Computer Science with Applications

Anne Rogers and Borja Sotomayor

Sep 26, 2022

CONTENTS

PREFACE

Welcome to the preface! In time, this preface will contain more information about the purpose of this book, suggested ways of reading it, etc. For now, it just contains some basic information on some notation you'll see throughout the book.

1 Example code

Some chapters in the book rely on example code that we provide through the following repository on GiHub:

https://github.com/cs-apps-book/examples

When referring to individual example files, we will use the full path within that repository. For example, `getting-started/code-organization/arithmetic.py`

2 Additional information boxes

Throughout the book, you will encounter four types of information boxes:

Technical Details

This kind of box provides deeper technical details about something that was just explained. These technical details are not essential to understanding the concept or skill that precedes this box, and are mostly provided for readers who want to dig a bit deeper (specially if you're the kind of learner who has an easier time understanding new concepts with those lower-level details).

This means that you can usually safely skip these boxes on your first read through a chapter. However, if you do so, we still recommend revisiting these boxes once you become more comfortable with the material.

Common Pitfalls

This kind of box alerts you to common pitfalls that beginners sometimes make. Make sure to read this box so you don't fall into those pitfalls yourself!

Debugging Tips

As we'll describe later in the book, debugging issues in your code can sometimes feel like solving a murder mystery where you are both the detective and the murderer. Pinpointing the cause of errors in your code is a skill that takes time to build, so we've made sure to include debugging tips throughout the book so you know what to look out for when your code fails in certain ways.

Note

Notes are used to clarify certain concepts, or to call out information that doesn't fit into any of the above boxes.

3 Special sections

Sometimes, entire sections of the book will revolve around the kind of technical details we would include in a Technical Details box (as described above). Look out for sections with the gears icon like this:

When you see a section like this, remember that you can safely skip it if you want to, but may want to revisit it later on.

Part I.

Getting Started

CHAPTER ONE

COMPUTATIONAL THINKING

When Hurricane Harvey made landfall in the US in August 2017, it caused a massive amount of damage and flooding, leaving many people stranded in their homes. While considerable resources were devoted to rescuing people from the floods, one of the biggest challenges was determining *who* needed to be rescued. Ordinarily, a call to 911 could alert the authorities about a rescue situation, but most 911 call centers were backlogged and could not process most incoming calls. Instead, many people took to social media to post about their situation, using the `#HarveySOS` hashtag to alert others that they were in need of help or needed to be rescued.

This ensured that information about who needed to be rescued was available to anyone who could act on it, including the Coast Guard and a number of volunteer organizations that sent boats into the flooded areas on rescue missions. However, sifting through all the uses of `#HarveySOS` is not easy: someone on a rescue boat can easily filter Twitter or Facebook posts by the `#HarveySOS` hashtag, but there is no easy way to find posts made close to the boat's location. On top of that, some posts may refer to people who have already been rescued, so a rescue boat could be acting on outdated information.

To facilitate rescue operations, the data needs to be presented to authorities and volunteers in a way that enables them to act on it, and there needs to be mechanisms to indicate whether someone who used the `#HarveySOS` hashtag has already been rescued. While this could be done manually, with volunteers reading each post and acting as intermediaries between rescuers and rescuees, this kind of task lends itself nicely to being automated using computational methods.

In fact, that is what ended up happening in this case: a group of volunteers around the US and around the world developed tools to easily extract the `#HarveySOS` posts, put them in a database, display them on an online map that allowed easy access to all the information that rescue teams needed. Thousands of people were rescued thanks to this effort.

The Harvey rescue map highlights how computation can be used to solve problems outside the realm of Computer Science. The people behind the Harvey rescue map were not developing the next killer app or some advanced piece of machinery. They had a problem that they cared deeply about, and they used computers to solve that problem faster and better.

In fact, this book is written for people who don't intend to become computer scientists or engineers. It is intended for people who want to harness the power of computation to solve problems that matter to them and, more generally, to their chosen field of study. This may be anything from analyzing some census data and testing a hypothesis about it to performing a complex simulation of housing segregation models.

Doing so involves a lot more than just learning "how to program." At the end of the day, learning how to write a computer program is not that hard. Take, for example, the following piece of Python code:

```python
print("Who's a good dog?")
if number_of_dogs == 1:
    print("You're a good dog!")
else:
    print("You're all good dogs!")
```

Without getting into the weeds, you can tell that there appears to be a mechanism to print messages (`print`) and a mechanism to perform one action or another based on some condition (`if number_of_dogs == 1 ... else`). While this piece of code is trivial, you can already tell that it is not written in some esoteric and cryptic language, understandable only by seasoned computer scientists. Computer programs actually tend to be fairly human-readable, and picking up the basics of a programming language is less intimidating than you may think.

Programming is ultimately a *tool*. The challenge lies in learning how to *think computationally*: how to translate a problem into terms that a computer can solve efficiently. This is not unlike what happens in many other disciplines and crafts: you could learn how to use a variety of woodworking tools in a few hours, but it takes a lot more time than that to learn how to build a decent cabinet or carve a sculpture out of wood.

In a sense, the goal of this book is to teach you to think computationally rather than to simply teach you to program. Don't get us wrong: you *will* learn a *lot* of programming in this book, but it will be in service of computational thinking. Our ultimate goal is for you to be able to take problems that matter to you and figure out how to get a computer to solve them.

So, what is computational thinking? Jeannette Wing, in a seminal 2006 article on the subject, wrote:

> Computational thinking involves solving problems, designing systems, and understanding human behavior, by drawing on the concepts fundamental to computer science.

While the field of computer science is enormous and Wing's article discusses many ideas, we will focus on a few concepts and skills that are integral to computational thinking:

- Decomposition and Abstraction
- Modeling
- Algorithms
- Complexity

In the rest of this chapter we will discuss each of these in more detail and, throughout the book, we will often highlight the material we present relates back to these four points.

1.1 Decomposition and abstraction

Computational thinking can help us solve large and complex problems in a more manageable way. For example, consider the following problem: how can we find out, after a snow storm in Chicago, whether a given street in the city has been plowed?

In 2012, the City of Chicago launched a website called Plow Tracker that, originally, only allowed users to see the location of the city's snow plows during a snow storm (at all other times, the Plow Tracker did not display anything). While that data is useful (in fact, the City of Chicago is a pioneer in openly publishing government data), it doesn't solve the problem we posed, unless we watch the plow tracker intently during a snow storm and wait to see whether a snow plow passes through a given street.

So, several civic data enthusiasts in Chicago (most of whom, it should be noted, were not computer scientists) set out to solve this problem and publish their solution. They built a website, called ClearStreets, that allows anyone to determine whether a snow plow has passed through any address in Chicago since the last snow storm. Building this system required breaking up this problem into multiple parts, which we will describe below.

Note

The ClearStreets website no longer collects snow plow data, not does it allow you to query their database, partially because the City of Chicago's own Plow Tracker website now provides the functionality that ClearStreets originally set out to provide.

However, ClearStreets is still a great case study on how ordinary citizens with some basic computational skills can build interesting and useful systems using openly-available data (which eventually nudged the city towards offering that same functionality themselves). The ClearStreet website itself still exists, and includes details and documentation on how the ClearStreets system was originally built. We encourage you to check it out after you've read this chapter!

1.1.1 Obtaining the plow coordinates

During a snow storm, the city's Plow Tracker produces a sequence of GPS locations for every active plow. ClearStreets needs a program to monitor the Plow Tracker during a snow storm and store those GPS coordinates for use in subsequent queries. They end up with a sequence of latitude and longitude coordinates at specific points in time, like this:

```
...
Plow 42 was at 41°47'16.0"N 87°36'03.0"W at 2017-03-14 11:55:19am
Plow 42 was at 41°47'16.0"N 87°35'53.9"W at 2017-03-14 11:56:20am
...
```

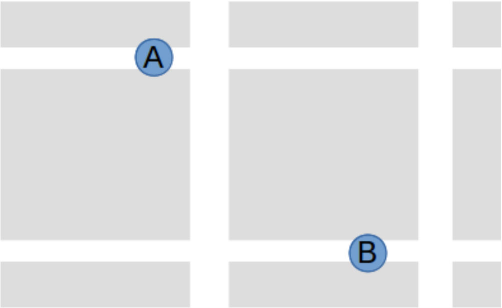
1.1.2 Converting the data into paths

The data obtained from the Plow Tracker aren't particularly user-friendly: if someone wanted to find out whether their street had been plowed, they would need to know the latitude and longitude of their home. However, even if they converted each latitude/longitude to a street address, the data would not include all the addresses passed by the snow plow. For example, the coordinates shown above correspond to the following two addresses:

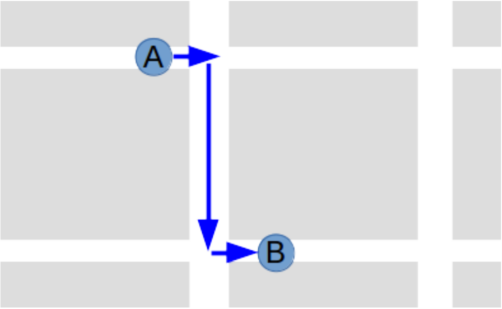
- 1010 East 59th Street
- 1130 East 59th Street

What if they live at 1116 East 59th Street? The coordinates of that address are not in the data we collected, but the snow plow must've passed through that address (East 59th Street is a one-way street, and you can't get from 1010 to 1130 without passing 1116).

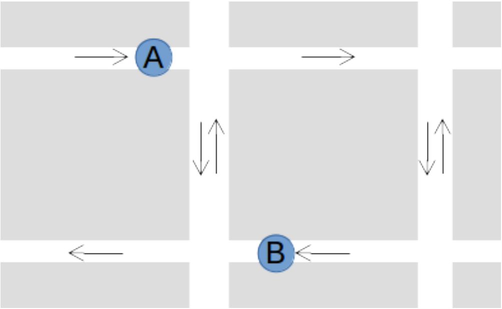
So, we need to extrapolate the paths of the snow plows from those sequences of coordinates. As it turns out, this can be a pretty complex problem. In the above example, we had a one-way street, but consider a plow that reports the following two locations:



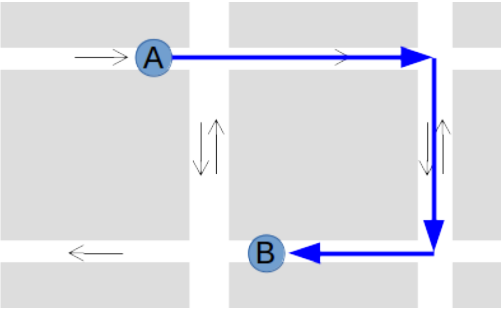
We might assume the plow took the shortest path from A to B:



But that doesn't take into account Chicago's many one-way streets:



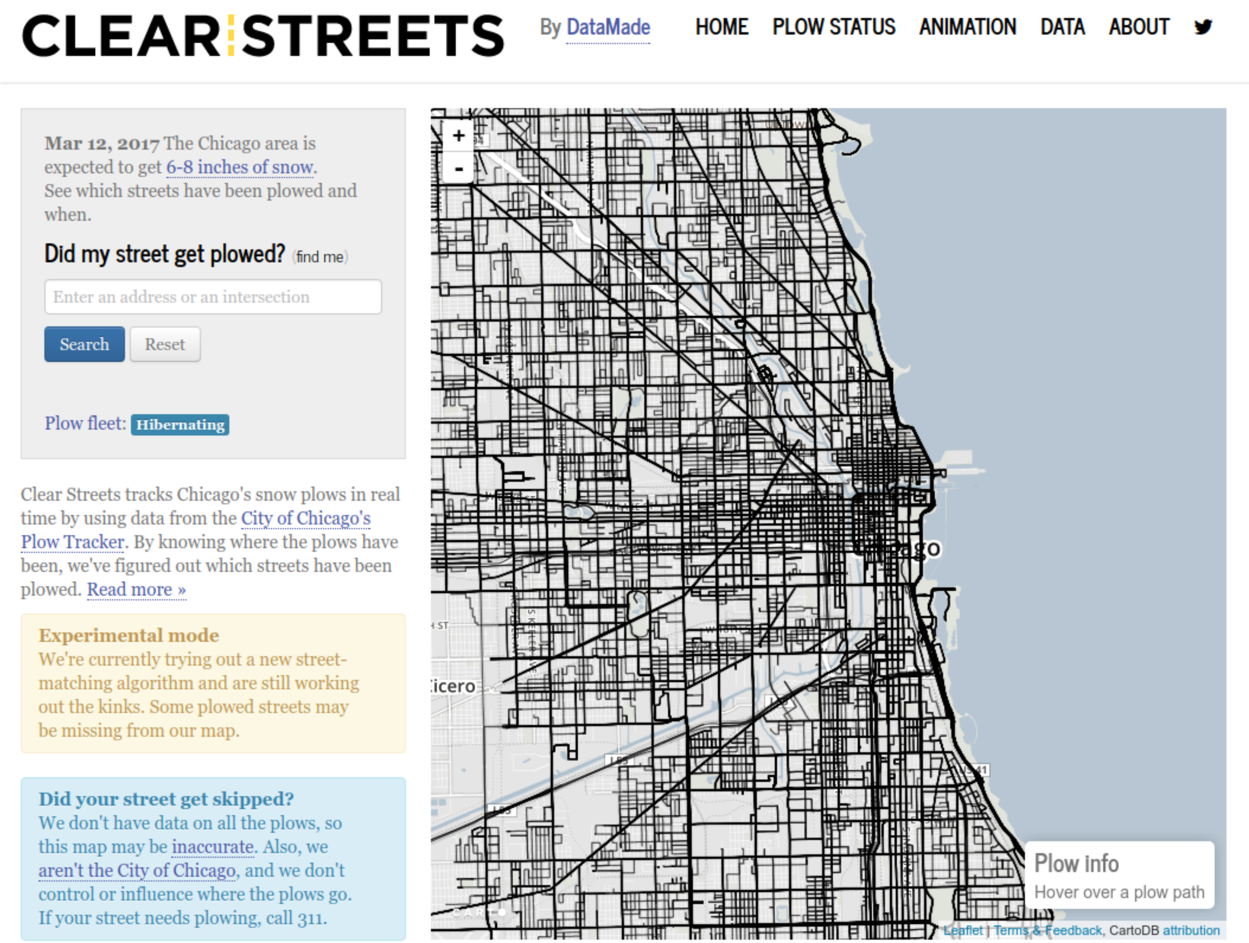
Taking the shortest path would actually involve going the wrong way down a one-way street! It's more likely that the plow followed this path:



Fortunately, converting GPS coordinates to paths is a well-studied problem, and there are many existing software solutions available. ClearStreets actually uses a system called OSRM (Open Source Routing Machine) to convert the GPS coordinates from the Plow Tracker into plausible paths on a street map.

1.1.3 Visualizing and querying the data

Finally, ClearStreets had to provide an interface for users to query the data and easily visualize it. These tasks are themselves quite complex and are handled using a combination of tools including CSS, HTML, JavaScript and CartoDB. The end result looked like this:



Mar 12, 2017 The Chicago area is expected to get 6-8 inches of snow. See which streets have been plowed and when.

Did my street get plowed? (find me)

Enter an address or an intersection

Search Reset

Plow fleet: Hibernating

Clear Streets tracks Chicago's snow plows in real time by using data from the City of Chicago's Plow Tracker. By knowing where the plows have been, we've figured out which streets have been plowed. Read more »

Experimental mode
We're currently trying out a new street-matching algorithm and are still working out the kinks. Some plowed streets may be missing from our map.

Did your street get skipped?
We don't have data on all the plows, so this map may be inaccurate. Also, we aren't the City of Chicago, and we don't control or influence where the plows go. If your street needs plowing, call 311.

By DataMade. Map hosting provided by CartoDB.

Through the ClearStreets website, users were able to type in an address, and then see on the map whether or not a plow passed through an area.

The ClearStreets example shows how computational thinking involves *decomposing* complex problems into smaller, more manageable pieces that provide a layer of *abstraction* over all the details of the problem. This means that, instead of trying to solve one large problem, we can solve several smaller problems in a way that makes it easier to divide up the work among a team, as well as making each individual piece easier to test and maintain in the long run.

This problem-solving approach has several benefits. First, it separates different parts of the program. The part of the program that converts coordinates to paths on a map only has to focus on solving that particular sub-problem, without being concerned with how the other parts of the program work. In fact, in a well-designed program, we should be able to replace the mechanism that converts points to paths (e.g., because we developed a better algorithm for this task) without the other parts of the program needing to know about this change. Similarly, if a given piece of code focuses on one specific task, it will be easier to test whether that code is implemented correctly.

It also allows us to reuse code across projects. ClearStreets used an existing piece of software, OSRM, so that it would not have to solve the coordinates-to-paths problem from scratch. In fact, in most situations, it is considered better to solve a problem by using existing code (either written by yourself or your team, or provided by some other project, like OSRM) instead writing the entire solution from scratch.

Of course, when we decompose a single problem into multiple parts, we must ensure that all of the pieces can work together. This task is especially challenging when different people write each piece, which is very common in many software projects. To address this challenge, each piece must clearly define some sort of *interface* that specifies what kind of data it expects and what kind of data it produces.

For example, it would be very easy to get a mismatch between the code that collects the plow coordinates and the code library used to convert coordinates to paths. The person who was in charge of writing the code to collect plow coordinates might decide to write code that produces a sequence of addresses as output, because that format seems intuitively useful for the overall task of answering questions about whether a specific street has been plowed. While this choice has some merit, it does not work well with the library that converts coordinates to paths, which expects a sequence of GPS coordinates (latitude/longitude) as input.

Throughout the book, we will see that there are various ways to decompose a problem into smaller pieces, but that it will be important to define good interfaces when we do so. We will also be emphasizing that these smaller pieces need to be testable, and we'll see that there are tools that allow us to automatically run tests on specific parts of our code.

1.2 Modeling

Another aspect of making complex problems more manageable is to distill the problem into its essential components, so that we don't waste time and effort on aspects of the problem that are irrelevant to solving it. This is different from decomposition and abstraction, in which we try to break down a complex problem into more manageable sub-problems. Now, we will focus coming up with a *model* that captures the essence of a given problem while allowing a computer program to solve it.

Modeling is not unique to computational thinking. For example, you may be familiar with economic models, statistical models, or mathematical models. In general, models formalize an aspect of the real world in a way that allows us to reason about it.

While modeling is not unique to computational thinking, it is a central aspect of it. As we will see throughout the book, a programmer has a number of *data structures* at their disposal when writing a program. Solving a problem computationally requires a programmer to figure out how to use data structures to represent aspects of the problem. Or, to put it another way, the programmer must figure out how to *model* the problem using those data structures before performing any computations.

For example, let's take a look at the problem of matching medical students to residency training programs. When someone graduates from medical school, they next complete a *residency* at a hospital under the supervision of an attending physician (a more senior physician that has already completed a residency). The National Resident Matching Program, also known, somewhat ominously, as "The Match", matches medical students to resident positions in hospitals.

This problem lends itself nicely to a computational solution but, before we can think about specific procedures to match students to hospitals (which we'll discuss in the next section), we first need to decide on a model for this problem. In particular, consider that a student could say something like, "I prefer a hospital in the West Coast, but I'm open to hospitals in the Midwest as long as they're highly-ranked", and a hospital could say, "We prefer students from top schools, but will take truly exceptional students from lower-ranked schools". We need to represent these preferences in a way that a computer program will be able to process.

One approach is to model every possible preference. For students, we would include location, quality of program, specialty, and so on, and for hospitals, we would include ranking of the student's medical school, the student's specialty, and so on. We would attach a numeric score to each, where 0 is "not important at all" and 10is "very important". So, if a student scored "West Coast location" with a 10 and "Midwest location" with a 5, the program would try to match that student with a West Coast hospital (or, possibly, a Midwest hospital if that hospital has other characteristics that score highly for that student). Notice that we would also have to model certain attributes of the students and the hospitals, like where the hospital is located and what specialty the student wants.

Coming up with a model that captures every possible preference can quickly become intractable. While there are some problems where such an approach may be appropriate, this model should boil down the problem to its very essence. If we think about it, students are ultimately providing a series of rules to determine whether one hospital is better than another. So, a much simpler model would be to ask each student to rank the hospitals they are interested in. Similarly, hospitals would be asked to rank the students they are willing to hire as residents. This is, in fact, what The Match asks them to do.

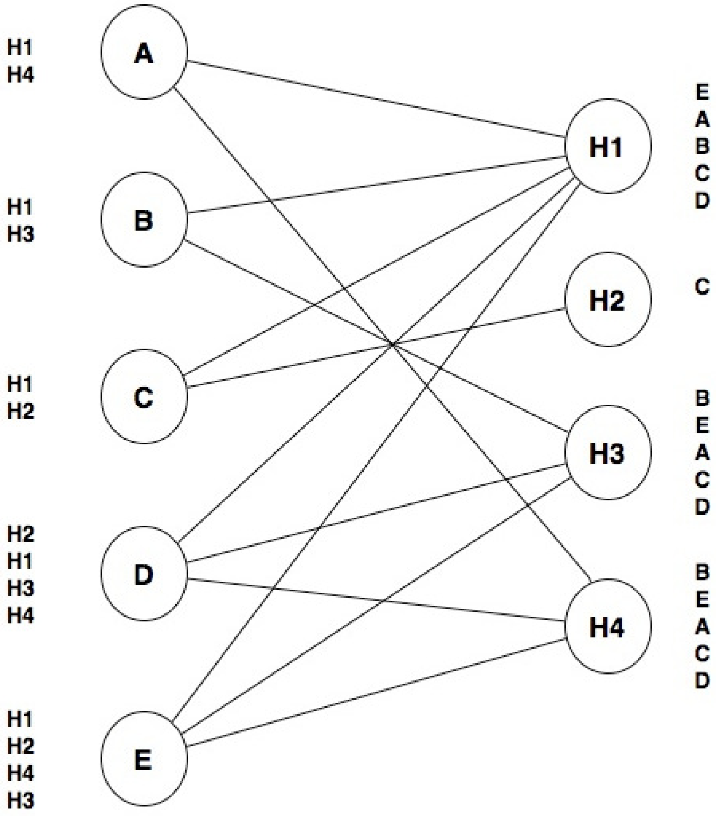
So, if we have five students (A, B, C, D, E) and four hospitals (H1, H2, H3, H4), we could model each student's preferences as a ranked list of hospitals:

- Student A: H1, H4
- Student B: H1, H3
- Student C: H1, H2
- Student D: H2, H1, H3, H4
- Student E: H1, H2, H4, H3

And each hospital's preferences as a ranked list of students:

- Hospital H1: E, A, B, C, D
- Hospital H2: C
- Hospital H3: B, E, A, C, D
- Hospital H4: B, E, A, C, D

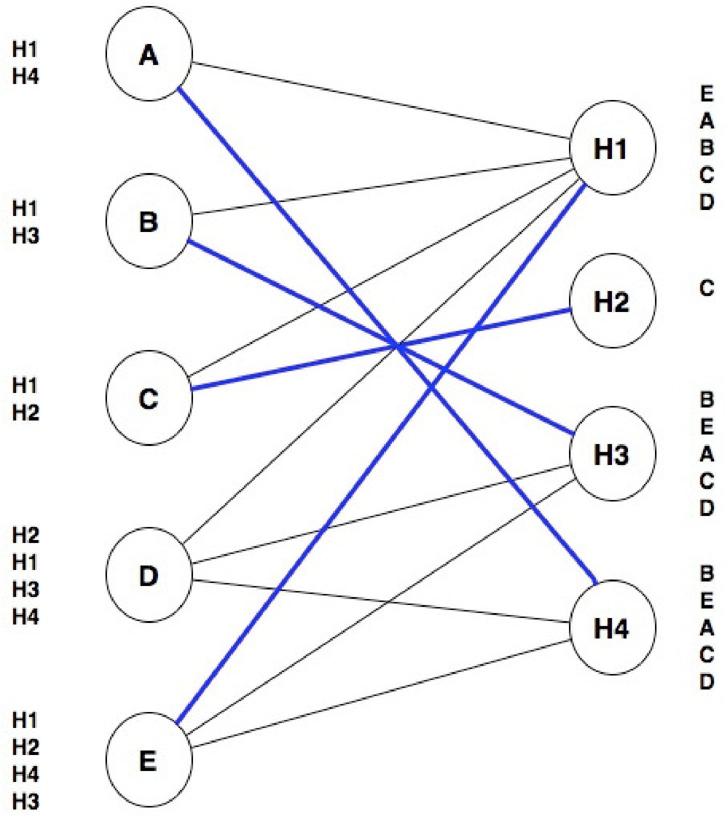
We can further simplify our model. It doesn't make sense to keep track of unrequited preferences. For example, Hospital H3 is willing to hire any of the five students, but only students B, D, and E listed H3 as a choice. We could represent these preferences graphically by connecting a student and a hospital with a line if and only if the student and the hospital both listed each other in their list of preferences. So, we would have something like this:



As it turns out, this is a data structure called a *graph*, where we model certain things as nodes (in this case, the students and hospitals) and we model a relationship between the nodes using lines or *edges* (in this case, whether a student and a hospital mutually prefer each other). Notice that there are no edges that connect students to other students or hospitals to other hospitals; this is a type of graph called a *bipartite graph* because the nodes are divided into two categories (students and hospitals) and each edge connects a node from one category with a node from the other.

In this graph, an edge indicates a *potential match*. Notice that each student also has a ranked list of hospitals so, given two potential matches, we can tell which one is more desirable to the student. Likewise, the hospitals have ranked lists of students.

Ultimately we have to produce the actual matches between students and hospitals, so our model needs to distinguish between edges that represent potential matches, and edges that represent actual matches. We can do so by having two types of edges: one for potential matches and one for actual matches. Graphically, we can represent actual matches with a thicker blue line:



Keep in mind that any student will be matched with at most one hospital. Notice that our model allows some students to be unmatched (in this case, Student D is not matched with any hospital).

Hospitals usually have multiple residency programs, each with multiple slots, but we're going to assume that each hospital hires at most one resident for simplicity.

For this particular problem, this graph is our model. Anything that can't be represented in this graph, and thus in our model, is outside of the scope of our computational solution. In an actual computer program, we would probably use an existing software library that allows us to manipulate and query graphs, allowing us to implement the model we just designed. This is an example of abstraction: we don't have to worry about the internal details of how a graph data structure is implemented, because we can probably find a software library that provides operations like "Add node called X to graph" , "Check whether node X and Y are connected", "Change edge between Y and Z from potential match to actual match", etc., in the same way that the developers of ClearStreets abstracted away the details of converting GPS coordinates to paths by using the OSRM library.

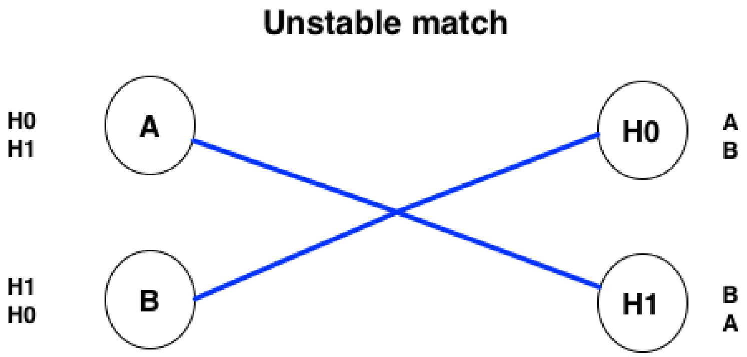
Now that we have defined our model, we should also define what it means to have a "good match" in the context of our model. As it turns out, computer scientists have thoroughly studied matching problems like this one, and there is something called a "stable match" that suits our needs. A stable match is one where, for every possible pairing of a student S with a hospital H, the following two conditions are not both true:

1. S is unmatched, or would prefer to go to H over the hospital that S is currently matched with.
2. H is unmatched, or would prefer S over the student it is currently matched with.

