CHAPTER 1

# How to Solve It

> We need to do away with the myth that computer science is about computers. Computer science is no more about computers than astronomy is about telescopes, biology is about microscopes or chemistry is about beakers and test tubes. Science is not about tools, it is about how we use them and what we find out when we do.
>
> Michael R. Fellows and Ian Parberry
> *Computing Research News (1993)*

> It has often been said that a person does not really understand something until after teaching it to someone else. Actually a person does not really understand something until after teaching it to a computer, i.e., expressing it as an algorithm.
>
> Donald E. Knuth
> *American Scientist (1973)*

COMPUTERS now touch almost every facet of our daily lives, whether we are consciously aware of them or not. Computers have changed the way we learn, communicate, shop, travel, receive healthcare, and entertain ourselves. They are embedded in virtually everything, from major feats of engineering like airplanes, spaceships, and factories to everyday items like microwaves, cameras, and toothbrushes. In addition, all of our critical infrastructure—utilities, transportation, finance, communication, healthcare, law enforcement—relies upon computers.

Since computers are the most versatile tools ever invented, it should come as no surprise that they are also employed throughout academia in the pursuit of new knowledge. Social scientists use computational models to better understand social networks, epidemics, population dynamics, markets, and auctions. Humanists use computational tools to gain insight into literary trends, authorship of ancient texts, and the macroscopic significance of historical records. Artists are increasingly incorporating digital technologies into their compositions and performances. Natural

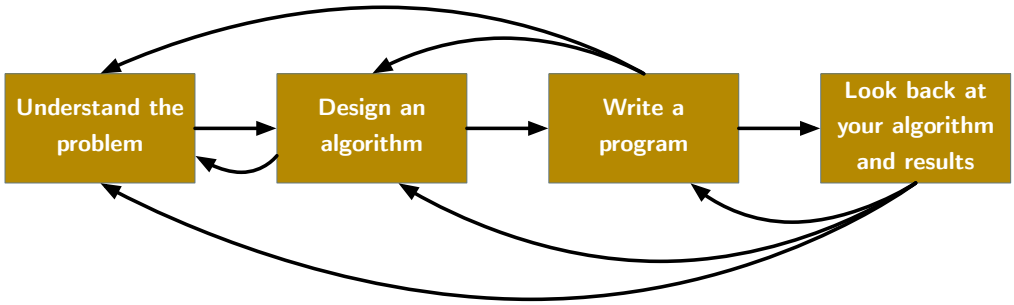



Figure 1.1 A simplified view of the problem solving process.

scientists use computers to collect and analyze immense quantities of data to make discoveries in environmental science, genomics, particle physics, neuroscience, pharmacology, and medicine.

But computers are neither productive nor consequential on their own. All of the computers now driving civilization, for good or ill, were taught by humans. Computers are amplifiers of human ingenuity. Without us, they are just dumb machines.

The goal of this book is to empower *you* to teach computers to solve problems and make discoveries. Computational problem solving is a process that you will find both familiar and new. We all solve problems every day and employ a variety of strategies in doing so. Some of these strategies, like breaking big problems into smaller, more manageable ones, are also fundamental to solving problems with a computer. Where computational problem solving is different stems from computers' lack of intellect and intuition. Computers will only do what you tell them to and nothing more. They cannot tolerate ambiguity or intuit your intentions. Computational problem solving, by necessity, must be more precise and intentional than you may be used to. The payoff though, paraphrasing Donald Knuth[1], is that teaching a computer to do something can also dramatically deepen our understanding of that thing.

The problem solving process that we will outline in this chapter is inspired by *How to Solve It*, a short book written by mathematician George Polya [50] in 1945. Polya's problem solving framework, having withstood the test of time, consists of four steps:

1. *First, understand the problem. What is the unknown? What are the data? What is the condition?*
2. *Second, devise a plan to solve the problem.*
3. *Third, carry out your plan, checking each step.*
4. *Fourth, look back. Check the result. Can you derive the result differently?*

These four steps, with some modifications, can be applied just as well to computational problem solving, as illustrated in Figure 1.1. In the first step, we make

[1] You can learn more about Donald Knuth at the end of this chapter.

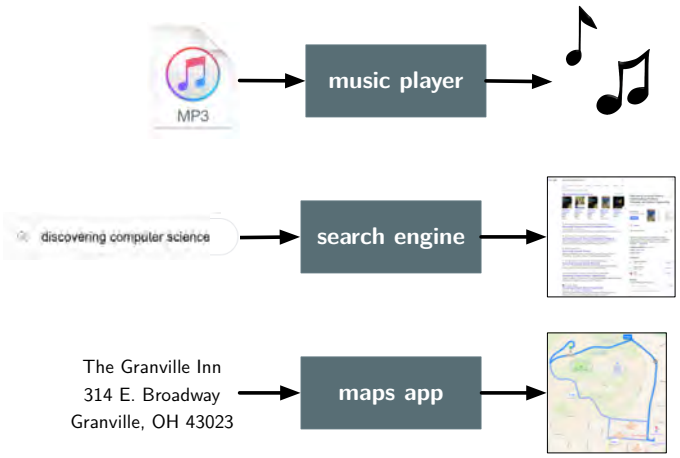



Figure 1.2 Some examples of computational problems.

sure that we understand the problem to be solved. In the second step, we devise an *algorithm*, a sequence of steps to solve the problem. In the third step, we translate our algorithm into a correct *program* that can be carried out by a computer. We will be using a programming language called Python throughout this book to write programs. Finally, in the fourth step, we look back on our results and ask whether we can improve them or the algorithm that derived them. Notice that this process is often not linear. Work on one step can refine our understanding of a previous step and nudge us backward, not unlike the process of writing a paper.

This chapter serves as a framework for your learning throughout the rest of the book. Each subsequent chapter will flesh out aspects of these steps and make them more concrete by focusing on new types of computational problems and the techniques used to solve them.

## 1.1 UNDERSTAND THE PROBLEM

> *First, understand the problem. What is the unknown? What are the data? What is the condition?*

In computer science, we think of a ***problem*** as a relationship between some initial information, an ***input***, and some desired result, the ***output***. To solve the problem, we need to teach a computer how to transform the input into the output. The steps that the computer takes to do this are called a ***computation***. In Polya's language, the "data" is the input, the "unknown" is the output, and the "condition" is the relationship between the two.

Figure 1.2 illustrates three common computational problems. In each, an input enters on the left and a corresponding output exits on the right. In between, a computation transforms the input into the correct output. When you listen to a song, your music

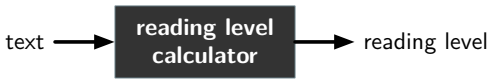
player performs a computation to convert a digital sound file (input) into a sound pattern that can be reproduced by your headphones (output). When you submit a web search request (input), your computer, and many others across the Internet, perform computations to get you results (outputs). And when you use an app on your phone to get directions, it computes the directions (output) based on your current position and desired destination (inputs).

Inputs and outputs are probably familiar to you from high school algebra. When you were given an expression like $y = 18x + 31$ or $f(x) = 18x + 31$, you may have thought about the variable $x$ as a representation of the input and $y$, or $f(x)$, as a representation of the output. In this example, when the input is $x = 2$, the output is $y = 67$, or $f(x) = 67$. The arithmetic that turns $x$ into $y$ is a very simple (and boring) example of a computation.

**Reflection 1.1** *What kinds of problems are you interested in? What are their inputs and outputs? Are the inputs and outputs, as you have defined them, sufficient to define the problems completely?*

## A first problem: computing reading level

Suppose you are a teacher who wants to evaluate whether some text is at an appropriate grade level for your class. In other words, you want to solve the problem illustrated below.




The input and output for this problem seem straightforward. But they actually aren't; once you start thinking carefully about the problem, you realize there are many questions that need to be answered. For example, are there any restrictions or special conditions associated with the input? What kinds of texts are we talking about? Should the solution work equally well for children's books, newspaper articles, scientific papers, and technical manuals? For what language(s) should the solution work? In what electronic format do the texts need to be? Is there a minimum or maximum length requirement for the text? It is important to formulate these kinds of questions and seek any needed clarifications right away; it is much better to do so immediately than to wait until you have spent a lot of time working on the wrong problem!

The same sorts of questions should be asked about the output. How is a reading level represented? Is an integer value corresponding to a school year? Or can it be a fraction? To what educational system should the grade levels correspond? Are their minimum and/or maximum allowed values? Once you have answers to your questions, it is a good idea to re-explain the problem back to the poser, either orally or in writing. The feedback you get from this exercise might identify additional points of misunderstanding. You might also draw a picture and work out some examples by hand to make sure you understand all of the requirements and subtleties involved.

We will answer these questions by clarifying that the solution should work for any English language text, available as a plain text file like those on Project Gutenberg.[2] The output will be a number like 4.2, indicating that the text is appropriate for a student who has completed 2/10 of fourth grade in the U.S. educational system. Negative reading level values will not make sense in this system, but any positive number will be acceptable if we interpret the number to mean the number of years of education required to understand the text.

### Functional abstraction

A problem at this stage, before we know how to solve it, is an example of a ***functional abstraction***.

> *A functional abstraction describes how to use a tool or technology without necessarily providing any knowledge about how it works.*

In other words, a functional abstraction is a "black box" that we know how to use effectively, without necessarily understand what is happening inside the box to produce the output. In the case of the reading level problem, now that we have a better handle on the specifics, if we had a black box that computed the reading level, we would know how to use it, even without understanding *how* the output was computed. Similarly, to use each of the technologies illustrated in Figure 1.2 we do not need to understand *how* the underlying computation transforms the input to the output.

We exist in a world of functional abstractions that we usually take for granted. We even think about our own bodies in terms of abstractions. Move your fingers. Did you need to understand how your brain triggered your nervous and musculoskeletal systems to make that happen? As far as most of us are concerned, a car is also an abstraction. To drive a car, do you need to know how turning the steering wheel turns the car or pushing the accelerator makes it go faster? We understand *what* should happen when we do these things, but not necessarily *how* they happen. Without abstractions, we would be paralyzed by an avalanche of minutiae.

**Reflection 1.2** *Imagine that it was necessary to understand how your phone works in order to use it. Or a car. Or a computer. How would this affect your ability to use these technologies?*

New technologies and automation have introduced new functional abstractions into everyday life. Our food supply is a compelling example of this. Only a few hundred years ago, our ancestors knew exactly where their food came from. Inputs of hard work and suitable weather produced outputs of grain and livestock to sustain a family. In modern times, we input money and get packaged food; the origins of our food have become much more abstract.

[2] Project Gutenberg (`http://www.gutenberg.org`) is a library of freely available classic literature with expired U.S. copyrights. The books are available in a variety of formats, but we will be interested in those in a plain text format like the version of *Walden* by Henry David Thoreau at `http://www.gutenberg.org/files/205/205-0.txt`.

**Reflection 1.3** *Think about a common functional abstraction that you use regularly, such as your phone or a credit card. How has this functional abstraction changed over time? Can you think of instances in which better functional abstractions have enhanced our ability to use a technology?*

Functional abstraction is a very important idea in computer science. In the next section, we will demonstrate how more complex problems are solved by breaking them into smaller functional abstractions that we can solve and then recombine into a solution for the original problem.

### Exercises

1.1.1. What is a problem in your life that you have to solve regularly? Define the input and output of the problem well enough for someone else to propose an algorithm to solve it. Here is an example.

Problem: scan pages in a book
Inputs: a book and page numbers to scan
Output: one PDF file containing all of the pages, one physical page per page in the file, full color, text recognized

1.1.2. What information is missing from each of the inputs and/or outputs of the following problem definitions? In each case, assume that you know how to complete the task given enough information about the input and output.

(a)* Problem: Make brownies
Inputs: butter, sugar, eggs, vanilla, cocoa powder, flour, salt, baking powder
Output: brownies

(b) Problem: Dig a hole
Inputs: a shovel
Output: a hole (of course)

(c) Problem: Plant a vegetable garden
Inputs: seeds
Output: a planted garden plot

1.1.3. Describe three examples from your everyday life in which an abstraction is beneficial. Explain the benefits of each abstraction versus what life would be like without it.

