number is an integer greater than 1 that is evenly divisible only by itself and 1. For example, 2, 3, 5, and 111,119 are primes, and 4, 6, 8 and 62,710,561 are not primes.

The simplest way to find out if an integer, x, greater than 3 is prime, is to divide x by each integer between 2 and, x-1. If the remainder of any of those divisions is 0, x is not prime, otherwise x is prime. The code in Figure 3-2 implements that approach. It first asks the user to enter an integer, converts the returned string to an int, and assigns that integer to the variable x. It then sets up the initial conditions for an exhaustive enumeration by initializing guess to 2 and the variable smallest_divisor to None—indicating that until proven otherwise, the code assumes that x is prime.

The exhaustive enumeration is done within a for loop. The loop terminates when either all possible integer divisors of x have been tried or it has discovered an integer that is a divisor of x.

After exiting the loop, the code checks the value of smallest_divisor and prints the appropriate text. The trick of initializing a variable before entering a loop, and then checking whether that value has been changed upon exit, is a common one.

```
# Test if an int > 2 is prime. If not, print smallest divisor
x = int(input('Enter an integer greater than 2: '))
smallest_divisor = None
for guess in range(2, x):
    if x%guess == 0:
        smallest_divisor = guess
        break
if smallest_divisor != None:
    print('Smallest divisor of', x, 'is', smallest_divisor)
else:
    print(x, 'is a prime number')
```

Figure 3-2 Using exhaustive enumeration to test primality

Finger exercise: Change the code in Figure 3-2 so that it returns the largest rather than the smallest divisor. Hint: if y*z = x and y is the smallest divisor of x, z is the largest divisor of x.

The code in Figure 3-2 works, but is unnecessarily inefficient. For example, there is no need to check even numbers beyond 2, since if an integer is divisible by any even number, it is divisible by 2. The code in Figure 3-3 takes advantage of this fact by first testing whether x is an even number. If not, it uses a loop to test whether x is divisible by any odd number.

While the code in Figure 3-3 is slightly more complex than the code in Figure 3-2, it is considerably faster, since half as many numbers are checked within the loop. The opportunity to trade code complexity for runtime efficiency is a common phenomenon. But faster does not always mean better. There is a lot to be said for simple code that is obviously correct and still fast enough to be useful.

```
# Test if an int > 2 is prime. If not, print smallest divisor
x = int(input('Enter an integer greater than 2: '))
smallest_divisor = None
if x%2 == 0:
    smallest_divisor = 2
else:
    for guess in range(3, x, 2):
        if x%guess == 0:
            smallest_divisor = guess
            break
if smallest_divisor != None:
    print('Smallest divisor of', x, 'is', smallest_divisor)
else:
    print(x, 'is a prime number')
```

Figure 3-3 A more efficient primality test

Finger exercise: Write a program that asks the user to enter an integer and prints two integers, root and pwr, such that 1 < pwr < 6 and root**pwr is equal to the in-

left. If it is too small, we know that the answer must lie to the right. We then repeat the process on the smaller interval. Figure 3-5 contains an implementation and test of this algorithm.

```
epsilon = 0.01
num_guesses, low = 0, 0
high = max(1, x)
ans = (high + low)/2
while abs(ans**2 - x) >= epsilon:
    print('low =', low, 'high =', high, 'ans =', ans)
    num_guesses += 1
    if ans**2 < x:
        low = ans
    else:
        high = ans
    ans = (high + low)/2
print('number of guesses =', num_guesses)
print(ans, 'is close to square root of', x)
```

Figure 3-5 Using bisection search to approximate square root

When run for x = 25, it prints

```
low = 0.0 high = 25 ans = 12.5
low = 0.0 high = 12.5 ans = 6.25
low = 0.0 high = 6.25 ans = 3.125
low = 3.125 high = 6.25 ans = 4.6875
low = 4.6875 high = 6.25 ans = 5.46875
low = 4.6875 high = 5.46875 ans = 5.078125
low = 4.6875 high = 5.078125 ans = 4.8828125
low = 4.8828125 high = 5.078125 ans = 4.98046875
low = 4.98046875 high = 5.078125 ans = 5.029296875
low = 4.98046875 high = 5.029296875 ans = 5.0048828125
low = 4.98046875 high = 5.0048828125 ans = 4.99267578125
low = 4.99267578125 high = 5.0048828125 ans = 4.998779296875
low = 4.998779296875 high = 5.0048828125 ans = 5.0018310546875
numGuesses = 13
5.00030517578125 is close to square root of 25
```

Notice that it finds a different answer than our earlier algorithm. That is perfectly fine, since it still meets the problem's specification.

More important, notice that at each iteration of the loop, the size of the space to be searched is cut in half. For this reason, the algorithm is called **bisection search**. Bisection search is a huge improvement over our earlier algorithm, which reduced the search space by only a small amount at each iteration.

Let us try `x = 123456` again. This time the program takes only 30 guesses to find an acceptable answer. How about `x = 123456789`? It takes only 45 guesses.

There is nothing special about using this algorithm to find square roots. For example, by changing a couple of 2's to 3's, we can use it to approximate a cube root of a nonnegative number. In Chapter 4, we introduce a language mechanism that allows us to generalize this code to find any root.