A more intuitive way to think about this is that a solution is stable if there is no (student, hospital) pair such that they both prefer each other over their current match (or lack of match).

Our earlier matches are stable, even in the case of student D. For all the possible pairings of D with a hospital (D with H1, D with H2, D with H3, and D with H4), the first condition is true, because D is unmatched, but the second condition is never true: all the hospitals either did not want D, or they're matched with a student that they ranked higher than D.

As a counter example, here is a simple unstable match:



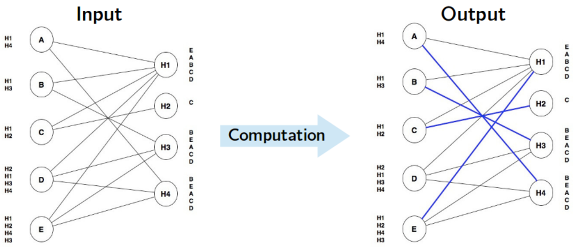
Both matches break the conditions for stability: A would prefer to go to H0, *and* H1 would prefer B over A. Similarly, B would prefer to go to H1, *and* H0 would prefer A over B.

This example highlights how we need to *model* a problem in a way that allows us to operate on it computationally before we can solve it. Since we haven't started writing code yet, the graph model we present may still seem abstract. Graphs, however, are a very common data structure. If we can model a problem as a graph, there is a good chance that we will be able solve it computationally.

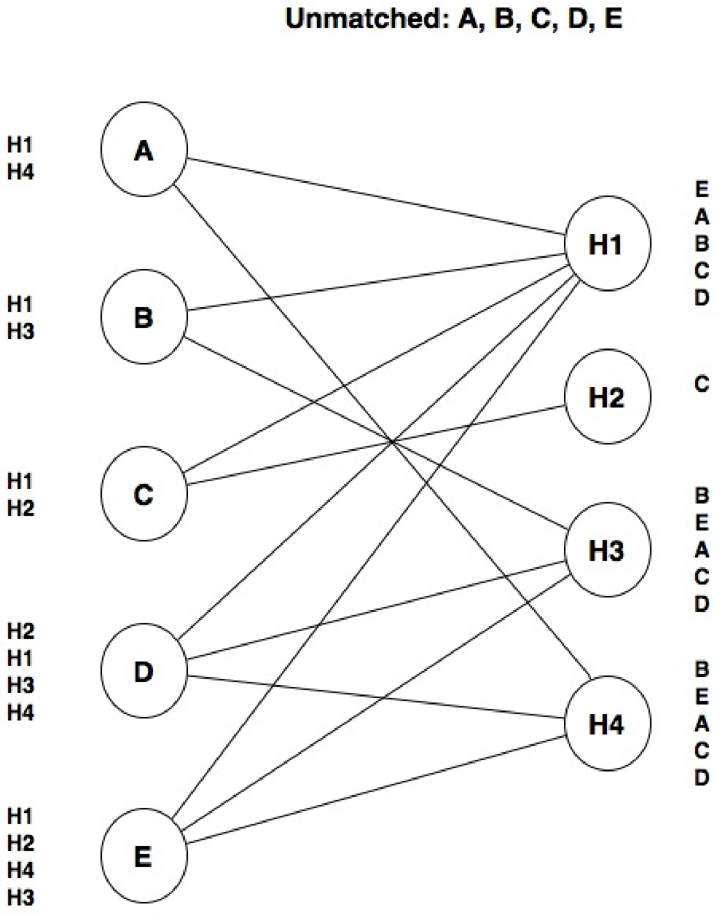
Of course, graphs are not the only way to model a problem. As we'll see throughout the book, we have many data structures at our disposal and picking the right data structure is crucial to modeling a problem. Not just that, we can also build new data structures using a paradigm called object-oriented programming that will give us considerable flexibility to model the information we need to solve a problem computationally.

1.3 Algorithms

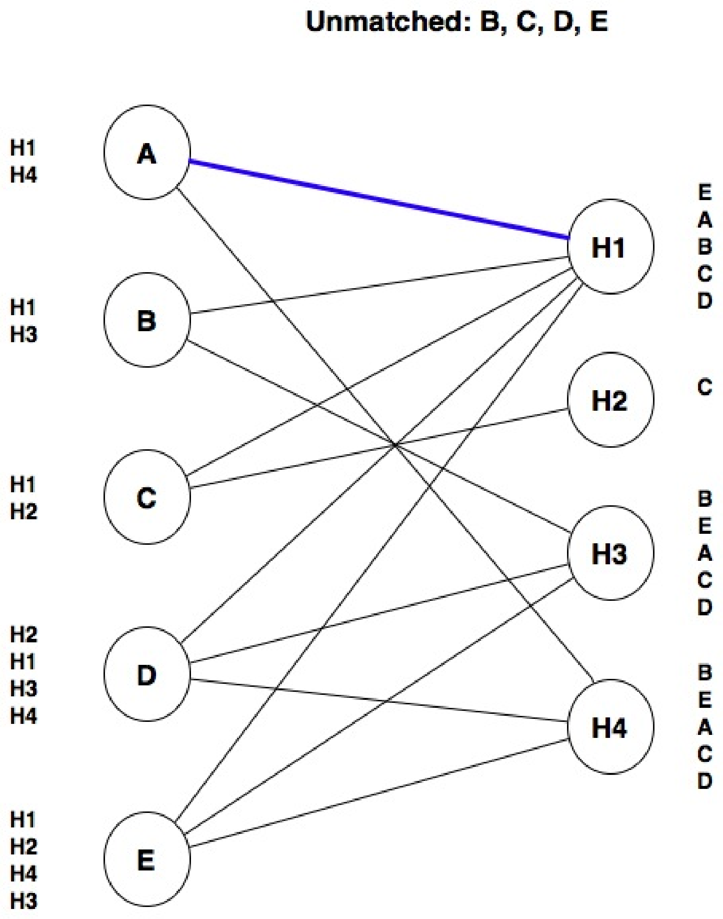
When explaining the National Resident Matching Program we provided a simple example where some students and hospitals ranked each other. This information, which we model as a graph, is the *input* of our problem, while the *output* of our problem is a stable match for the given input. In fact, at its most basic level, this is what a program does: it takes some input and, based on that input, performs a series of computations that produce some output:



However, while we defined what constitutes a stable match, we did not explain *how* we arrived at the stable match we presented. We actually did so by going through each student, and trying to make the best possible match for that student, possibly unmatching another student if doing so results in a better match. Let's work through this one step at a time. Our input looks like this:

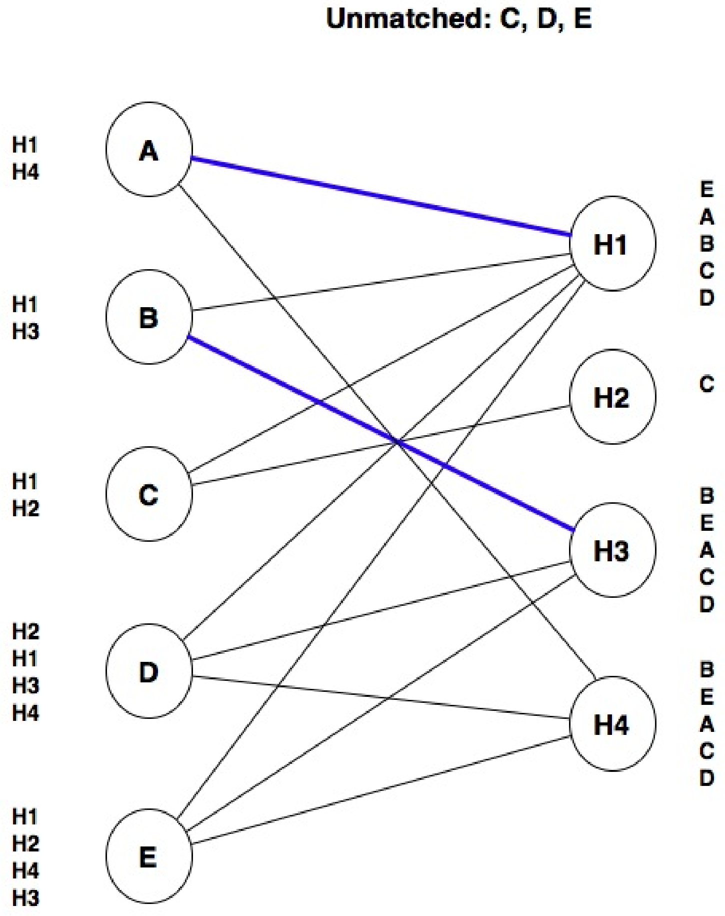


Notice that, at this point, all students are unmatched. We will start with student A. Since A's top choice is H1, we will make that match:

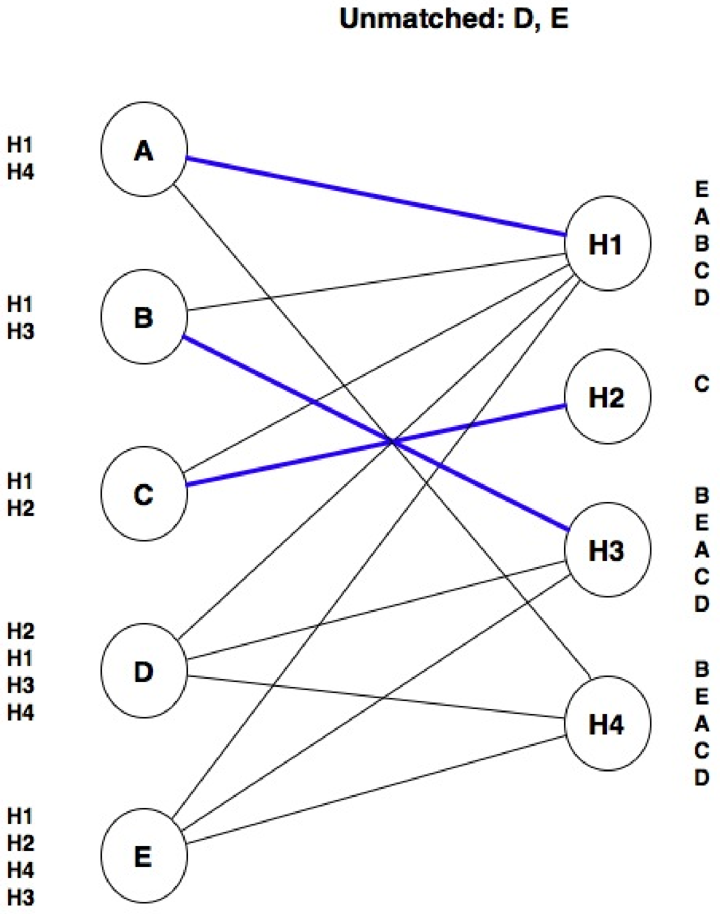


At this point, this is a *tentative* match because there could find a better match as we look at the rest of the students.

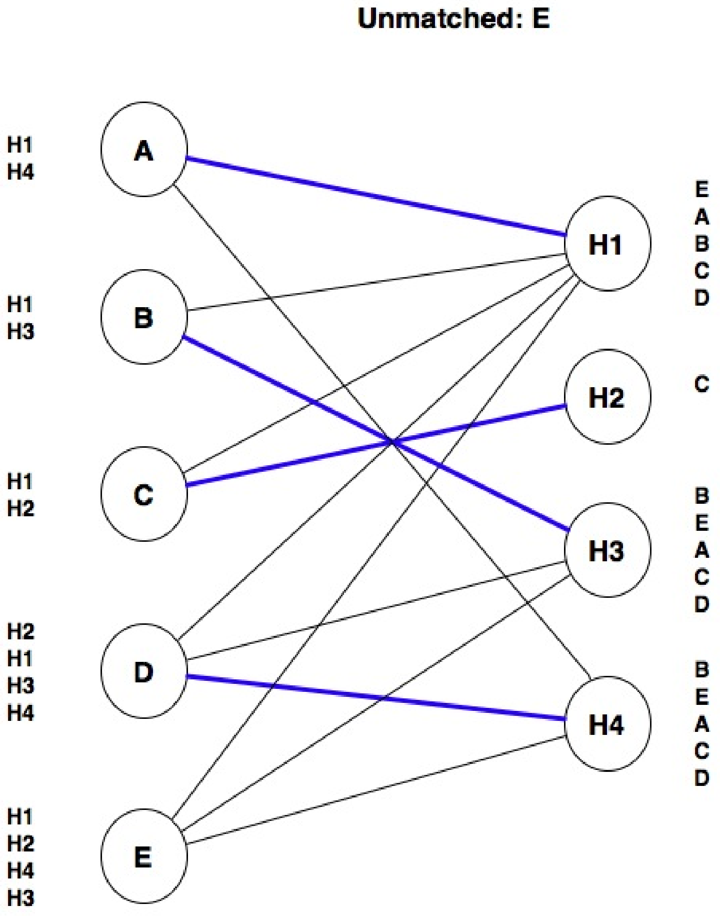
We next look at B: that student also wants hospital H1. Because H1 prefers A over B, we don't match B with H1. Instead, we tentatively match B with its next choice, H3:



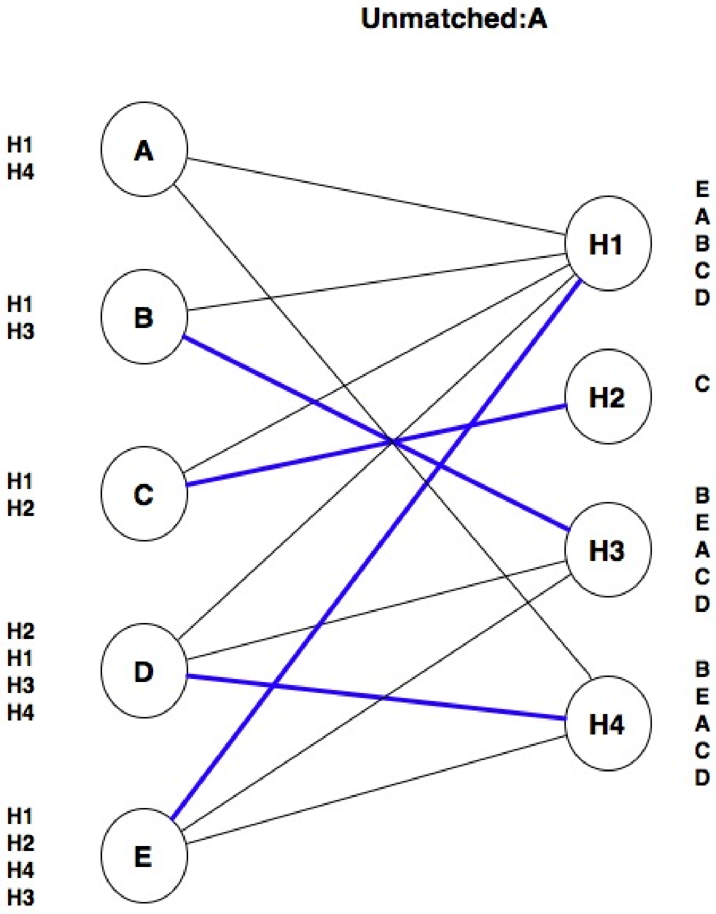
Similarly, C won't get matched with H1 because that hospital ranks A (its current tentative match) higher than C. Instead, we tentatively match C with H2:



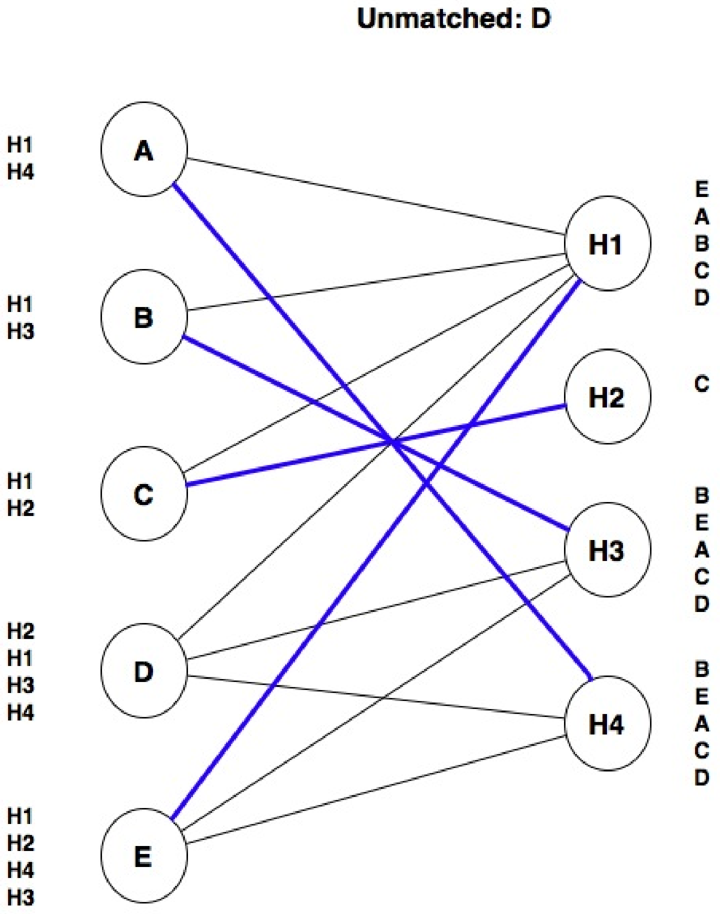
Next, D cannot be matched with H1 (it prefers A), or with H3 (it prefers B), but we can tentatively match D with H4:



Things get interesting when we look at E. E's top choice is H1, and H1 is tentatively matched with A. However, H1 prefers E over A, so we *unmatch* A and tentatively match E with H1:



At this point, we have made one pass through all of the students, but we do not yet have a stable match: A is unmatched, and one of the hospitals A wants, H4, is matched with a student, D, that is ranked lower than A. So we repeat the process for A: we can't match A with H1, because H1 prefers E, but we can match A with H4, unmatching D in the process:



Finally, we make one last (unsuccessful) attempt to match D.

At this point, we have arrived at a stable match and stop. We could define our steps a bit more formally like this:

1. Look at the first student. We will refer to that student as *S*.
2. Look at the first hospital ranked by student *S*. We will refer to this hospital as *H*. Do the following:
   a. If *H* is not matched to any student, tentatively match *H* with *S*, and go to step 3 (no need to look at other hospitals).
   b. If *H* is matched with another student *S'* and *H* prefers *S* to *S'*, then unmatch *S'*, tentatively match *H* with *S*, and go to step 3.
   c. If *H* is matched with another student *S'* and *H* does not prefer *S* to *S'*, then repeat steps 2(a)-2(c) with the next-ranked hospital as *H* (i.e., look at the next hospital for student *S*).
3. If we have made a pass through all the students without changing any matches, we have arrived at a stable match. Stop the program.
4. Otherwise, we repeat steps 2-3 with the next unmatched student as *S* (if we're looking at the last unmatched student, we return to the first one).

Note that we don't actually have to check whether a match is stable or not. Instead, we repeat a series of steps until we have made a pass through all the students without changing any matches.

We call a concrete set of steps like this an *algorithm*. While we have described it here in English, it could instead be written in a programming language like Python, Java, or C++. As we'll see in the next chapter, programming languages provide syntax to specify such steps, and can indicate whether to repeat some steps or to take some action based on the outcome of some test.

Once we have written these steps in a programming language, a computer will be able to run through those steps automatically; we won't need to manually go step-by-step like we did above. However, as we'll soon see, programming languages are much less expressive than human languages like English. This means we can't describe the steps in a hand-wavy fashion and trust that the computer can figure out the gaps. Instead, we must specify the steps to take precisely and without ambiguity.

This algorithmic approach to solving problems, where we must specify the concrete steps we want to take, is a key aspect of computational thinking. It makes us better problem solvers because it requires us to think hard about the exact steps that will take us from a given input to some desired output.

While we will have to figure out many algorithms on our own, there are also many existing algorithms that we can use and, more generally, various algorithmic approaches that we can try when we tackle a given problem. The algorithm shown above is an example of a *greedy* algorithm, where the general approach is to choose the best tentative match possible for each student. We happen to know this algorithm reaches a stable match because other computer scientists have formally proven that property for this particular algorithm.

In fact, the algorithm we just described is essentially the same algorithm used in the National Resident Matching Program, and it produces very good results: in 2016, fourth year medical students in the US (of which there were more than 18,000) had a match rate of 93.8% and 96.2% of the positions were filled. The NRMP website used to include a fairly detailed description of the algorithm, which has since been replaced with this video.

1.4 Complexity

The greedy algorithm is not the only algorithm for finding a stable match. For example, we could've taken a different approach. For example, we could have considered every possible match between students and hospitals and, for each, checked the match's stability. This seems simpler than the greedy algorithm, but it is also a pretty terrible one: for N students and M hospitals, we would need to look at up to $\frac{N!}{(N-M)!}$ possible matches and, for each of them, we would need to inspect every possible pair of students and hospitals (a total of $N \cdot M$ pairs) to check the stability conditions. This is known as a *brute-force* algorithm, because we take the approach of producing every possible solution and checking whether it meets some conditions (in this case, whether the match is stable).

Programs that implement brute-force algorithms typically take a very long time to run and, in most cases, inefficiently solve a given problem. The greedy algorithm, on the other hand, requires at most $N \cdot M$ operations on the graph (this is much much much less than the $\frac{N! \cdot N \cdot M}{(N-M)!}$ operations required by the brute-force algorithm).

It's not enough to select *an* algorithm, we must know how to compare algorithms and determine whether one is more efficient than another. Later in the book, we will present a more concrete way of thinking about this but, in a nutshell, we will want to be mindful of how the number of operations an algorithm performs grows relative to the size of the input. For example, the brute-force algorithm might actually be fine for small values of N and M, but its running time will grow exponentially with N and M (while the running time of the greedy algorithm won't).

Similarly, we must also know how to recognize when we are confronted by a hard problem:

In fact, there are entire categories of problems that cannot be solved efficiently by a computer. For example, we could tweak our model to allow students to specify partner constraints: some students may be in a relationship so they want to be matched with the same hospital, or with hospitals that are reasonably close to each other. In the following graph, the dotted lines represent relationships between students and hospitals that are close to each other:

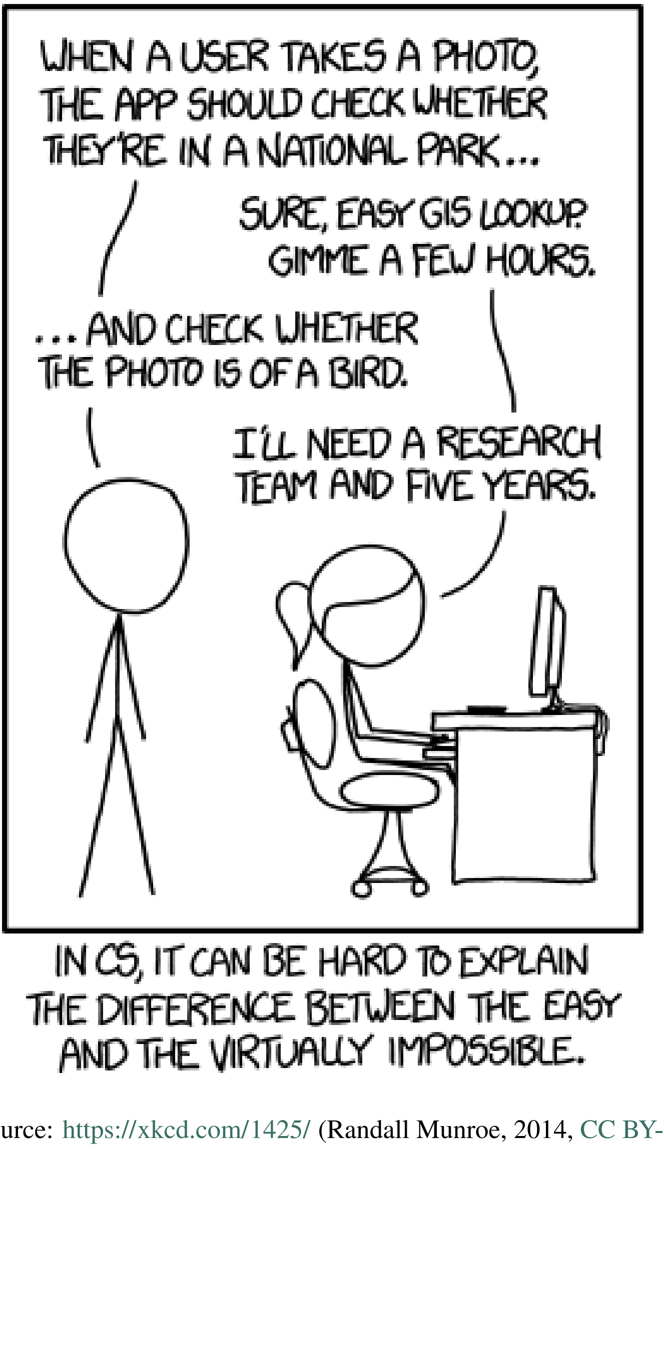
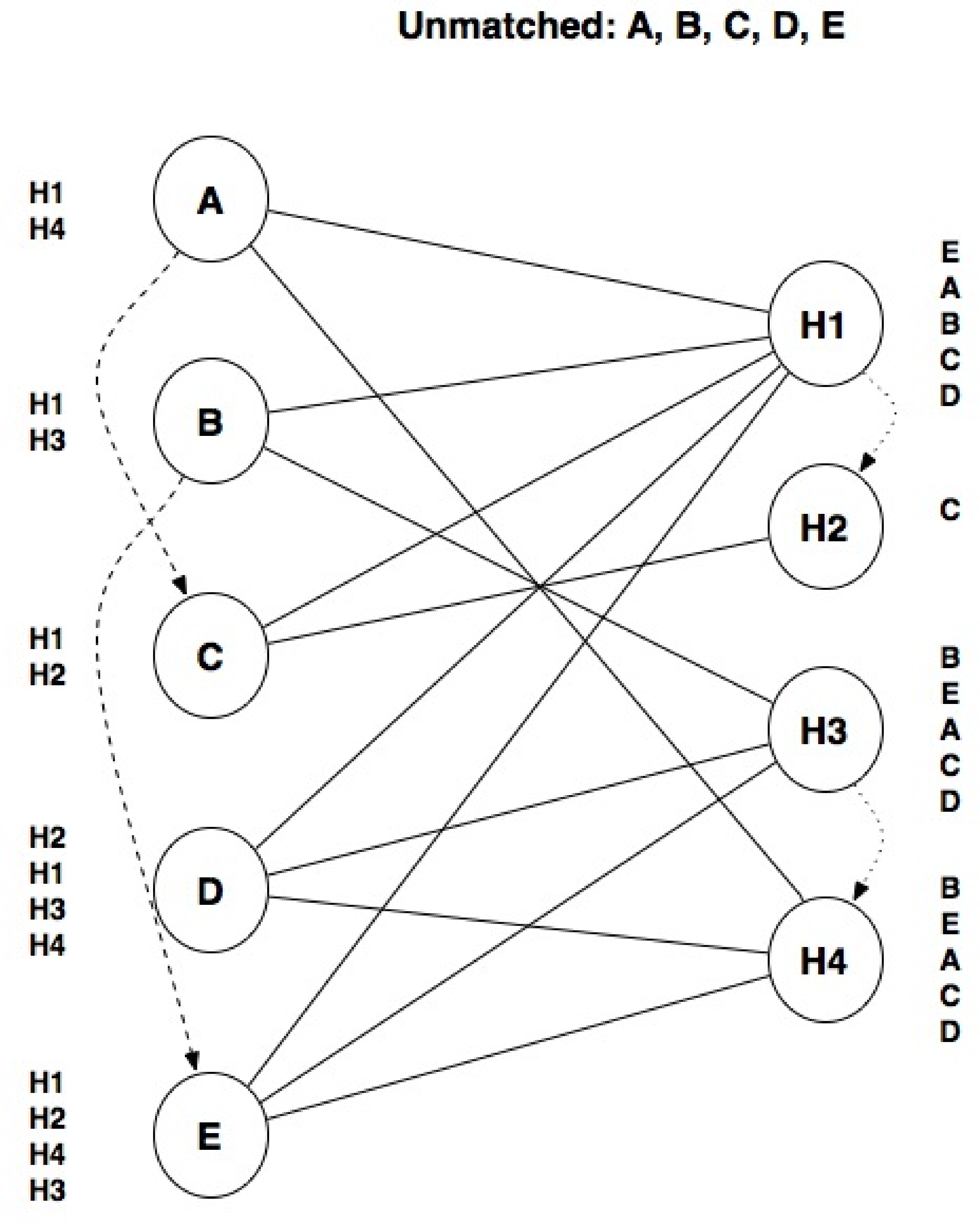

WHEN A USER TAKES A PHOTO, THE APP SHOULD CHECK WHETHER THEY'RE IN A NATIONAL PARK...
SURE, EASY GIS LOOKUP. GIMME A FEW HOURS.
...AND CHECK WHETHER THE PHOTO IS OF A BIRD.
I'LL NEED A RESEARCH TEAM AND FIVE YEARS.
IN CS, IT CAN BE HARD TO EXPLAIN THE DIFFERENCE BETWEEN THE EASY AND THE VIRTUALLY IMPOSSIBLE.