## 1.2 DESIGN AN ALGORITHM

*Second, devise a plan to solve the problem.*

To compute reading level, we will use the well-known *Flesch-Kincaid grade level score*, which approximates the grade level of a text using the formula

$$0.39 \times \text{average words per sentence} + 11.8 \times \text{average syllables per word} - 15.59\,.$$

**Reflection 1.4** *To better understand how the Flesch-Kincaid grade level formula works, apply it to the first epigraph (Fellows and Parberry) at the beginning of this chapter. What does the formula output as the grade level of this quote?*

The 3 sentences in the quote contain 14, 24, and 20 words, respectively, so the average number of words per sentence is $(14 + 24 + 20)/3 \approx 19.33$. There are 90 total syllables in the quote's 58 words, so the average number of syllables per word is $90/58 \approx 1.55$. Plugging these values into the formula, we get

$$0.39 \times 19.33 + 11.8 \times 1.55 - 15.59 \approx 10.24.$$

So the formula says that the quote is at about a tenth grade reading level.

You may be surprised to hear that this formula does not provide nearly enough detail for a computer to carry it out. *You* can figure out how to find the average number of words per sentence and the average number of syllables per word, but a computer definitely cannot without a lot more help. Instead, this formula is more appropriately thought of as a more detailed description of what the output "reading level" means.

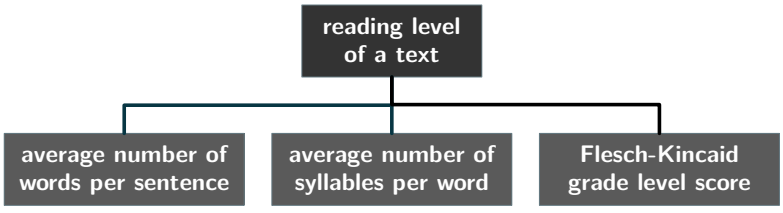
To teach a computer how to apply the Flesch-Kincaid formula to any text, we need to replace the black box with a detailed sequence of steps that transforms the input (text) into the correct output (reading level). This sequence of steps is called an ***algorithm***. An algorithm is how we teach a computer how to solve a problem.

### Take it from the top

To make designing an algorithm more manageable, we can decompose it into simpler ***subproblems***. A subproblem is an easier problem that, once solved, will make solving the original problem more straightforward.

**Reflection 1.5** *Look at the Flesch-Kincaid formula again. What are the two subproblems we need to solve before we can apply the formula?*

As we saw when we applied the formula, we need to determine two things about the text: the average number of words per sentence and the average number of syllables per word. So these are two subproblems of the overall problem, together with the actual calculation of the Flesch-Kincaid grade level score. We can represent this as follows.




This diagram shows us that there are three subproblems involved in solving the reading level problem. If we had functional abstractions, "black boxes," for these three subproblems, then we could easily solve the reading level problem by getting the

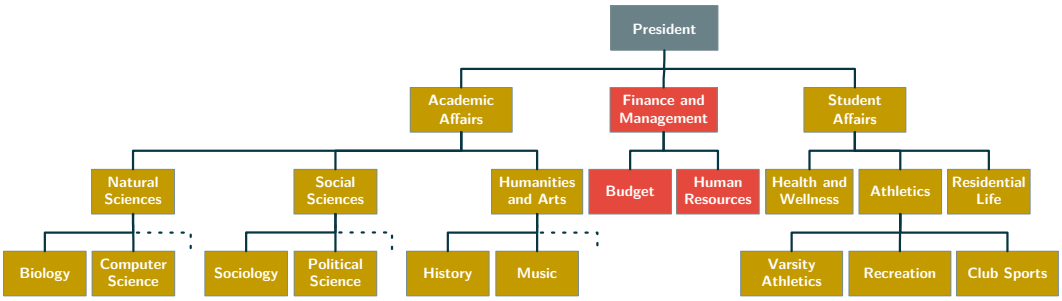



Figure 1.3 A simplified organizational chart of a hypothetical college.

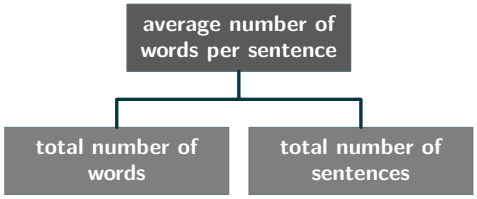
required values from the two leftmost subproblems and then plugging their outputs into the third subproblem. This technique is called ***top-down design*** because it involves starting from the top, the problem to be solved, and then breaking it down into smaller pieces. The final result of this process is called a ***functional decomposition***.

Top-down design and functional decomposition are commonly used to make all sorts of things more manageable. For example, suppose you are the president of a college. Because you cannot effectively manage every detail of such a large organization, you hire a vice president to oversee each of three divisions, as illustrated in Figure 1.3. You expect each vice president to keep you informed about the general activity and performance of their division, but insulate you from the day-to-day details. In this arrangement, each division becomes a functional abstraction to you; you know what each division does, but not necessarily how it does it, freeing you to focus on more important organization-level activity. Each vice president may utilize a similar arrangement within their division. Indeed, organizations are often subdivided many times until the number of employees in a unit is small enough to be overseen by a single manager.

Similarly, each of the subproblems in a functional decomposition might be further broken down, until we arrive at subproblems that are straightforward to solve.

**Reflection 1.6** *Can the left subproblem in the reading level problem, "average number of words per sentence," be computed directly? Or can it be decomposed further? (Think about how you computed the reading level in Reflection 1.2.)*

We saw above that the average number of words per sentence is equal to the total number of words divided by the total number of sentences, so we can decompose this subproblem into two even simpler subproblems:



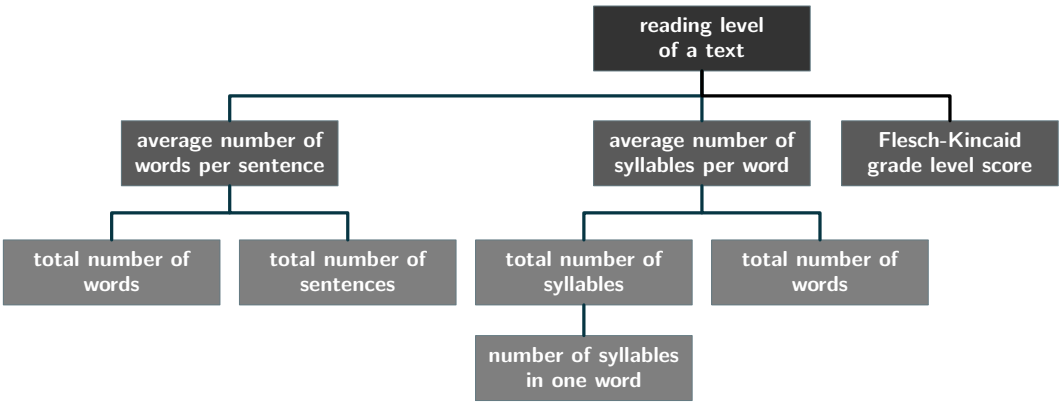
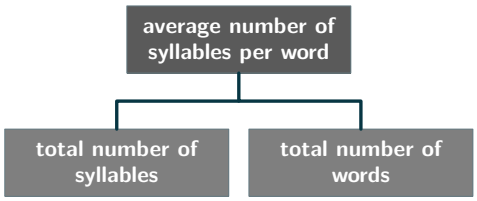



Figure 1.4 Functional decomposition of the reading level problem.

Similarly, we can also decompose the problem of computing the average number of syllables per word into two subproblems:




Taken altogether, we are now left with three relatively simple subproblems to solve: (a) counting the total number of words, (b) counting the total number of sentences, and (c) counting the total number of syllables.

**Reflection 1.7** *Can computing the total numbers of words, sentences, or syllables be broken down further?*

Counting the total numbers of words and sentences seems pretty straightforward. But finding the total number of syllables is not as simple because even finding the number of syllables in one word is not trivial for a computer, especially with all of the oddities of the English language. Thus it makes sense to further decompose finding the *total* number of syllables into the subproblem of finding the number of syllables in just *one* word.

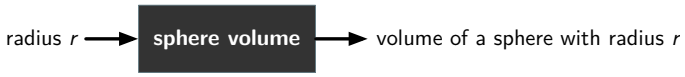
A diagram of the final functional decomposition is shown in Figure 1.4. These kinds of diagrams are called ***trees*** because they resemble an upside down tree with the *root* at the top and branches spreading out below. Nodes at the bottom of the tree are called *leaves*.

## Pseudocode

The next step is to write an algorithm for each of the subproblems, starting with the leaves at the bottom of the tree and working our way up to the root. We will get to this shortly, but first let's write an algorithm for a more straightforward problem.

### *Computing the volume of a sphere*

This simple problem can be visualized as follows.




To compute the output from the input, we can simply use the well-known formula $V = (4/3)\pi r^3$. Although this is much closer to an algorithm than the Flesch-Kincaid formula, it still does not explicitly specify a sequence of steps; there are several alternative sequences that one could follow to carry it out.[3] For example, we could cube $r$ first, then multiply that result by the rest of the terms, or we could cube $r$ last, or we could multiply $r$ by $(4/3)\pi$ then by $r^2$, etc. Here is one algorithm that follows the formula:

**Algorithm** SPHERE VOLUME

**Input:** the radius $r$ of the sphere

1. Multiply $r \times r \times r$.
2. Multiply the previous result by $\pi$.
3. Multiply the previous result by 4.
4. Divide the previous result by 3.

**Output:** the final result, which is the volume of the sphere

At the top of the algorithm, we note the input and at the bottom we note the output. In between, the individual lines are called ***statements***. Carrying out the sequence of statements in an algorithm is called ***executing*** or ***running*** the algorithm.

The informal style in which this algorithm is written is known as ***pseudocode***, to differentiate it from *code*, which is another name for a computer program. In common usage, the prefix *pseudo* often has a negative connotation, as in *pseudo-intellectual* or *pseudoscience*, but here it simply connotes a relaxed manner of writing algorithms that is meant to be read by a human rather than a computer. The flexibility afforded by pseudocode allows us to more clearly focus on how to solve the problem at hand without becoming distracted by the more demanding requirements of a programming language. Once we have refined the algorithm adequately and convinced ourselves that it is correct, we can translate it into a formal program. We'll talk more about that in the next section.

[3] In Python, we can actually use this formula more or less directly, but to facilitate this simple example, we'll pretend otherwise for now.

Here is a different algorithm that also computes the volume of a sphere. We are calling this algorithm a "draft" because, like other kinds of writing, algorithms also require rounds of revisions. We will revise this algorithm two more times.

**Algorithm** Sphere Volume 2 — Draft 1

**Input:** the radius $r$ of the sphere

1. Divide 4 by 3.
2. Multiply the previous result by $\pi$.
3. Repeat the following three times: multiply the previous result by $r$.

**Output:** the final result, which is the volume of the sphere

To better understand what an algorithm is doing, we can execute it on an example, using a ***trace table***. In a trace table, we *trace* through each step of the algorithm and keep track of what is happening. The following trace table shows the execution of our draft Sphere Volume 2 algorithm with input value $r = 10$.

**Trace input:** $r = 10$

| Step | Line | Result | Notes |
|---|---|---|---|
| 1 | 1 | $1.\bar{3}$ | $4 \div 3 = 1.\bar{3}$ |
| 2 | 2 | $4.18\bar{6}$ | multiplying the previous result ($1.\bar{3}$) by $\pi$ |
| 3 | 3 | $41.8\bar{6}$ | multiplying the previous result ($4.18\bar{6}$) by 10 |
| 4 | 3 | $418.\bar{6}$ | multiplying the previous result ($41.8\bar{6}$) by 10 |
| 5 | 3 | $4{,}186.\bar{6}$ | multiplying the previous result ($418.\bar{6}$) by 10 |

**Output:** $4{,}186.\bar{6}$

The four columns keep track of the number of steps executed by the algorithm, the line number in the algorithm being executed, the result after that line is executed, and notes explaining what is happening in that line.

The first two steps are pretty self-explanatory. Then, because line 3 of the algorithm instructs us to repeat something three times, line 3 is executed 3 times in the trace table. A statement that repeats like this is called a ***loop***. When algorithms contain loops, the number of steps is not necessarily the same as the number of lines.

Because we will eventually want to translate our pseudocode algorithms into *actual* code, it will be important to adhere to some important principles that will make this translation easier. First, we must strive to eliminate any ***ambiguity*** from our algorithms. In other words, the steps in an algorithm must never require creative interpretation by a human being. As we will see in Section 3.1, computers are, at their core, only able to perform very simple instructions like arithmetic and comparing two numbers, and are incapable of creative inference. Second, the steps of an algorithm must be ***executable*** by a computer. In other words, they must correlate to things

a computer can actually do. The definition of *executable* will become clearer as we learn more about programming.

These two requirements are not really unique to computer algorithms. For example, we hope that new surgical techniques are unambiguously presented with references to actual anatomy and real surgical tools. Likewise, when an architect designs a building, they must use only available materials and be precise about their placement. And when an author writes a novel, they must write to their audience, using appropriate language and culturally familiar references.