Bisection search is a widely useful technique for many things other than finding roots. For example, the code in Figure 3-6 uses bisection search to find an approximation to the log base 2 of x (i.e., a number, ans, such that 2**ans is close to x). It is structured exactly like the code used to find an approximation to a square root. It first finds an interval containing a suitable answer, and then uses bisection search to efficiently explore that interval.

```
# Find lower bound on ans
lower_bound = 0
while 2**lower_bound < x:
    lower_bound += 1
low = lower_bound - 1
high = lower_bound + 1
# Perform bisection search
ans = (high + low)/2
while abs(2**ans - x) >= epsilon:
    if 2**ans < x:
        low = ans
    else:
        high = ans
    ans = (high + low)/2
print(ans, 'is close to the log base 2 of', x)
```

Figure 3-6 Using bisection search to estimate log base 2

Bisection search is an example of a **successive approximation** method. Such methods work by making a sequence of guesses with the property that each guess is closer to a correct answer than the previous guess. We will look at an important successive approximation algorithm, Newton's method, later in this chapter.

Finger exercise: What would the code in Figure 3-5 do if `x = -25`?

Finger exercise: What would have to be changed to make the code in Figure 3-5 work for finding an approximation to the cube root of both negative and positive numbers? Hint: think about changing `low` to ensure that the answer lies within the region being searched.

Finger exercise: The Empire State Building is 102 stories high. A man wanted to know the highest floor from which he could drop an egg without the egg breaking. He proposed to drop an egg from the top floor. If it broke, he would go down a floor, and try it again. He would do this until the egg did not break. At worst, this method requires 102 eggs. Implement a method that at worst uses seven eggs.

3.3 A Few Words about Using Floats

Most of the time, numbers of type `float` provide a reasonably good approximation to real numbers. But "most of the time" is not all of the time, and when they don't, it can lead to surprising consequences. For example, try running the code

```
x = 0.0
for i in range(10):
    x = x + 0.1
if x == 1.0:
    print(x, '= 1.0')
else:
    print(x, 'is not 1.0')
```

Perhaps you, like most people, find it surprising that it prints,

```
0.9999999999999999 is not 1.0
```


Why does it get to the `else` clause in the first place?

To understand why this happens, we need to understand how floating-point numbers are represented in the computer during a computation. To understand that, we need to understand **binary numbers.**

When you first learned about decimal numbers—i.e., numbers base 10—you learned that any decimal number can be represented by a sequence of the digits 0123456789. The rightmost digit is the 10^0 place, the next digit towards the left is the 10^1 place, etc. For example, the sequence of decimal digits 302 represents $3*100 + 0*10 + 2*1$. How many different numbers can be represented by a sequence of length n? A sequence of length 1 can represent any one of 10 numbers (0-9); a sequence of length 2 can represent 100 different numbers (0-99). More generally, a sequence of length n can represent 10^n different numbers.

Binary numbers—numbers base 2—work similarly. A binary number is represented by a sequence of digits each of which is either 0 or 1. These digits are often called **bits**. The rightmost digit is the 2^0 place, the next digit towards the left is the 2^1 place, etc. For example, the sequence of binary digits 101 represents $1*4 + 0*2 + 1*1 = 5$. How many different numbers can be represented by a sequence of length n? 2^n.

Finger exercise: What is the decimal equivalent of the binary number 10011?

Perhaps because most people have ten fingers, we like to use decimals to represent numbers. On the other hand, all modern computer systems represent numbers in binary. This is not because computers are born with two fingers. It is because it is easy to build hardware **switches**, i.e., devices that can be in only one of two states, on or off. That computers use a binary representation and people a decimal representation can lead to occasional cognitive dissonance.

In modern programming languages non-integer numbers are implemented using a representation called **floating point**. For the moment, let's pretend that the internal representation is in decimal. We would represent a number as a pair of integers—the **significant digits** of the number and an **exponent**. For example,

part of a program that compares the efficiency of Newton–Raphson and bisection search. (You should discover that Newton–Raphson is far more efficient.)

3.5 Terms Introduced in Chapter

decrementing function
guess-and-check
exhaustive enumeration
approximation
total ordering
bisection search
successive approximation
binary numbers
bit
switch
floating point
significant digits
exponent
precision
rounding
Newton–Raphson
polynomial
coefficient
degree
root

4 FUNCTIONS, SCOPING, AND ABSTRACTION

So far, we have introduced numbers, assignments, input/output, comparisons, and looping constructs. How powerful is this subset of Python? In a theoretical sense, it is as powerful as you will ever need, i.e., it is Turing complete. This means that if a problem can be solved using computation, it can be solved using only those linguistic mechanisms you have already seen.

But just because something can be done, doesn't mean it should be done! While any computation can, in principle, be implemented using only these mechanisms, doing so is wildly impractical. In the last chapter, we looked at an algorithm for finding an approximation to the square root of a positive number, see Figure 4-1.

```
#Find approximation to square root of x
if x < 0:
    print('Does not exist')
else:
    low = 0
    high = max(1, x)
    ans = (high + low)/2
    while abs(ans**2 - x) >= epsilon:
        if ans**2 < x:
            low = ans
        else:
            high = ans
        ans = (high + low)/2
    print(ans, 'is close to square root of', x)
```

Figure 4-1 Using bisection search to approximate square root of x