Fig. 1: Source: https://xkcd.com/1425/ (Randall Munroe, 2014, CC BY-NC 2.5)



This seemingly simple tweak to our model actually makes the problem much harder to solve. In fact, not only is the greedy algorithm no longer valid, it turns out that the only guaranteed way to arrive at a stable match is by using the brute-force algorithm *and* there is no guarantee that such a stable match will exist. In a case like this, we would have to rely on other types of algorithms, such as probabilistic algorithms, that are not guaranteed to find the best possible match, but can usually find a near-optimal match in much less time than the brute force algorithm.

The “difficulty” of a problem or of a given algorithm is referred to as its *complexity*. Throughout the book, we will highlight how we must be aware of the complexity of the algorithms we write, as well as of what data structures we choose for a given problem.

CHAPTER

TWO

PROGRAMMING BASICS

A computer program is, at its core, a collection of instructions that the computer must perform. It could be as simple as telling the computer to add two numbers or as complex as performing a complex simulation of drug interactions. In this chapter, you will learn some of the most basic instructions we can include in a program, and you will run your first computer program.

As you read through this chapter, it may seem like you won't be able to do much with these basic instructions. This is normal: the concepts presented in this chapter lay the foundation for understanding more complex concepts introduced in subsequent chapters.

In this book, we will be using the Python programming language and, more specifically, Python 3. Python is a widely-used modern language with a fairly low overhead for getting started. Don't forget, however, that the goal of this book isn't to teach you Python specifically; our goal is to teach you how to think computationally and how to write programs that provide a computational solution to a given problem. Python just happens to be a very convenient language to get started with, but many of the concepts and skills you will learn will carry over to other languages.

2.1 Tools

Before we get started, let's review the tools you'll need to follow along as we introduce concepts. Specifically, you'll need:

1. a shell and a terminal application,
2. a code or text editor, and
3. the Python 3 interpreter.

Both MacOS and Linux come with a terminal application pre-installed. You may need to install one, however if you are running Windows. The program that runs within a terminal window and processes the commands the you type is called a *shell*. There are lots of shells—Bash, ksh, PowerShell, etc. For our purposes, it does not matter which one you use as long as you know how to run basic commands and create directories and move among them.

When you use the shell, you will type commands at the command-line prompt. We will use `$` as the shell command-line prompt in our examples. You may see something different depending on exactly which shell you are using. The exact prompt does not matter, just remember that you do not need to type the `$` when you enter a command.

Programmers often have *very* strong opinions about text editors. One of us is a long-time Emacs fan, the other uses graphical IDEs like Visual Studio Code (VSCode) and PyCharm. We currently recommend VSCode to our students. If you do not already have a favorite text editor, try out a few and pick the one that works best for *you* and then do your best to ignore all the comments coming from your friends and colleagues about *their* favorite editor being the one true editor. Whichever editor you choose, you will need to be able to create new files and edit existing files.

As for Python, depending on how your machine is configured you may already have Python 3 or you may need to install it. You can check which, if any, version of Python you have installed by running the following command at the command-line prompt in a terminal window:

```
$ python --version
Python 3.8.3
```

If the command fails with a `command not found` error, then you will need to install Python 3. If the output shows a version number starting with 2 (e.g., `Python 2.7`), try running `python3` instead of just `python` to make sure you run Python 3. The version numbers displayed may be different on your computer. Please make sure you're running at least Python 3.4.

While we have chosen to recommend using an editor and a separate interpreter, many programmers prefer to use integrated development environments (IDE), such as PyCharm or Juypter notebooks. You might want to try one or more IDEs as well. As long as you have a way to edit code and run it, you will be in good shape,

If you choose to use different tools, please keep in mind that the prompts and the format of output may be slightly different from our examples below.

2.2 Your First Program

Traditionally, the first program you write is a simple program that instructs the computer to print out the following message on the screen:

```
Hello, world!
```

In Python, we can do this task with a single line of code:

```
print("Hello, world!")
```

Don't worry about the exact syntactical details just yet. However, do notice that the above line makes sense intuitively: we're telling the computer to *print* something on the screen: `Hello, world!`

Not just that, that single line is the entirety of the program. In most programming languages, a program is specified in plain text typically stored in one or more files. So, if you create a text file called `hello.py` with that single line of code, that file will contain your first program.

Of course, now we have to *run* that program. In general, all programming languages have some mechanism to take the contents of a file containing a program and run the instructions contained in that file. Because Python is an *interpreted* language, we have second way of running the program: interactively entering Python code into an *interpreter*.

These two ways of running code (storing it in a file or running it interactively in an interpreter) are complementary. We will see how to do both in Python, and then discuss when it makes sense to use one or the other.

2.2.1 Using the Python interpreter

The Python interpreter is a program that allows us to interactively run Python code one line at a time. So, let's start the Python interpreter! From the terminal, run the `python` command. You should see something like the following:

```
Python 3.8.3 (default, Jul  2 2020, 11:26:31)
[Clang 10.0.0 ] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Note

The exact date and version numbers shown when you start the interpreter will likely be different on your computer. As noted earlier, please make sure you're running at least Python 3.4.

The >>> symbol is called the *prompt*. If you write a line of Python code and press the Enter key, the Python interpreter will run that single line of code, print any output resulting from running that code, and will finally return to the prompt so you can write more code. So, try typing in the "Hello, world!" program and then pressing "Enter". The interpreter should look something like this:

```
Python 3.8.3 (default, Jul  2 2020, 11:26:31)
[Clang 10.0.0 ] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello, world!")
Hello, world!
>>>
```

Notice that, after the user pressed Enter, the Python interpreter *printed* `Hello, world!` before returning to the prompt. This is called the *output* of the program.

For the remainder of the book, whenever we want to show code that is intended to be run in the Python interpreter, we will include the >>> prompt in the code examples. However, this does *not* mean you have to type >>> yourself; it is simply intended to distinguish between the code you type into the interpreter, and the expected output of that code. For example:

```
>>> print("Hello, world!")
Hello, world!
```

Before we continue, it is worth noting that Python (and pretty much all programming languages) are very picky when it comes to code syntax (i.e., the required elements and form of a piece of code). For example, code is usually case-sensitive, meaning that typing `Print` instead of `print` will result in an error:

```
>>> Print("Hello, world!")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'Print' is not defined
```

Every bit of syntax, even if it seems redundant, plays a role, so forgetting to include quotation marks will similarly result in an error:

```
>>> print(Hello, world!)
  File "<stdin>", line 1
    print(Hello, world!)
                      ^
SyntaxError: invalid syntax
```

If you type a piece of code into the interpreter and get an error back, especially a `SyntaxError`, double-check the code you typed to make sure you included all of the necessary syntax and did not introduce any typos.

You'll encounter many errors as you learn to write code. In a couple of chapters, we'll explain how to interpret the information presented in error messages in more detail. For now, you can ignore most it; just look at the last line to find out type of error occurred.

2.2.2 Running code from a file

Instead of typing and running a program line by line in the interpreter, we can also store that program in a file, typically named with a `.py` extension, and tell Python to read the file and run the program contained in it. In fact, when we use the term "a Python program" we typically refer to a `.py` file (or a collection of `.py` files; for now we'll work with just one) that contains a sequence of Python instructions.

Let's write our "Hello World!" program using this approach: create a blank text file called `hello.py` and edit it to contain this single line:

```
print("Hello, world!")
```

To run this program, open a terminal and, in the same directory that contains your `hello.py` file, run the following:

```
$ python hello.py
```

This command should produce the following output:

```
Hello, world!
```

And then immediately return to the terminal.

2.2.3 Running code interactively vs. from a file

We have seen two ways of running Python code: by entering the code line by line ourselves into the interpreter, and by saving the code in a text file and telling Python to run the contents of that file.

Entering code into the interpreter line by line is very useful for trying out small pieces of code. For example, let's say you wanted to experiment with the "Hello, world!" code to see what happened if you included different messages:

```
>>> print("Hello, world!")
Hello, world!
>>> print("Hello, reader!")
Hello, reader!
>>> print("Hello, interpreter!")
Hello, interpreter!
```

If we were running code from a file, we would have to open the `hello.py` file, edit it, save it, and re-run `python hello.py`. This process quickly becomes tedious, and using an interactive tool like the interpreter makes it much easier to experiment with small pieces of code.

In fact, this type of tool is common in other interpreted programming languages, such as Ruby, JavaScript, R, and others. It is more formally called a REPL environment: Read-Evaluate-Print-Loop. The tool *reads* the code, *evaluates* it, *prints* any output it produces, and *loops* (i.e., allows you to start the process all over again by prompting you for more code).

By contrast, *compiled* programming languages like C, C++, Java, and C# typically don't offer such an environment. In those languages, you must write code in a file, and then run it from the file (or, more precisely, code is first *compiled* into a binary format that the computer can understand, and then it is actually run).

While the interpreter is a great tool, it is most useful for testing *small* pieces of code. Imagine that you had a complex program with hundreds or even thousands of lines of codes: typing it line by line into the interpreter *every time* you wanted to run it would be cumbersome. Instead, we store the program in a file (or several files) and run it from there.

This reasoning, however, doesn't mean that all small programs are run in the interpreter and all large programs are run from files. Instead, the approaches are complementary. When writing a program, a common workflow is to start writing the program in a file and to use the interpreter to help you figure out specific pieces of code. In other words, you may use the interpreter to work through tricky bits of code, adding them to the text file only when they are correct. Later in the book, we will see specific examples of this workflow.

Interpreters are also helpful for gaining familiarity with a new *software library*. For example, Python itself includes a vast library of code to handle common tasks (such as performing common math operations and generating random numbers) and, while this code is very well-documented, it usually pays off to familiarize yourself with it in the interpreter before using it in programs. Later in the book we will see examples of how you can experiment with Python's standard library in the interpreter, as well as other *third-party libraries* (such as Pandas for data manipulation and Matplotlib for visualizing data).

2.3 The Basics

So far, the only "instruction" we've seen is `print`, which allows us to print some text on the screen (as we'll see later on, `print` is actually something called a "function" and `print("Hello, world!")` is a "call" to that function). Of course, there is a lot more you can do in Python. We'll see that there are instructions for doing many things:

- Simple arithmetic
- Performing one action or another based on the result of a previous one
- Repeating an action multiple times
- And so on

For the remainder of this chapter we are going to focus on three fundamental concepts found in nearly every programming language:

- Variables
- Types
- Expressions

As we said at the start of this chapter, there is little we'll be able to do with these constructs alone, so don't worry if they seem a bit abstract at first. In no time, you will find yourself using variables, types, and expressions in all of your programs.

2.4 Variables

A fundamental part of writing a computer program is keeping track of certain information throughout the lifetime of the program (i.e., while the program is running). For example, if you were writing a simple program to compute the average of a series of measurements, you would need to keep track of the running total of those measurements. This kind of information is stored in your computer's memory while a program is running.

However, you will rarely (if ever) have to interact with your computer's memory directly. Instead, most programming languages provide a convenient abstraction for storing information: *variables*. A variable is a *symbolic name* representing a location in the computer's memory. You can store a specific *value*, such as a number or piece of text, in a variable and retrieve that value later on.

In Python, you can *assign* a value to a variable like this:

```
variable = value
```

The equals sign is called the *assignment operator*. It tells Python to take the value on the right-hand side and assign it to the variable on the left-hand side. The whole code fragment is called an *assignment statement*.

For example:

```
message = "Hello, world!"
```

In Python, it doesn't matter whether the `message` variable already existed or not. Whenever you perform an assignment on a previously-unseen variable, Python will choose a location in memory to store whatever value is assigned to that variable (in this case, the text `"Hello, world!"`). You don't have to worry about the low-level details, as Python handles them under the hood.

Go ahead and try running the above assignment in the interpreter. You should see something like this:

```
>>> message = "Hello, world!"
```

After you press Enter, the interpreter will return straight to the >>> prompt. Unlike the `print` function, an assignment does not produce any output. It simply alters the state of your computer. More specifically, this example stored the value `"Hello, world!"` in a location in the computer's memory identified by the name `message`.

To print the value of a variable, we can use the `print` function:

```
>>> print(message)
Hello, world!
```

In fact, you can also just write the name of a variable in the interpreter, and the interpreter will *evaluate* (or look up the value of) the variable and print out its value:

```
>>> message
'Hello, world!'
```

You can ignore the quotation marks around `Hello, world!` in the above output; we will revisit this detail later in this chapter.

Over the lifetime of a program, we can assign new values to variables using the assignment operator. For example, notice that we assign a sequence of new values to the `message` variable:

```
>>> print(message)
Hello, world!
>>> message = "Hello, reader!"
>>> print(message)
Hello, reader!
>>> message = "Hello, interpreter!"
>>> print(message)
Hello, interpreter!
```

Assigning a new value to an existing variable is often referred to as *updating* the variable.

2.5 Types

In the above example, the `message` variable contained a piece of text (`"Hello, world!"`). However, variables can also contain other *types* of data. Most programming languages (including Python) support at least three basic types:

- **Numbers**: Numbers usually encompass both integer numbers and real numbers.
- **Strings**: Strings are how we refer to "text" in most programming languages (in the sense that text is a "string" of characters). We've actually already seen an example of a string: `Hello, world!` (the character `H` followed by the character `e` followed by the character `l`, etc.).
- **Booleans**: Booleans represent truth values (*True* or *False*).

Additionally, Python also supports a special “None” type, to indicate the *absence* of a value. In this section, we will describe the above three types, as well as the special “None” type, in more detail.

In most programming languages, each variable in a program has a specific type associated with it. For example, `message` has a string type: it stores a string of characters. Notice, however, that we didn’t need to tell Python “`message` is a variable of type string”. This is because Python is a *dynamically-typed* language: it can figure out the type of a variable on-the-fly. In this case, the fact that we assigned a string value (`"Hello, world!"`) to `message` was enough for Python to determine that `message` was a string variable. In addition, as we will see later in this chapter, the type of a variable can change dynamically during the lifetime of a program.

Other languages, like Java and C/C++, are *statically-typed* and require the programmer to specify the type of *every* variable. For example, in Java we would need to declare the variable like this:

```
String message = "Hello, world!";
```

In a statically-typed language, once the type of a variable is set, it remains the same for the remainder of the program’s lifetime.

2.5.1 Integers

An integer is a number without a fractional component. We can use the assignment operator to assign an integer value to a variable:

```
>>> a = 5
>>> b = -16
```

In the above code, `5` and `-16` are what we call *literal values*, in the sense that `5` is *literally* the integer 5, while `a` is the *symbolic* name of a variable. Note how we can also specify *negative* integers.

Right now, there is not much we can do with integers, other than print them:

```
>>> print(a)
5
>>> print(b)
-16
```

As we will see soon, we will also be able to perform common arithmetic operations with integers.

2.5.2 Real numbers

Similarly, we can also assign real numbers to variables:

```
>>> x = 14.3
```

Computers, however, can only represent real numbers up to a finite precision. In other words, while there are infinitely many real numbers between 0.0 and 1.0, computers can only represent a finite subset of those numbers. Similarly, π has infinitely many decimal places (3.14159265359. . .) but a computer can only store a finite number of them.

Computers store real numbers using an internal representation called *floating point*. In fact, these numbers are commonly referred to as *floats* in programming languages. The floating point representation *approximates* the value of the real number and may not always store its exact value. For example, the number `2.0` could, in some cases, be represented internally as `1.99999999999999`.

Integers, on the other hand, use an *exact* internal representation. The *range* of integers that can be represented is still finite, but the internal representation of an integer is always an exact value. For example, the integer `2`, if stored in a variable, will always be exactly `2` (instead of an approximation like `1.99999999999999`).

The difference between an integer literal and a floating point literal is the presence of a decimal point. If a decimal point is present in the number, it will be represented internally as a floating point value, even if the fractional part is zero. Otherwise, it will be represented as an integer.

For example:

```
>>> x = 15.0
>>> b = 15
>>> print(x)
15.0
>>> print(b)
15
```

Conceptually, `x` and `b` both store the name number (fifteen), but they have different types: `x` is a float and `b` is an integer. Python actually has a built-in `type` function that will tell us the exact type of a variable:

```
>>> print(x)
15.0
>>> type(x)
<class 'float'>
>>> print(b)
15
>>> type(b)
<class 'int'>
```

Technical Details

Types are represented using classes in Python, which is why the output of the `type` function says `<class 'float'>` rather than `<type 'float'>` or simply `'float'`. We will introduce classes later in the book.

This function also allows us to see how Python is able to recognize the type of a variable has changed dynamically based on the value it stores:

```
>>> c = 10
>>> type(c)
<class 'int'>
>>> c = 10.5
>>> type(c)
<class 'float'>
```

Notice how variable `c` first stored an integer, but then switched to being a `float` variable once we assigned a float to it.

2.5.3 Strings

We have already seen one way to assign a string value to a variable:

```
>>> message = "Hello, world!"
```

One thing to note, though, is that the value that is associated with the variable does not include the quotation marks. The quotation marks are simply used to delimit the start and end of the string literal. This is why the quotation marks are not included when we print a string variable:

```
>>> print(message)
Hello, world!
```

But are included when we supply the name of the variable to the interpreter:

```
>>> message
'Hello, world!'
```

The result of evaluating the name `message` includes the quotation marks to indicate that the value is a string.

You can also use single quotes to delimit the start and end of a string literal:

```
>>> message2 = 'Hello, universe!'
```

When using single or double quotes, the string cannot span multiple lines. Instead, you can use triple-quotes (either three single quotes or three double quotes) to specify strings that span multiple lines:

```
>>> message3 = """This message
... spans multiple
... lines"""
>>> print(message3)
This message
spans multiple
lines
>>> message4 = '''And
... this
... one
... does
... too!'''
>>> print(message4)
And
this
one
does
too!
```

When a user is writing a piece of code that spans multiple lines, the interpreter will use three periods `...` to indicate that it is expecting more code before it can run anything.

You might reasonably wonder why there are so many different ways to delimit a string. One answer is that having different methods makes it easier to include quotation marks in your strings. For example, I might want to have a string to represent the sentence: `He said, "Hello world!"`. I can represent this value in Python using single quotes `'He said, "Hello world"'`. Because we are using single quotes to delimit the string, the two occurrences of `"` inside the string are treated as normal characters that have no special meaning. In other languages, we would need to use a special pair of characters `\"`, known as an escape sequence, to indicate that the inner quotes are part of the value.

Finally, note that you must always use some type of quotes to delimit the start and end of a string. If you don't, Python cannot distinguish between a string literal and a variable name. For example, try this:

```
>>> message5 = Hello
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'Hello' is not defined
```

When Python interprets the above code, it will assume that `Hello` is a the name of a variable, not a string. And since we haven't defined a `Hello` variable, we will get a `NameError` telling you as much.

2.5.4 Booleans

Variables can also contain a *boolean* value. This value can be either True or False:

```
>>> a = True
>>> print(a)
True
>>> b = False
>>> print(b)
False
```

As we noted earlier, Python is case sensitive, which means that capital letters in True and False are required. Typing true into the Python interpreter, yields an error

```
>>> true
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'true' is not defined
```

because Python interprets true as the name of a non-existent variable, rather than as a boolean value.

Right now, there's not much we can do with boolean variables or values, but we'll see soon that they'll come in handy.

2.5.5 The value None

Sometimes, we may want to define a variable but not assign any value to it just yet. In some cases, we can simply assign a reasonable default value to a variable. For example, if we're writing a program to process sales, we may need to apply a discount in certain cases (e.g., a client who is part of a loyalty program). This program could include a variable to store the total discount, and we could simply initialize it to zero:

```
>>> discount = 0.0
```

If it turns out no discounts are applicable, then the default value of zero works well.

In some cases, however, we need to distinguish *absent* values explicitly. For example, suppose a survey includes an optional question where a customer can specify the number of children in their household (which could be used for demographic classification purposes). A default value of zero wouldn't work in this case, because we would need to distinguish between "the customer didn't provide a value" and "the customer did provide a value, and it is zero". We would need some way of indicating that a variable simply has no value. In Python, we can use the special value None:

```
>>> num_children = None
```

Besides using None directly in our programs, we will also see that there are a number of Python operations that will use None to indicate that the operation did not produce any value at all.

Not all programming languages have this kind of special value. In languages without the special value None, a variable would be assigned an *impossible* value instead. For example, we could assign a value of -1 to num_children because it is impossible for someone to have "negative one" children and thus that value would actually mean that "num_children has no value". You may encounter this convention now and then but, in Python, you should remember to use None to indicate the absence of a value.

Technical Details

Since we mentioned that every value has a type, you might be wondering about the type of None. It has the type NoneType, which has exactly one value: None.

2.5.6 Scalar types

Atomic values, those of type `int`, `float`, `bool`, and `NoneType` plus values from a couple of types–`complex` and `bytes` –that we will not discuss, are often referred to as *scalars*. If you have experience with linear algebra, you might recognize this term as referring to a value that has a magnitude but no direction (as opposed to a vector which has both a magnitude and a direction). If you don't have experience with linear algebra, just remember that the term scalar is used to mean a single value–like `5`, `5.0`, or `True` –that cannot be broken into smaller components.

2.6 Expressions

Now that we've seen variables, some basic types, and their corresponding literals, we can combine them together into *expressions*. An expression is a piece of Python code that gets *evaluated* to produce a new value. For example, we can combine integer literals using simple arithmetic operators to produce new integer values. For example:

```
>>> 2 + 2
4
```

Note that whenever you enter an expression in the Python interpreter, the interpreter will evaluate the expression and then print out the resulting value.

We can also assign the result of evaluating an expression to a variable:

```
>>> a = 2 + 2
>>> print(a)
4
```

And we can use variables in expressions themselves. For example, we can add an integer literal and an integer variable:

```
>>> a = 10
>>> a + 5
15
```

Or we can add two integer variables:

```
>>> a = 10
>>> b = 20
>>> a + b
30
```

We will focus on only two types of expressions for now: arithmetic expressions, which produce integer or float values, and boolean expressions, which produce a boolean value.

For arithmetic expressions, addition, subtraction, multiplication (`*`) and division (`/`) work pretty much like you would expect them to:

```
>>> 2 + 2
4
>>> 10 - 3
7
>>> 3 * 3
9
>>> 10 / 3
3.3333333333333335
```

Notice, however, that division will always produce a floating point number even when its operands are integers, *even* if the divisor evenly divides the dividend:

```
>>> 9 / 3
3.0
```

We can verify this claim using the `type` function that we introduced earlier as a way to determine the type of a variable. When this function is used with an expression, Python will first evaluate the expression and *then* determine the type of the result.