By these standards, both of the previous algorithms are less than ideal in at least two ways. First, references to the "previous result" are not precise and will not get us very far in more complex algorithms where multiple intermediate values need to be remembered. This kind of imprecision can easily lead to problematic ambiguity in our algorithms. Second, in none of the steps did we explicitly state that we needed to remember a result to be used later. When you executed the algorithms, you could infer this necessity, but this is an example of the kind of ambiguity that we need to avoid when writing algorithms for a computer.

To remedy these issues, algorithms use ***variables*** to give names to values that need to be remembered later. To make our algorithms understandable to a human reader, we will use descriptive variable names, unlike the single letter $x$ and $y$ variables that are common in mathematics. In our pseudocode algorithms, we will indicate variables in italics and ***assign*** a value to a variable with the notation

*variable* ← value

The left-facing arrow indicates that the value on the right is being assigned to the variable on the left. For example, *eggs* ← 12 would assign the value 12 to the variable named *eggs*. Using variables, the Sphere Volume 2 algorithm can be rewritten as follows.

**Algorithm** Sphere Volume 2 — Draft 2

**Input:** the radius *r* of the sphere

1 *volume* ← 4 ÷ 3
2 *volume* ← previous value of *volume* × $\pi$
3 repeat the following three times:
4 *volume* ← previous value of *volume* × *r*

**Output:** the value of *volume*

In this version, we have also formatted the loop a little bit differently, indenting the statement that is being repeated on a separate line (line 4). The statements that are executed repeatedly by a loop are called the ***body*** of the loop. So line 4 is the body of the loop that starts on line 3. The following trace table, again with input value *r* = 10, illustrates how the revised algorithm works.

| **Trace input:** $r = 10$ | | | |
|---|---|---|---|
| Step | Line | *volume* | Notes |
| 1 | 1 | $1.\overline{3}$ | *volume* ← $4 \div 3 = 1.\overline{3}$ |
| 2 | 2 | $4.18\overline{6}$ | *volume* ← previous *volume* × $\pi$ = $1.\overline{3} \times 3.14 = 4.18\overline{6}$ |
| 3 | 3 | ″ | *volume* unaffected; execute line 4 three times |
| 4 | 4 | $41.8\overline{6}$ | *volume* ← previous *volume* × $r$ = $4.18\overline{6} \times 10 = 41.8\overline{6}$ |
| 5 | 4 | $418.\overline{6}$ | *volume* ← previous *volume* × $r$ = $41.8\overline{6} \times 10 = 418.\overline{6}$ |
| 6 | 4 | $4{,}186.\overline{6}$ | *volume* ← previous *volume* × $r$ = $418.\overline{6} \times 10 = 4{,}186.\overline{6}$ |
| **Output:** *volume* = $4{,}186.\overline{6}$ | | | |

Notice that we have replaced the generic "Result" column with a column that keeps track of the value of the introduced variable, which we named *volume* because it will eventually be assigned the volume of the sphere. The first line of the trace table shows that the variable named *volume* is assigned the result of dividing 4 by 3. In line 2, the value of *volume*, which is now $1.\overline{3}$, is multiplied by $\pi$ (which we truncate to 3.14), and the result, which is $4.18\overline{6}$, is assigned to *volume*. Notice how much less ambiguous this is, compared to a reference to a "previous result." Also note that this assignment has *overwritten* the previous value of *volume*. Next, line 3 does not do anything on its own; it just instructs us to repeat line 4 three times. (The "ditto" marks indicate no change to *volume*.) Each execution of line 4 multiplies the value of *volume* by 10, and overwrites the value of *volume* with this result. At the end, *volume* corresponds to the value $4{,}186.\overline{6}$, which is output by the algorithm.

**Reflection 1.8** *Trace through the algorithm again with input value $r = 5$. Create a new trace table to show your progress. (The final answer should be $523.\overline{3}$.)*

Let's make one more refinement to our algorithm. In line 2 (and similarly in line 4), the algorithm refers to the "previous value of" *volume* on the righthand side of the assignment:

$$\textit{volume} \leftarrow \text{previous value of } \textit{volume} \times \pi$$

We included this language for clarity, but it is not actually necessary; the statement can be abbreviated to

$$\textit{volume} \leftarrow \textit{volume} \times \pi \ .$$

In any assignment statement, the righthand side after the arrow must be evaluated first, *before* the result of this evaluation is assigned to the variable on the lefthand side. Therefore, when *volume* is referenced on the righthand side of this assignment, it *must* refer to the previous value of *volume*, assigned in the previous line, as illustrated below.

$$\overbrace{\textit{volume} \leftarrow}^{\text{second step}} \ \overbrace{\textit{volume} \times \pi}^{\text{first step}}$$

$$\begin{array}{cc} \uparrow & \uparrow \\ \text{next value} & \text{previous value} \\ (4.18\overline{6}) & (1.\overline{3}) \end{array}$$

With this revision, the algorithm now looks like this:

**Algorithm** Sphere Volume 2 — Final

**Input:** the radius $r$ of the sphere

1 $volume \leftarrow 4 \div 3$
2 $volume \leftarrow volume \times \pi$
3 repeat the following three times:
4 $\quad volume \leftarrow volume \times r$

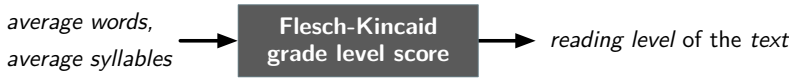
**Output:** *volume*

A trace table for this algorithm looks exactly like the previous one.

### Implement from the bottom

We are now prepared to return our attention to the reading level problem. Based on our decomposition tree in Figure 1.4, there are two ways that we can proceed. Our first option is to start at the top of the tree and work our way down. For this to work, we would need to assume that algorithms for the three main subproblems already exist. Although it is possible to work this way, it is trickier because we cannot test whether anything is working correctly until we have written algorithms for everything in the tree.

Instead, we will start at the bottom of the decomposition tree and work our way up, in what we call a ***bottom-up implementation***. The subproblems that are leaves of the tree are not dependent on any other subproblems, so we can design algorithms for these first, make sure they work, and then call upon them in algorithms for subproblems one level higher. If we continue this process until we reach the root of the tree, we will have a complete algorithm. Let's start with an algorithm for the "Flesch-Kincaid grade level score" subproblem, which takes as inputs the average number of words per sentence and the average number of syllables per word, and outputs the grade level of the text according to the Flesch-Kincaid formula.




In pseudocode, we can write this algorithm as follows:

**Algorithm** Flesch Kincaid

**Input:** *average words*, *average syllables*

1 $reading\ level \leftarrow 0.39 \times average\ words + 11.8 \times average\ syllables - 15.59$

**Output:** *reading level*

This one-line algorithm simply uses its two input values to compute the grade level, assigns this value to the variable named *reading level*, and then outputs this value.

Let's next write an algorithm to compute the number of syllables in a word. This algorithm will take a single word as input and output the number of syllables.


**Reflection 1.9** *How do you count the number of syllables in a word? Do you think your method can be "taught" to a computer?*

As you might imagine, a computer cannot use the "clapping method" or something similar to compute the number of syllables in a word. Instead, a syllable-counting algorithm will need to "look at" the letters in the word and follow some rules based on those letters. Since the number of syllables in a word is defined to be the number of distinct vowel sounds, a first approximation would be to simply count the number of vowels in the word.

**Algorithm** SYLLABLE COUNT — DRAFT 1

**Input:** a *word*
1 *count* ← the number of vowels in *word*
**Output:** *count*

**Reflection 1.10** *What does it mean for an algorithm to be correct? Is this algorithm correct? If it is not, why not?*

An algorithm is ***correct*** if it gives the correct output for *every* possible input. This algorithm is obviously too simplistic to be correct. For example, the algorithm will over-count the number of syllables in words containing diphthongs, such as *rain* and *loan*, and in words ending with a silent *e*.

**Reflection 1.11** *There is also some ambiguity in this one-line syllable-counting algorithm. Do you see what it is?*

The ambiguity arises from the definition of a vowel in the English language. The letters *a*, *e*, *i*, *o*, and *u* are always vowels but sometimes so is *y*. So our algorithm needs to clarify this. Incorporating these insights (and ignoring *y* as a vowel) leads to the following enhanced algorithm.

**Algorithm** SYLLABLE COUNT — DRAFT 2

**Input:** a *word*

1 *count* ← the number of vowels (*a*, *e*, *i*, *o*, *u*) in *word*
2 repeat for each pair of adjacent *letters* in *word*:
3  if the *letters* are both vowels, then subtract 1 from *count*
4 if *word* ends in *e*, then subtract 1 from *count*

**Output:** *count*

Notice two new kinds of pseudocode statements in this algorithm. First, line 2, is a different kind of loop. Imagine yourself looking carefully for adjacent vowels in a very long word like *consanguineous*. You would probably scan along the word visually or with your finger, from left to right, repeatedly checking pairs of adjacent letters. This is also what lines 2 and 3 are doing; for each pair of adjacent letters you look at, check if they are both vowels and subtract one from the *count* if they are. Another name for the repetitive process carried out by a loop is ***iteration***; implicit in line 2 is a process of *iterating over* the letters of the word.

Lines 3 and 4 both illustrate the second new type of statement, called a ***conditional statement***, or sometimes an ***if-then statement***. Simple conditional statements like this are self-explanatory: if the condition after the *if* is true, do the thing after *then*. We will work with more sophisticated conditional statements in Chapter 5.

Let's use a trace table to show the execution of this algorithm on the word "ancient."

**Trace input:** *word* = `"ancient"`

| Step | Line | *count* | *letters* | Notes |
|---|---|---|---|---|
| 1 | 1 | 3 | — | there are three vowels in `"ancient"` |
| 2 | 2 | ″ | `"an"` | first pair of adjacent letters in `"ancient"` is `"an"` |
| 3 | 3 | ″ | ″ | `"an"` are not both vowels; no change to *count* |
| 4 | 2 | ″ | `"nc"` | next pair of adjacent letters is `"nc"` |
| 5 | 3 | ″ | ″ | `"nc"` are not both vowels; no change to *count* |
| 6 | 2 | ″ | `"ci"` | next pair of adjacent letters is `"ci"` |
| 7 | 3 | ″ | ″ | `"ci"` are not both vowels; no change to *count* |
| 8 | 2 | ″ | `"ie"` | next pair of adjacent letters is `"ie"` |
| 9 | 3 | 2 | ″ | `"ie"` are both vowels; subtract 1 from *count* |
| 10 | 2 | ″ | `"en"` | next pair of adjacent letters is `"en"` |
| 11 | 3 | ″ | ″ | `"en"` are not both vowels; no change to *count* |
| 12 | 2 | ″ | `"nt"` | next pair of adjacent letters is `"nt"` |
| 13 | 3 | ″ | ″ | `"nt"` are not both vowels; no change to *count* |
| 14 | 4 | ″ | ″ | `"ancient"` does not end in *e*; no change to *count* |

**Output:** *count* = 2

The horizontal lines in the trace table make it easier to see the individual iterations of the loop. In the first step, we count the number of vowels in the input and assign *count* to this value. In step 2, we begin the loop by considering the first pair of adjacent letters in the input, `"an"`. In step 3, we check if these are both vowels. Since they are not, we leave *count* alone. In step 4, we repeat the loop by executing line 2 again with the next pair of adjacent letters, `"nc"`. In step 5, we repeat line 3 which, again, has no effect on the *count*. In the third iteration, in lines 6–7, the same thing happens. In the fourth iteration of the loop, starting in step 8, we do find a pair of adjacent letters that are both vowels, so we subtract one from *count*. The loop continues until we run out of letters from the input. Finally, in step 14, we execute line 4, which finds that the input does not end in *e*, so *count* remains 2.

**Reflection 1.12** *Use trace tables to also execute the algorithm on the words "create" and "syllable." Do you get the correct numbers of syllables?*

From these examples, you can see that our algorithm is still not correct. Indeed, designing a computer algorithm that correctly counts syllables for every word in the English language is virtually impossible; there are just too many exceptions! But we can certainly get closer than we are now. We will leave it as an exercise for you to draft further improvements.

To continue our bottom-up implementation of the reading level algorithm, we will use the Syllable Count algorithm to solve the "total number of syllables" problem. The idea of the Total Syllable Count algorithm is simple: for each word in the text, call upon the Syllable Count algorithm for the number of syllables in that word, and add this number to a running sum of the total number of syllables.

