorough unit test for each of the following functions. If you discover errors during your testing, identify and fix them.*

5.5.9*
```
def assignGP(score):
    """Assign a grade point to a score between 0 and 100."""

    if score >= 90:
        return 4
    if score >= 80:
        return 3
    if score >= 70:
        return 2
    if score >= 60:
        return 1
    return 0
```

5.5.10.
```
def power(base, exponent):
    """Compute base ** exponent."""

    count = 0
    result = 1
    while count <= exponent:
        result = result * base
        count = count + 1
    return result
```

5.5.11.
```
def factorial(n):
    """Compute n! = 1 * 2 * 3 * ... * n."""

    result = 1
    for number in range(n):
        result = result * number
    return result
```

5.5.12* Design a thorough unit test for the `volumeSphere` function in Exercise 5.5.1.

5.5.13. Design a thorough unit test for the `windChill` function in Exercise 5.5.4.

5.5.14. Design a thorough unit test for the `decayC14` function in Exercise 5.5.7.

*Add a `try/except` clause within a loop to each of the two following functions to re-prompt for input if converting it to a number causes an exception.*

5.5.15*
```
def daysAlive():
    text = input('How old are you? ')
    age = int(text)
    print('You have been alive for', round(age * 365.25), 'days!')
```

5.5.16.
```
def guessMyNumber():
    myNumber = random.randrange(1, 11)
    guess = int(input('Guess a number from 1 to 10: '))
    if guess == myNumber:
        print('Nice guess!')
    else:
        print('Nope.')
```

5.5.17. An alternative way to deal with a possible division by zero error in the `fair` function on page 182 is to let the division happen and catch the `ZeroDivisionError` exception if it occurs. Modify the function, shown below, in this way. The function should return 0 if division by zero is detected.

```
def fair(employee, ceo, ratio):
    return (ceo / employee) <= ratio
```

## 5.6 GUESS MY NUMBER

In this section, we will design a function to play the classic "I'm thinking of a number" game to practice a bit more with complex conditionals and `while` loops. Here is a first attempt at a function to play the game.

```
import random

def guessingGame(maxGuesses):
    """Plays a guessing game.  The human player tries to guess
       the computer's number from 1 to 100.

    Parameter:
        maxGuesses: the maximum number of guesses allowed

    Return value: None
    """

    secretNumber = random.randrange(1, 101)
    for guesses in range(maxGuesses):
        myGuess = input('Please guess my number: ')
        try:
            myGuess = int(myGuess)
        except ValueError:
            myGuess = 0                 # count this as a miss

        if myGuess == secretNumber:     # win
            print('You got it!')
        else:                           # try again
            print('Nope.  Try again.')
```

The `randrange` function returns a random integer that is at least the value of its first argument, but less than its second argument (similar to the way the `range` function interprets its arguments). After the function chooses a random integer between 1 and 100, it enters a `for` loop that will allow us to guess up to `maxGuesses` times. The function prompts us for our guess with the `input` function, and then assigns the response to `myGuess` as a string. Because we want to interpret the response as an integer, we use the `int` function to convert the string. If `myGuess` cannot be converted to an `int`, we catch the exception and assign `myGuess` to `0` so it will count as a missed guess. Once it has `myGuess`, the function uses the `if/else` statement to tell us whether we have guessed correctly. After this, the loop will give us another guess, until we have used them all up.

**Reflection 5.23** *Try playing the game by calling* `guessingGame(20)`*. Does it work? Is there anything we need to improve?*

You may have noticed three issues:

1. After we guess correctly, unless we have used up all of our guesses, the loop iterates again and gives us another guess. Instead, we want the game to end.
2. It would be much friendlier for the game to tell us whether an incorrect guess is too high or too low.
3. If we do not guess correctly in at most `maxGuesses` guesses, the last thing we see

is `'Nope. Try again.'` before the function ends. But there is no opportunity to try again; instead, it should tell us that we have lost.

We will address these issues in order.

### Ending the game nicely

Our current `for` loop, like all `for` loops, iterates a prescribed number of times. Instead, we want the loop to only iterate for as long as we need another guess. So this is another instance that calls for a `while` loop. In this case, we need the loop to iterate while we have not guessed the secret number, in other words, while `myGuess != secretNumber`. But we also need to stop when we have used up all of our guesses, as counted by the index variable `guesses`. So we also want our `while` loop to iterate while `guesses < maxGuesses`. Since both of these conditions must be `True` for us to keep iterating, our desired `while` loop condition is:

```
while (myGuess != secretNumber) and (guesses < maxGuesses):
```