```
>>> type(9 / 3)
<class 'float'>
```

When an integer result is desirable, we can use *integer division*:

```
>>> 10 // 3
3
>>> type( 10 // 3)
<class 'int'>
```

While the previous example suggests that this operator, known as *floor division*, just throws away the fractional part of the result, it actually rounds down towards negative infinity:

```
>>> - 10 // 3
-4
```

There is also a *modulus* operator that will produce the remainder of dividing two integers:

```
>>> 10 % 3
1
```

And an exponentiation operator that will raise a value to a power:

```
>>> 2 ** 3
8
```

When an expression contains multiple operators, Python evaluates the operations in a specific order based on their relative *precedence*. Most notably, multiplication and division are always performed before addition and subtraction, so the expression `10 - 2 * 3` is equivalent to $10 - (2 \cdot 3)$:

```
>>> 10 - 2 * 3
4
>>> 10 - (2 * 3)
4
```

In technical terms, we say that multiplication and division have *higher* precedence than addition and subtraction or alternatively, we could say that addition and subtraction have *lower* precedence than multiplication and division. We will describe Python's precedence rules in more detail below after we have introduced a few more operators.

If we want to force a different order of evaluation, we can use parentheses:

```
>>> (10 - 2) * 3
24
>>> 10 - (2 * 3)
4
```

All of the above operators are *binary* operators, meaning that they operate on two operands (one on the left and one on the right). Python also has *unary* operators that operate on a single operand. For example, unary negation will take an expression that evaluates to a number, and will produce its negative value:

```
>>> - (3 - 5)
2
>>> - (10 / 3)
-3.3333333333333335
>>> - (10 / -3)
3.3333333333333335
```

When an arithmetic expression involves both integers and floats, the entire expression will yield a float, *even* if the float's fractional part is zero:

```
>>> 1 + 3.2
4.2
>>> 2 * 3.0
6.0
```

The expressions we have seen that operate on numbers all produce a numeric value, but we can also use *relational operators* on numbers. These include operators such as "greater than" (>), "greater than or equals" (>=), "less than" (<), "less than or equals" (<=), "equals" (==), and "not equals" (`!=`) to compare two values. The result of the comparison will be a boolean value: `True` or `False`. For example:

```
>>> 10 > 5
True
>>> 100 < 2
False
>>> 7 >= 7
True
>>> 42 <= 37
False
>>> 5 == 5
True
>>> 5 != 5
False
>>> 10 == 6
False
>>> 10 != 6
True
```

Either side of the relational operator can be a literal value, a variable, or *any* expression that produces a number. For example:

```
>>> a = 5
>>> 5 + 5 < a * 3
True
```

In the above expression, the left side of the < evaluates to 10, and the right side evaluates to 15, meaning that the comparison becomes `10 < 15` (which evaluates to `True`). We do not need to enclose the expressions `5 + 5` and `a * 3` in parenthesis because relational operators have lower precedence than arithmetic operators. Whether or not to include them for clarity is largely a matter of personal preference.

The equality and inequality operators can also be used with the value `None`:

```
>>> num_children = None
>>> tax_rate = 15.0
>>> num_children == None
True
>>> tax_rate == None
False
```

Python also includes two operators, `is` and `is not`, that are similar, but not identical, to == and `!=`:

```
>>> a is 5
True
>>> a is not 10
True
>>> num_children is None
True
>>> tax_rate is None
False
>>> tax_rate is not None
True
```

The differences between == and `is` are very subtle and we will not concern ourselves with them here. However, by convention, == and `!=` are used to compare integers, floats, strings, and booleans, while `is` and `is not` are used to check whether a value is `None` or not. Later in the book, we will see that there are differences between these operators that become important when the operands have more complex Python data types, such as lists.

On top of all this, we can combine boolean expressions using *logical* operators, where each side of the operator must evaluate to a boolean value. The `and` operator evaluates to `True` if both sides of the operator evaluate to `True` and evaluates to `False` otherwise:

```
>>> a = 10
>>> b = -5
>>> a > 0 and b > 0
False
```

The above expression checks whether both `a` and `b` are positive non-zero numbers. Since `b` is not, the whole expression evaluates to `False`.

The `or` operator evaluates to `True` if either or both sides of the operator evaluate to `True`, and evaluates to `False` only if both sides of the operator are `False`. For example:

```
>>> a = 10
>>> b = -5
>>> a > 0 or b > 0
True
```

The above expression evaluates to `True` if `a` is a positive non-zero number, if `b` is a positive non-zero number, or if both `a` and `b` are positive non-zero numbers. Since `a` is positive, the expression evaluates to `True`. This operation is known as *inclusive or*, because it "includes" as `True` the case where both operands are true.

Finally the `not` operator takes only a single operand on its right side and negates a boolean value. For example:

```
>>> a = 10
>>> b = -5
>>> not (a > 0 and b > 0)
True
```

The above expression yields `True` when either `a` or `b` are negative or zero, but `False` if they are both positive and non-zero. In other words, it yields the opposite of the expression that we saw earlier.

At this point, while you can probably relate to the need to compute the value of an arithmetic expression, you may be wondering about the purpose of boolean expressions. We will see in the next chapter that boolean expression will be used to determine whether an action has to be performed or not, or for how many times an action should be repeated. For example, if you are writing a stock-trading application, you might need a way to express that a given stock should be sold *if* its price (stored in a variable called `price`) reaches or exceeds a certain target price (stored in a variable called `target_price`). The boolean expression that controls this action could look something like this:

```
price >= target_price
```

We can combine different logical operations to describe complex rules. For example, in the United States Senate, a bill can be brought to the floor for debate in a few ways:

- all of the senators present in the chamber agree on a *motion to proceed*, also known as unanimous consent, or
- at least 60 senators vote in favor of a motion to proceed, or
- a quorum of at least 51 senators is present, a majority of the senators present vote in favor of a *motion to proceed*, and the bill either is not or cannot be filibustered (defined roughly as "talked to death"), or
- a quorum is present, half the senators present vote in favor of a *motion to proceed*, the bill either is not or cannot be filibustered, and the Vice President is present and votes in favor of the motion to proceed.

Given variables for:

- the number of votes in favor of the motion to proceed (`num_yea`),
- the number of votes against proceeding (`num_nay`),
- a boolean that indicates whether the bill is being filibustered (`is_filibuster`), and
- a boolean that indicates whether the Vice President in present and votes "yea" (`is_vp_yea`).

and a couple of constants:

```
FILIBUSTER_LIMIT = 60
QUORUM = 51
```

we can translate these rules into a boolean expression:

```
(num_yea >= FILIBUSTER_LIMIT) or \
((num_yea + num_nay > QUORUM) and \
 (not is_filibuster) and \
 ((num_yea > num_nay) or ((num_yea == num_nay) and is_vp_yea)))
```

This expression is long. To prevent the line of code from getting too long to read easily, we split it across multiple lines using backward slash (\) to indicate that the expression continues on the next line. Alternatively, we could have wrapped the whole expression in parentheses:

```
((num_yea >= FILIBUSTER_LIMIT) or
 ((num_yea + num_nay > QUORUM) and
  (not is_filibuster) and
  ((num_yea > num_nay) or ((num_yea == num_nay) and is_vp_yea))))
```

We could have chosen to use the numbers 60 and 51 in the expression directly, but it is better to give these types of values names rather than have them appear as *magic numbers* in an expression. It is traditional to name *constants*, that is, variables whose values are fixed and will not change during the lifetime of a program, using capital letters.

You might notice that we did not include a special case for unanimous consent. We handle this case by setting num_yea equal to the filibuster limit and num_nay to zero. Alternatively, we could introduce a new boolean variable, unanimous_consent for tracking this situation and add a new clause to the expression:

```
(unanimous_consent or
 (num_yea >= FILIBUSTER_LIMIT) or
 ((num_yea + num_nay > QUORUM) and
  (not is_filibuster) and
  ((num_yea > num_nay) or ((num_yea == num_nay) and is_vp_yea))))
```

Notice that we wrote unanimous_consent rather than unanimous_consent == True for the new clause that we added to the expression. The latter form frequently appeals to new programmers, but adding == True is redudant and should be avoided. We'll come back to the appropriate values for num_yea, num_nay, etc in the case that the senators approve the motion to proceed by unanimous consent shortly.

Operator precedence

You might be asking yourself whether all the parentheses in the expression above necessary? To answer that question we need to understand the relative precedence of the different operations used in the expression. Here is a subset of Python's precedence rules taken from Operator Precedence section of the Python Language Reference.

Operator	Description
`**`	exponentiation
`-x`, `+x`	unary negation, unary plus
`*`, `/`, `//`, `%`	multiplication, division, floor division, remainder
`+`, `-`	addition, subtraction
`<`, `<=`, `>`, `>=`, `!=`, `==`, `is`, `is not`	relational, comparison, and identity operations
`not x`	logical negation
`and`	logical and
`or`	logical (inclusive) or

Operators higher in the table have higher precedence than operators lower in the table. For example, exponentiation has higher precedence than unary negation. Operators in the same row have the same precedence and are evaluated left to right if they occur together. For example, the expression `2 / 3 * 4` is equivalent to `(2 / 3) * 4`. Similarly, with the exception of exponentiation, multiple instances of the same operator are evaluated left to right, so the expression `2 / 3 / 4` is equivalent to `(2 / 3) / 4`. Exponentiation, in constrast, is evaluated right to left, so the expression `2 ** 3 ** 4` is equivalent to `2 ** (3 ** 4)`.

In addition to saying that one operator has higher precedence than another, programmers also use the phrase *binds more tightly* to mean that one operator has higher precedence than another. For example, multiplication binds more tightly than addition.

Since arithmetic operations bind more tightly than relational operations, which in turn, bind more tightly than logical operations, we can write the expression for describing when a bill can be brought to the floor of the United States Senate for debate with many fewer parenthesis:

```
num_yea >= FILIBUSTER_LIMIT or \
num_yea + num_nay > QUORUM and \
not is_filibuster and \
(num_yea > num_nay or num_yea == num_nay and is_vp_yea)
```

Only one set–those around the expression `num_yea > num_nay or num_yea == num_nay and is_vp_yea`–is required to express the rules of the senate. As noted earlier, whether to include the rest of the parentheses or not is largely a matter personal preference.

As noted, the operands for nearly all binary operators evaluated left to right. Logical operators exploit this evaluation order to provide a very useful feature: they *short circuit*, that is, if the value of the left operand determines the result of the operation (`True` for `or`, `False` for `and`), then the right operand is not evaluated. For example, given the expression:

```
(y != 0) and (x % y == 0)
```

the subexpression `(x % y == 0)` will not be evaluated if `y` has the value zero. Why? Because if `y` is zero, then the result of the `and` is guaranteed to be `False`. Conveniently, short circuiting allows us to avoid dividing by zero in this case.

This property of `and` and `or` makes the order of the operands important. This expression:

```
(x % y == 0) and (y != 0)
```

will fail with a ZeroDivisionError error when y is zero, because the left operand is evaluated first.

Now that we have discussed short circuiting, let's return to the filibuster example:

```
(unanimous_consent or
 (num_yea >= FILIBUSTER_LIMIT) or
 ((num_yea + num_nay > QUORUM) and
  (not is_filibuster) and
  ((num_yea > num_nay) or ((num_yea == num_nay) and is_vp_yea))))
```

Python will stop evaluating this expression as soon as one of the three or clauses evaluates to True. As a result, the values of num_yea, num_nay, etc can be set to zero or to None or not at all, for that matter, if unanimous_consent is true.

2.6.1 String expressions

Strings also support many of the operations we just described. Addition, for example, works with strings and results in the concatenation, or joining, of those strings:

```
>>> "abc" + "def"
'abcdef'
>>> name = "Alice"
>>> "Hello, " + name
'Hello, Alice'
```

One thing to note: while you can mix integers and floats when using the addition operator, mixing integers and strings generates a type error:

```
>>> tax = 15.0
>>> "Your tax rate is " + tax
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "float") to str
```

We will describe a solution to this problem in the next section.

The equality and identity operations can also be used on strings:

```
>>> name = "Alice"
>>> name == "Bob"
False
>>> name != "Bob"
True
>>> name == "Alice"
True
>>> name is "Alice"
True
>>> name is not "Bob"
True
```

Mixing types with the equality operator

While, as we noted, you cannot mix strings and numbers when using arithmetic operations, such as addition, you *can* mix types when using the == and != operators, but be careful: they check whether the two values have the same type. For example:

```
>>> 5 == "Hello"
False
```

Naturally, the result is `False`: the number 5 is not the same as the string `"Hello"`. However, evaluating this expression is also yields `False`:

```
>>> 10 == "10"
False
```

When comparing two values of a different type, Python won't make any attempt to convert one to the other's type before making the comparison. Instead, the above expression returns `False` because an integer is not the same thing as a string, even if semantically they refer to the same thing.

The relational operations can also be used on strings:

```
>>> "Alice" < "Bob"
True
>>> "Alex" > "Alice"
False
```

The relational operators use lexicographic ordering when used to compare strings: the result of the operation is determined by comparing the first two characters that differ. In the first example, the strings differ in the first character and "A" comes before "B" and so, the result of evaluating `"Alice" < "Bob"` is `True`. In the second example, the strings differ first at the third character and since "e" does not come after "i", the result of evaluating the expression is `False`.

Character encoding and relational operators

What does it mean for one character to “come before” another? Strings in Python are represented using a *character encoding* which associates each individual character (like `A` and `B`) with an integer value. You can actually see that integer value by using the built-in `ord` function:

```
>>> ord("A")
65
>>> ord("B")
66
>>> ord("ñ")
241
>>> ord("🤔")
129300
```

Notice how `A` has a lower numerical value than `B`, so it is considered to come before `B`. Also, notice how letters from other alphabets, such as `ñ` from the Spanish alphabet, and emojis are valid characters in Python. This is because the default character encoding in Python is Unicode, which supports a wide array of characters, including practically all non-English characters.

If you are working with English language text (or, more specifically, with the 26-letter Roman alphabet), then in practice, you do not need to be concerned with this technical detail. When it comes to the relational operators and strings, you can assume they support the standard English dictionary ordering on strings.

If you are working with text that includes characters outside the 26-letter Roman alphabet, the ordering created by the Unicode encoding may not always produce the expected result. For example, in Spanish, the letter `ñ` comes after `n` and before `o`, which means the word `original` must come after `ñoño`. However:

```
>>> "original" > "ñoño"
False
```

Ordering string can thus get a bit complicated in these cases, and often requires the use of external libraries, like PyICU.

We have only described a small subset of the operations supported by strings here. We discuss lots more later in *Lists, Tuples, and Strings*.

2.7 Casting

There are times when we need to convert a value from one type to another. For example, we might get a real number represented as a string from a user interface and want to compute with that value as a number. To do so, we first need to convert it from a string to a floating point value. We convert or *cast* the string into a floating point value using the `float` function:

```
>>> approx_pi = "3.1415"
>>> x = float(approx_pi)
>>> x * 2
6.283
```

Or we might want to cast a float into a string (using `str`) and then combine it with another string:

```
>>> approx_pi = 3.1415
>>> s = str(approx_pi) + " is an approximation to Pi"
>>> print(s)
3.1415 is an approximation to Pi
```

When we cast a float to an integer, we not only change the type, we also change the value by throwing away the fractional part:

```
>>> approx_pi = 3.1415
>>> int(approx_pi)
3
```

Note that throwing away the fractional part is equivalent to rounding towards zero. If the value is positive, as in the above example, `int` will round it down towards zero. In contrast, if the value is negative, `int` will round up towards zero:

```
>>> neg_pi = -3.1415
>>> int(neg_pi)
-3
```

2.8 Dynamic Typing Revisited

Now that we've seen some basic types as well as expressions, we can see some of the implications of dynamic typing in a bit more detail. Like we said earlier, Python is dynamically-typed, which means it infers the type of a variable when we assign a value to it. As we saw earlier, we can see the exact type of a variable by writing `type(v)` (where `v` is a variable).

Notice that Python correctly infers that `a` should be an integer (or `int`), that `x` should be a float, and that `s` should be a string (or `str`) based on the values supplied on the right-hand side of the assignment statements:

```
>>> a = 5
>>> type(a)
<class 'int'>
>>> x = 3.1415
>>> type(x)
<class 'float'>
>>> s = "foobar"
>>> type(s)
<class 'str'>
```

Not just that, we can assign a value of one type to a variable and, later on, assign a value of a different type, and Python will *dynamically* change the type of the variable. In contrast, a statically-typed language would likely generate an error pointing out that you're trying to assign a value of an incompatible type.

```
>>> b = 5
>>> type(b)
<class 'int'>
>>> b = "Hello"
>>> type(b)
<class 'str'>
```

One consequence of this property is that an operation that succeeds in one line, may not succeed a few lines later. For example,

```
>>> b = 5
>>> c = b + 7
>>> print(c)
12
>>> b = "hello"
>>> c = b + 7
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

The first time we evaluate the expression `b + 7`, the evaluation succeeds and yields the value `12`. The second time, however, the evaluation fails, because `b` is now a string and we cannot add a string and an integer.

When working with a dynamically-typed language like Python, we must be careful to use types consistently. Just because a variable can change its type throughout the lifetime of a program doesn't mean it should. As you take your first steps with Python, you should try to be disciplined about choosing a type for a variable and then sticking with it throughout the program.

2.9 Code Comments

Before we move on to the next chapter, there's one final element of Python syntax we need to discuss: code comments. When writing code in a Python file, you can include notes for the reader, known as *comments*, in your code using the hash character, and Python will ignore everything that appears after the hash character:

```
# This is a comment
a = 5

# This is a comment that
# spans multiple
# lines
b = 5

c = a + b  # Comments can also appear at the end of lines
```

You will see comments in many of the examples we will provide, and it is good practice to use comments to document your code, especially when your code is not self-explanatory.

CHAPTER

THREE

CONTROL FLOW STATEMENTS

In the previous chapter, we saw that a program is essentially a collection of instructions to a computer. We saw two specific instructions: assignment statements, which assign a value to a variable, and the `print` function, which prints a value on the screen. For example:

```
n = 7
print("n is", n)
n = 10
print("n is now", n)
```

```
n is 7
n is now 10
```

The "instructions" in a program are called *statements*. The above program has four statements: two assignment statements and two function call statements (remember that `print` is something called a "function"; we will explain this construct in more detail in the next chapter).

You might notice that these calls to `print` look different than the calls you have seen thus far. Here, we pass two comma-separated values of different types to the function. In general, you can pass multiple values with different types to `print`, and the function will print each of them separated by spaces.

Note

Notice that we used a different format for showing code above. Instead of showing the execution of the program line by line in the interpreter, we showed all of the code followed by all of its output. In fact, if you saved the above code in a file named `statements.py` and ran `python statements.py` from the terminal, you would see the output shown above.

Of course, you can still run the above code in the interpreter. We are showing you the code and the output in a different way because, as we move on to more complex pieces of code, this format will be more readable than the interpreter format (which can get a bit cluttered with the >>> prompt for every statement). However, we will still be using the interpreter format for small pieces of code.

These statements are run in sequence (i.e., one after the other). Sometimes, though, we want the program to run a statement based on the outcome of a previous statement, or we may want to repeat a block of statements multiple times. To support this, most programming languages include *control flow statements*. In this chapter, we will focus on three types of control flow statements:

- **Conditional statements**, also known as "if statements" or "if-then-else statements", allow us to run a piece of code only if a given condition (expressed as a boolean expression) is true, and optionally allow us to run an alternate piece of code if the condition is false. For example, given a variable `x` containing an integer, we may want to perform a different action when `x` is positive than when `x` is zero or negative (we could express this condition using the boolean expression `x > 0`).

- **Sequence-based loops**, also known as “for loops”, allow us to repeat a piece of code based on a sequence of values. For example, given a list of numbers, we may want to determine, for each number, whether the number is prime. So, we want to “repeat” the code for testing primality for each number in the list.
- **Condition-based loops**, also known as “while loops”, allow us to repeat a piece of code based on whether a boolean expression is true. For example, when performing a simulation, we may want to continue running the simulation until some stopping condition is met.

In the next chapter, we will see additional statements that relate to *functions* and which also allow us to alter the control flow of our program.

3.1 Conditional Statements

In the previous chapter, we learned that a boolean expression is an expression that evaluates to either `True` or `False`:

```
>>> n = 5
>>> n > 0
True
```

A conditional statement allows the program to perform different actions based on the value of a boolean expression. For example:

```
if n > 0:
    print(n, "is a positive number")
```

In the above code, the `print` statement will be run *only* if `n` is greater than zero.

The conditional statement starts with a special keyword `if`, which is part of the syntax of the Python language. The keyword `if` is followed by a boolean expression and a colon. The next line contains the first line of the the code to run *if* the boolean expression evaluates to `True`. The spacing before `print` is important, and we’ll return to this detail soon.

Let’s try out this code in the interpreter. Make sure you type four spaces before the call to `print`:

```
>>> n = 5
>>> if n > 0:
...     print(n, "is a positive number")
...
5 is a positive number
```

The `if` statement spans multiple lines and the interpreter won’t run the statement until you’ve finished writing it. After you’ve written the `if` line, the interpreter will switch the prompt to `...` to indicate that it is still expecting input as part of a multi-line statement. After you write the `print` line, the interpreter will still show the `...` prompt because it isn’t sure whether you’re going to provide more lines in the `if` statement. Pressing Enter on a blank line tells the interpreter that the code is ready to be run.

Note

Remember that the `...` prompt is an artifact of the Python interpreter. If you were to write this code in a file, you would not include the `...` characters. In fact, try saving this code in an `if.py` file:

```
n = 5
if n > 0:
    print(n, "is a positive number")
```

And running `python if.py` from the terminal. The output should be:

```
5 is a positive number
```

Now, try it with a negative value of `n`:

```
>>> n = -5
>>> if n > 0:
...     print(n, "is a positive number")
...
```

Notice that, after running the `if` statement, the interpreter didn't produce any output. The value of `n` is not greater than zero and so, the `print` statement was not executed.

In the above example, the `if` statement contained a single statement to execute conditionally. However, conditional statements can contain multiple statements to execute as well. For example:

```
n = 5
if n > 0:
    print(n, "is a positive number")
    n = -n
    print("And now the number is negative:", n)
```

```
5 is a positive number
And now the number is negative: -5
```

In this case, if the condition is true, then Python will run all of the statements under the `if`. More specifically, the *block* of statements under the `if`, that is, all the statements at the same level of indentation (i.e., with the same number of spaces before them), will be run. For example, consider this other version of the code:

```
n = 5
if n > 0:
    print(n, "is a positive number")
    n = -n
print("And now the number is:", n)
```

```
5 is a positive number
And now the number is: -5
```

And again with a negative number:

```
n = -5
if n > 0:
    print(n, "is a positive number")
```

(continues on next page)

(continued from previous page)

```
    n = -n
print("And now the number is:", n)
```

```
And now the number is: -5
```

Notice that the first `print` and the `n = -n` statement are run only if the condition is true, but the second `print` is *always* run. The second `print` is not part of the `if` statement but occurs *after* the `if` statement.

Python's syntax uses indentation to delimit blocks of code, typically with four spaces corresponding to one level of indentation. As with other syntactical elements, Python is very picky about this detail and incorrect indentation will produce errors. For example, we'll see an error if we do not indent the code in an `if statement`:

```
if (n % 2) == 1:
print(n, "is odd")
```

```
  File "<stdin>", line 2
    print(n, "is odd")
    ^
IndentationError: expected an indented block
```

Or if we use inconsistent indentation inside the same block of code:

```
if (n % 2) == 1:
    print(n, "is odd")
     print("This concludes the odd branch.")
```

```
  File "<stdin>", line 3
    print("This concludes the odd branch.")
    ^
IndentationError: unexpected indent
```

For this first `if` example, we walked you through all the syntactical elements of the statement but, as you get more comfortable with Python, it is easier to learn about new elements of the language by getting a more formal description of their syntax. In the case of the `if` statement, a formal description would look like this:

```
if <boolean expression>:
    <statements>
```

We will use this convention often when describing new statements and other aspects of Python. Words in **bold** represent keywords that are part of the language itself, whereas anything delimited with < and > means "substitute this for ..." (i.e., you do not write the < and > characters themselves). Additionally, remember that indentation matters, so the statement or statements that constitute `<statements>` all have the same level of indentation.

So, now we can easily introduce a variant of the `if` statement in which we specify an alternate block of statements to run if the condition is *false*:

```
if <boolean expression>:
    <statements>
else:
    <statements>
```

For example:

```
n = 6
if (n % 2) == 1:
```

(continues on next page)

(continued from previous page)

```
    print(n, "is odd")
else:
    print(n, "is even")
```

```
6 is even
```

When written in this format, the block under the `if` is usually called the *if branch*, and the block under the `else` is called the *else branch*.

In this example, each block has a single statement but, as we saw earlier, each block can have multiple statements:

```
n = 5
if (n % 2) == 1:
    print(n, "is odd")
    print("This concludes the odd branch.")
else:
    print(n, "is even")
    print("This concludes the even branch.")
```

```
5 is odd
This concludes the odd branch.
```

An `if` statement can also have multiple if branches (but only one `else`, which is run only if none of the conditions are met). After the first `if` branch, subsequent branches start with the keyword `elif` (which stands for "else if"). For example:

```
if <boolean expression>:
    <statements>
elif <boolean expression>:
    <statements>

...

else:
    <statements>
```

For example:

```
n = 17
if n < 0:
    print(n, "is negative")
elif n > 0:
    print(n, "is positive")
else:
    print(n, "is zero")
```

```
17 is positive
```

When we have multiple branches, as soon as Python finds a true boolean condition, it will evaluate that branch and ignore all others. Python first evaluates the `if` branch's condition, and moves to subsequent `elif` branches only if no prior condition is true. If no branch is true, Python will run the code in the `else` branch.

An important note: for any `if` statement, *at most one branch will be run*, even if the conditions in multiple branches are true. For example:

```
n = 17
if n < 0:
    print(n, "is negative")
elif n > 0:
    print(n, "is positive")
elif n % 2 == 1:
    print(n, "is odd")
elif n % 2 == 0:
    print(n, "is even")
```

```
17 is positive
```

In the above example, there are two conditions that are true when `n` is 17: `n > 0` and `n % 2 == 1`. Python, however, will only run the code for the *first* branch with a true condition (in this case, `n > 0`). In general, it is good practice to make the conditions in an `if` statement mutually exclusive (i.e., at most one of them can be true).

You should also take care to distinguish between an `if` statement with multiple branches and multiple `if` statements. For example, the code below is *not* equivalent to the code above:

```
n = 17
if n < 0:
    print(n, "is negative")
if n > 0:
    print(n, "is positive")
if n % 2 == 1:
    print(n, "is odd")
if n % 2 == 0:
    print(n, "is even")
```

```
17 is positive
17 is odd
```

The code above has four *separate* `if` statements (often referred to as *parallel* if statements), while the previous example had a *single* `if` statement with four branches. Since we have four separate statements, each conditional statement is evaluated and Python will run the block of code for any true conditional (in this case, the second and third `if` statements).

A Common Pitfall

Using parallel `if` statements when you need a single `if` statement can lead to bugs that can be very hard to find. For example,

```
n = 7
if n > 0:
   print(n, "is positive")
   n = -n
elif n < 0:
   print(n, "is negative")
```

```
7 is positive
```

will yield a very different result than:

```
n = 7
if n > 0:
   print(n, "is positive")
   n = -n

if n < 0:
   print(n, "is negative")
```

```
7 is positive
-7 is negative
```

Why? In the first example, the first branch of the conditional is true and so the second branch is never tested. In the second example, the second conditional is a *separate* statement. Its test *will* be evaluated and since, the value of n changes during the execution of the first conditional the result of that test will be True and its statement will be executed.

Handling multiple conditions

If you have a bug in code that has a conditional statement with multiple conditions, a good first step is to verify that the conditions (i.e. the boolean expressions) are mutually exclusive. If not, ask whether your application requires overlapping conditions. If not, rewrite the conditions to make them mutually exclusive. If so, check that the order you have chosen for the conditions makes sense for the application. Also, make sure that you test each of the possible conditions. Finally, verify that you have not used parallel if statements when a single if statement with multiple conditions is required.

In general, it is a good practice to try out your code with values that will trigger the different conditions in a conditional statement.

3.2 for Loops

Loops provide a mechanism for repeating work in a program. Loops are often used when we need to process a set of values and perform the same action for each. For example, given a list of prices, we may want to compute the tax on each price as well as keep a running total of all the taxes paid. To do this task, we will use a for loop (in the next section, we will see a different type of loop, the while loop, that operates differently).

for loops have the following syntax:

```
for <variable> in <sequence>:
    <statements>
```

For example:

```
for p in [10, 25, 5, 70, 10]:
    print("The price is", p)
```

```
The price is 10
The price is 25
The price is 5
The price is 70
The price is 10
```

Notice that the loop repeated the print statement five times, once for each value in a sequence of values. In each repetition, or *iteration*, of the loop, the variable p is set to the value of a specific element in the provided sequence of values, starting with the first and advancing through the provided sequence. Once all of the elements have been processed, the loop ends.

The sequence in the above `for` loop (`[10, 25, 5, 70, 10]`) is called a *list*. We will take a deeper look at lists and other types of sequences in Chapter *Lists, Tuples, and Strings*, but for now, will only use them in `for` loops. Lists are values and can be assigned to variables so instead of including the list of values directly in the `for` loop, we could first store it in a variable:

```
prices = [10, 25, 5, 70, 10]

for p in prices:
    print("The price is", p)
```

```
The price is 10
The price is 25
The price is 5
The price is 70
The price is 10
```

While in this case, we are specifying the value of `prices` directly in the program, you could imagine these prices being read from a file, or being scanned at the checkout counter of a store.