**Algorithm** Total Syllable Count

**Input:** a *text*

```
1  total count ← 0
2  repeat for each word in the text:
3      number ← Syllable Count (word)
4      total count ← total count + number
```

**Output:** *total count*

In line 1 of the algorithm, we initialize the *total count* of syllables to zero. Line 2 is a loop that iterates over all of the words in the text. For every *word*, we execute lines 3 and 4, indented to indicate that these comprise the body of the loop. In line 3, we call upon the Syllable Count algorithm to get the number of syllables in the *word* that is being considered in that iteration. Syllable Count (*word*) is shorthand for "execute the Syllable Count algorithm with input *word*," where *word* is the variable name representing the word that is being examined in each iteration of the loop. The output of the Syllable Count algorithm is then assigned to the variable named *number*. So altogether, line 3 is shorthand for

> "Execute the SYLLABLE COUNT algorithm, with input *word*, and assign the output to the variable named *number*."

Then in line 4, we add the *number* of syllables in the *word* to the *total count* of syllables.

This is a lot to take in, so let's once again illustrate with a trace table, using the first three words of the United Nations charter as input.

**Trace input:** *text* = "We the peoples"

| Step | Line | *total count* | *word* | *number* | Notes |
|---|---|---|---|---|---|
| 1 | 1 | 0 | — | — | initialize *total count* to zero |
| 2 | 2 | ″ | "We" | — | do the loop body with *word* ← "We" |
| 3 | 3 | ″ | ″ | 1 | get the number of syllables in "We" |
| 4 | 4 | 1 | ″ | ″ | add *number* to *total count* |
| 5 | 2 | ″ | "the" | ″ | do the loop body with *word* ← "the" |
| 6 | 3 | ″ | ″ | 1 | get the number of syllables in "the" |
| 7 | 4 | 2 | ″ | ″ | add *number* to *total count* |
| 8 | 2 | ″ | "peoples" | ″ | do the loop body with *word* ← "peoples" |
| 9 | 3 | ″ | ″ | 2 | get the number of syllables in "peoples" |
| 10 | 4 | 4 | ″ | ″ | add *number* to *total count* |

**Output:** *total count* = 4

In the trace table, horizontal lines identify the three iterations of the loop, one for each word in the *text*. In the first iteration, in step 2, the loop assigns *word* to be "We", as the first word in the *text*. Then, in step 3, *number* is assigned the output of SYLLABLE COUNT ("We"), which means that the SYLLABLE COUNT algorithm is called upon to get the number of syllables in "We" and this value (1) is assigned to the variable *number*. In step 4, the value of *number* is added to the *total count*. Remember that, in an assignment statement, the righthand side is evaluated first, so the statement in line 4 is assigning to *total count* the sum of the previous value of *total count* and *number*, as illustrated below:

$$\underset{\text{next value}\ (1)}{\textit{total count}} \leftarrow \underset{\text{previous value}\ (0)}{\textit{total count}} + \underset{(1)}{\textit{number}}$$

In the second iteration, starting in step 5, the loop assigns *word* to be "the", the SYLLABLE COUNT algorithm is called upon to get the number of syllables in "the" and this value is added to *total count*, bringing its value to 2. Finally, in the third iteration, starting on line 8, the process repeats with *word* assigned to be "peoples", bringing the total number of syllables to 4, which is the output of the algorithm.

To flesh out the entire reading level algorithm, we would continue designing algorithms for subproblems at the bottom of the tree, and work our way up, calling upon

algorithms at lower levels from algorithms for levels above. This would continue until we get to the root of the tree.

Suppose, for the moment, that we have worked our way up the decomposition tree and that, in addition to the Flesch-Kincaid algorithm, algorithms for the other two main subproblems have been written—one that computes the average number of words per sentence and one that computes the average number of syllables per word. Also suppose that we have named these algorithms Average Words Per Sentence and Average Syllables Per Word, respectively. Then the final reading level algorithm would look like the following.

**Algorithm** Reading Level

**Input:** a *text*

1 *average words* ← Average Words Per Sentence (*text*)

2 *average syllables* ← Average Syllables Per Word (*text*)

3 *reading level* ← Flesch-Kincaid (*average words, average syllables*)

**Output:** *reading level*

Line 1 of the algorithm is shorthand for

> "Execute the Average Words Per Sentence algorithm, with input *text*, and assign the output to the variable named *average words*."

Similarly, line 2 of the algorithm is shorthand for

> "Execute the Average Syllables Per Word algorithm, with input *text*, and assign the output to the variable named *average syllables*."

Finally, line 3 calls upon the Flesch-Kincaid algorithm, with the values of these two variables as input, to compute the Flesch-Kincaid grade level score. The output of this algorithm is then assigned to the variable named *reading level*, which is output by the algorithm.

We invite you to take a stab at writing the remaining algorithms in the exercises.

## Exercises

1.2.1. Decompose each of the following problems into subproblems. Continue the decomposition until you think each subproblem is sufficiently simple to solve. Explain your rationale for stopping the decomposition where you did.

(a)* an exercise routine from warmup to cool down

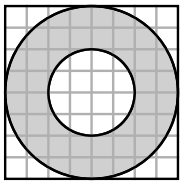
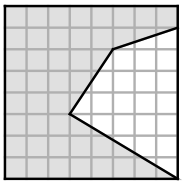
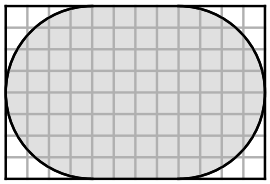
(b) your complete laundry routine

(c) writing a paper for a class

(d) your morning routine

(e) planning a multiple course menu

1.2.2. Suppose you want to find the area of each of the following shaded regions. In each of the diagrams, one square represents one square unit. Decompose each problem into subproblems that make finding the solution easier. (You do not need to actually find the areas.)

(a)*  (b)  (c) 

1.2.3. Look up the organizational chart for your school. Choose one division and explain how the organization of that division supplies a functional abstraction to the office that oversees the division.

1.2.4* Use a trace table to show how the final Sphere Volume 2 algorithm executes with input $r = 7$.

1.2.5. Use a trace table to show how the second draft of the Syllable Count algorithm executes on the word *algorithm*. Is the result correct?

1.2.6. The following algorithm computes the surface area of a box.

**Algorithm** Surface area of a box

**Input:** *length*, *width*, *height*

1 *area 1* ← *length* × *width*
2 *area 2* ← *length* × *height*
3 *area 3* ← *width* × *height*
4 *surface* ← *area 1* + *area 2* + *area 3*
5 *surface* ← *surface* × 2

**Output:** *surface*

Use a trace table (started below) to show how the algorithm executes with inputs *length* = 4, *width* = 5, and *height* = 2.

| **Trace input:** *length* = 4, *width* = 5, *height* = 2 | | | | | | |
|---|---|---|---|---|---|---|
| Step | Line | *area 1* | *area 2* | *area 3* | *surface* | Notes |
| 1 | 1 | 20 | — | — | — | *area 1* ← *length* × *width* |
| 2 | 2 | | | | | |
| ⋮ | | | | | | |
| **Output:** | | | | | | |

1.2.7* The following algorithm determines the winner of an election between two candidates, Laura and John. The input is a list votes like [Laura, Laura, John, Laura, . . . ].

**Algorithm** COUNT VOTES

**Input:** *votes*

1 *laura* ← 0
2 *john* ← 0
3 repeat once for each entry in *votes*:
4 if the entry is for Laura, then add 1 to *laura*
5 otherwise, add 1 to *john*
6 if *laura* > *john*, then *winner* ← Laura
7 otherwise, *winner* ← John

**Output:** *winner*

Use a trace table (started below) to show how the algorithm executes with input *votes* = [John, Laura, Laura, John, Laura].

**Trace input:** *votes* = [John, Laura, Laura, John, Laura]

| Step | Line | *laura* | *john* | *winner* | Notes |
|---|---|---|---|---|---|
| 1 | 1 | 0 | — | — | *laura* set to 0 |
| 2 | 2 | | | | |
| ⋮ | | | | | |
| **Output:** | | | | | |

1.2.8. There is a subtle mistake in the algorithm in Exercise 1.2.7. Describe and fix it.

1.2.9* Revise the original SPHERE VOLUME algorithm on page 10 so that it also uses a variable instead of referring to the "previous result."

1.2.10. Write yet another algorithm for finding the volume of a sphere.

1.2.11. Write an algorithm to sort a stack of any 5 cards by value in ascending order. In each step, your algorithm may compare or swap the positions of any two cards.

1.2.12. Write an algorithm to walk between two nearby locations, assuming the only legal instructions are "Take $s$ steps forward," and "Turn $d$ degrees to the left," where $s$ and $d$ are positive integers.

1.2.13. The term *algorithm* was derived from the name of Muḥammad ibn Mūsā al-Khwārizmī (c. 780–c. 850), a Persian mathematician who introduced both Arabic numerals and algebra to the world. The term *algebra* is derived from the Latin translation of the title of his book, "The Compendious Book on Calculation by Completion and Balancing" [4], which introduced algebra. The following algorithm for a common algebraic operation is from an English translation of this work.

*You know that all mercantile transactions of people, such as buying and selling, exchange and hire, comprehend always two notions and four numbers, which are stated by the enquirer; namely, measure and price, and quantity and sum. The number which expresses the measure is inversely proportionate to the number*

*which expresses the sum, and the number of the price inversely proportionate to that of the quantity. Three of these four numbers are always known, one is unknown, and this is implied when the person inquiring says "how much?" and it is the object of the question. The computation in such instances is this, that you try the three given numbers; two of them must necessarily be inversely proportionate the one to the other. Then you multiply these two proportionate numbers by each other, and you divide the product by the third given number, the proportionate of which is unknown. The quotient of this division is the unknown number, which the inquirer asked for; and it is inversely proportionate to the divisor.*

*Examples.—For the first case: If you are told "ten for six, how much for four?" then ten is the measure; six is the price; the expression how much implies the unknown number of the quantity; and four is the number of the sum. The number of the measure, which is ten, is inversely proportionate to the number of the sum, namely, four. Multiply, therefore, ten by four, that is to say, the two known proportionate numbers by each other; the product is forty. Divide this by the other known number, which is that of the price, namely, six. The quotient is six and two-thirds; it is the unknown number, implied in the words of the question "how much?" it is the quantity, and inversely proportionate to the six, which is the price.*

---

There are four variables identified in the passage: *measure*, *price*, *sum*, and *quantity*. Write an algorithm in pseudocode that answers the "how much?" question posed in the example when the first three quantities are given as input.

1.2.14* Using the Syllable Count algorithm as a guide, write an algorithm named Word Count that approximates the total number of words in a text. Your algorithm should take a text as input and output a count of words. Like the Syllable Count algorithm, use a loop to look at each letter in the text and adjust a count as appropriate. As with counting syllables, this problem is fraught with complexity arising from the English language, so your algorithm need not be perfect.

1.2.15. Write an algorithm named Sentence Count to count the total number of sentences in a text. Your algorithm should take a text as input and output a count of sentences. The guidance from the previous exercise also applies.

1.2.16* Using Figure 1.4 and the Reading Level algorithm on page 19 as guides, design the Average Words Per Sentence algorithm. Call upon your Word Count and Sentence Count algorithms to do most of the work.

1.2.17. Using Figure 1.4 and the Reading Level algorithm on page 19 as guides, design the Average Syllables Per Word algorithm. Call upon your Word Count algorithm and the Total Syllable Count algorithm to do most of the work.

1.2.18. Enhance the Syllable Count (Version 2) algorithm on page 16 so that it correctly counts the number of syllables in

(a) plural words

(b) words ending in a consonant plus *le* (e.g., *syllable*)

(c) words containing a *y* that acts like a vowel

(d) the word *algorithm*

## 1.3 WRITE A PROGRAM

> *Third, carry out your plan, checking each step.*

The next step in the problem solving process is to "carry out your plan" by translating your algorithm into a ***program*** that a computer can execute. A program must adhere to a set of grammatical rules, called ***syntax***, that are defined by a particular ***programming language***. In this book, we will use a programming language called *Python*. You will find that programming in Python is not too different from writing algorithms in pseudocode, which is why it is a great first language. But Python is not a toy language either; it has become one of the most widely used programming languages in the world, especially in data science, bioinformatics, and digital humanities.

Writing programs (or "programming") is a hands-on activity that allows us to test our algorithms, apply them to real inputs, and harness their results, in tangible and satisfying ways. Learning how to program empowers us to put our algorithms into production. Solving problems and writing programs should also be fun and creative. Guido van Rossum, the inventor of Python understood this when he named Python after the British comedy series "Monty Python's Flying Circus!"