Because we are replacing our `for` loop with this `while` loop, we will now need to manage the index variable manually. We do this by initializing `guesses = 0` before the loop and incrementing `guesses` in the body of the loop. Here is the updated function with these changes highlighted:

```
def guessingGame(maxGuesses):
    """ (docstring omitted) """

    secretNumber = random.randrange(1, 101)
    myGuess = 0
    guesses = 0
    while (myGuess != secretNumber) and (guesses < maxGuesses):
        myGuess = input('Please guess my number: ')
        try:
            myGuess = int(myGuess)
        except ValueError:
            myGuess = 0
        guesses = guesses + 1           # increment # of guesses
        if myGuess == secretNumber:     # win
            print('You got it!')
        else:                           # try again
            print('Nope.  Try again.')
```

**Reflection 5.24** *Notice that we have also included* `myGuess = 0` *before the loop. Why do we bother to assign a value to* `myGuess` *before the loop? Is there anything special about the value 0? (Hint: try commenting it out.)*

If we comment out `myGuess = 0`, we will see the following error on the line containing the `while` loop:

```
UnboundLocalError: local variable 'myGuess' referenced before assignment
```

This error means that we have referred to an unknown variable named `myGuess`. The

name is unknown to the Python interpreter because we had not defined it before it was first referenced in the `while` loop condition. Therefore, we need to initialize `myGuess` before the `while` loop, so that the condition makes sense the first time it is tested. Recall from Section 4.3 that this was one of the two important things to remember about `while` loops. We initialize `myGuess` to 0, a value that cannot be the secret number, to make sure the loop iterates at least once.

The second important consideration for `while` loops is to ensure that the condition will eventually become false. For the `and` expression in this `while` loop to become false, either `(myGuess != secretNumber)` must be false or `(guesses < maxGuesses)` must be false. This reasoning is the same as De Morgan's first law that we discussed in the previous section.

**Reflection 5.25** *How do the statements in the body of the loop ensure that eventually* `(myGuess != secretNumber)` *or* `(guesses < maxGuesses)` *will be* `False`*?*

Prompting for a new guess creates the opportunity for the first part to become `False`, while incrementing `guesses` ensures that the second part will eventually become `False`. Therefore, we cannot have an infinite loop.

### Friendly hints

Inside the loop, we currently handle two cases: (1) we win, and (2) we do not win but get another guess. To be friendlier, we should split the second case into two subcases: (2a) our guess was too low, and (2b) our guess was too high. We can accomplish this by replacing the not-so-friendly `print('Nope. Try again.')` with another `if/else` that decides between the two new subcases:

```
if myGuess == secretNumber:                 # win
    print('You got it!')
else:                                       # try again
    if myGuess < secretNumber:              # too low
        print('Too low.  Try again.')
    else:                                   # too high
        print('Too high.  Try again.')
```

Now, if `myGuess == secretNumber` is false, we execute the first `else` clause, the body of which is the new `if/else` construct. If `myGuess < secretNumber` is true, we print that the number is too low; otherwise, we print that the number is too high.

**Reflection 5.26** *Do you see a way in which the conditional construct above can be simplified?*

The conditional construct above is really just equivalent to a decision between three disjoint possibilities: (a) the guess is equal to the secret number, (b) the guess is less than the secret number, or (c) the guess is greater than the secret number. In other words, it is equivalent to:

```
if myGuess == secretNumber:          # win
    print('You got it!')
elif myGuess < secretNumber:         # too low
    print('Too low.  Try again.')
else:                                # too high
    print('Too high.  Try again.')
```

### A proper win/lose message

Now inside the loop, we currently handle three cases: (a) we win, and (b) our guess is too low and we get another guess, and (c) our guess is too high and we get another guess. But we are missing the case in which we run out of guesses. In this situation, we want to print something like `'Too bad. You lose.'` instead of one of the "try again" messages.

**Reflection 5.27** *How can we augment the* `if/elif/else` *statement so that it correctly handles all four cases?*

In the game, if we incorrectly use our last guess, then two things must be true just before the `if` condition is tested: `myGuess` is not equal to `secretNumber` and `guesses` is equal to `maxGuesses`. So we can incorporate this condition into the `if/elif/else` too:

```
if (myGuess != secretNumber) and (guesses == maxGuesses):   # lose
    print('Too bad.  You lose.')
elif myGuess == secretNumber:        # win
    print('You got it!')
elif myGuess < secretNumber:         # too low
    print('Too low.  Try again.')
else:                                # too high
    print('Too high.  Try again.')
```

Notice that we have made the previous first `if` statement into an `elif` statement because we only want one of the four messages to be printed. However, here is an alternative structure that is more elegant:

```
if myGuess == secretNumber:          # win
    print('You got it!')
elif guesses == maxGuesses:          # lose
    print('Too bad.  You lose.')
elif myGuess < secretNumber:         # too low
    print('Too low.  Try again.')
else:                                # too high
    print('Too high.  Try again.')
```

By placing the new condition second, we can leverage the fact that, if we get to the first `elif`, we already know that `myGuess != secretNumber`. Therefore, we do not need to include it explicitly.