Like `if` statements, we can include multiple statements in the loop:

```
prices = [10, 25, 5, 70, 10]

for p in prices:
    tax = 0.10 * p
    total = p + tax
    print("The price (with tax) is", total)
```

```
The price (with tax) is 11.0
The price (with tax) is 27.5
The price (with tax) is 5.5
The price (with tax) is 77.0
The price (with tax) is 11.0
```

The block of statements contained in a loop is usually called the *body* of the loop.

Our loop example uses a list with five values as the sequence and the loop iterate five times and prints one line of output for each value. If we used a list with ten values for the sequence, the loop would iterate ten times, once per value. If the list has zero values, which known as the empty list and is written as `[]`, then the loop will iterate zero times.

```
prices = []

print("Before the loop")
for p in prices:
    tax = 0.10 * p
    total = p + tax
    print("The price (with tax) is", total)
print("After the loop")
```

```
Before the loop
After the loop
```

Notice that the calls to `print` before and after the loop, which are not controlled by the loop, are executed and generate output. The call to `print` in the body of the loop, in contrast, is never executed and thus generates no output, because the loop iterates zero times.

Testing loops

A good rule of thumb for testing loops: try the loop with a list that has zero values, a list that has exactly one value, and a list that has many values.

3.2.1 Nested statements

Suppose we wanted to compute the total tax on all the prices. We could accomplish this task with the following loop:

```
prices = [10, 25, 5, 70, 10]

total_tax = 0

for p in prices:
    tax = 0.10 * p
    total_tax = total_tax + tax

print("The total tax is", total_tax)
```

```
The total tax is 12.0
```

Notice that we used an additional variable, `total_tax` to add up the values of the tax. This kind of variable is usually referred to as an *accumulator* variable because it is used to add up (or accumulate) a set of values.

Now, suppose that prices with a value less than 15 are not taxed. This means that, when computing the total tax, we should only add up the taxes of the prices that meet the following condition: `p >= 15`.

So far, the body of an `if` statement or a `for` loop has been a sequence of assignments or `print` statements (the only two other statements we know). It is certainly possible, however, for the body of a `for` loop to include `if` statements or even other `for` loops.

For example:

```
prices = [10, 25, 5, 70, 10]

total_tax = 0

for p in prices:
    if p >= 15:
        tax = 0.10 * p
        total_tax = total_tax + tax
    else:
        print("Skipping price", p)

print("The total tax is", total_tax)
```

```
Skipping price 10
Skipping price 5
Skipping price 10
The total tax is 9.5
```

Notice the two levels of indentation: one level for the body of the loop, and another for the if and else branches. To be clear, we could have other statements at the same level of indentation as the if statement:

```
prices = [10, 25, 5, 70, 10]

total_tax = 0

for p in prices:
    print("Processing price", p)
    if p >= 15:
        print("This price is taxable.")
        tax = 0.10 * p
        total_tax = total_tax + tax
    else:
        print("This price is not taxable.")
    print("---")

print("The total tax is", total_tax)
```

```
Processing price 10
This price is not taxable.
---
Processing price 25
This price is taxable.
---
Processing price 5
This price is not taxable.
---
Processing price 70
This price is taxable.
---
Processing price 10
This price is not taxable.
---
The total tax is 9.5
```

Suppose there were three tax rates (5%, 10%, and 15%) and we wanted to find the total value of the tax under each tax rate for each of the prices. If we only had one price to worry about, we could use this single loop to compute the different taxes:

```
p = 10
tax_rates = [0.05, 0.10, 0.15]

print("Price", p)
for tr in tax_rates:
    tax = tr * p
    print("Taxed at", tr, "=", tax)
```

```
Price 10
Taxed at 0.05 = 0.5
Taxed at 0.1 = 1.0
Taxed at 0.15 = 1.5
```

If we had multiple prices to worry about, however, we would use a *nested* for loop. We can combine both of the single loop examples by *nesting* the loop over the prices inside of the loop over tax rates to find the total tax for different tax rates.

```
prices = [10, 25, 5, 70, 10]
tax_rates = [0.05, 0.10, 0.15]

for tr in tax_rates:
    total_tax = 0
    for p in prices:
        if p >= 15:
            tax = tr * p
            total_tax = total_tax + tax
    print("Taxed at", tr, "the total tax is", total_tax)
```

```
Taxed at 0.05 the total tax is 4.75
Taxed at 0.1 the total tax is 9.5
Taxed at 0.15 the total tax is 14.25
```

Notice that we replaced the fixed tax rate (`0.10`) from the original loop over the prices with the outer loop variable (`tr`). Let's walk through this computation. For each iteration of the outer loop:

- the variable `total_tax` is set to zero, then
- the inner loop iterates through all the prices, checks whether each price (`p`) is taxable, and if so, updates the total tax using the current value of `tr`, and finally
- the print statement outputs the current tax rate (that is, the value of `tr`) and the computed total tax.

When you use nested loops, it is very important to think carefully about where to initialize variables. As a thought experiment, consider what would happen if we initialized `total_tax` to zero *before* (rather than in) the body of the outer loop.

```
prices = [10, 25, 5, 70, 10]
tax_rates = [0.05, 0.10, 0.15]
total_tax = 0

for tr in tax_rates:
    for p in prices:
        if p  >= 15:
            tax = tr * p
            total_tax = total_tax + tax
    print("Taxed at", tr, "the total tax is", total_tax)
```

```
Taxed at 0.05 the total tax is 4.75
Taxed at 0.1 the total tax is 14.25
Taxed at 0.15 the total tax is 28.5
```

Notice that rather than restarting at zero for each new tax rate, the total tax from the previous iteration of the outer loop carries over. As a result, the answers are wrong.

Do keep in mind that the right answer is context dependent. If instead of representing different possible tax rates for a given taxing entity (such as, a city), the rates instead represented the tax rates for different tax domains, say the city, county, and state. If your goal was to compute the total tax paid for all the prices across all the taxing domains, then it would make sense to initialize `total` before the loop nest. In this use case, you would likely move the `print` statement out of the loops altogether as well.

Nested loops and initialization

If you have a bug in code that uses nested loops, verify that you are initializing *all* the variables used in the loop in the right place for your specific application.

3.2.2 Iterating over ranges of integer

A common way to use `for` loops is to do something with all the integers in a given *range*. Let's say we wanted to print out which numbers between 1 and 10 are odd and even. We could do it with this simple loop, which also features a nested `if` statement:

```
nums = [1,2,3,4,5,6,7,8,9,10]
for n in nums:
    if (n % 2) == 1:
        print(n, "is odd")
    else:
        print(n, "is even")
```

```
1 is odd
2 is even
3 is odd
4 is even
5 is odd
6 is even
7 is odd
8 is even
9 is odd
10 is even
```

What if we wanted to do this computation with the numbers between 1 and 100? While we could manually write out all those numbers, there is a better way of doing this task: we can use Python's built-in `range` function:

```
for n in range(1, 11):
    if (n % 2) == 1:
        print(n, "is odd")
    else:
        print(n, "is even")
```

```
1 is odd
2 is even
3 is odd
4 is even
5 is odd
6 is even
7 is odd
8 is even
9 is odd
10 is even
```

We specify two numbers for `range`: a lower bound and an upper bound. The lower bound is inclusive, but the upper bound is not; i.e., `range` will allow you to iterate through every integer starting at the lower bound and up to but not including the upper bound. To change this loop to do our computation on the numbers between 1 and 100, we would simply change the upper bound in the call to `range` from 11 to 101 (*i.e.*, `range(1, 101)`).

A Common Pitfall

You should think very carefully about the appropriate bounds when using `range`. In particular, make sure you don't make an "off-by-one" error, where you set a bound to be too low or too high by one. This type of mistake is very common and can be difficult to track down.

3.2.3 Example: Primality testing

Let's move on to a slightly more involved example: testing whether a number is prime. While there are very elaborate methods for testing primality, a naive way of testing whether a number n is prime is to divide that number by every integer between 2 and $n - 1$. If any such integer evenly divides n, then n is *not* prime.

Assume that we have a variable `n` containing the number we are testing. We could use a `for` loop that uses `range` to iterate over every integer between 2 and $n - 1$. Inside the loop, we would check whether each of integer evenly divides `n` using the modulus operator (if the remainder of dividing `n` by another integer is zero, then that number can't be prime).

Let's take a first stab at this problem by setting up the loop, but not solving the problem entirely:

```
n = 81
print("Testing number", n)
for i in range(2, n):
    if n % i == 0:
        print(i, "divides", n)
print("Testing done")
```

```
Testing number 81
3 divides 81
9 divides 81
27 divides 81
Testing done
```

This code simply prints out the divisors of `n`. Now let's try it with a prime number:

```
n = 83
print("Testing number", n)
for i in range(2, n):
    if n % i == 0:
        print(i, "divides", n)
print("Testing done")
```

```
Testing number 83
Testing done
```

Since 83 is a prime number and doesn't have any divisors (other than 1 and itself), the program doesn't print any divisors. So, it looks like our code is on the right track, even if it doesn't give us a clean "yes" or "no" answer to whether `n` is prime or not.

Solving a simple version of a complex problem is common practice in programming. Avoid the temptation to solve an entire problem at once, instead try solving simpler versions of the problem first to make sure you are on the right track. For example, the code above reassures us that we're checking divisibility correctly.

So, how do we modify this code to check whether a number is prime? Well, if n has even a single divisor, it will not be prime. So, all we need to do is keep track of whether we have encountered a divisor or not. While we could do this with an integer (by keeping track of how many divisors we have encountered, and then checking whether the number is zero or not), we can get away with using only a boolean variable, since all we care about is whether we have encountered a divisor or not. We initialize this variable to False, since we haven't encountered any divisors yet at the start of the program.

```
encountered_divisor = False
n = 83
for i in range(2, n):
    if n % i == 0:
        encountered_divisor = True

if encountered_divisor:
    print(n, "is NOT prime")
else:
    print(n, "is prime")
```

```
83 is prime
```

Notice that the value of encountered_divisor will not change once it is set to True. What would happen if we added an else clause that set the value of encounter_divisor to False when n is not evenly divisible by i? We'd have a bug: our code would declare all numbers to be prime.

Finally, as an additional example of nesting one loop inside another, suppose we want to print whether each integer in a given range is prime or not. We could simply nest the primality testing code we wrote above inside another for loop that uses range to iterate over a sequence of integers, each of which we will test for primality:

```
for n in range(2,32):
    encountered_divisor = False
    for i in range(2, n):
        if n % i == 0:
            encountered_divisor = True

    if encountered_divisor:
        print(n, "is NOT prime")
    else:
        print(n, "is prime")
```

```
2 is prime
3 is prime
4 is NOT prime
5 is prime
6 is NOT prime
7 is prime
8 is NOT prime
9 is NOT prime
10 is NOT prime
11 is prime
12 is NOT prime
13 is prime
14 is NOT prime
15 is NOT prime
16 is NOT prime
17 is prime
18 is NOT prime
```

(continues on next page)

(continued from previous page)

```
19 is prime
20 is NOT prime
21 is NOT prime
22 is NOT prime
23 is prime
24 is NOT prime
25 is NOT prime
26 is NOT prime
27 is NOT prime
28 is NOT prime
29 is prime
30 is NOT prime
31 is prime
```

3.2.4 The `break` statement

The primality testing algorithm we just saw is a correct algorithm: for all values of `n` (where $n \geqslant 1$), the algorithm will correctly determine whether `n` is prime or not. However, it is nonetheless a *terrible* algorithm.

Why? Suppose you wanted to test whether 2^{61} is prime. It is clearly not prime (it is divisible by 2), but the algorithm we presented would nonetheless waste a lot of time iterating over all numbers from 2 to 2^{61}, even though we would know `n` isn't prime the moment we divided it by 2. To make the cost more concrete, if we can perform 15,000 iterations of the loop in 1 microsecond, then determining whether 2^{61} is prime using the code above would take nearly five years.

We need to tweak our algorithm so that we can "bail out" of the `for` loop as soon as we've found our answer. We can use a `break` statement, which immediately exits the loop, to do this:

```
encountered_divisor = False
n = 83
for i in range(2, n):
    if n % i == 0:
        encountered_divisor = True
        break

if encountered_divisor:
    print(n, "is NOT prime")
else:
    print(n, "is prime")
```

```
83 is prime
```

Notice how all we had to do was add a single `break` statement in the case when we find a divisor. Making this simple change reduces the time to determine that 2^{61} is not prime to under a microsecond.

Even so, this algorithm is *still* not particularly great. For example, $2^{61} - 1$ happens to be prime and, if we used the above algorithm to test this number, we would still end up performing $2^{61} - 1$ divisions (about two quintillions, or $2 \cdot 10^{18}$).

So, adding the `break` statement only allowed us to optimize certain cases (which might be fine if we expect to run our algorithm only with small integers).

Don't forget that, as we discussed in Chapter *Computational Thinking*, computational thinking includes thinking about the *complexity* of computational solutions. This small example highlights that you shouldn't settle for an algorithm simply because it produces the correct answer. Instead, you should also ask yourself whether the algorithm is sufficiently efficient for your purposes.

Sometimes, this process is as simple as asking yourself how your algorithm will perform in a couple representative cases. For example, the performance of this algorithm with a composite (not prime) number is probably fine, but the performance with very large prime numbers will be very bad. Other times, more elaborate analysis is required. We will be revisiting the notion of complexity several more times throughout this book.

Before we move on, let's change our nested loop example to use `break` and see what happens:

```
for n in range(2,32):
    encountered_divisor = False
    for i in range(2, n):
        if n % i == 0:
            encountered_divisor = True
            print("Breaking:", i, "evenly divides", n)
            break

    if encountered_divisor:
        print(n, "is NOT prime")
    else:
        print(n, "is prime")
```

```
2 is prime
3 is prime
Breaking: 2 evenly divides 4
4 is NOT prime
5 is prime
Breaking: 2 evenly divides 6
6 is NOT prime
7 is prime
Breaking: 2 evenly divides 8
8 is NOT prime
Breaking: 3 evenly divides 9
9 is NOT prime
Breaking: 2 evenly divides 10
10 is NOT prime
11 is prime
Breaking: 2 evenly divides 12
12 is NOT prime
13 is prime
Breaking: 2 evenly divides 14
14 is NOT prime
Breaking: 3 evenly divides 15
15 is NOT prime
Breaking: 2 evenly divides 16
16 is NOT prime
17 is prime
Breaking: 2 evenly divides 18
18 is NOT prime
19 is prime
Breaking: 2 evenly divides 20
20 is NOT prime
Breaking: 3 evenly divides 21
```

(continues on next page)

(continued from previous page)

```
21 is NOT prime
Breaking: 2 evenly divides 22
22 is NOT prime
23 is prime
Breaking: 2 evenly divides 24
24 is NOT prime
Breaking: 5 evenly divides 25
25 is NOT prime
Breaking: 2 evenly divides 26
26 is NOT prime
Breaking: 3 evenly divides 27
27 is NOT prime
Breaking: 2 evenly divides 28
28 is NOT prime
29 is prime
Breaking: 2 evenly divides 30
30 is NOT prime
31 is prime
```

Upon encountering a `break`, Python exits the innermost enclosing loop. We added a `print` statement to highlight this behavior. As you can see from the output, upon finding a value of `i` that evenly divides the current value of `n`, Python prints a message with the values of `i` and `n` and moves on to the conditional that follows the inner loop.

Keep in mind that `break` statements are always used in a loop guarded by a conditional statement. A loop with an unguarded `break` statement will execute at most once. This loop, for example,

```python
prices = [10, 25, 5, 70, 10]

for p in prices"
    print("The value of p is:", p)
    break
```

```
  File "<stdin>", line 3
    for p in prices"
                   ^
SyntaxError: EOL while scanning string literal
```

executes outputs exactly one line. If list of prices happens to be empty, then the loop would not generate any output, which explains why we wrote that a loop with an unguarded break would execute "at most" once.

3.2.5 `continue` statement

Python will immediately exit a loop when it encounters a `break` statement. Sometimes, you just want to stop the current iteration and move on to the next one. To illustrate this concept, we will rewrite an earlier example using `continue`:

```python
prices = [10, 25, 5, 70, 10]

total_tax = 0

for p in prices:
    if p < 15:
        print("Skipping price", p)
```

(continues on next page)

(continued from previous page)

```
        continue
    tax = 0.10 * p
    total_tax = total_tax + tax

print("The total tax is", total_tax)
```

```
Skipping price 10
Skipping price 5
Skipping price 10
The total tax is 9.5
```

In this version, we flipped the test to be `p < 15` (rather than `p >= 15`), added a use of `continue` after the call to `print` in the conditional, and reduced the indentation for the rest of the loop body.

While `continue` should be used sparingly as it can lead to confusing code, it can help improve readability when you have a complex deeply-nested piece of code that needs to be executed in some situations and not others. Writing this code:

```
for <variable> in <sequence>:
    if not <simple test>:
        <complex, deeply nested code>
```

this way:

```
for <variable> in <sequence>:
    if <simple test>:
        continue
    <complex, deeply nested code>
```

may yield code that is easier to understand. And so, while our example above is correct, we would not normally choose to write code that simple using `continue`.

As with `break`, it does not make sense to use `continue` without a corresponding conditional to guard its execution as the code following an unguarded `continue` would never be executed.

3.3 `while` loops

`while` loops are a more general type of loop that, instead of repeating an action *for* each element in a sequence, will repeat an action *while* a condition is true. The condition is expressed using a boolean expression, which allows us to write much more complex loops than `for` loops.

The syntax for `while` loops is the following:

```
while <boolean expression>:
    <statements>
```

Python starts executing a `while` loop by evaluating the boolean expression and, if it is true, running the block of statements (otherwise, the `while` loop is skipped entirely). After the statements are run, the boolean expression is evaluated *again*, and if it remains true, the statements are run again. This process is repeated until the boolean expression becomes false.

For example, this `while` loop will add up all of the integers between 1 and `N`:

```
N = 10
i = 1
sum = 0

while i <= N:
    sum = sum + i
    i = i + 1

print(sum)
```

```
55
```

We use a `sum` variable to store the sum of all the integers, and initialize it to 0. Then, we use a variable `i` to represent the integer we are adding to the sum. We initialize it to one and, in each iteration of the loop, we add `i` to the sum, and increment `i` by one. As long as `i` is less than `N`, the loop will keep adding `i` to `sum`.

Unlike a `for` loop, a `while` loop does not require a sequence of values. Instead, we *must* keep track of and update the integer we add in each iteration of the loop (stored in variable `i`).

This difference is often the source of a common programming error: forgetting to include code to increment `i` (that is, the statement: `i = i + 1`). Without this statement, the loop becomes an *infinite loop*: the``while`` condition (`i <= N`) will never become false because the value of `i` will always be one (and thus will never become greater than `N`). When this happens, your program (or the Python interpreter) will appear stuck. In most computer environments, you can force your program to exit by pressing the "Control" key and the "C" key (known as "Control-C") at the same time. You may have to type "Control-C" a few times to get Python's attention.

In fact, precisely because of this issue, the above piece of code should be implemented using a `for` loop, which is less error prone:

```
N = 10
sum = 0

for i in range(N+1):
    sum = sum + i

print(sum)
```

```
55
```

Notice that we cannot possibly fall into an infinite loop with this implementation, because the `for` loop will *only* iterate over the specified range of values.

So when should we use a `for` loop instead of a `while` loop? As a rule of thumb, any time you need to iterate over a sequence of values, using a `for` loop is typically the best option. Although a `while` loop can get the job done, it can be more error-prone.

There are, however, certain algorithms where the loop cannot naturally be stated as iterating over a sequence of values, so we need the more general mechanism provided by a boolean expression. We will see one such example in the next section.

3.4 Putting it all together

In this section we are going to work through a slightly longer example that combines conditionals and loops and involves non-trivial logic.

A few years ago Gil Kalai posed a question on his blog Combinatorics and more:

> You throw a dice until you get 6. What is the expected number of throws (including the throw giving 6) conditioned on the event that all throws gave even numbers.

The answer, somewhat paradoxically, is 1.5. As noted by Dan Jacob Wallace, the question can be answered using some probability theory and pre-calculus, but it can also be answered by simulating a simple game many times and averaging the number of throws per game.

In the game, we will roll one fair six-sided die over and over again and count the number of rolls. Since the problem is conditioned on all the throws being even numbers, we will stop a given game and declare it to be invalid as soon as we roll an odd number. For valid games, we will keep rolling the die until we hit a six.

Before we can translate this informal description into code, we need to find a way to simulate rolling a die. We can use a function, `randint` from Python's built-in `random` library for this purpose. Given a lower bound and an upper bound, the function chooses a value uniformly at random between the lower bound and upper bound inclusive (that is, including both end points). Since we are working with a six-sided die, we will call `random.randint(1, 6)` repeatedly to get a randomly-chosen stream of values that range between `1` and `6` inclusive.

Note: `random.randint` is a *function*, which we will be discussing in more detail in the next chapter. For now, all you need to so is that, like the `print` function, this function will perform a task for us and, in this case, it will be choosing a number randomly between `1` and `6`. In chapter *Basics of Code Organization* we will also discuss Python's built-in libraries, like `random` in more detail.

To simulate the game, we will need to roll the die over and over again until one of two conditions is met: either (1) a six is rolled, which ends the (valid) game or (2) an odd number is rolled, which signals that the game is invalid and should end. `while` loops are designed for exactly this type of computation: we want to execute a piece of code repeatedly while some conditions holds. Here's one–quite direct–way to translate this description into code:

```python
# Simulate a new game
is_valid_game = True
# first roll
roll = random.randint(1, 6)
game_num_rolls = 1
while roll != 6:
    if roll % 2 == 1:
        # stop: hit an odd number making the game invalid
        is_valid_game = False
        break
    # subsequent rolls
    roll = random.randint(1, 6)
    game_num_rolls = game_num_rolls + 1
```

At the end of the game, the variable `game_num_rolls` will hold the number of times the die was thrown in the game and the variable `is_valid_game` will tell us whether the game was valid.

Here is another less direct, but easier-to-read version:

```python
# first roll
roll = random.randint(1, 6)
```

(continues on next page)

(continued from previous page)

```
game_num_rolls = 1
while roll == 2 or roll == 4:
    # subsequent rolls
    roll = random.randint(1, 6)
    game_num_rolls = game_num_rolls + 1
is_valid_game = (roll == 6)
```

In this version, we keep rolling the die until we hit something other than a 2 or a 4 (i.e., a 1, 3, 5, or 6). We mark the game as valid only if the last roll, the one that caused the game to stop, was a 6.

These two versions illustrate a common phenomenon: there is often more than one way to implement a task and thinking about it in a slightly different way may yield simpler code.

Let's move on to simulating our game many times and computing the answer. We can run multiple games or trials by wrapping a for-loop around the code for simulating a game. Our goal is to compute the average number of rolls per valid game. To do so, we will track the number of valid games using the variable `num_valid_games` and the total number of rolls made in *valid* games using the variable `total_rolls_valid_games`.

Here's code that for the full task:

```
num_trials = 1000
num_valid_games = 0
total_rolls_valid_games = 0

for _ in range(num_trials):
    # run a new game
    # first roll
    roll = random.randint(1, 6)
    game_num_rolls = 1
    while roll == 2 or roll == 4:
        # subsequent rolls
        roll = random.randint(1, 6)
        num_rolls = num_rolls + 1
    if roll == 6:
        # the game was valid
        num_valid_games = num_valid_games + 1
        total_rolls_valid_games = total_rolls_valid_games + game_num_rolls

if num_valid_games > 0:
    ans = total_rolls_valid_games/num_valid_games
    print("Average number of rolls in valid games:", ans)
else:
    print("No valid games.  Try again.")
```

The value of the loop variable for the outer `for` loop is not needed in the computation and so, we followed Python convention and named it _ (underscore).

Note that we dropped the name `is_valid_game` and simply used the expression `roll == 6` as the test for a conditional that ensures that `num_valid_games` and `total_rolls_valid_games` are updated only for valid games.

Finally, let's review our decisions about where to initialize the variables used in the computation. The variables `num_valid_games` and `total_rolls_valid_games` are initialized before the outer loop because they are used to accumulate values across the whole computation. In contrast, `roll` and `game_num_rolls` are re-initialized for every iteration of the outer loop, because they need to be reset to their initial values for each new game.

CHAPTER

FOUR

INTRODUCTION TO FUNCTIONS

Up to this point, we have seen enough programming concepts to express non-trivial algorithms, such as determining whether a number is prime or not:

```
encountered_divisor = False
n = 83
for i in range(2, n):
    if n % i == 0:
        encountered_divisor = True
        break

if encountered_divisor:
    print(n, "is NOT prime")
else:
    print(n, "is prime")
```

```
83 is prime
```

The input to that algorithm was a number (stored in variable `n` in the code above) and the result was a simple “yes” or “no” answer, printed in the final `if` statement.

However, we can run into a couple of issues with this code:

- What if we wanted to re-run that program to test the primality of a different number? If we were using the interpreter, we would have to type in all the code again (changing the value of `n` as we go). If we had saved the code to a file, we would have to edit the file manually to change the value of `n`.
- What if we wanted to use the *result* of the algorithm in a different context? For example, we could be writing a more complex program that, at some point, performs a specific action depending on whether a number is prime (certain cryptography algorithms, for example, need to make this choice). We would have to re-write the primality testing code and change the final `if`.
- What if we needed to use the primality testing code in multiple parts of our program? We would have to repeat the above code in multiple places, which can be very error-prone. For example, if we decided to update the algorithm, we would have to make sure to update every copy of it.

Functions address these issues. Functions allow us to name a piece of code and re-use it multiple times with different inputs (in fact, some programming languages refer to them as *subroutines* or *subprograms*).

This mechanism is called a function because, like a mathematical function, functions in Python take some input values, called the *parameters*, and produce a new value, called the *return value*:

We have already *seen* uses of functions in some of our examples, like the function `random.randint`, which returns a random integer:

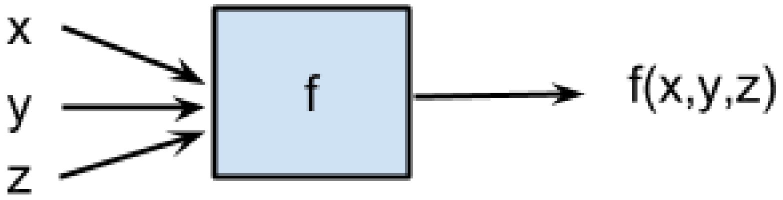


Fig. 1: Function

```
>>> import random
>>> n = random.randint(0, 1000)
>>> print("The value of n is", n)
The value of n is 873
```

When calling an existing function, you specify parameters in parentheses, with each parameter separated by a comma. `random.randint` takes two parameters: one to specify a lower bound and one to specify an upper bound. The code in `random.randint` takes the value of those parameters, computes a random number between them, and *returns* the random number. In the above example, we assign the returned value to variable `n`.