In this section, we will not be able to fully realize our reading level algorithm as a program just yet. Some of the steps that are easy to write as pseudocode, such as breaking a text into individual words, are actually more involved than they look on paper. But we will be able to implement the Flesch-Kincaid algorithm at the bottom of our decomposition tree, and get oriented for what awaits in future chapters. Before long, you will be able to implement everything from the previous section and much more!

### Welcome to the circus

As you work through this book, we highly recommend that you do so in front of a computer. The only way to learn how to program is to do it, so every example we provide is meant to be tried by you. Then go beyond the examples, and experiment with your own ideas. Instead of just wondering, "What would happen if I did this?", type it in and see! To get started, launch the application called IDLE that comes with every Python distribution (or another programming environment recommended by your instructor). You should see a window appear with something like this at the top:

```
Python 3.8.4 (v3.8.4:dfa645a65e, Jul 13 2020, 10:45:06)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license()" for more information.
>>>
```

The program executing in this window is known as a *Python shell*. The first line tells you which version of Python you are using (in this case, 3.8.4). The programs in this book are based on Python version 3.6 and higher. If you need to install a

newer version, you can find one at `http://python.org`. The symbol `>>>` on the fourth line in the IDLE window is called the ***prompt*** because it is prompting you to type in a Python statement. To start, type in `print('Hello world!')` at the prompt and hit return.

```
>>> print('Hello world!')
Hello world!
>>>
```

Congratulations, you have just written your first program! This one-statement program simply prints `Hello world!` on the screen.

Notice that the Python shell responded to your command with a result, and then gave you a new prompt. The shell will continue this "prompt → compute → respond" cycle until we quit (by typing `quit()`). In the "compute" step, as we will see in Section 3.1, the computer does not really understand what we are typing. Instead, each Python statement is transparently translated into ***machine language***, which is the only language a computer actually understands. Then the shell executes the machine language instructions and prints the result. The part of the shell that does this translation is called the ***interpreter***. Python programs can also be executed in "program mode," where the Python interpreter executes an entire program containing multiple statements all at once. We will introduce program mode in the next chapter.

A programming language like Python provides a rich set of abstractions that enable us to solve a wide variety of interesting problems. Your one-line program demonstrates two of these. The sequence of characters in quotes, `'Hello world!'`, is called a *character string* or just a ***string***. Strings, which can be enclosed in either single quotes (`'`) or double quotes (`"`), are how Python represents and stores text, from single characters up to entire books. To display this string, we used the `print` function. ***Functions*** are how functional abstractions are implemented in Python. A function, like the algorithms we developed in the previous section, takes one or more inputs, called ***arguments***, and produces an output, called the ***return value***. We call upon functions to compute things for us with the familiar notation that we used in the previous section to call upon algorithms. The `print` function takes the string `'Hello world!'` as an argument (in parentheses) and prints it to the screen. Alternatively, we could have assigned the string to a ***variable*** and then passed this variable to the `print` function like this:

```
>>> message = 'Hello world!'
>>> print(message)
Hello world!
>>>
```

In our pseudocode algorithms, we used a left-facing arrow to assign values to variables to emphasize that assignment is a two-step, right-to-left process:

1. Evaluate the expression on the righthand side of the assignment operator.
2. Assign the resulting value to the name on the lefthand side of the assignment operator.

| | Operators | Description |
|---|---|---|
| 1. | `( )` | parentheses |
| 2. | `**` | exponentiation (power) |
| 3. | `+, -` | unary positive and negative, e.g., `-(4 * 9)` |
| 4. | `*, /, //, %` | multiplication and division |
| 5. | `+, -` | addition and subtraction |

Table 1.1 Arithmetic operator precedence, highest to lowest. Operators with the same precedence are evaluated left to right.

In Python, assignment works exactly the same way, but the ***assignment operator*** is the equal sign (`=`) instead of an arrow.

Python can also crunch numbers, of course. Computing the volume of a sphere looks like this:

```
>>> radius = 10
>>> pi = 3.14159
>>> volume = (4 / 3) * pi * radius ** 3
```

We created two new variables above named `radius` and `pi`, and used these variables to compute the volume using the formula $(4/3)\pi r^3$. The `/`, `*`, and `**` symbols perform division, multiplication, and exponentiation, respectively. The spaces around the operators in the arithmetic expression are optional and ignored by the interpreter. In general, Python does not care if you include spaces in expressions, but you always want to make your programs readable to others, and spaces often help. The interpreter evaluates arithmetic operators in the usual order, summarized in Table 1.1 (i.e., PEMDAS). This precedence may be overridden by parentheses. You can also use parentheses, even when unnecessary, to make expressions easier to understand, as we did above with parentheses around `4 / 3`.

Assignment statements do not print any results, but you can display the value of a variable by either typing its name or using `print`.

```
>>> volume
4188.786666666666
>>> print(volume)
4188.786666666666
```

In the shell, both methods do the same thing. (When we start writing programs in the next chapter, using `print` will be necessary.)

Similarly, let's compute the Flesch-Kincaid reading level of a hypothetical text (since we cannot yet analyze a real text) with an average of 16 words per sentence and 1.78 syllables per word, using the formula on page 6.

```
>>> averageWords = 16
>>> averageSyllables = 1.78
>>> readingLevel = 0.39 * averageWords + 11.8 * averageSyllables - 15.59
>>> print(readingLevel)
11.654
```

The print function can also take multiple arguments, separated by commas. A space will be inserted between arguments when they are displayed.

```
>>> print('The reading level is', readingLevel, '.')
The reading level is 11.654 .
```

The first and last arguments are strings, and the second argument is the variable we defined above. Notice that there are no quotes around the variable name readingLevel.

**Reflection 1.13** *Why do you think quotation marks are necessary around strings? Try removing them and see what happens.*

```
>>> print(The reading level is, readingLevel, .)
```

The quotation marks are necessary because otherwise Python has no way to distinguish text from a variable or function name. Without the quotation marks, the Python interpreter will try to make sense of each argument, assuming that each word is a variable or function name, or a reserved word in the Python language. Since this sequence of words does not follow the syntax of the language, and most of these names are not defined, the interpreter will print an error.

Every value in Python has a ***type*** associated with it. Understanding this is very important when programming because the behaviors of operators and functions often depend upon the type of data they are given. You can see the different types of values assigned to our variables so far by using the type function.

```
>>> type(message)
<class 'str'>
>>> type(averageWords)
<class 'int'>
>>> type(averageSyllables)
<class 'float'>
>>> type(readingLevel)
<class 'float'>
```

A *class*, for our purposes at the moment, is a synonym for *type*. (We will talk about classes in more detail in the next chapter.) So this is telling us that the value assigned to message is a string (str), the value assigned to averageWords is an ***integer*** (int), and the value assigned to averageSyllables is a ***float*** (short for *floating point* number). We will have more to say about integers and floats in Section 3.2; for now, suffice to say that any number without a decimal point is an integer and any number with a decimal point is a float. The value assigned to readingLevel is also a float because the type of any arithmetic expression involving a float will also be a float. So in the print statement above, we actually combined two different types of values: strings and a float. The print function transparently converted readingLevel to a string before combining it with the other two strings into a longer string to print.

To suppress the extra space that gets inserted before the period in this print statement, we can build a string manually using the + operator which, when applied

to strings, is called the ***concatenation operator***. Concatenation combines strings into longer strings. For example,

```
>>> first = 'Monty'
>>> last = 'Python'
>>> name = first + ' ' + last
>>> print(name)
Monty Python
```

To concatenate `'The reading level is'`, `readingLevel`, and `'.'`, we need to first convert `readingLevel` to a string using the `str` function. The `str` function can take just about any type of value as an argument and it returns the argument represented as a string. For example, try this:

```
>>> readingLevelString = str(readingLevel)
>>> readingLevelString
'11.654'
```

When we use a function like this by passing an argument to it, it is called a ***function call***, or a *function invocation*. Calling `str(readingLevel)` returns (i.e., outputs) a string representation of `readingLevel` and assigns this value to the variable `readingLevelString`. Notice the quotes in `'11.654'`, indicating that it is a string rather than a float. The `str` function does not change the value of `readingLevel` though; it remains the same afterwards, as you can confirm:

```
>>> readingLevel
11.654
```

The easiest way to use `str` in the `print` statement is to skip the intermediate variable like this:

```
>>> print('The reading level is ' + str(readingLevel) + '.')
The reading level is 11.654.
```

**Reflection 1.14** *Try the previous statement without the* `str` *function. What happens and why?*

If we do not convert `readingLevel` to a string first, then we are trying to "add" a string to a float, which doesn't make any sense.

Some other useful functions are `float`, `int`, and `round`. The `float` and `int` functions return float and integer versions of their arguments, similar to the way the `str` function returns a string version of its argument.

```
>>> float(3)
3.0
>>> int(-1.618)
-1
```

The `int` function converts its argument to an integer by truncating it, i.e., removing the fractional part to the right of the decimal point. This might be helpful in our reading level computation, since we probably do not really want all of the digits to the right of the decimal point.

```
>>> readingLevel = int(readingLevel)
>>> readingLevel
11
```

Notice that we have overwritten the old value of `readingLevel` with the truncated value. Alternatively, we could have used the `round` function to round the reading level. Function arguments can be more complex than just single constants and variables; they can be anything that evaluates to a value. For example, we could get the rounded reading level like this too:

```
>>> readingLevel = round(0.39*averageWords + 11.8*averageSyllables - 15.59)
>>> readingLevel
12
```

The expression in parentheses is evaluated first, and then the result of the expression is used as the argument to the `round` function.

Not all functions have return values. For example, the `print` function, which simply prints its arguments to the screen, does not. For example, try this:

```
>>> result = print(readingLevel)
12
>>> print(result)
None
```

The variable `result` was assigned whatever the `print` function returned, which is different from what it printed. When we print `result`, we see that it was assigned something called `None`. `None` is a Python keyword that essentially represents "nothing." Any function that does not define a return value itself returns `None` by default. We will see this again shortly when we learn how to define our own functions.

### What's in a name?

Let's remind ourselves of a few reasons why variable names are so important.

1. Assigning descriptive names to values can make our algorithms much easier to understand. In the "real world," programming is almost always a collaborative endeavor, so it is important to always write programs that are easy to understand by others. Our goal should be to use sufficient descriptive variable names to create ***self-documenting programs*** that require as little as possible explanation outside the program itself. To see the value of self-documenting programs, just consider if we had written the reading level computation above like this instead:

   ```
   >>> a = 16
   >>> b = 1.78
   >>> c = 0.39 * a + 11.8 * b - 15.59
   ```

   Would you have any idea what these statements did?

2. As we did in our pseudocode algorithms, naming inputs will allow us to generalize algorithms so that, instead of being tied to one particular input,

| | | | | | | |
|---|---|---|---|---|---|---|
| and | break | elif | for | in | not | True |
| as | class | else | from | is | or | try |
| assert | continue | except | global | lambda | pass | while |
| async | def | False | if | None | raise | with |
| await | del | finally | import | nonlocal | return | yield |

Table 1.2 The 35 Python keywords.

they work for a variety of possible inputs. We will discuss this further in Section 2.5.

3. Names will serve as labels for computed values that we wish to use later, eliminating the need to compute them again at that time.

Variable names in Python can be any sequence of characters drawn from letters, digits, and the underscore (_) character, but they may not start with a digit. And, unlike some of our pseudocode variable names, they may not contain spaces. You also cannot use any of Python's ***keywords***, shown in Table 1.2. Keywords are elements of the Python language that have predefined meanings. We will encounter most of these keywords as we progress through this book.

Let's try breaking some of these naming rules to see what happens.

```
>>> average words = 6
            ^
SyntaxError: invalid syntax
```

A ***syntax error*** indicates a violation of the syntax, or grammar, of the Python language. It is completely *normal* for programmers to encounter syntax errors; it is part of the programming process. With practice, it will often become immediately obvious what you did wrong, you will fix the mistake, and move on. Other times, you will need to look harder to discover the problem but, with practice, these instances too will become easier to diagnose. In this case, the problem is the space we are trying to use in the variable name.

Next, try this one.