There is a third way to handle this situation that is perhaps even more elegant. Notice that both of the first two conditions are going to happen at most once, and at the end of the program. So it might make more sense to put them *after* the loop.

Doing so also exhibits a nice parallel between these two events and the two parts of the `while` loop condition. As we discussed earlier, the negation of the `while` loop condition is

```
(myGuess == secretNumber) or (guesses >= maxGuesses)
```

So when the loop ends, at least one of these two things is true. Notice that these two events are exactly the events that define a win or a loss: if the first part is true, then we won; if the second part is true, we lost. So we can move the win/loss statements after the loop, and decide which to print based on which part of the `while` loop condition became false:

```
if myGuess == secretNumber:        # win
    print('You got it!')
else:                              # lose
    print('Too bad.  You lose.')
```

In the body of the loop, with these two cases gone, we will now need to check if we still get another guess (mirroring the `while` loop condition) before we print one of the "try again" messages:

```
if (myGuess != secretNumber) and (guesses < maxGuesses):
    if myGuess < secretNumber:          # too low
        print('Too low.  Try again.')
    else:                               # too high
        print('Too high.  Try again.')
```

**Reflection 5.28** *Why is it not correct to combine the two* `if` *statements above into a single statement like the following?*

```
if (myGuess != secretNumber) and (guesses < maxGuesses) \
                             and (myGuess < secretNumber):
    print('Too low.  Try again.')
else:
    print('Too high.  Try again.')
```

*Hint: what does the function print when* `guesses < maxGuesses` *is false and* `myGuess < secretNumber` *is true?*

All of these changes are incorporated into the final game shown below. As you play it, think about what the best strategy is. Exercise 5.6.7 asks you to write a Monte Carlo simulation to compare three different strategies for playing the game.

```
import random

def guessingGame(maxGuesses):
    """ (docstring omitted) """

    secretNumber = random.randrange(1, 101)
    myGuess = 0
    guesses = 0
    while (myGuess != secretNumber) and (guesses < maxGuesses):
        myGuess = input('Please guess my number: ')
        try:
            myGuess = int(myGuess)
        except ValueError:
            myGuess = 0
        guesses = guesses + 1         # increment # of guesses used

        if (myGuess != secretNumber) and (guesses < maxGuesses):
            if myGuess < secretNumber:          # guess is too low
                print('Too low.  Try again.')   #   give a hint
            else:                               # guess is too high
                print('Too high.  Try again.')  #   give a hint

    if myGuess == secretNumber:        # win
        print('You got it!')
    else:                              # lose
        print('Too bad.  You lose.')

def main():
    guessingGame(10)

main()
```

## Exercises

5.6.1* Write a function

```
ABC()
```

that prompts for a choice of A, B, or C and keeps prompting until it receives one of those strings. Your function should return the final choice.

5.6.2. Write a function

```
numberPlease()
```

that prompts for an integer between 1 and 100 (inclusive) and continues to prompt until it receives a number within this range. Your function should return the final number.

5.6.3. Write a function

```
differentNumbers()
```

that prompts for two different numbers. The function should use a `while` loop

to keep prompting for a pair of numbers until the two numbers are different, and then print the final numbers.

5.6.4. Write a function

```
rockPaperScissorsLizardSpock(player1, player2)
```

that decides who wins in a game of rock-paper-scissors-lizard-Spock.[2] Each of `player1` and `player2` is a string with value `'rock'`, `'paper'`, `'scissors'`, `'lizard'`, or `'Spock'`. The function should return `1` if player 1 wins, `-1` if player 2 wins, or `0` if they tie. Test your function by playing the game with the following main program:

```
def main():
    player1 = input('Player1: ')
    player2 = input('Player2: ')
    outcome = rockPaperScissorsLizardSpock(player1, player2)
    if outcome == 1:
        print('Player 1 wins!')
    elif outcome == -1:
        print('Player 2 wins!')
    else:
        print('Player 1 and player 2 tied.')
```

5.6.5* Write a function

```
yearsUntilDoubled(amount, rate)
```

that returns the number of years until `amount` is doubled when it earns the given rate of interest, compounded annually. Use a `while` loop.