We can also call a function to perform an action rather than to compute a value. The `print` function, which we have used extensively, is an example of such a function.

```
>>> print("Hello, world!")
Hello, world!
>>> n = 5
>>> print("The value of n is", n)
The value of n is 5
```

The `print` function runs code necessary to print input parameters on the screen.

All Python functions return some value. Functions like `print` that perform an action rather than compute a value return the special value `None`.

As we will explore, functions are an important *abstraction* mechanism. They allow us to *encapsulate* a piece of code and then reuse that code, without being concerned with how that piece of code works (other than knowing its purpose, parameters and return value). For example, when we used `random.randint`, we did not have to worry about the exact algorithm used to produce a random number. We just incorporated this functionality into our code, *abstracting* away the internal details of `random.randint`.

In this chapter, we will cover the basics of writing our own functions, and also dig into some of the lower-level aspects of how functions work. As we will point out then, you can safely skip those low-level details for now, but may want to revisit them once you're more comfortable with functions, and definitely before we the Functional Programming and Recursion chapters, where we explore more advanced concepts involving functions.

4.1 Anatomy of a function

To describe the basic elements of a Python function, we will start with a very simple function that takes two integers and returns the product of those two integers:

```
def multiply(a, b):
    '''
    Compute the product of two values.

```

(continues on next page)

(continued from previous page)

```
    Inputs:
      a, b: the values to be multiplied.

    Returns: the product of the inputs
    '''

    n = a * b
    return n
```

Let's break down the above code:

- The `def` keyword indicates that we are *def*-ining a function. It is followed by the name of the function (`multiply`).
- The name of the function is followed by the names of the *parameters*. These names appear in parentheses, with parameters separated by commas, and are followed by a colon. (The parenthesis are required, even if the function has no parameters.) The parameters are the *input* to the function. In this case, we are defining a function to multiply two numbers, so we must define two parameters (the numbers that will be multiplied). Sometimes we refer to these names as the *formal parameters* of the function to distinguish them from the actual values provided when the function is used. The line(s) containing keyword `def`, the function name, the parameters, and the colon are known as the *function header*.
- A docstring and the body of the function follow the colon, both indented one level from the function header.
- A *docstring* is a multi-line string (delimited by triple quotes, either `'''` or `"""`) that contains at least a brief description of the purpose of the function, the expected inputs to the function, and the function's return value. While docstrings are not required by the syntax of the language, it is considered good style to include one for *every* non-trivial function that you write.
- The body of the function is a block of code that defines what the function will do. Notice that the code operates on the parameters. As we'll see later on, the parameters will take on specific values when we actually run the function. At this point, we are just defining the function, so none of the code in the body of the function is run just yet; it is simply being associated with a function called `multiply`.
- Notice that the body of the function contains a `return` statement. This statement is used to specify the return value of the function (in this case, `n`, a variable that contains the product of parameters `a` and `b`). The last line of the body of the function is typically a `return` statement but, as we'll see later on, this statement is not strictly required.

Try typing in the function into the interpreter:

```
>>> def multiply(a, b):
...     '''
...     Compute the product of two values.
...
...     Inputs:
...       a, b: the values to be multiplied.
...
...     Returns: the product of the inputs
...     '''
...
...     n = a * b
...     return n
...
```

Once the `multiply` function is defined, we can call it directly from the interpreter. To do so, we just write the name of the function followed by the values of the parameters in parentheses, with the parameters separated by commas:

```
>>> multiply(3, 4)
12
```

This use of `multiply` is referred to as a *function call*, and the values passed to the function are referred to as either the *actual parameters* or the *arguments*. This call will run the code in the `multiply` function, initializing `a` with the value 3 and `b` with the value 4. More specifically, remember that the body of the function was this code:

```
n = a * b
return n
```

When the function is called with parameters 3 and 4, the function effectively executes this code:

```
a = 3
b = 4
n = a * b
return n
```

And then the Python interpreter prints `12`, the value returned by the function called with parameters 3 and 4.

Later on, we'll see that passing arguments to functions is more complicated, but for now, you can think of a function simply initializing formal parameters in the function body from the actual parameter values specified in the function call.

You can also include function calls in expressions:

```
>>> 2 + multiply(3, 4)
14
```

When Python evaluates the expression, it calls `multiply` with actual parameters 3 and 4 and uses the return value to compute the value of the expression. In general, you can use a function call in any place where its return value would be appropriate. For example, the following code is valid because the `print` function can take an arbitrary number of arguments and both string and integer are acceptable argument types:

```
>>> print("2 x 3 =", multiply(2, 3))
2 x 3 = 6
```

Of course, context matters. The following code is not valid because the function call returns an integer and Python can't add strings and integers:

```
>>> "2 x 3 = " + multiply(2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

Finally, parameter values can also be expressions, as long as the expression yields a value of the expected type. For example:

```
>>> x = 3
>>> y = 5
>>> multiply(x, y)
15
>>> multiply(x-1, y+1)
12
```

The actual parameter expressions are evaluated *before* the multiply function is called. As a result, the first call to multiply uses 3 and 5 (the values of x and y respectively) as parameters. The second call to multiply uses x-1, or 2, and y+1, or 6, as the initial values for a and b.

In fact, the parameters to a function can themselves be function calls:

```
>>> multiply(4, multiply(3, 2))
24
```

In this case, Python first evaluates the inner call to multiply (that is, multiply(3, 2)) and, then uses the call's value (6) as the second parameter to multiply. The outer call essentially becomes multiply(4, 6).

A Common Pitfall

The distinction between a function that *prints* something and a function that *returns* something is important, but often misunderstood. Our multiply function, for example, returns the product of its two arguments.

Let's look a similar function that prints the result instead:

```
def print_multiply(a, b):
    '''
    Print the product of two values.

    Inputs:
      a, b: the values to be multiplied.

    Returns: None
    '''

    n = a * b
    print(n)
```

When we call this function:

```
>>> print_multiply(5, 2)
10
```

It appears to return 10. The Python interpreter displays integers returned from a function and integers printed using print in the same way (other Python interpreters, like IPython, explicitly distinguish between the two). We can see this difference if we explicitly assign the return value to a variable and then print it:

```
>>> rv = multiply(5, 2)
>>> print("The return value is:", rv)
The return value is: 10
```

```
>>> rv = print_multiply(5, 2)
10
>>> print("The return value is:", rv)
The return value is: None
```

Notice that print_multiply still printed the value 10, but the return value of the function is the special value None.

Additionally, although it is valid to use the result of one call to multiply as an argument to another, it is not valid do use a call to print_multiply in the same way.

```
>>> multiply(5, multiply(2, 3))
30
```

```
>>> print_multiply(5, print_multiply(2, 3))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 11, in print_multiply
TypeError: unsupported operand type(s) for *: 'int' and 'NoneType'
```

In general, if you are writing a function that produces a value that you want to use elsewhere, make sure that you *return* that value. Printing it is not sufficient.

4.2 Encapsulating primality testing

We can use a function to encapsulate our primality testing code, allowing us to use it easily and in more places. Our function will take an integer as a parameter and return `True` if the integer is prime and `False` if it is not.

```
def is_prime(n):
    '''
    Determines whether the input is prime.

    Inputs:
      n (int): value to be checked

    Returns (boolean): True if the input is prime and False
      otherwise
    '''

    encountered_divisor = False
    for i in range(2, n):
        if n % i == 0:
            encountered_divisor = True
            break

    return not encountered_divisor
```

Once we type this function into the interpreter, we can run it as many times as we want.

```
>>> is_prime(4)
False
>>> is_prime(17)
True
>>> is_prime(81)
False
```

This is a big improvement from the previous chapter, where we either had to type the code in all over again or edit a file to modify the value of the number we were testing and rerun the code.

Instead of typing the function into the interpreter, you can create a file named `primes.py` that contains the above function, and then run the following in the interpreter:

```
>>> import primes
```

You will be able to call the function from the interpreter like this:

```
>>> primes.is_prime(4)
False
>>> primes.is_prime(7)
True
```

Similarly, you can use `import primes` in another Python file to get access to the function. In this context, we would refer to `primes` as a *module*. Python already includes many built-in modules that provide access to a large collection of useful functions. For example, earlier in the chapter we used Python's `random` module and, more specifically, the `randint` function contained in that module.

Before we move on to the next topic, let's fix a bug you might have noticed in our implementation of `is_prime`: it does not handle one as an argument properly:

```
>>> primes.is_prime(1)
True
```

We could fix this problem by using a more complex expression in the return statement:

```
def is_prime(n):
    '''
    Determines whether the input is prime.

    Inputs:
      n (int): value to be checked

    Returns (boolean): True if the input is prime and False
      otherwise
    '''

    encountered_divisor = False
    for i in range(2, n):
        if n % i == 0:
            encountered_divisor = True
            break

    return (n != 1) and (not encountered_divisor)
```

```
>>> is_prime(1)
False
>>> is_prime(4)
False
>>> is_prime(7)
True
```

But as you will see in the next section, there is a better way to resolve this problem.

4.3 Return statements

The functions we've seen so far have a single return statement at the end of the function. A return statement can appear anywhere in the function and can appear multiple times. For example:

```
def absolute(x):
    '''
    Compute the absolute value of a number.

    Inputs:
        n (number): operand

    Returns (number): the magnitude of the input
    '''

    if x < 0:
        return -x
    else:
        return x
```

```
>>> absolute(3)
3
>>> absolute(-3)
3
```

We can use multiple return statements to simplify the return statement in our second version of the `is_prime` function. Specifically, we can modify the function to treat one as a special case, immediately returning `False` when the function is called with `1` as the parameter.

```
def is_prime(n):
    '''
    Determines whether the input is prime.

    Inputs:
      n (int): value to be checked

    Returns (boolean): True if the input is prime and False
    otherwise
    '''

    if n == 1:
        return False

    encountered_divisor = False
    for i in range(2, n):
        if n % i == 0:
            encountered_divisor = True
            break

    return not encountered_divisor
```

In fact, we can tweak the function further to avoid using the `encountered_divisor` variable altogether. If we find that `n` is divisible by `i`, then we can return `False` right away. If we make it through the whole `for` loop, then `n` must be prime:

```
def is_prime(n):
    '''
    Determines whether the input is prime.

    Inputs:
      n (int): value to be checked

    Returns (boolean): True, if the input is prime and False
    otherwise
    '''

    if n == 1:
        return False

    for i in range(2, n):
        if n % i == 0:
            return False

    return True
```

```
>>> is_prime(1)
False
>>> is_prime(4)
False
>>> is_prime(7)
True
```

Python computes the return value and leaves the function *immediately* upon encountering a `return` statement. For example, when it executes the first call above (`is_prime(1)`), Python encounters the `return` statement within the first conditional and exits the function before it reaches the loop. In contrast, in the second call (`is_prime(4)`), Python enters the loop and encounters the return statement during its first iteration. Python reaches the final return statement only during the third call (`is_prime(7)`).

Tip

As with `break` and `continue`, every `return` statement in a loop should be guarded by a conditional statement. Because Python returns from the current function as soon as it encounters a `return` statement, a loop with an unguarded `return` will never execute more than one iteration (and possibly, a partial iteration at that).

4.4 Advantages of using functions

Functions help us organize and abstract code. We'll look at simulating *Going to Boston*, a simple game with dice, to help illustrate this point.

Going to Boston is played with three dice by two or more players who alternate turns. When it is a player's turn, they first roll all three dice and set aside the die with the largest face value, then roll the remaining two dice and set aside the largest one, and finally, roll the remaining die. The sum of the resulting face values is their score for the round. The players keep a running total of their scores until one reaches 500 and wins.

4.4.1 Reusability

Let's think about how to implement the code for a single round. We could write code that implements the rules directly:

```python
def play_round():
    '''
    Play a round of the game Going to Boston using three dice.

    Inputs: none

    Return (int): score earned
    '''

    NUM_SIDES = 6
    score = 0

    # roll 3 dice, choose largest
    die1 = random.randint(1, NUM_SIDES)
    die2 = random.randint(1, NUM_SIDES)
    die3 = random.randint(1, NUM_SIDES)
    largest = max(die1, max(die1, die2))
    score +=  largest

    # roll 2 dice, choose largest
    die1 = random.randint(1, NUM_SIDES)
    die2 = random.randint(1, NUM_SIDES)
    largest = max(die1, die2)
    score += largest

    # roll 1 die, choose largest
    largest = random.randint(1, NUM_SIDES)
    score += largest

    return score
```

This code is not very elegant, but it does the job and we can call it over and over again to play as many rounds as we want:

```
>>> play_round()
13
>>> play_round()
10
>>> play_round()
9
```

Although this implementation works, it has repeated code, a sign of a poor design. We should abstract and separate the repeated work into a function. At times, this task will be easy because your implementation will have a block of code repeated verbatim. More commonly, you will have several blocks of similar but not identical code. When faced with the latter case, think carefully about how to abstract the task into a function that can be used, with suitable parameters, in place of original code.

In the function `play_round`, for example, we have three variants of code to roll dice and find the largest face value (keep in mind the largest face value of a single die roll is the value of the single die). To abstract this task into a function, we will take the number of dice as a parameter and replace the calls(s) to `max` with a single call.

```python
def get_largest_roll(num_dice):
    '''
    Roll a specified number of dice and return the largest face
    value.

    Inputs:
      num_dice (int): the number of dice to roll

    Returns (int): the largest face value rolled
    '''

    NUM_SIDES = 6

    # initialize largest with a value smaller than the smallest
    # possible roll.
    largest = 0
    for i in range(num_dice):
        roll = random.randint(1, NUM_SIDES)
        largest = max(roll, largest)

    return largest
```

Given this function, we can replace the similar blocks of code with functions calls that have appropriate inputs:

```python
def play_round():
    '''
    Play a round of the game Going to Boston using three dice.

    Inputs: None

    Returns: the score earned by the player as an integer.
    '''

    score = get_largest_roll(3)
    score += get_largest_roll(2)
    score += get_largest_roll(1)
    return score
```

This version is easier to understand and yields a new function, `get_largest_roll`, that may be useful in other contexts.

As an aside, we chose to implement the rules for playing a round of Going to Boston with exactly three dice. We could have chosen to generalize it by taking the number of dice as a parameter. For example:

```python
def play_round_generalized(num_dice):
    '''
    Play a round of the game Going to Boston.

    Inputs:
      num_dice (int): the number of dice to use

    Returns (int): score earned
    '''

```

(continues on next page)

(continued from previous page)

```
    score = 0
    for nd in range(1, num_dice+1):
        score += get_largest_roll(nd)
    return score
```

A good rule of thumb is to start with a simple version of a function and generalize as you find new uses.

4.4.2 Composability

In addition to reducing repeated code, we also use functions as building blocks for more complex pieces of code. For example, we can use our `play_round` function to simulate a two-person version of Going to Boston.

```
def play_going_to_boston(goal):
    '''
    Simulate one game of Going to Boston.

    Inputs:
      goal (int): threshold for a win.

    Returns: None
    '''

    player1 = 0
    player2 = 0

    while (player1 < goal) and (player2 < GOAL):
        player1 += play_round()
        if player1 >= goal:
            break
        player2 += play_round()

    if player1 > player2:
        print("player1 wins")
    else:
        print("player2 wins")
```

This function is more complex than a simple call to `play_round`, so let's look at it carefully. Notice that we take the winning score as a parameter instead of hard-coding it as 500. Next, notice that in the body of the loop, we play a round for `player2` only if `player1` does not reach the winning score. The loop ends when one player reaches the target score; we then print the winner.

Since both of our implementations of `play_round` complete the same task and have the same interface, we could use either with our implementation `play_go_to_boston`. In fact, it is common to start with a straightforward algorithm, like our very basic implementation of `play_round`, for a task and only come back to replace if it becomes clear that an algorithm that is more efficient or easier to test is needed.

In general, we want to design functions that allow the function's user to focus on the function's purpose and *interface* (the number and types of arguments the function expects and the number and types of value the function returns), and to ignore the details of the implementation. In this case, we didn't need to understand how rounds are played to design the loop; we only needed to know that `play_round` does not take any arguments and returns the score for the round as an integer.

A word about our design choices: the decision to make the winning score a parameter seems like a good choice, because it makes our function more general without adding substantial burden for the user. The decision to combine the code for running one game and printing the winner, on the other hand, has fewer benefits. Our implementation fulfills the stated task but yields a function that is not useful in other contexts. For example, this function would not be useful for answering the question of "is there a significant benefit to going first?" A better design would separate these tasks into two functions:

```
def play_one_game(goal):
    '''
    Simulate one game of Going to Boston.

    Inputs:
      goal (int): threshold for a win

    Returns (boolean): True if player1 wins and False, if player2
      wins.
    '''

    player1 = 0
    player2 = 0

    while (player1 < goal) and (player2 < goal):
        player1 += play_round()
        if player1 < goal:
            player2 += play_round()

    return player1 > player2


def play_going_to_boston(goal):
    '''
    Simulate one game of Going to Boston and print the winner.

    Inputs:
      goal (int): threshold for a win

    Returns: None
    '''

    if play_one_game(goal):
        print("player1 wins")
    else:
        print("player2 wins")
```

This design allows us to simulate many games to see if the first player has an advantage:

```
def simulate_many_games(num_trials, goal):
    '''
    Simulate num_trials games of Going to Boston and print the
    average number of wins.

    Inputs:
      num_trials (int): number of trial games to play
      goal (int): threshold for a win

```

(continues on next page)

(continued from previous page)

```
    Returns: None
    '''

    wins = 0
    for i in range(num_trials):
        if play_one_game(goal):
            wins = wins + 1
    print(wins/num_trials)
```

Simulating 10,000 trials with a goal of 500 shows that there is a big advantage to going first: `player1` wins roughly 60% of the time.

```
>>> simulate_many_games(10000, 500)
0.6029
>>> simulate_many_games(10000, 500)
0.5958
>>> simulate_many_games(10000, 500)
0.6057
```

4.4.3 Testability

Finally, functions make code easier to test. A piece of code that fulfills a specific purpose (e.g., to determine whether a number is prime or roll N dice and return the largest face value) is far easier to test when it is encapsulated inside a function.

As we saw with primality testing earlier, once our code is encapsulated as a function, we can test it informally by running the function by hand with parameters for which we know the correct return value. Of course, manually testing functions in the interpreter can still be cumbersome. The process of testing code can be largely automated using *unit test frameworks* that allow us to specify a series of tests, and easily run all of them.

4.5 Variable scope

In Python and other programming languages, variables have a specific *scope*, meaning that they are only valid and can only be used in a specific part of your code. This concept is especially relevant for functions because any variables that are defined within a function have *function scope*, meaning that they are only valid within that function (i.e., within the *scope* of that function).

Variables that are only valid in a specific scope, such as the scope of a function, are commonly referred to as *local variables* (as opposed to *global variables*, which we will discuss later). A function's formal parameters are local variables because they are valid only within the scope of the function.

Calling a function alters the control flow of a program: when Python reaches a function call, it "jumps" to the function, runs through the statements in the function, and finally, returns to the point in the code where the function was called. Let's look more carefully at what happens during a call to `play_round`:

```
def play_round():
    '''
    Play a round of the game Going to Boston using three dice.

    Inputs: None

```

(continues on next page)

(continued from previous page)

```
    Returns: the score earned by the player as an integer.
    '''

    score = get_largest_roll(3)
    score += get_largest_roll(2)
    score += get_largest_roll(1)
    return score
```

The first statement in `play_round` introduces a new variable, named `score` and sets its initial value to the result of evaluating a function call (`get_largest_roll(3)`). To evaluate this call, Python will:

- evaluate the argument (3),
- create the parameter `num_dice` and initialize it to the result of evaluating the argument (3),
- transfer control to the body of `get_largest_roll`,
- execute the first two statements, which create new variables named `NUM_SIDES` and `largest`, and initialize them to 6 and 0, respectively,
- execute the loop, which itself introduces new variables, named `i` and `roll`, that are reset for each iteration,
- set the return value of the function call to be the value of `largest`,
- discard the variables it created during the function evaluation (e.g., `num_dice` and `NUM_SIDES`), and
- transfer control back to the first statement of `play_round`.

The variables `NUM_SIDES`, `largest`, `i`, and `roll` are valid only within `get_largest_roll`. We would get an error if we tried to use one them or the parameter (`num_dice`) outside of the context of this function. For example, this code will fail,

```
>>> score = get_largest_roll(3)
>>> print(largest)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'largest' is not defined
```

Similarly, the variable `score` is visible inside `play_round`, but not within `get_largest_roll.` Any attempt to use `score` inside `get_largest_roll` would also fail.

The second and third statements in `play_round` also call `get_largest_roll` and update the variable `score`. Python will go through exactly the same steps to evaluate the second call, including creating *fresh* versions of the parameter (`num_dice`) and the rest of the local variables (`NUM_SIDES`, `largest`, `i`, and `roll`), initializing them appropriately, and eventually discarding them when the function returns. Finally, it will complete the same process all over again to evaluate the third call.

In all, three distinct sets of the local variables for `get_largest_roll` will be created and discarded over the course of a single execution of `play_round`.

4.6 Parameters

Understanding how parameters work is an important aspect of learning how to design and write functions. So far, we have seen some fairly simple parameters, but, as we'll see in this section, parameters have several features and nuances that we need to be aware of when writing functions.

4.6.1 Call-by-value

As discussed above, expressions used as arguments for a function call are evaluated *before* the function call, and their *values* are used to initialize fresh copies of the formal parameters. This type of parameter passing is known as *call-by-value*.

Let's consider a trivial example to illustrate one impact of this design. Here's a function that updates the value of its formal parameter and three functions that use it:

```
def add_one(x):
    print("The value of x at the start of add_one is", x)
    x = x + 1
    print("The value of x at the end of add_one is", x)
    return x

def f():
    y = 5
    print("The value of y before the the call to add_one is", y)
    z = add_one(y)
    print("The value returned by the call to add_one is", z)
    print("The value of y after the the call to add_one is", y)

def g():
    y = 5
    print("The value of y before the the call to add_one is", y)
    z = add_one(5)
    print("The value returned by the call to add_one is", z)
    print("The value of y after the the call to add_one is", y)

def h():
    x = 5
    print("The value of x before the the call to add_one is", x)
    z = add_one(x)
    print("The value returned by the call to add_one is", z)
    print("The value of x after the the call to add_one is", x)
```

Here's what happens when we call function `f`:

```
>>> f()
The value of y before the the call to add_one is 5
The value of x at the start of add_one is 5
The value of x at the end of add_one is 6
The value returned by the call to add_one is 6
The value of y after the the call to add_one is 5
```

As expected, we do not see any changes to `y`. When Python reached the call to `add_one` in `f`, it evaluated the expression (`y`) and initialized a fresh copy of `x`, the formal parameter of `add_one`, to the resulting value (`5`). As the function executed, the value of that copy of x was read, updated, and then read again. Once add_one returned to f, the copy of x was discarded.

From the perspective of the add_one function, there is no difference between the call in function f and the call in function g.

```
>>> g()
The value of y before the the call to add_one is 5
The value of x at the start of add_one is 5
The value of x at the end of add_one is 6
The value returned by the call to add_one is 6
The value of y after the the call to add_one is 5
```

We can even reuse the name x in place of the name y, as seen in function h, and the values printed by our code will not change:

```
h()
```

```
The value of x before the the call to add_one is 5
The value of x at the start of add_one is 5
The value of x at the end of add_one is 6
The value returned by the call to add_one is 6
The value of x after the the call to add_one is 5
```

Why? Because the variable x defined in function h and the formal parameter x in add_one are *different* variables that just happen to have the same name.

4.6.2 Default parameter values

Suppose we wanted to write a function that simulates flipping a coin n times. The function itself would have a single parameter, n, and would return the number of coin flips that landed on heads. The implementation of the function needs a for loop to perform all of the coin flips and uses random.randint to randomly produce either a 0 or a 1 (arbitrarily setting 0 to be heads and 1 to be tails):

```
import random

def flip_coins(n):
    '''
    Flip a coin n times and report the number that comes up heads.

    Input:
      n (int): number of times to flip the coin

    Returns (int): number of flips that come up heads.
    '''

    num_heads = 0

    for i in range(n):
        flip = random.randint(0,1)
        if flip == 0:
            num_heads = num_heads + 1
```

(continues on next page)

(continued from previous page)

```
    return num_heads
```

```
>>> flip_coins(100)
55
```

As expected, if we flip the coin a large number of times, the number of flips that come out heads approaches 50%:

```
>>> n = 100000
>>> heads = flip_coins(n)
>>> print(heads/n)
0.49912
```

Now, let's make this function more general. Right now, it assumes we're flipping a fair coin (i.e., there is an equal probability of getting heads or tails). If we wanted to simulate a weighted coin with a different probability of getting heads, we could add an extra parameter `prob_heads` to provide the probability of getting heads (for a fair coin, this value would be `0.5`):

```
def flip_coins(n, prob_heads):
```

Of course, we also have to change the implementation of the function. We will now use a different function from the `random` module: the `uniform(a,b)` function. This function returns a *float* between `a` and `b`. If we call the function with parameters `0.0` and `1.0`, we can simply use values less than `prob_heads` as indicating that the coin flip resulted in heads, and values greater than or equal to `prob_heads` to indicate tails:

```
def flip_coins(n, prob_heads):
    '''
    Flip a weighted coin n times and report the number that come up
    heads.

    Input:
      n (int): number of times to flip the coin
      prob_heads (float): probability that the coin comes up heads

    Returns (int): number of flips that came up heads.
    '''

    num_heads = 0

    for i in range(n):
        flip = random.uniform(0.0, 1.0)
        if flip < prob_heads:
            num_heads = num_heads + 1

    return num_heads
```

Like before, we can informally validate that the function seems to be working correctly when we use large values of `n`:

```
>>> n = 100000
>>> heads = flip_coins(n, 0.7)
>>> print(heads/n)
0.69942
```

(continues on next page)

(continued from previous page)

```
>>> heads = flip_coins(n, 0.5)
>>> print(heads/n)
0.49649
```

However, it's likely that we will want to call `flip_coins` to simulate a fair coin most of the time. Fortunately, Python allows us to specify a *default value* for parameters. To do this, we write the parameter in the form of an assignment, with the default value "assigned" to the parameter:

```
def flip_coins(n, prob_heads=0.5):
    '''
    Flip a weighted coin n times and report the number that come up
    heads.

    Input:
      n (int): number of times to flip the coin
      prob_heads (float): probability that the coin comes up heads
        (default: 0.5)

    Returns (int): number of flips that came up heads.
    '''

    num_heads = 0

    for i in range(n):
        flip = random.uniform(0.0, 1.0)
        if flip < prob_heads:
            num_heads = num_heads + 1

    return num_heads
```

We can still call the function with the `prob_heads` parameter:

```
>>> flip_coins(100000, 0.7)
69750
>>> flip_coins(100000, 0.35)
35227
```

But, if we omit the parameter, Python will use `0.5` by default:

```
>>> flip_coins(100000)
49912
```

When you specify a function's parameters, those without default values must come first, followed by those with default values (if there are any). Notice that the following code fails:

```
def flip_coins(prob_heads=0.5, n):
    '''
    Flip a weighted coin n times and report the number that come up
    heads.