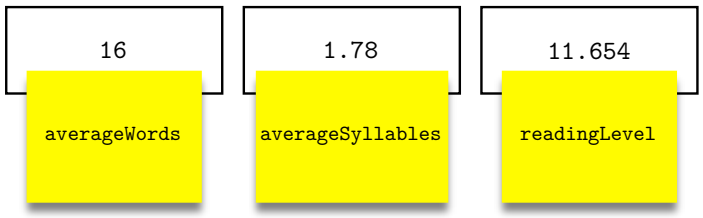
```
>>> average-words = 6
SyntaxError: cannot assign to operator
```

This syntax error is referring to the dash/hyphen/minus sign symbol (-) that we have in our name. Python interprets the symbol as the minus operator, which is why it is not allowed in names. Instead, we can use the underscore (_) character (i.e., `average_words`) or vary the capitalization (i.e., `averageWords`) to distinguish the two words in the name.

To develop a more nuanced understanding of what an assignment statement really does, we need to know a little bit about how values are stored. A computer's memory consists of billions of ***memory cells***, each of which can store one value. These cells are analogous to post office boxes, each with a unique address. And a variable name

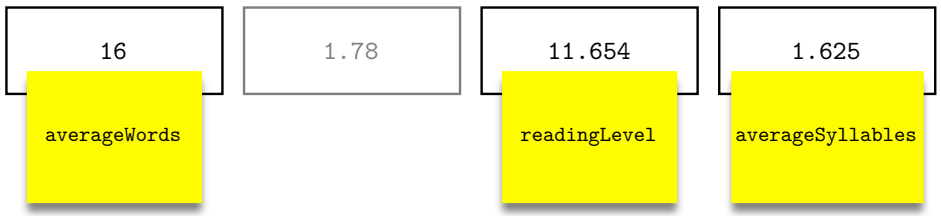
is like a "Sticky note"[4] attached to the front of one of those boxes. As we will see in Section 3.2, our programs are also stored in the same memory while they are executing.

The picture below represents the outcome of the three assignment statements in our reading level computation. Each of the three rectangles represents a memory cell, and a variable name (on a sticky note) is attached to each one.




Like a sticky note, a variable name can easily be reassigned to a different value at any time. For example, suppose we change the average number of syllables to 1.625:

```
>>> averageSyllables = 1.625
```




The reassignment caused the `averageSyllables` sticky note to move to a different memory cell containing the value `1.625`. The old value `1.78` may briefly remain in memory without a name attached to it, but since we can no longer access that value without a reference to it, the Python "garbage collection" mechanism will soon free up the memory that it occupies and allow it to be overwritten with something new.

**Reflection 1.15** *Did the value of* `readingLevel` *change when we changed the value of* `averageSyllables`*?*

Try it:

```
>>> readingLevel
11.654
```

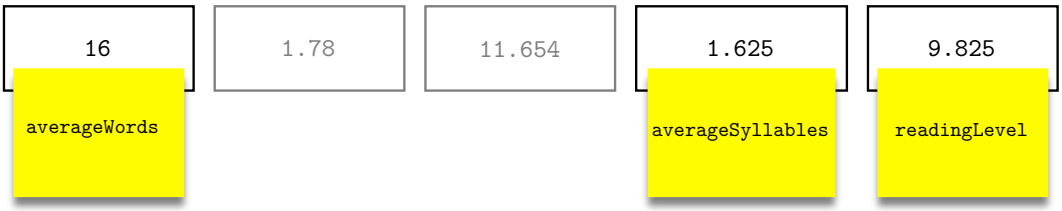
While the value assigned to `averageSyllables` has changed, the value assigned to `readingLevel` has not. This example demonstrates that assignment is a one-time event; Python does not "remember" how the value of `readingLevel` was computed. Put another way, an assignment is *not* creating an equivalence between a name and a computation. Rather, it performs the computation on the righthand side of the assignment operator only when the assignment happens, and then assigns the result to the name on the lefthand side. That value remains assigned to the name until the

[4] "Sticky note" is a registered trademark of the BIC Corporation.

name is explicitly assigned some other value or it ceases to exist. To compute a new value for `readingLevel` based on the new value of `averageSyllables`, we would need to perform the reading level computation again.

```
>>> readingLevel = 0.39 * averageWords + 11.8 * averageSyllables - 15.59
>>> readingLevel
9.825
```

Now the value assigned to `readingLevel` has changed, due to the explicit assignment statement above.




Yet another way to reinforce the nature of assignment is to look at what happens if we add one to `averageWords`:

```
>>> averageWords = averageWords + 1
```

If the equals sign denoted equality, then this statement would not make any sense! However, if we interpret it using the two-step process, it is perfectly reasonable. First, the expression on the righthand side is evaluated, ignoring the lefthand side entirely. Since, at this moment, `averageWords` is `16`, the righthand side evaluates to $16 + 1 = 17$. Second, the value `17` is assigned to `averageWords`. So this statement has added `1` to, or ***incremented***, the value of `averageWords`.

What if we had not assigned a value to `averageWords` before we tried to increment it? To find out, try this:

```
>>> tryThis = tryThis + 1
NameError: name 'tryThis' is not defined
```

This ***name error*** occurred because, when the Python interpreter tried to evaluate the righthand side of the assignment, it found that `tryThis` was not assigned a value, i.e., it was not defined. So we need to make sure that we define any variable before we refer to it. This may sound obvious but, in the context of some larger programs later on, it might be easy to forget.

### Interactive computing

We can interactively query for string input in our programs with the `input` function. The `input` function takes a string prompt as an argument and returns a string value that is typed in response. For example, the following statement prompts for your name and prints a greeting.

```
>>> name = input('What is your name? ')
What is your name? George
>>> print('Howdy, ' + name + '!')
Howdy, George!
```

The call to the `input` function above prints the string `'What is your name? '` and then waits. After you type something (above, we typed `George`, shown in red) and hit the return key, the text that we typed is returned by the `input` function as a string and assigned to the variable called `name`. The value of `name` is then used in the `print` function.

To adapt this format to a reading level program, we will need to convert the strings returned by the `input` function to numbers. Luckily, we can do this easily with the `float` and `int` functions.

```
>>> text = input('Average words per sentence: ')
Average words per sentence: 4.5
>>> averageWords = float(text)
>>> averageSyllables = float(input('Average syllables per word: '))
Average syllables per word: 2.1
>>> readingLevel = 0.39 * averageWords + 11.8 * averageSyllables - 15.59
>>> print('The reading level is ' + str(round(readingLevel)) + '.')
The reading level is 11.
```

In response to the first prompt above, we typed `4.5`. Then the `input` function assigned what we typed to the variable `text` as a string, in this case `'4.5'` (notice the quotes). Then, using the `float` function, the string is converted to the numeric value `4.5` (no quotes) and assigned to the variable `averageWords`. In the second prompt, we combined these two steps by passing the return value of the `input` function in as the argument of `float`. Either way, now that `averageWords` and `averageSyllables` are numerical values, they can be used in the arithmetic expression to compute the reading level. In the last `print` statement, notice how we composed the `str` and `round` functions so that the return value of `round` is being used as the argument to `str`.

**Reflection 1.16** *Type the statements above again, omitting the* `float` *function:*

```
>>> averageWords = input('Average words per sentence: ')
>>> averageSyllables = input('Average syllables per word: ')
>>> readingLevel = 0.39 * averageWords + 11.8 * averageSyllables - 15.59
```

*What happened? Why?*

## Looking ahead

In just this section, we have nearly achieved a full Python implementation of our Flesch Kincaid algorithm from page 14. What our implementation is missing is the ability to call upon it as a functional abstraction like we did in the Reading Level algorithm on page 19:

> *reading level* ← Flesch-Kincaid (*average words, average syllables*)

Here's a sneak peek at how we will do that in the next chapter:

```
def fleschKincaid(averageWords, averageSyllables):
    return 0.39 * averageWords + 11.8 * averageSyllables - 15.59
```

This defines `fleschKincaid` to be a *function* that takes two arguments as input and returns the corresponding reading level as its output. With this function, we will be able to do things like this:

```
readingLevel = fleschKincaid(16, 1.78)
```

More to come...

## Exercises

*Use the Python interpreter to answer the following questions. Where appropriate, provide both the answer and the Python expression you used to get it.*

1.3.1* You may have seen a meme that challenges you to find the correct answer for the expression $8 \div 2(2+2)$. Use Python to do this.

1.3.2* The Library of Congress stores its holdings on 838 miles of shelves. Assuming an average book is one inch thick, how many books would this hold?

1.3.3. If I gave you a nickel and promised to double the amount you have every hour for the next 24, how much money would you have at the end? What if I only increased the amount by 50% each hour, how much would you have? Use exponentiation to compute these quantities.

1.3.4. The Library of Congress stores its holdings on 838 miles of shelves. How many round trips is this between Granville, Ohio and Columbus, Ohio?

1.3.5. What is wrong with each of the following Python names? Suggest a fixed version for each.

(a) `word count`

(b) `here:there`

(c) `'minutes'`

(d) `4ever`

(e) `#thisisavariable`

1.3.6. (a) Assign a variable named `radius` to have the value 10. Using the formula for the area of a circle ($A = \pi r^2$), assign to a new variable named `area` the area of a circle with radius equal to your variable `radius`. (The number 10 should not appear in the formula.)

(b) Now change the value of `radius` to 15. What is the value of `area` now? Why?

1.3.7* The formula for computing North American wind chill temperatures, in degrees Celsius, is

$$W = 13.12 + 0.6215\,t + (0.3965\,t - 11.37)\,v^{0.16}$$

where $t$ is the ambient temperature in degrees Celsius and $v$ is the wind speed in km/h.[5]

(a) Compute the wind chill for a temperature of −3° C and wind speed of 13 km/h by assigning this temperature and wind speed to two variables `temperature` and `windChill`, and then assigning the corresponding wind chill to another variable `windChill` using the formula above.

(b) Change the value of `temperature` to 4.0 and then check the value of `windChill`. Why did the value of `windChill` not change? How would you update the value of `windChill` to reflect the change in `temperature`?

1.3.8* Suppose we want to swap the values of two variables named `left` and `right`. Why doesn't the following work? Show a method that does work.

```
left = right
right = left
```

1.3.9. What are the values of `apples` and `oranges` at the end of the following?

```
apples = 12.0
oranges = 2 * apples
apples = 6
```

1.3.10. What is the value of `number` at the end of the following?

```
number = 0
number = number + 1
number = number + 1
number = number + 1
```

1.3.11. In the previous exercise, what happens if you omit the first statement (`number = 0`)? Explain why `number` must be assigned a value before the executing the statement `number = number + 1`.

1.3.12. What are the values of `apples` and `oranges` at the end of the following?

```
apples = 12.0
oranges = 6
oranges = oranges * apples
```

1.3.13. String values can also be manipulated with the `*` operator. Applied to strings, the `*` operator becomes the ***repetition operator***, which repeats a string some number of times. The operand on the left side of the repetition operator is a string and the operand on the right side is an integer that indicates the number of times to repeat the string.

```
>>> last * 4
'PythonPythonPythonPython'
>>> print(first + ' ' * 10 + last)
Monty          Python
```

(a) Explain why `18 * 10` and `'18' * 10` give different values.

[5] Technically, wind chill is only defined at or below 10° C and for wind speeds above 4.8 km/h.

(b) Use the repetition operator to create a string consisting of 20 asterisks separated by spaces.

(c) The special character `'\n'` (really two characters, but it represents a single character) is called the *newline* character, and causes printing to continue on the next line. To see its effect, try this:

```
>>> print('Hello\nthere')
```

Use the newline character and the repetition operator to create a string that will display a *vertical* line of 20 asterisks when printed.

(d) Combine the techniques from parts (b) and (c) (with some modification) to create a string that will display a $20 \times 20$ square of asterisks when printed.

1.3.14* Modify your wind chill computation from Exercise 1.3.7 so that it gives the wind chill rounded to the nearest integer.

1.3.15. Show how you can use the `int` function to truncate any floating point `number` to two places to the right of the decimal point. In other words, you want to truncate a number like `3.1415926` to `3.14`. Your expression should work with any value of `number`.

1.3.16* Show how you can use the `int` function to find the fractional part of any positive floating point number. For example, if the value 3.14 is assigned to `number`, you want to output 0.14. Your expression should work with any value of `number`.

1.3.17. Show how to round a floating point number to the nearest tenth in Python.

1.3.18. What happens when you execute the following statements? Explain why.

```
>>> value = print(42)
>>> print(value * 2)
```

1.3.19* Fix the following sequence of statements

```
>>> radius = input('Radius of your circle? ')
>>> area = 3.14159 * radius * radius
>>> print('The area of your circle is ' + area + '.')
```

1.3.20. Write a sequence of statements that

(a) prompt for a person's age,

(b) compute the number of days that person has been alive (assume 365.25 days in a year to account for leap years),

(c) round the number of days to the nearest integer, and then

(d) print the result, nicely formatted.