5.6.6. The *hailstone numbers* are a sequence of numbers generated by the following simple algorithm. First, choose any positive integer. Then, repeatedly follow this rule: if the current number is even, divide it by two; otherwise, multiply it by three and add one. For example, suppose we choose the initial integer to be 3. Then this algorithm produces the following sequence: $3, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1, 4, 2, 1, \ldots$ For every initial integer ever tried, the sequence always reaches one and then repeats the sequence 4,2,1 forever after. Interestingly, however, no one has ever *proven* that this pattern holds for every integer! Write a function

```
hailstone(start)
```

that prints the hailstone number sequence starting from the parameter `start`, until the value reaches one. Your function should return the number of integers in your sequence. For example, if `start` were `3`, the function should return `8`. (Use a `while` loop.)

5.6.7. In this exercise, you will design a Monte Carlo simulation to compare the effectiveness of three strategies for playing the guessing game. Each of these strategies will be incorporated into the guessing game function we designed in this chapter, but instead of checking whether the player wins or loses, the function will continue until the number is guessed, and then return the number of guesses used. We will also make the maximum possible secret number a parameter, so that we can compare the results for different ranges of secret numbers.

[2] See `http://en.wikipedia.org/wiki/Rock-paper-scissors-lizard-Spock` for the rules.

The first strategy is to make a random guess each time, ignoring any previous guesses:

```
def guessingGame1(maxNumber):
    """Play the guessing game by making random guesses."""

    secretNumber = random.randrange(1, maxNumber + 1)
    myGuess = 0
    guesses = 0
    while (myGuess != secretNumber):
        myGuess = random.randrange(1, maxNumber + 1)
        guesses = guesses + 1
    return guesses
```

The second strategy is to avoid duplicate guesses by trying every number from 1 to 100. This function is identical, except it replaces the red statement above as follows:

```
def guessingGame2(maxNumber):
    """Play the guessing game by making incremental guesses."""
    ⋮
        myGuess = myGuess + 1
    ⋮
```

Finally, the third strategy uses previous outcomes to narrow in on the secret number:

```
def guessingGame3(maxNumber):
    """Play the guessing game intelligently by narrowing in
       on the secret number."""

    secretNumber = random.randrange(1, maxNumber + 1)
    myGuess = 0
    low = 1
    high = maxNumber
    guesses = 0
    while (myGuess != secretNumber):
        myGuess = (low + high) // 2
        guesses = guesses + 1
        if myGuess < secretNumber:
            low = myGuess + 1
        elif myGuess > secretNumber:
            high = myGuess - 1
    return guesses
```

Write a Monte Carlo simulation to compare the expected (i.e., average) behavior of these three strategies. Use a sufficiently high number of trials to get consistent results. Similarly to what we did in Section 5.1, run your simulation for a range of maximum secret numbers, specifically $5, 10, 15, \ldots, 100$, and plot the average number of guesses required by each strategy for each maximum secret number. (The $x$-axis of your plot will be the maximum secret number and the $y$-axis will be the average number of guesses.) Explain the results. In general, how many guesses on average do you think each strategy requires to guess a secret number between 1 and $n$?

## 5.7 SUMMARY AND FURTHER DISCOVERY

In previous chapters, we designed deterministic algorithms that did the same thing every time we executed them, if we gave them the same inputs. Giving those algorithms different arguments, of course, could change their behavior, whether it be drawing a different size shape, modeling a different population, or experimenting with a different investment scenario. In this chapter, we started to investigate a new class of algorithms that can change their behavior "on the fly," so to speak. These algorithms all make choices using *Boolean logic*, the same Boolean logic on which computers are fundamentally based. By combining *comparison operators* and *Boolean operators*, we can characterize any decision. By incorporating these Boolean expressions into *conditional statements* (`if/elif/else`) and *conditional loops* (`while`), we vastly increase the diversity of algorithms that we can design. These are fundamental techniques that we will continue to use and develop over the next several chapters, as we start to work with textual and numerical data that we read from files and download from the web.

### Notes for further discovery

This chapter's epigraph is a famous "Yogiism," from Hall of Fame catcher, coach, and manager Yogi Berra [8].

If you would like to learn more about Robert Brown's experiments, and the history and science behind them, visit the following website, titled "What Brown Saw and You Can Too."

`http://physerver.hamilton.edu/Research/Brownian/index.html`

*The Drunkard's Walk* by Leonard Mlodinow [41] is a very accessible book about how randomness and chance affect our lives. For more information about generating random numbers, and the differences between PRNGs and true random number generators, visit

`https://www.random.org/randomness/` .

The Park-Miller random number generator is due to Keith Miller and the late Steve Park [46].

The Roper Center for Public Opinion Research, at Cornell University, maintains some helpful educational resources about random sampling and errors in the context of public opinion polling at

`https://ropercenter.cornell.edu/learn/polling-and-public-opinion` .

## *5.8 PROJECTS

This section is available on the book website.