    Input:
      prob_heads (float): probability that the coin comes up heads
        (default: 0.5)
```

(continues on next page)

(continued from previous page)

```
      n (int): number of times to flip the coin

    Returns (int): number of flips that came up heads.
    '''

    num_heads = 0

    for i in range(n):
        flip = random.uniform(0.0, 1.0)
        if flip < prob_heads:
            num_heads = num_heads + 1

    return num_heads
```

```
  File "<stdin>", line 1
SyntaxError: non-default argument follows default argument
```

4.6.3 Positional and keyword arguments

So far, whenever we have called a function, we have specified the arguments in that function call in the same order as they appear in the function's list of parameters (with the ability to omit some of those parameters that have a default value). These are referred to as *positional arguments*, because how they map to specific parameters depends on their *position* in the list of arguments. For example:

```
>>> flip_coins(100000, 0.7)
69897
```

In this case, `100000` will be the value for the `n` parameter and `0.7` will be the value for the `prob_heads` parameter.

It is also possible to specify the exact parameter that we are passing a value by using *keyword arguments*. This type of argument follows the same syntax as an assignment; for example:

```
>>> flip_coins(n=100000, prob_heads=0.7)
69992
```

Notice that, because we explicitly provide a mapping from values to parameters, the position of the arguments no longer matters:

```
>>> flip_coins(prob_heads=0.7, n=100000)
70169
```

Keyword arguments can make a function call easier to read, especially for functions that have many parameters. With keyword arguments, we do not need to remember the exact position of each parameter.

It is possible to use both positional arguments and keyword arguments, although in that case the positional arguments must come first.

For example, this call works:

```
>>> flip_coins(100000, prob_heads=0.7)
69976
```

But this call doesn't:

```
>>> flip_coins(prob_heads=0.7, 100000)
  File "<stdin>", line 1
SyntaxError: positional argument follows keyword argument
```

4.6.4 Dynamic typing revisited

Let's return to our `multiply` function and revisit the impact of dynamic typing. In this case, we've defined a function that is intended to work with numbers. One of the nice things about Python is that, as defined, the function will also seamlessly work with floats:

```
>>> multiply(2.5, 3.0)
7.5
```

It even works when one parameter is an integer and one is a string:

```
>>> multiply("hello ", 3)
'hello hello hello '
```

but it does not work for all combinations of types. Because Python does not verify the types of parameters when the code is loaded, passing incompatible values can lead to errors at runtime:

```
>>> multiply(3.0, "hello")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 11, in multiply
TypeError: can't multiply sequence by non-int of type 'float'
```

This behavior is both an advantage and a disadvantage of dynamically-typed languages. It is an advantage because we can often use the same code for different types of inputs. It is a disadvantage because we do not learn about these types of errors until runtime and it is very discouraging to have a long-running computation fail with a type error. In a statically-typed language, we would specify the types of the parameters. Doing so can make the definition more rigid, but enables us to catch errors more quickly.

4.7 Global variables

As mentioned earlier, variables defined in a function are known as *local* variables. Variables defined outside the context of a function are known as *global* variables.

Let's say we have an application that needs to compute the maximum of N rolls of a die in some places and the sum of N rolls of a die in other places and that the number of sides in a die is constant (that is, the value does not change once it is set). To avoid defining variables for this constant in multiple places in our code, we could move the definition out of the functions altogether.

```
NUM_SIDES = 6

def get_largest_roll(num_dice):
    '''
    Roll a specified number of dice and return the largest face
    value.

    Inputs:
```

(continues on next page)

(continued from previous page)

```
      num_dice (int): the number of dice to roll

    Returns (int): the largest face value rolled
    '''

    # initialize largest with a value smaller than the smallest
    # possible roll.
    largest = 0
    for i in range(num_dice):
        roll = random.randint(1, NUM_SIDES)
        largest = max(roll, largest)

    return largest

def sum_throw(num_dice):
    '''
    Throw a specified number dice and sum up the resulting rolls.

    Inputs:
      num_dice (int): the number of dice to roll

    Returns (int): the sum of the face values rolled
    '''

    total = 0
    for i in range(num_dice):
        total = total + random.randint(1, NUM_SIDES)

    return largest
```

In this example, the variable `NUM_SIDES` is defined and then used in both functions. As an aside, it is common to use names with all capital letters to signal that a value is a constant.

This is an excellent use of a global variable. We avoid repeated code (i.e., multiple definitions of the same value) and, since the value does not change, there is never any confusion about the variable's current value.

Common Pitfalls

It can be tempting to use global variables as a way to reduce the number of parameters that are passed to a function. If, for example, let's say you found a new use for `get_largest_roll`, but with an eight-sided die instead of a six-sided die. You might be tempted to simply update the value of `NUM_SIDES` to `8` before calling `get_largest_roll`:

```
>>> NUM_SIDES = 8
>>> get_largest_roll(3)
6
```

Don't. This design is bad. If you forget to set `NUM_SIDES` back to `6`, subsequent calls to `get_largest_roll` (especially those related to its orginal purpose) might be incorrect in a particularly pernicious way: the answer looks fine (e.g., a `3` is returned) even though the function is no longer modelling the desired behavior.

The appropriate way to extend the use of `get_largest_roll` is to use an optional parameter to allow the user to provide information about the die in some cases and not others:

```
NUM_SIDES = 6

def get_largest_roll(num_dice, die_num_sides=NUM_SIDES):
    '''
    Roll a specified number of dice and return the largest face
    value.

    Inputs:
        num_dice (int): the number of dice to roll
        die_num_sides (int): the number of sides on a die␣
↪(default: 6)

    Returns (int): the largest face value rolled
    '''

    # initialize largest with a value smaller than the smallest
    # possible roll.
    largest = 0
    for i in range(num_dice):
        roll = random.randint(1, die_num_sides)
        largest = max(roll, largest)

    return largest
```

Using this implementation, the original uses of the function will work without change, while new uses can supply the number of sides for the die as needed.

To avoid writing code that is hard to understand and debug, beginning programmers should limit themselves to using global variables for constants. Even more experienced programmers should use them with great caution.

When a global variable and a local variable have the same name, the local variable *shadows* the global variable. That is, the local variable takes precedence. Here is a simple example of this behavior:

```
>>> c = 5
>>> def add_c(x, c):
```

(continues on next page)

(continued from previous page)

```
...     '''
...     Add x and c
...     '''
...     return x + c
...
...
>>> add_c(10, 20)
30
```

When the call to `add_c` is made, a fresh local variable `c` is created with the initial value of `20`. This local shadows the global `c` and is the variable used in the computation.

Here is a related example that illustrates another aspect of shadowing:

```
>>> c = 5
>>> def add_10(x):
...     '''
...     Add 10 to x
...     '''
...     c = 10
...     return x + c
...
...
>>> add_10(10)
20
>>> c
5
```

When the call to `add_10` reaches the assignment to `c` it defines a *local* variable named `c`. It does *not* change the value of the global variable of the same name; the global `c` still has the value `5` after the call.

Technical Details

It is possible to override Python's default behavior to update a global variable within a function by declaring that it is `global` *before* is it used in the function. Here is a trivial use of this feature:

```
>>> c = 5
>>> def update_c(new_c):
...     '''
...     Update the value of the global variable c
...     '''
...
...     global c
...     c = new_c
...
...
>>> c
5
>>> update_c(10)
>>> c
10
```

Notice that in this function, the value of the global variable named `c` has changed after the call to `update_c` because Python was informed that it should use the global `c` before it reached the assignment statement in the function `update_c`. As a result, it it does not create and set a fresh local variable. Instead, it updates the global variable `c`.

This mechanism should be used with great care as indiscriminate use of updates to global variables often yields code that is buggy and hard to understand.

4.8 The function call stack

Understanding what happens under the hood during a function call helps us understand scoping and many other nuances of functions. This section delves into some of the lower-level details of how functions work. You can safely skip for now it if you like, but you should revisit it once you become more comfortable with functions.

Programs, including Python programs, usually have a location in memory called the *call stack* (usually referred to as the *stack*) that keeps track of function calls that are in progress. When a program begins to run, the call stack is empty because no functions have been called. Now, let's suppose we have these three simple functions:

```
def tax(p, rate):
    '''
    Compute the tax for a given price.

    Inputs:
      p (float): price
      rate (float): tax rate

    Returns (float): the computed tax
    '''
    t = p * rate
    return t

def taxed_price(price, t):
    '''
```

(continues on next page)

(continued from previous page)

```
    Compute the price with tax.

    Inputs:
      price (float): price
      rate (float): tax rate

    Returns (float): price with tax
    '''

    price = price + tax(price, t)
    return price

def main():
    '''
    Compute and print the price 100 with tax.

    Inputs: none

    Returns: None
    '''

    p = 100

    tp = taxed_price(p, 0.10)

    print("The taxed price of", p, "is", tp)
```

`taxed_price` takes a price (`price`) and a tax rate (`t`) and computes the price with tax. `tax` takes the same parameters and computes just the tax (although note how the tax rate parameter is called `rate`; this naming choice was a conscious decision). The `main` function calls `taxed_price` and prints information about the price with and without taxes.

Now, let's say we call the `main` function:

```
main()
```

This call will add an entry to the call stack:

Function: `main`
Parameters: None
Local Variables: None
Return Value: None

This entry, known as a *stack frame*, contains all of the information about the function call. This diagram shows the state of the function call at the moment it is called, so we do not yet have any local variables, nor do we know what the return value will be. However, the moment we run the statement `p = 100`, the stack frame will be modified:

Function: `main`
Parameters: None
Local Variables:

- p: 100

Return Value: None

Next, when we reach this line:

```
tp = taxed_price(p, 0.10)
```

A local variable, `tp` , will be created, but its value will remain undefined until `taxed_price` returns a value. So, the frame for `main` on the call stack will now look like this:

Function: `main`
Parameters: None
Local Variables:

- `p`: 100
- `tp`: *undefined*

Return Value: None

Remember that, when we call a function, the code that is currently running (in this case, the statement `tp = taxed_price(p, 0.10)`) is, in a sense, put on hold while the called function runs and returns a value. Internally, an additional frame is added to the call stack to reflect that a call to `taxed_price` has been made. By convention, we draw the new frame stacked below the existing frame:

Function: `main`
Parameters: None
Local Variables:

- `p`: 100
- `tp`: *undefined*

Return Value: None

Function: `taxed_price`
Parameters:

- `price`: 100
- `t`: 0.10

Variables: None
Return Value: None

Notice that the call stack retains information about `main`. The program needs to remember the state `main` was in before the call to `taxed_price` (such as the value of its local variables) so that it can return to that exact same state when `taxed_price` returns.

Next, notice that the value of parameter `price` is set to 100. Why? Because we called `taxed_price` like this:

```
taxed_price(p, 0.10)
```

We can now see why passing a variable as a parameter to a function doesn't modify that variable. The function receives the *value* of the variable, not the variable itself. This means that changes made inside the function won't change the variable itself. In this case, `taxed_price` receives the value of `p` (100), but does not modify `p` itself.

Now, in `taxed_price` we will run the following statement:

```
price = price + tax(price, t)
```

Once again, we are calling a function. As a result, the execution of `taxed_price` is paused while we run the `tax` function, which adds another frame to the call stack:

Function: `main`
Parameters: None
Local Variables:

- `p`: 100
- `tp`: *undefined*

Return Value: None

Function: `taxed_price`
Parameters:

- `price`: 100
- `t`: 0.10

Variables: None
Return Value: None

Function: `tax`
Parameters:

- `p`: 100
- `rate`: 0.10

Variables: None
Return Value: None

The order of the functions in the stack diagram is important: notice that `tax` appears under `taxed_price` (or is *stacked* below `taxed_price`) and `taxed_price` is below `main`. This means that `tax` was called from `taxed_price` which, in turn, was called from `main`. In other words, the stack contains information about not only each function call, but also the order in which those calls were made.

Now, let's get back to `tax`. It has the following two statements:

```
t = p * rate
return t
```

The first statement creates a new local variable `t`, and the second specifies the function's return value and terminates the function. So, after `return t` is run, the most recent frame of the call stack will look like this:

Function: `tax`
Parameters:

- `p`: 100
- `rate`: 0.10

Variables:

- `t`: 10

Return Value: 10

Once the *calling* function, `taxed_price`, has retrieved the return value, this frame will be *removed* from the stack. After `tax` returns, and the `price = price + tax(price, t)` statement in `taxed_price` is run, the stack will look like this:

Function: `main`
Parameters: None
Local Variables:

- `p`: 100
- `tp`: *undefined*

Return Value: None

Function: `taxed_price`
Parameters:

- `price`: 100
- `t`: 0.10

Variables:

- `price`: 110

Return Value: None

All of the parameters and local variables in a function's scope *disappear* as soon as the function returns. As we see above, the frame for `tax` is *gone*, along with all of the information associated with it, including its local variables. In addition, calling `tax` again will create a *fresh* frame for the call stack: the values of the parameters and local variables from previous calls will not carry over into new calls.

Similarly, once we execute the statement `return price` in the function `taxed_price`, its return value will be set to `110`, and Python will plug that return value into the statement `tp = taxed_price(p, 0.10)` in `main` (which effectively becomes `tp = 110`). At this point, the call stack will look like this:

Function: `main`
Parameters: None
Local Variables:

- `p`: 100
- `tp`: 110

Return value: None

The `main` function will then call `print`, which create a frame for `print` stacked below `main`'s frame. After `print` returns, `main` itself doesn't return anything explicitly, which means the return value of `main` will default to `None`:

```
>>> rv = main()
The taxed price of 100 is 110.0
>>> print(rv)
None
```

Although all of the above may seem like a lot of under-the-hood details, there are a few important takeaways:

- Every time we call a function, the values of all of its parameters and local variables are stored in a freshly-created stack frame and only exist while the function is running. We cannot access variables or parameters from any other stack entry unless they were passed as parameters to the current function and, even then, we will only get their values.
- When a function returns, the values of its parameters and variables are discarded and they do not persist into future function calls. For example, if we called `tax` again, it would not "remember" that a previous call already set a value for its local variable `t`.

CHAPTER

FIVE

BASICS OF CODE ORGANIZATION

Back in *Programming Basics*, we wrote our first program:

```
print("Hello, world!")
```

And saw that we could run it directly in the Python interpreter, like this:

```
>>> print("Hello, world!")
Hello, world!
```

Or by placing the code in a file called `hello.py`, and running that file from the terminal:

```
$ python hello.py
Hello, world!
```

In the previous chapter, we also saw that it is possible to write function definitions in a Python file, such as the `primes.py` file that contained our `is_prime` function, and them "import" the contents of that file into the interpreter:

```
>>> import primes
>>> primes.is_prime(4)
False
>>> primes.is_prime(7)
True
```

So, it seems that Python files can serve two purposes: I can *run* a Python file, which may produce some result (like printing `Hello, world!`) or I can *import* a Python file, which allows me to use functions that are defined inside that file. In this chapter, we will expand on this distinction (running vs. importing) by introducing the notion of Python *modules*, and providing an introduction to how to organize Python code in a program.

5.1 Python modules

A Python module is, quite simply, a file containing Python code. The `hello.py` file mentioned above is a Python module, as is the `primes.py` file from the previous chapter. However, as we saw earlier, we used each of these files in very different ways: we "ran" the `hello.py` file, but we `import`-ed the `primes.py` file. To explore this distinction, we'll start by elaborating on what it means to "import" a module. To do this, we will use two modules which you can find in our *example code*: `getting-started/code-organization/primes.py` and `getting-started/code-organization/mersenne.py` In Python, modules are usually referred to without the `.py` extension, so we will refer to these as the "`primes` modules" and the "`mersenne` module".

If you look at the `primes` module, you'll see it contains two functions, `print_primes` and `is_prime`:

```
def print_primes(max_n):
    # ...

def is_prime(n):
    # ...
```

The actual implementation of these functions won't be relevant to our discussion of modules, as we'll be focusing on how these functions are *called* across modules, and not on how they work internally. So, we won't be digging into their implementation, but we nonetheless encourage you to take a quick look at the implementation of the `is_prime` function: it is a bit more complex than the one we saw in the previous chapter, because we need a faster algorithm for this chapter's example, and it provides several examples of using conditional and looping statements in a non-trivial way.

As we've seen previously, we can use the `import` statement to access the contents of the `primes` module from the interpreter:

```
>>> import primes
>>> primes.is_prime(4)
False
>>> primes.is_prime(7)
True
```

The `import` statement also allows us to import only specific functions from a module, like this:

```
>>> from primes import is_prime
>>> is_prime(17)
True
>>> is_prime(42)
False
```

We can actually also import modules from *other* modules as well. In particular, the `mersenne` module provides a number of functions related to *Mersenne primes*, or prime numbers of the form $2^p - 1$, where p is itself a prime number. One of these functions needs to call the `is_prime` function, located in the `primes` module, so we will need to import the `primes` module from the `mersenne` module, which looks something like this:

```
import primes

def is_mersenne_prime(p):
    # ...

def is_power_of_two(n):
    # ...

def print_mersenne_primes(max_p):
    # ...
```

As with the `primes` module, the exact implementation of these functions won't be relevant to our discussion, but notice how the `print_mersenne_primes` function uses the `is_prime` function from the `primes` module:

```
if not primes.is_prime(p):
```

We are able to use this function because we included the `import primes` statement at the top of the `mersenne` module.

Now, let's try using a function from the `mersenne` module from the interpreter. Before doing so, exit the interpreter and start it again, so we can make sure the previous `import primes` statement we ran from the interpreter isn't affecting this next example.

```
>>> import mersenne
>>> mersenne.print_mersenne_primes(100)
M1 is 2
M2 is 3
M3 is 5
M4 is 7
M5 is 13
M6 is 17
M7 is 19
M8 is 31
M9 is 61
M10 is 89
```

Notice how we are able to call the `print_mersenne_primes` function in the `mersenne` module, which internally requires using the `is_prime` function, located in a different module. However, we are not required to run `import primes` ourselves in the interpreter, because it is already being imported from inside the `mersenne` module.

5.2 Running vs importing a module

So far, we've seen that Python modules can be imported from the interpreter and from other modules, but modules can also be *run* from the command-line. To better understand this distinction, we will use an `arithmetic` module which you can find in the `getting-started/code-organization/` directory of the examples. You'll see that there are three versions of this module: `arithmetic`, `arithmetic_nomain`, and `arithmetic_main`.

We'll start by looking at the `arithmetic` module, which contains two very simple functions:

```
def add(x, y):
    return x + y + 1


def multiply(x, y):
    return x * y
```

As expected, we can import this module and use it from the interpreter:

```
>>> import arithmetic
>>> arithmetic.add(2, 10)
13
>>> arithmetic.multiply(2, 10)
20
```

But what happens if we *run* this module from the command-line?

```
$ python3 arithmetic.py
$
```

Nothing happens: Python returns immediately. The reason for this is that Python ran through all the code in the `arithmetic.py` file, and only encounters function definitions. Python internally makes a note that these functions have been defined, but there are no statements in the file that would make Python do something, like print a message or call the functions.

So, let's take a look at this slightly modified version, `arithmetic_nomain`:

```python
def add(x, y):
    return x + y


def multiply(x, y):
    return x * y


a = add(2, 10)
m = multiply(2, 10)

print("add(2, 10) = ", a)
print("multiply(2, 10) =", m)
```

This version includes some statements after the function definitions. If we run this module we'll see the following:

```
$ python3 arithmetic_nomain.py
add(2, 10) =  12
multiply(2, 10) = 20
$
```

What's happening here is that Python runs through the code, makes a note of that functions `add` and `multiply` have been defined, and then encounters code that calls those functions and prints something, and runs that code as well.

This may seem like a convenient way to define a few functions, and then include some basic code to informally test those functions, but there is a snag: that code will also run when we import the module, resulting in this:

```
>>> import arithmetic_nomain
add(2, 10) =  12
multiply(2, 10) = 20
>>> arithmetic_nomain.add(2, 10)
12
>>> arithmetic_nomain.multiply(2, 10)
20
```

The reason for this is that importing a module also causes Python to run through all the code in the corresponding Python file, which is why we are then able to use the functions defined in that module. However, we may want to be selective about what code is run exactly and, in particular, we may want to separate out the code that should only run when the module is run from the command line. We can do this by placing the code in a *main block*. We can see what this looks like in the `arithmetic_main` module:

```python
def add(x, y):
    return x + y


def multiply(x, y):
    return x * y


if __name__ == "__main__":
    a = add(2, 10)
    m = multiply(2, 10)

    print("add(2, 10) = ", a)
```

(continues on next page)

(continued from previous page)

```
    print("multiply(2, 10) =", m)
```

When we run it, the code under `if __name__ == "__main__":` runs as expected:

```
$ python3 arithmetic_main.py
add(2, 10) =  12
multiply(2, 10) = 20
$
```

But that code *won't* be run if we import the module:

```
>>> import arithmetic_main
>>> arithmetic_main.add(2, 10)
12
>>> arithmetic_main.multiply(2, 10)
20
```

All that said, this doesn't mean that every Python module has to have a main block. In the next section, we will elaborate on what a computer program is, and how Python programs often span multiple modules (where typically only one module will have a main block).

Reloading modules

When you import a module from the interpreter, Python will import the current version of that module, and won't track changes in that module. This means that, if you make a change to the module, that change won't automatically propagate to the interpreter. For example, try doing the following:

```
>>> import arithmetic
>>> arithmetic_main.add(2, 10)
12
```

Now, try modifying the `arithmetic.py` file modify the add function to look like this:

```
def add(x, y):
    return x + y + 1
```

Let's also add the following function:

```
def subtract(x, y):
    return x - y
```

If you try to use the add function, you'll see it still behaves according to the original (correct) version. Python will also tell you it can't find a `subtract` function:

```
>>> arithmetic_main.add(2, 10)
12
>>> arithmetic_main.subtract(100, 10)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: module 'arithmetic_main' has no␣
↪attribute 'subtract'
```

Interestingly, importing the module again won't actually resolve the situation:

```
>>> import arithmetic
>>> arithmetic_main.add(2, 10)
12
>>> arithmetic_main.subtract(100, 10)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: module 'arithmetic_main' has no␣
↪attribute 'subtract'
```

You need to explicitly reload the module using Python's built-in `importlib` module:

```
>>> import importlib
>>> importlib.reload(arithmetic)
<module 'arithmetic' from 'arithmetic.py'>
>>> arithmetic.add(2, 10)
13
>>> arithmetic.subtract(100, 10)
90
```

5.3 Computer programs revisited

In *Programming Basics*, we said that “a computer program is, at its core, a collection of instructions that the computer must perform”. We now have a better sense of what these instructions look like: `if` statements, `for` and `while` loops, assignments, function definitions, function calls, etc. And, as we’ve seen a few times already, the vessel for these instructions is text file with a name ending in `.py`, i.e., a Python module.

However, this doesn’t mean that a program is composed of exactly one module. While simple programs can often be implemented in one module, it is very common for programs to span multiple modules. To see an example of this, take a look at the `prime-checker` module in the `getting-started/code-organization/` examples directory. This is a module with a main block that asks the user to enter a number, and will then print out some information about whether the number is prime or not, and whether it is a Mersenne prime or not:

```python
from primes import is_prime
from mersenne import is_mersenne_prime, is_power_of_two

if __name__ == "__main__":
    n = input("Enter a number: ")

    n = int(n)

    if not is_prime(n):
        print(f"{n} is not a prime number.")
    else:
        p = is_power_of_two(n+1)
        if p is not None:
            if is_mersenne_prime(n):
                print(f"{n} is a double Mersenne prime: both {n} and 2^{n}-1 are␣
↪Mersenne primes.")
            else:
                print(f"{n} is a Mersenne prime ({n} == 2^{p}-1)")
        else:
            if is_mersenne_prime(n):
                print(f"{n} is a prime number, but not a Mersenne prime (however, 2^{n}-
↪1 is a Mersenne prime).")
            else:
                print(f"{n} is a prime number, but not a Mersenne prime (and neither is␣
↪2^{n}-1).")
```

Here are some sample runs of this program:

```
$ python3 prime-checker.py
Enter a number: 16
16 is not a prime number.
```

```
$ python3 prime-checker.py
Enter a number: 23
23 is a prime number, but not a Mersenne prime (and neither is 2^23-1).
```

```
$ python3 prime-checker.py
Enter a number: 61
61 is a prime number, but not a Mersenne prime (however, 2^61-1 is a Mersenne prime).
```

```
$ python3 prime-checker.py
Enter a number: 127
127 is a double Mersenne prime: both 127 and 2^127-1 are Mersenne primes.
```

However, all the logic involved in testing each number's primality is contained in two other modules: `primes` and `mersenne`. Our program, thus, spans three modules: `prime-checker`, `primes`, and `mersenne`.

This is a further example of *decomposition*: while we could have placed all the code in a single module, dividing it into distinct modules, each with a related set of functions, makes the code more manageable. It also improves the reusability of our code: if we wanted to write a different program that involves checking a number's primality, all we need to do is import our `primes` module.

All that said, this doesn't mean that "a collection of modules" is a program. A program is specifically something that is *executable*, which is the technical term for "something I can run" (in the manner we've described above, as opposed to just importing a module). In a Python program, this often means that at least one of the modules must include a `__main__` block.

On the other hand, when we have a collection of modules that provides some useful functionality, but which is not executable, that is what we would call a *software library*, or simply a *library*. For example, we could distribute the `primes` and `mersenne` modules as a "prime number library". Neither of these modules has a `__main__` block and that is totally fine: these modules are not meant to be run but, rather, to be imported by other modules.

While we may not develop that many libraries ourselves, we will almost certainly *use* existing libraries in our code. In particular, Python itself includes a vast collection of modules, called the Python Standard Library, that we can use in our code and which we describe next.

5.4 The Python Standard Library

When you install Python on your computer, you are not only getting a Python interpreter, but also access to a huge collection of modules that is already included with Python. This is known as the Python Standard Library, or PSL, and it provides all sorts of functionality that can come in handy when writing our code. It's hard to overstate how large and useful this library is: it includes modules for most common tasks you can imagine, such as string processing, math functions, file and directory access, network utilities, and much, much more. You can see the full content of the PSL here: https://docs.python.org/3/library/

We have actually already used two of the modules included with the PSL: the `random` module and the `math` module. Now that we know what modules are, we can better understand what is happening when we do this:

```
>>> import random
>>> random.randint(1,100)
61
```

Based on what we saw earlier this chapter, it would seem that running `import random` requires that there be a `random.py` file in the same directory as our code. Python *will* look for a `random.py` file in the same directory as our code first but, if it does not find it, it will check whether the PSL includes such a module (which it does). So, somewhere in the official Python code, there is a `random.py` file containing a bunch of functions related to random number generation, including a `randint` function (the actual `random.py` is actually a bit more complicated, and involves classes and objects, which we have not yet seen).

Where exactly is the PSL?

When you install any piece of software, some of that software will usually be installed in a "system directory", separate from the directories were you (a regular user) keep your own files. Without going to deep into how operating systems organize their filesystems, it is enough to know that the location of these system directories is well-known by applications running in your computer, including the Python interpreter.

This means that the Python interpreter knows to search through those system directories if we ask it to import a module (and that module can't be found in the same directory as our code). For example, this is where you would find the `random.py` in most operating systems:

- Windows: `C:\Program Files\Python 3.8\lib\random.py`
- MacOS: `/Library/Frameworks/Python.framework/Versions/3.8/lib/python3.8/random.py`
- Linux: `/usr/lib/python3.8/random.py`

CHAPTER

SIX

UNDERSTANDING ERRORS AND CATCHING EXCEPTIONS

When Python detects something wrong in your code, it will *raise* an *exception* to indicate that an error condition has occurred and it is severe enough that Python can't continue running the rest of your code.

A very simple example of an exception occurs when we try to divide a number by zero:

```
>>> x = 42 / 0
```

Trying to run the above code will result in an error like this:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

Let's start with the last line: it tells us the name of the exception (`ZeroDivisionError`) and a description of why the exception was raised (`division by zero`).

The error message contains a *stack trace* or a synopsis of the state of the call stack when the error is detected.

Specifically, in this example, the first two lines constitute the stack trace. Because we directly typed `x = 42 / 0` into the interpreter, these lines don't provide a lot of information in this case (it is immediately apparent that the error originated in the code we just typed).

The stack trace is much more useful when an exception is raised inside a function, as it will tell us the exact line inside that function where the exception was raised. Not just that, it will provide us with the complete sequence of function calls that led to the exception.

For example, suppose we wanted to write a simple program that prints the result of dividing an integer N by all the positive integers less than N.