1.3.21* Repeat Exercise 1.3.14, but this time prompt for the temperature and wind chill using the `input` function, and print the result formatted like

```
The wind chill is -2 degrees Celsius.
```

1.3.22. The following program implements a Mad Lib.

```
adj1 = input('Adjective: ')
noun1 = input('Noun: ')
noun2 = input('Noun: ')
adj2 = input('Adjective: ')
noun3 = input('Noun: ')

print('How to Throw a Party')
print()
print('If you are looking for a/an', adj1, 'way to')
print('celebrate your love of', noun1 + ', how about a')
print(noun2 + '-themed costume party?  Start by')
print('sending invitations encoded in', adj2, 'format')
print('giving directions to the location of your', noun3 + '.')
```

Write your own Mad Lib program, requiring at least five parts of speech to insert. (You can download the program above from the book website to get you started.)

1.3.23* Write a sequence of statements that accepts three numbers as input, one at a time, and prints the running sum of the numbers after each input. Use only two variables, one for the input number and one for the running sum. Here is an example of what your program should print (omitting the statements you type at the prompt):

```
Number 1: 5.1
The current sum is 5.1.
Number 2: 7
The cu
rrent sum is 12.1.
Number 3: 12.3
The final sum is 24.4.
```

## 1.4 LOOK BACK

> *Fourth, look back. Check the result. Can you derive the result differently?*

Not all algorithms are *good* algorithms, even if they are correct. And just about any algorithm can be made better. Like writing prose or poetry, writing algorithms and programming involve continual refinement. At every step of the process, we should "look back" on what we have created to see if it can be improved.

**Reflection 1.17** *What characteristics might make one algorithm or program better than another?*

Here are some questions we should always ask about our algorithms and programs:

1. Is your program easy to understand?

   Are you using descriptive variable names? Is there anything extraneous that could be omitted? Is there a more elegant way to accomplish the same thing?

2. Does your program work properly?

Is it solving the correct problem? Does it give the correct output for every possible input?

3. How long does your algorithm take? Is it as efficient as it could be?

   Is your algorithm doing too much work? Is there a way to streamline it? Does your program use too much memory? Can it be done with less? Our algorithms at this point are too simple to worry about this too much but, as they grow more complex, we will see that efficiency becomes an issue of paramount importance.

4. Are there ethical ramifications to consider?

   How will your algorithm or program affect human welfare? Will your algorithm unfairly impact some groups more than others? Are any of your assumptions based on unexamined cultural or racial prejudices? Are there related privacy or intellectual property issues? What is the environmental impact?

   These are essential questions to ask at every step of the problem-solving process. Sometimes even determining what problem you should solve requires careful judgment. Similarly, poorly chosen inputs to some problems, e.g., facial recognition and risk assessment algorithms used in the criminal justice system, can have severely damaging effects on entire groups of people. And when designing an algorithm, you may find that ethical considerations are at odds with efficiency; shortcuts and overly simple solutions can lead to damaging results.

   Questions such as these are both complex and essential, but largely beyond the scope of this book. Some additional resources well worth exploring are given in Section 1.5.

We will discuss the first point in more detail in the next chapter, as we start to develop more complete programs. We elaborate on the second and third points below.

### Testing

It should go without saying that we want our programs to be correct. That is, we want our algorithms to produce the correct output for every possible input, and we want our programs to be faithful to the design of our algorithms. There are techniques that we can use to increase the likelihood that our functions and programs are correct. The first two steps in our problem solving process are a good start: making sure that we thoroughly understand the problem we are trying to solve and spending quality time designing a solution, well before we start typing any code.

However, despite the best planning, errors, or "bugs," will still creep into your programs. To root out bugs from our programs, i.e., *debug* them, we have to test them thoroughly with a variety of carefully chosen inputs. We started to do this when

we refined our syllable-counting algorithm in Section 1.2. There are four important categories of inputs that you should be thinking about as we move forward:

1. If there are *disallowed inputs* that don't make sense for the problem and are not guaranteed to work, this should be stated explicitly in the documentation, as we will discuss further in the next section. We will also talk about how to more formally specify and check for these inputs in Section 5.5.
2. Once you have identified the range of legal inputs, test your program with several *common inputs* to make sure that its basic functionality is intact. It is important to test with inputs that are representative of the entire range of possibilities. For example, if your input is a number, try both negative and positive integers and floats.
3. *Boundary cases* are inputs that rests on a boundary of the range of legal inputs. For example, if your allowed inputs are all numbers between 0 and 100, be sure to test both 0 and 100. In many problems, testing boundary cases can identify issues with your algorithm that are easy to overlook.
4. Finally, *corner cases* are any other kind of rare input that might cause the program to break. These are usually the hardest to identify and tend to be quite specific to the problem being solved.

To illustrate, let's look at the simple problem of converting an average course grade between 0 and 100 to a GPA on a standard four-point scale, where 90–100 is a 4, 80–89 is a 3, 70–79 is a 2, 60–69 is a 1, and < 60 is a 0. (For simplicity, we will ignore +/− grades.). As a first stab at an algorithm, we notice that dividing by 10 to get the tens place of the input grade and then subtracting 5 seems to work.

**Algorithm** CONVERT GRADE — DRAFT

**Input:** *grade*

1 *tens place* ← the digit in the tens place of *grade*
2 *GPA* ← *tens place* − 5

**Output:** *GPA*

In Python, we can implement this algorithm in one line:

```
>>> grade = 87
>>> GPA = int(grade / 10) - 5
>>> GPA
3
```

**Reflection 1.18** *What are the disallowed inputs for this algorithm?*

Assuming that no extra credit is possible, any grade less than zero or greater than 100 should be disallowed. We'll look at how to more formally specify this in the coming chapters. Next, we should try to some common inputs from 0 to 100. To start, trying a grade in each of the five GPA categories makes sense.

**Reflection 1.19** *Try a grade in each of the five GPA categories. What do you find?*

**Reflection 1.20** *What boundary cases should you try?*

Did you try grades on the boundaries of the categories (e.g., 60 and 90), grades below 50, and the boundary cases of 0 and 100? Here we have some issues.

```
>>> grade = 42
>>> GPA = int(grade / 10) - 5
>>> GPA
-1
>>> grade = 0
>>> GPA = int(grade / 10) - 5
>>> GPA
-5
>>> grade = 100
>>> GPA = int(grade / 10) - 5
>>> GPA
5
```

To fix these problems, we want to ensure that `GPA` never falls below zero or exceeds four. We can accomplish this with the `min` and `max` functions, which return the minimum and maximum values among their arguments. To fix the negative GPA issue, we want to return the maximum of `GPA` and zero.

```
>>> grade = 42
>>> GPA = int(grade / 10) - 5
>>> GPA = max(GPA, 0)
>>> GPA
0
```

To fix a GPA exceeding four, we want to return the minimum of the GPA and 4.

```
>>> grade = 100
>>> GPA = int(grade / 10) - 5
>>> GPA = min(GPA, 4)
>>> GPA
4
```

To fix both problems, we need to combine these solutions:

```
>>> GPA = int(grade / 10) - 5
>>> GPA = max(GPA, 0)
>>> GPA = min(GPA, 4)
```

At this point, you should try all of the test cases again to make sure everything works correctly. We will revisit testing in Section 5.5.

### Algorithm efficiency

Now let's look a little more closely at the third question at the beginning of this section:

3. How long does your algorithm take? Is it as efficient as it could be?

To determine how much time an algorithm requires, we could implement it as a program and execute it on a computer. However, this approach presents some problems. First, which programming language do we use? Second, which inputs do we use for our timing experiments? Third, which computer do we use? Once we make these choices, will they give us a complete picture of our algorithm's efficiency? Will they allow us to predict the time required to execute the algorithm on different computers? Will these predictions still be valid ten years from now?

A better way to predict the amount of time required by an algorithm is to count the number of ***elementary steps*** that are required, independent of any particular computer. An elementary step is one that always requires the same amount of time, regardless of the input. Examples of elementary steps are

- arithmetic operations,
- assignments of values to variables,
- testing a condition involving numbers or a character, and
- examining a character in a string or a number in a list.

Each of these things takes the same amount of time regardless of the numbers being operated upon, the types of values being assigned, or the values being examined. The number of elementary steps required by an algorithm is called the algorithm's ***time complexity***. By determining an algorithm's time complexity, we can estimate how long an algorithm will take on any computer, relative to another algorithm for the same problem.

### *Constant-time algorithms*

To make this more concrete, let's count how many elementary steps there are in our final Sphere Volume 2 algorithm on page 14, beginning with line 1:

```
1 | volume ← 4 ÷ 3
```

Line 1 contains two elementary steps: an arithmetic operation followed by an assignment of the result to a variable. Assignment and arithmetic (with two operands) are elementary steps because they always require the same amount of time regardless of the variable or the operands. Line 2, below, also contains two elementary steps for the same reason.

```
2 | volume ← volume × π
```

Lines 3–4 consist of a loop that instructs us to perform a similar arithmetic/assignment statement three times:

```
3 | repeat the following three times:
4 |     volume ← volume × r
```

Line 4 takes two elementary steps by itself, but it is executed three times, so lines 3–4 require a total of six elementary steps. Therefore, all together, this algorithm requires $2 + 2 + 6 = 10$ elementary steps.

The most important takeaway from this analysis, however, is that the SPHERE VOLUME 2 algorithm requires the same number of elementary steps regardless of what the input is. It executes ten elementary steps whether the input is 10 or 10,000. Therefore, we call it a ***constant-time algorithm***.

### *Linear-time algorithms*

Next let's analyze our last SYLLABLE COUNT algorithm from page 16. The first statement in this algorithm counts the number of vowels in the input *word*:

```
1 | count ← the number of vowels (a, e, i, o, u) in word
```

Although this may look like one elementary step at first glance, it is not. As we will discuss more in Chapter 6, a computer algorithm cannot just look at a word and instantly tell you how many vowels it has. Instead, it will need to check each letter, one at a time, counting the number of vowels that it sees. In other words, line 1 is equivalent to the following:

```
(a) | count ← 0
(b) | repeat for each letter in word:
(c) |     if letter is a vowel (a, e, i, o, u), then add 1 to count
```

Writing it in this way makes it more apparent that the number of elementary steps required by line 1 depends on the number of letters in *word*. More specifically, a *word* with $n$ letters will require $n$ iterations of the loop in lines (b)–(c); the longer the *word* is (i.e., the bigger $n$ is), the longer this will take. The body of the loop in line (c) requires at most two elementary steps: one to examine a letter and one to add to *count*. Therefore altogether, including the initialization of *count* to zero in line(a), there are $2n + 1$ elementary steps here.

Lines 2–3 of the algorithm contain a more explicit loop that is very similar to our rewritten version of line 1:

```
2 | repeat for each pair of adjacent letters in word:
3 |     if the letters are both vowels, then subtract 1 from count
```

The only difference from lines (b)–(c) above is that this loop looks at pairs of letters instead of individual letters and it subtracts from, rather than adds to, *count*. Regardless, once again, the number of elementary steps depends on the length of *word*; if there are $n$ letters in *word*, lines 2–3 repeat $n - 1$ times (because there are $n - 1$ pairs of adjacent letters in a *word* with $n$ letters). You can see this more explicitly in the trace table in page 16. In that case, the input contained $n = 7$ letters and there were $n - 1 = 6$ iterations of the loop. In the body of the loop, there are at most two elementary steps, so the entire loop contains $2(n - 1)$ elementary steps.

Finally, line 4 is much simpler:

```
4 | if word ends in e, then subtract 1 from count
```

We can safely say that the number of elementary steps in this line does *not* depend on the length of *word*. This conclusion relies on the assumption, which will be

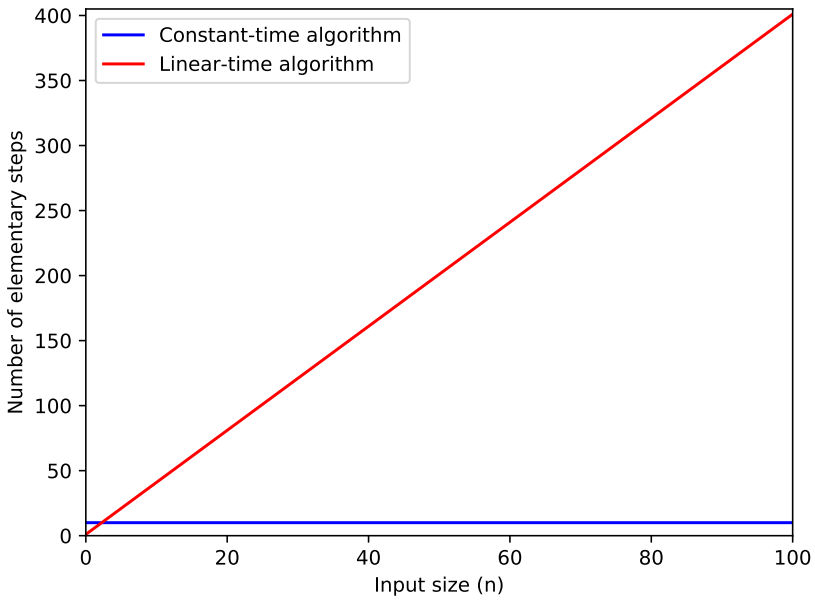



Figure 1.5 A comparison of constant vs. linear time complexity.

verified in Chapter 6, that we can look at the end of any *word* directly, regardless of its length. So we can say that this line requires at most two elementary steps, one to check the last letter in *word* and one to subtract from *count*.

Putting it all together then, the entire algorithm requires about $(2n+1)+2(n-1)+2 = 4n+1$ elementary steps. As with the sphere-volume algorithm, the exact number is not terribly important. The important thing to notice is that the time complexity of this algorithm is linearly proportional to the length of the input *word*. A linear function is one that contains $n$ but no higher powers of $n$ like $n^2$ or $n^3$. We call algorithms with time complexities that are linearly proportional to $n$, like SYLLABLE COUNT, ***linear-time algorithms***.

The real issue underlying time complexity is ***scalability***: how quickly the running time grows as the input gets very large. The difference between a constant-time algorithm and a linear-time algorithm is illustrated in Figure 1.5. The blue line represents the time complexity of a constant-time algorithm that always requires ten elementary steps regardless of how large the input becomes. The red line represents the time complexity of a linear-time algorithm that requires $4n+1$ elementary steps when the input has size $n$. Notice that the number of elementary steps in the linear-time algorithm grows proportionally to the size of the input. As the input gets larger, the difference between the constant-time algorithm and the linear-time algorithm grows much larger. Therefore, for a particular problem, if we could choose between a constant-time algorithm and a linear-time algorithm, especially if $n$ is

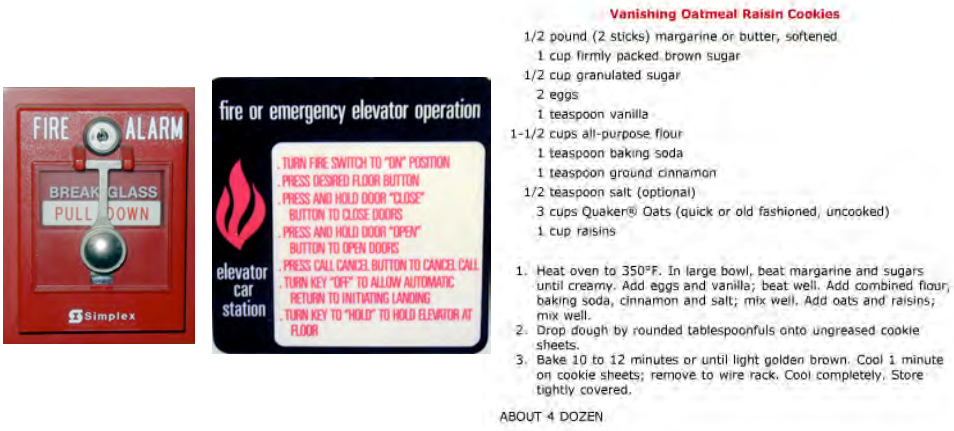



Figure 1.6 Examples of everyday algorithms: a fire alarm, an elevator, and a recipe.

large, we would clearly favor the constant-time algorithm. But even linear-time algorithms are generally considered to be very fast. As will see later, some problems require a lot more time to solve.

A linear-time algorithm is also said to have time complexity $\mathcal{O}(n)$ (pronounced "big oh of $n$"). The uppercase letter $\mathcal{O}$ is shorthand for "order;" we can also say that a linear-time algorithm has "order of $n$" time complexity. A constant-time algorithm is said to have $\mathcal{O}(1)$ time complexity. We will study time complexity in more detail in Section 6.7.

Although we have presented "looking back" as the last step in a four-step process, the issues we have discussed here are important to keep in mind in every step. The better your product is at each step, the less work you will have to do later to clean it up. Designing algorithms can be a tricky business and first impressions can sometimes be deceiving. For example, you have already seen that the number of lines in an algorithm is often unrelated to its time complexity. Similarly, techniques that seem to work at first glance may not work for some inputs. As in any worthwhile endeavor, a careful and deliberate approach will pay dividends in the long run.

## Exercises

1.4.1. Identify three algorithms from your everyday life and critique them with respect to readability, correctness, and efficiency. Some examples of everyday algorithms are shown in Figure 1.6.

1.4.2. What characteristics, other than the ones we discussed, might make one algorithm better than another?

1.4.3. For each of the following algorithms, demonstrate that it is correct by testing it with at least two common inputs and all boundary inputs you can identify. Say what the algorithm's output is in each case. Also, if there are any inputs that should not be allowed, identify those.

(a)* your revised SPHERE VOLUME algorithm from Exercise 1.2.9

(b)* the COUNT VOTES algorithm from Exercise 1.2.7

(c) the SURFACE AREA OF A BOX algorithm from Exercise 1.2.6

(d)

**Algorithm** MAXIMUM VALUE IN A LIST

**Input:** a list of *numbers*

```
1  maxSoFar ← first item in numbers
2  repeat for each item in numbers:
3      if item > maxSoFar, then assign maxSoFar ← item
```

**Output:** *maxSoFar*

(e)

**Algorithm** DISTANCE TO LIGHTNING STRIKE

**Input:** elapsed number of *seconds*

```
1  speed of sound ← 343 m/s
2  distance ← seconds × speed of sound
3  distance ← distance ÷ 1000
```

**Output:** *distance*

(f)

**Algorithm** RAISE TO THE FIFTH

**Input:** a *number*

```
1  product ← 1
2  repeat 5 times:
3      product ← product × number
```

**Output:** *product*

(g)

**Algorithm** COUNT PRONOUNS

**Input:** a *text*

```
1  countFeminine ← 0
2  countMasculine ← 0
3  repeat for each word in the text:
4      if the word is she, then add one to countFeminine
5      if the word is he, then add one to countMasculine
6  ratio ← countFeminine ÷ countMasculine
```

**Output:** *ratio*

1.4.4. For each of the algorithms in the previous exercise, estimate the number of elementary steps and decide whether it is a constant-time or a linear-time algorithm.

1.4.5. Suppose that you have been asked to organize a phone tree for your organization to personally alert everyone with a phone call in the event of an emergency. You have to make the first call, but after that, you can delegate others to make calls as well. Who makes which calls and the order in which they are made constitutes an algorithm. For example, suppose there are only eight people in the organization, and they are named Amelia, Beth, Caroline, Dave, Ernie, Flo, Gretchen, and Homer. Then here is a simple algorithm:

**Algorithm** ALPHABETICAL PHONE TREE

**Input:** a list of eight people and their phone numbers

1 Amelia calls Beth.
2 Beth calls Caroline.
3 Caroline calls Dave.
⋮
7 Gretchen calls Homer.

**Output:** none (but phone calls were made)

For simplicity, assume that everyone answers the phone right away and every phone call takes the same amount of time.

(a) For the phone tree problem, identify at least two criteria that would make one algorithm better than another. For each criterion, design an algorithm that satisfies it.

(b) In the interest of safety, one criterion for a phone tree would be to ensure that everyone is notified as soon as possible. Design an algorithm that ensures that all eight people are notified in a chain of at most three calls. At the outset, only Amelia is aware of the emergency. (Multiple calls can be made simultaneously.)

(c) Extend the algorithm you designed in the previous question to an arbitrarily large number of people. In general, how many people are called simultaneously during any step $t = 1, 2, 3, \ldots$? First, think about how many calls are made during steps 1, 2, and 3 in your algorithm. Then think about how many calls would be made during time steps 4, 5, and 6. Can you generalize this process to any step $t$?

## 1.5 SUMMARY AND FURTHER DISCOVERY

In this chapter, we outlined the four steps in the computational problem solving process. First, we need to understand the problem we are trying to solve, viewed as the relationship between its inputs and the desired output. This may sound obvious, but you would be surprised at how much time is often wasted solving the incorrect problem! Solving some small examples by hand at this point can often help, and

illuminate potential pitfalls. At the end of this step, the problem is viewed as a functional abstraction because we understand what an algorithm for it should do, but not yet what the algorithm looks like.

Second, we want to design an algorithm to solve the problem. It helps to use top-down design to decompose the problem into smaller subproblems. Then we can design algorithms for the simplest problems first and work our way up the decomposition tree in a bottom-up fashion. Algorithm design is often the most challenging of the four steps, which is why writing algorithms in pseudocode is so valuable. Pseudocode allows us to think about how to solve the problem without being distracted by the more demanding requirements of a programming language. In our algorithms, we saw four categories of algorithmic statements:

1. assignment statements that assign a value to a variable,
2. arithmetic statements,
3. loops that repeat a set of statements some number of times, and
4. conditional statements that make decisions.

It may surprise you to know that these four types of statements are sufficient to write any algorithm imaginable! So writing algorithms, and programs, in large part amounts to putting this small palette to work in creative ways. As we progress through this book, we will incrementally learn how to use these kinds of statements in myriad combinations to solve a wide variety of problems.

In the third step, we translate the algorithm into a program in Python. We started our introduction to programming by using variables, arithmetic, and simple functions. In the next chapter, you will begin to write your own functions and incorporate them into longer programs. By the end of this book, you will be amazed by the kinds of things you can do!

Fourth, we need to remember to "look back" at our algorithms and programs in a process of continual refinement. Just because a program seems to work on a few simple inputs does not mean that it cannot be improved. We always want to strive for the clearest, most efficient, and fairest solution we can.

### Notes for further discovery

The first epigraph at the beginning of this chapter is from an article by computer scientists Michael Fellows and Ian Parberry [16]. A similar quote is often attributed to the late Dutch computer scientist Edsger Dijkstra.

The second epigraph is from the great Donald Knuth [33], Professor Emeritus of The Art of Computer Programming at Stanford University. When he was still in graduate school in the 1960s Dr. Knuth began his life's work, a multi-volume set of books titled, *The Art of Computer Programming* [31]. In 2011, he published the first part of Volume 4, and has plans to write seven volumes total. Although incomplete, this work was cited at the end of 1999 in *American Scientist*'s list of

"100 or so Books that Shaped a Century of Science" [42]. Dr. Knuth also invented the typesetting program TEX, which was used to write this book. He is the recipient of many international awards, including the Turing Award, named after Alan Turing, which is considered to be the "Nobel Prize of computer science."

Guido van Rossum is a Dutch computer programmer who invented the Python programming language. IDLE is an acronym for "Integrated DeveLopment Environment," but is also considered to be a tribute to Eric Idle, one of the founders of Monty Python.

The "Hello world!" program is the traditional first program that everyone learns when starting out. See `http://en.wikipedia.org/wiki/Hello_world_program` for an interesting history.

As you continue to learn Python, it will be helpful to add the following documentation site to your "favorites" list: `https://docs.python.org/3/index.html`. There are also a list of links and references for commonly used classes and functions (Appendix A) on the book website.

There are many good resources for learning more about ethics in computing and data science. The ACM Code of Ethics and Professional Conduct (`https://www.acm.org/code-of-ethics`) is the main code followed by computing practitioners around the world. For more in-depth coverage of moral theories and ethics, we recommend *Computer Ethics* by Deborah Johnson [27], *Ethics of Big Data* by Kord Davis [11], and *Ethical and Secure Computing* by Joseph Migga Kizza [30]. *Race after Technology* by Ruha Benjamin [6], *Algorithms of Oppression: How Search Engines Reinforce Racism* by Safiya Umoja Noble [44], and *Weapons of Math Destruction* by Cathy O'Neil [45] delve deeper into the potentially damaging social impacts of computing.

Finally, a note about "big oh" notation. Our use of $\mathcal{O}(n)$ is actually a slight, but common, abuse of notation. Formally, to say that an algorithm has $\mathcal{O}(n)$ time complexity means that its time complexity is asymptotically *at most* linearly proportional to $n$. In other words, a constant-time algorithm also has $\mathcal{O}(n)$ time complexity! The correct notation is $\Theta(n)$ ("big theta of $n$"), but "big oh" notation is used so frequently in practice that we chose to also use it, despite some discomfort.