```
def divide(a, b):
    ''' divide a by b '''
    return a / b

def print_divisions(N):
    ''' Print result of dividing N by integers less than N. '''
    for i in range(N):
        d = divide(N, i)
        print(N, "/", i, "=", d)
```

If we store this code in a file named `exc.py`, import it into python, and then call the `print_divisions` function, we'll see something like this:

```
>>> import exc
>>> exc.print_divisions(12)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "exc.py", line 8, in print_divisions
    d = divide(N, i)
  File "exc.py", line 3, in divide
    return a / b
ZeroDivisionError: division by zero
```

Notice how this stack trace is much more informative. In particular, it tells us than an exception happened after the following sequence of events:

- Python ran `exc.print_divisions(12)` in the interpreter (shown in the stack trace as as `File "<stdin>", line 1, in <module>`).
- During the call to `print_divisions`, Python ran the line `d = divide(N, i)`, located in line 8 in the file `exc.py`.
- During the call to `divide`, Python ran the line `return a / b`, located in line 3 of `exc.py`. This last entry in the stack trace produced a `ZeroDivisionError` exception.

Notice that this is simply a printed version of the function call stack, which we covered in section *The function call stack* of the previous chapter.

There is clearly something wrong in our code, and the exception's stack trace can be very useful to figure out exactly what is wrong. Frequently there is a difference between the line that raises the exception, and the code where the error actually originates. For example, in this case, they are not the same! While it is true that the `a / b` statement is the point at which the division by zero occurs, there is actually nothing wrong with that particular line or the `divide` function itself. As a result, we turn our attention to the next entry in the call stack:

```
File "exc.py", line 8, in print_divisions
   d = divide(N, i)
```

This line contains a hint at the source of the problem: `i` must have taken on the value `0` at some point to have triggered the `ZeroDivisionError` error. Where does the `i` get its value? From the `for` loop in `print_division`:

```
for i in range(N):
```

This `for` loop will iterate over the values 0, 1, 2, … N-1; we want to avoid the value zero, so the `range` should start at one:

```
for i in range(1, N):
```

If we make this change to `exc.py` and run `exc.print_divisions(12)` within the same session of the python interpreter, we'll get the exact same error as before:

```
>>> exc.print_divisions(12)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "exc.py", line 8, in print_divisions
    d = divide(N, i)
  File "exc.py", line 3, in divide
    return a / b
ZeroDivisionError: division by zero
```

Wait, didn't we fix this problem? Yes, but Python does not automatically reload your code when you change it. We can use the reload function from importlib to fix our problem:

```
>>> import importlib
>>> importlib.reload(exc)
<module 'exc' from 'exc.py'>
>>> exc.print_divisions(12)
12 / 1 = 12.0
12 / 2 = 6.0
12 / 3 = 4.0
12 / 4 = 3.0
12 / 5 = 2.4
12 / 6 = 2.0
12 / 7 = 1.7142857142857142
12 / 8 = 1.5
12 / 9 = 1.3333333333333333
12 / 10 = 1.2
12 / 11 = 1.0909090909090908
```

Now, we get the result we expect. The loop starts at 1 and no longer triggers the divide-by-zero exception.

So, when your code raises an exception, try not to fixate on the exact line that raises the exception. While that line could be wrong, it is just as likely that the actual origin of the exception is somewhere else in your code. The stack trace provides some hints as to where to look for the error. Systematically adding print statements that highlight the value of crucial variables can help you isolate the source of the error.

6.1 Catching exceptions

While exceptions can alert us to errors in our code, they can also be *caught* and handled in a way that is consistent with the goals of the application. In the case of our divide function, we'll just return None to indicate to the client that the value of a / b is not defined when b is zero.

We can catch an exception with a try statement, also known as a try .. except block. For example:

```
def divide(a, b):
    ''' divide a by b '''
    try:
        ret_val = a / b
    except ZeroDivisionError:
        # Send None back to the caller to signal
        # that a/b is not defined.
        ret_val = None
    return ret_val
```

If we replace the divide function with the above version, reload the file, and call print_divisions, we will see:

```
>>> importlib.reload(exc)
<module 'exc' from 'exc.py'>
>>> exc.print_divisions(12)
12 / 1 = None
12 / 2 = 6.0
12 / 3 = 4.0
12 / 4 = 3.0
```

(continues on next page)

(continued from previous page)

```
12 / 5 = 2.4
12 / 6 = 2.0
12 / 7 = 1.7142857142857142
12 / 8 = 1.5
12 / 9 = 1.3333333333333333
12 / 10 = 1.2
12 / 11 = 1.0909090909090908
```

A `try` statement allows us to “try” a piece of code, which we write after the `try` and, if it raises an exception, run an alternate piece of code, which can be found after the `except`. In this case, the division in the first call to `divide` will trigger the exception and the code in the `except` clause will be run and will set `ret_val` to `None`. Once the code in the `except` clause is finished, the `return` statement that follows the `try` statement will be executed and the value of `ret_val` will be returned to `print_divisions`, which, in turn, will simply print it as the result of the division. None of the subsequent calls to `divide` in the loop will raise the exception. In these cases, `ret_val` will simply be set to the result of the division and returned as expected.

The `divide` function as written now handles division by zero without failing. Notice, however, that it can still fail. For example, notice what happens if we pass it non-numeric arguments:

```
>>> exc.divide("abc", "a")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "exc.py", line 4, in divide
    return a / b
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

This usage raises a different type of exception, a `TypeError`. Our `try` statement catches the `ZeroDivisionError` exception, but not the `TypeError` exception. As a result, our program stops, and the stack trace message shown above is printed,

Fortunately, we can catch multiple types of exceptions in the same `try` statement:

```
def divide(a, b):
    try:
        ret_val = a / b
    except ZeroDivisionError:
        # Send None back to the caller to signal
        # that a/b is not defined.
        ret_val = None
    except TypeError as err:
        # Fail: no way to move forward.
        print("Type error:, err)
        sys.exit(1)
    return ret_val
```

Now when we call `divide` on strings, our error message is printed and the execution ends on the call to `sys.exit(1)`:

```
>>> exc.divide("abc", "a")
Type error: unsupported operand type(s) for /: 'str' and 'str'
```

This example also illustrates another feature of exceptions: we can use the keyword `as` to give a name to the exception that was caught. In this case, we use the name `err` and pass it to `print` along with the string `"Type error:"`. Using a mechanism that we’ll discuss in the chapter on *Classes and Objects*, `print` extracts a string that describes the exception that occurred from `err` and prints it.

We might not know all of the possible exceptions that can be raised by a given piece of code when we first write it or the set of possible exceptions might change over time (say, because a function we use has changed and can now raise a broader set of exceptions). If we want to make sure to deal with all possible types of exceptions, we can catch the generic exception Exception. This exception is best used to handle unexpected exceptions, as in:

```
>>> def divide(a, b):
...     try:
...         ret_val = a / b
...     except ZeroDivisionError:
...         # Send None back to the caller to signal
...         # that a/b is not defined.
...         ret_val = None
...     except TypeError as err:
...         # Fail: no way to move forward.
...         print("Type error:", err)
...         sys.exit(1)
...     except Exception as err:
...         # Fail: no way to move forward.
...         print("Unexpected error:", err)
...         sys.exit(1)
...
...     return ret_val
...
```

The last except clause will only be executed, if the code in the try block throws an exception other than ZeroDivisionError or TypeError. You might be tempted to use a generic Exception to catch everything. Don't. It is likely that your application will be better served by handling different exceptions in different ways.

The try statement also has an optional finally clause that gets run whether an exception is raised or not. This clause is useful when there are any cleanup operations that need to be performed (closing files, closing database connections, etc.) regardless of whether the code succeeded or failed. For example:

```
def divide(a,b):
    ''' Divide a by b and catch exceptions'''

    try:
        ret_val = a / b
    except ZeroDivisionError:
        ret_val = None
    except TypeError as err:
        print("Type error:", err)
        sys.exit(1)
    except Exception as err:
        print("Unexpected Error:", err)
        sys.exit(1)
    finally:
        print("divide() was called with {} and {}".format(a, b))

    return ret_val
```

```
>>> divide(6, 2)
divide() was called with 6 and 2
3.0
>>> divide(6, 0)
```

(continues on next page)

(continued from previous page)

```
divide() was called with 6 and 0
>>> divide(6, "foo")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 10, in divide
NameError: name 'sys' is not defined
```

Before we close this chapter, let's look at what happens when we catch some exceptions close to the source, but leave others to be handled higher up the call stack. Specifically, we'll return to the example from the start of the chapter. We've modified `divide` to catch the `TypeError`, but not `ZeroDivisionError`. Instead, we'll handle that error in `print_divisions`.

```
def divide(a,b):
    ''' Divide a by b and catch exceptions'''
    try:
        ret_val = a / b
    except TypeError as err:
        print("Type error:", err)
        sys.exit(1)

    return ret_val

def print_divisions(N):
    ''' Print result of dividing N by integers less than N. '''
    for i in range(N):
        try:
            d = divide(N, i)
            print(N, "/", i, "=", d)
        except ZeroDivisionError:
            print(N, "/", i, "is undefined")

print_divisions(12)
```

```
12 / 0 is undefined
12 / 1 = 12.0
12 / 2 = 6.0
12 / 3 = 4.0
12 / 4 = 3.0
12 / 5 = 2.4
12 / 6 = 2.0
12 / 7 = 1.7142857142857142
12 / 8 = 1.5
12 / 9 = 1.3333333333333333
12 / 10 = 1.2
12 / 11 = 1.0909090909090908
```

In the first iteration of the loop in `print_division`, `i` will be zero, which will cause the division operation in `divide` to raise an exception. The `try` statement in `divide` catches type errors, but not divide-by-zero errors. So, Python will propagate the error to call site in `print_division` to see if the call to `divide` is nested within a `try` statement that knows how to handle divide-by-zero errors. In this case it is, and so, the exception is handled by the `except` clause in `print_divisions`. In general, an exception will be propagated up the call stack until it is caught by an enclosing `try` statement or Python runs out of functions on the stack.

We have only skimmed the surface of exceptions in this chapter. You now know enough to read error messages and handle simple exception processing. We'll return to the topic of exceptions later in the book to look ways to catch related types of errors in one `except` clause and how to define and raise your own exceptions.

CHAPTER

SEVEN

EXAMPLE: A GAME OF CHANCE

Hazard is a game of chance that was popular in medieval England. To play, a designated player named the *caster* throws a pair of six-sided die and any number of other players can place bets on the caster's rolls. In this chapter, we will first describe the game and then write code to simulate it.

The game proceeds in rounds. At the start of a round, the caster chooses an integer, known as the *main*, in the range 5 to 9 inclusive. The caster then rolls both dice in the *come out roll*. The terminology for this part of Hazard is complex and does not add a lot to our understanding of the game. Without affecting the accuracy of our simulation, we will simply refer to the the sum of the come out roll as the *chance*.

Here are the rules for the come out roll:

- If the chance is equal to the caster's chosen main, the caster immediately wins the round.
- If the chance is 2 or 3, the caster immediately loses the round.
- If the chance is 11 or 12, then:
  - If the caster's main is 5 or 9, the caster loses the round.
  - If the caster's main is 6 or 8, the caster wins the round if the chance is 12 and loses the round otherwise.
  - If the caster's main is 7, the caster wins the round if the chance is 11 and loses the round otherwise.

If none of these conditions is met, the caster will continue to roll the dice until a throw matches the main or the chance. The caster wins the round if their last throw matches the chance and loses the round if it matches the main.

Here is a table that summarizes these rules:

Come out roll

Main	*Caster wins immediately on a chance of*	*Caster loses immediately on a chance of*	*Caster continues to throw on chance of*
5	5	2, 3, 11, 12	Any other value
6	6, 12	2, 3, 11,	Any other value
7	7, 11	2, 3, 12	Any other value
8	8, 12	2, 3, 11	Any other value
9	9	2, 3, 11, 12	Any other value

The caster continues playing rounds until they lose two consecutive rounds, at which point another player becomes the caster.

To make this description more concrete, let's play a few rounds with a chosen main of 5.

Round One			
Main	*Roll*	*Result*	*Notes*
5	9	Continues	Come out roll, chance is 9
5	7	Continues	Does not match main or chance
5	7	Continues	Does not match main or chance
5	9	Wins	Roll matches chance

In this round, because the caster rolls a 9 in the come out roll, the chance is 9. This value does not match any of the special cases for the first roll, so the caster continues to roll the dice. Neither of the next two rolls (7 and 7) match either the chance or the main, and so the caster rolls once more. In their final roll, the caster throws a 9, which matches the chance, and wins.

Round Two			
Main	*Roll*	*Result*	*Notes*
5	4	Continues	Come out roll, chance is 4
5	6	Continues	Does not match main or chance
5	11	Continues	Does not match main or chance
5	12	Continues	Does not match main or chance
5	5	Loses	Roll matches main

In the second round, because the caster rolls a 4 in the come out roll, the chance is 4. This value does not match any of the special cases, so the caster continues to roll the dice. The next three rolls (6, 11, and 12) do not match the chance or the main, so the caster rolls the dice a fifth time. The final roll is a 5, which matches the main, and the caster loses the round.

Round Three			
Main	*Roll*	*Result*	*Notes*
5	11	Loses	Come out roll

In the third round, the caster rolls an 11 in the come out roll and immediately loses. Since the caster has lost two rounds in a row, they pass the dice on to another player.

Given that brief explanation, let's look at some code for simulating this game. The first thing we need to do is simulate rolling a pair of dice.

7.1 Rolling dice

In Hazard, we throw of pair of dice and add the face values together. Because we will use this computation often in our simulation, it makes sense to write a very short function to do it:

```python
def throw_dice():
    '''
    Throw a pair of six-sided dice

    Returns (int): the sum of the dice
    '''
    NUM_SIDES = 6
    val = random.randint(1,NUM_SIDES) + random.randint(1,NUM_SIDES)
    return val
```

We chose to use a constant for number of sides for the die, but we could have easily written this function to take the number of sides as an argument.

7.2 Playing a round

The work needed to play a round is sufficiently complex that it makes sense to encapsulate it in a function. Since the caster chooses the main, this function will need to take the main as a parameter. The only detail that matters for the rest of the game is whether the caster won the round. We can return this information as a boolean: `True` for a win and `False` for a loss. Putting these design decisions together gives us the following function header:

```
def play_one_round(chosen_main):
    '''
    Play one round of Hazard.

    Inputs:
        chosen_main (int): a value between 5 and 9 inclusive.

    Returns (boolean): True, if player wins the round and False, otherwise.
    '''
```

To simulate the come out roll, we first call our `throw_dice` function and save the result in a variable named `chance`.

Next, we need to translate the rules for the come out roll. At the top-level, this computation requires a conditional with four branches. The first two branches are straightforward: if each condition holds, the caster wins in the first case (`return True`), and loses in the second (`return False`).

```
    if chance == chosen_main:
        return True
    elif (chance == 2) or (chance == 3):
        return False
```

To determine the outcome of the third case, we need a nested conditional that branches on the value of `chosen_main`.

```
    elif (chance == 11) or (chance == 12):
        if (chosen_main == 5) or (chosen_main == 9):
            return False
        elif (chosen_main == 6) or (chosen_main == 8):
            return (chance == 12)
        else:
            # chosen_main is 7
            return (chance == 11)
```

Notice that the outcome of the second branch of the inner conditional is not `True` or `False` but instead depends on whether the value of `chance` is 11 or 12. Since `chance` can only have one of these two values, we can compute the output with a simple boolean expression: `chance == 12`. The output of the third branch is computed similarly, using the boolean expression `chance == 11`.

Finally, if none of the first three conditions of the outer conditional statement hold, then Python executes the body of the `else` branch.

```
    roll = throw_dice()
    while not ((roll == chance) or (roll == chosen_main)):
        roll = throw_dice()
```

(continues on next page)

(continued from previous page)

```
    return (roll == chance)
```

In this case, the caster continues to throw the dice until a roll matches either the chance or the main. Let's look at how to encode this condition in two parts: The expression `(roll == chance) or (roll == chosen_main)` will be true when the roll matches either the chance or the main. Since our loop should continue only when that condition does not hold, we can simply negate this expression to get the proper test for the loop: `(not ((roll == chance) or (roll == chosen_main)))`

Notice that we again return the result of evaluating a boolean expression.

A note about using parenthesis in expressions: because `not` has higher precedence than `or`, the parenthesis surrounding the expression `((roll == chance) or (roll == main))` in the while loop test are necessary for correctness. The rest of parenthesis surrounding expressions in this function, in contrast, are not necessary for correctness. Some programmers include parenthesis to increase clarity, while others consider them unnecessary distractions. You can decide for yourself.

We can put these pieces together to get the following complete function:

```
def play_one_round(chosen_main):
    '''
    Play one round of Hazard.

    Inputs:
        chosen_main (int): a value between 5 and 9 inclusive.

    Returns (boolean): True, if player wins the round and False, otherwise.
    '''

    chance = throw_dice()

    if chance == chosen_main:
        return True
    elif (chance == 2) or (chance == 3):
        return False
    elif (chance == 11) or (chance == 12):
        if (chosen_main == 5) or (chosen_main == 9):
            return False
        elif (chosen_main == 6) or (chosen_main == 8):
            return (chance == 12)
        else:
            # chosen_main is 7
            return (chance == 11)

    roll = throw_dice()
    while not ((roll == chance) or (roll == chosen_main)):
        roll = throw_dice()

    return (roll == chance)
```

7.3 Simulating a player

We will also encapsulate the code for simulating an individual caster's turn in a function. Similar to play_one_round, this function will take the chosen main as a parameter. The betting rules for Hazard are complex so, for now, our simulation will simply count the number of rounds that the caster wins.

Recall that the caster plays rounds until they lose two in a row. Keeping track of the number of consecutive losses is the trickiest part of this function. We'll use a variable, consecutive_losses, that we initialize to zero because the caster has not yet played or lost any rounds. We will call play_one_round to simulate a round and update consecutive_losses as appropriate: on a loss, we will increment its value. On a win, we will reset it to zero.

As soon as the caster has lost two consecutive rounds, the test for the while loop will be false and the function will return the number of rounds won.

```python
def simulate_caster(chosen_main):
    '''
    Simulate rounds until the caster loses two rounds in a row.

    Inputs:
        chosen_main (int): a value between 5 and 9 inclusive.

    Returns (int): the number of rounds won
    '''

    num_wins = 0
    consecutive_losses = 0

    while consecutive_losses < 2:
        if play_one_round(chosen_main):
            consecutive_losses = 0
            num_wins = num_wins + 1
        else:
            consecutive_losses = consecutive_losses + 1

    return num_wins
```

7.4 Estimating win rate

Finally, we can easily reuse our function play_one_round to print a table of estimated per-round win rates for the different possible choices for main. There is a tradeoff between accuracy and computation time: simulating more rounds takes more time to compute but produces a more accurate result. Rather than hard-coding a specific number of rounds, we'll write our function to take the number of rounds as a parameter and let the user make this tradeoff.

We will use a pair of nested loops to generate the table. The outer loop will iterate over the possible values for main (range(5, 10)). Why 10? Recall that range generates a sequence of values from the specified lower bound up to, but not including, the specified upper bound. The inner loop will both simulate the desired number of rounds and keep track of the number of rounds won for the current value of chosen_main.

```python
def print_win_rate_table(num_rounds):
    '''
    Print a table with the win rates for the possible choices for main

    Inputs:
```

(continues on next page)

(continued from previous page)

```
        num_rounds (int): the number of rounds to simulate
    '''

    for chosen_main in range(5, 10):
        num_wins = 0
        for i in range(num_rounds):
            if play_one_round(chosen_main):
                num_wins = num_wins + 1
        print(chosen_main, num_wins/num_rounds)
```

Notice that for this computation, we need to start with a fresh win count for each possible value of `chosen_main`, so the value of `win` gets reset to zero for each iteration of the outer loop. As we noted in our earlier discussion of loops, initializing variables in the wrong place is a very common source of bugs.

Part II.

Data Structures

CHAPTER

EIGHT

LISTS, TUPLES, AND STRINGS

Up to this point, we have worked with simple data types: integers, floats, strings, and booleans. Whenever we used these data types, a variable contained a single value. For example, the variable `price` contains a single integer:

```
>>> price = 10
>>> price
10
```

Most programming languages allow us to construct a more complex data structure out of these basic data types. In fact, we caught a glimpse of this in *Control Flow Statements*, where we wanted to compute the total price (with tax) of a collection of prices:

```
prices = [10, 25, 5, 70, 10]

for p in prices:
    tax = 0.10 * p
    total = p + tax
    print("The price (with tax) is", total)
```

```
The price (with tax) is 11.0
The price (with tax) is 27.5
The price (with tax) is 5.5
The price (with tax) is 77.0
The price (with tax) is 11.0
```

The `prices` variable contains a *list* of integers, as opposed to a single integer. In Python, a list allows us to store and manipulate an ordered sequence of values. In this chapter, we will dig deeper into lists and learn how to create and manipulate them. Towards the end of the chapter, we will introduce a related data type, *tuples*. We will also discuss a number of features of strings that are easier to understand once we know how lists work.

In general, most programming languages provide some way of working with *collections* of values. Python's lists are a very versatile and powerful data type, arguably more so than similar data types provided in other languages. They are not, however, the be-all and end-all of working with multiple values: later in the book we will see other types, such as dictionaries and classes, that allow us to work with collections of values in other ways.

8.1 Creating lists

To create a list, we type the values in the list separated by commas and delimit the beginning and end of the list with square brackets. For example, here we assign a list of strings to a variable named `lang`:

```
>>> lang = ["C", "C++", "Python", "Java"]
>>> lang
['C', 'C++', 'Python', 'Java']
```

And here is a list of integers:

```
>>> nums = [3, 187, 1232, 53, 21398]
>>> nums
[3, 187, 1232, 53, 21398]
```

Since Python is a dynamically-typed language, a list can have values of different types. So, for example, this is a valid list:

```
>>> lst = ["string", 4, 5.0, True]
>>> lst
['string', 4, 5.0, True]
```

For the most part, we will avoid mixed-type lists because we typically want to operate on lists of values of the same type. In fact, statically-typed languages usually *require* all the values in a list to have the same type.

To create an empty list, we use square brackets with no values between them:

```
>>> lst = []
>>> lst
[]
```

We can also create *new* lists by concatenating existing lists using the + operator:

```
>>> lst0 = [1, 2, 3]
>>> lst1 = [4, 5, 6]
>>> lst2 = lst0 + lst1
>>> lst2
[1, 2, 3, 4, 5, 6]
```

The new list (`lst2`) contains the values from the first operand followed by the values from second operand.

Sometimes, we will want to create a list of a specific size with all the entries in the list set to the same initial value. For example, we may want to create a list with ten zeroes. We could do this task by explicitly writing out all the values:

```
>>> lst = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
>>> lst
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Of course, this approach would be cumbersome for long lists. So, we'll use the multiplication operator (*) to do this task instead:

```
>>> lst = [0] * 10
>>> lst
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

In general, given a list `l` and an integer value `n`, the expression `l * n` concatenates `n` copies of the list `l` to create a new list. The list `l` can even contain more than one value:

```
>>> lst = [0, 1] * 10
>>> lst
[0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1]
```

If the idea of multiplying a list and an integer seems odd to you, recall that multiplication is repeated addition and that adding two lists simply concatenates their values in a new list.

Once we have created a list, we can obtain its length using the built-in `len` function:

```
>>> lang
['C', 'C++', 'Python', 'Java']
>>> len(lang)
4
>>> nums
[3, 187, 1232, 53, 21398]
>>> len(nums)
5
>>> lst
[0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1]
>>> len(lst)
20
```

8.2 Accessing elements in a list

Once we have a list, we can access and update individual values within that list. To do so, we specify the variable containing the list followed by the position, or *index*, of the element we want to access enclosed in square brackets. Indexes are numbered from zero so if we wanted to access the third element in the list, we would use index `2`. For example:

```
>>> lang
['C', 'C++', 'Python', 'Java']
>>> lang[2]
'Python'
```

We can use individual elements from a list wherever a value of the element's type is appropriate. For example:

```
>>> nums
[3, 187, 1232, 53, 21398]
>>> (nums[0] + nums[1]) * nums[2]
234080
```

Lists are *mutable*: we can change both the contents and the size of a list. To update the value at an index, we assign a value to the list element at that index using the same indexing notation as when we read a value from the list:

```
>>> lang
['C', 'C++', 'Python', 'Java']
>>> lang[2] = "Python 3"
>>> lang
['C', 'C++', 'Python 3', 'Java']
```

Indexes are specified as integers, so the square brackets can contain any expression that evaluates to an integer. For example:

```
>>> i = 1
>>> lang[i]
'C++'
>>> lang[i + 1]
'Python 3'
```

Whether we are accessing or updating a value, we must always use a valid index. If we do not, our code will fail. For example, let's try to access the fifth element of the `lang` list, which only has four elements. Remember that, since lists are indexed from zero, we will use index `4` in our attempt to access the (non-existent) fifth element:

```
>>> lang[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

`IndexError` is an *exception*, which, as we saw in *Understanding Errors and Catching Exceptions*, is Python's way of informing us that something went wrong at runtime. In this case, the exception is telling us that our code failed with a `list index of out range` error. When you encounter an error of this variety, you will want to print the actual index and then look at your code and the stack trace provided in the error message to identify the origin of the bad index. (Alternatively, you may need to determine why your list is shorter than expected.)

Python also allows programmers to use negative indexes, counting from the end of the list, to access the list. So, index `-1` refers to the last element in the list, index `-2` is the next-to-last element of the list, and so on.

```
>>> lang
['C', 'C++', 'Python 3', 'Java']
>>> lang[-1]
'Java'
>>> lang[-2]
'Python 3'
```

Note that `lang[len(lang)-1]` and `lang[-1]` are equivalent ways to extract the last element in the list; the `len(lang)` part of the index expression is implicit in the second version.

As with positive indexes, Python will generate an `IndexError` exception at runtime if we specify a negative index that is out of range:

```
>>> lang[-5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

8.2.1 List slicing

In addition to accessing a single value from a list, we can also access all of the elements between two specified indexes of a list. This operation is called *slicing* and requires specifying the two indexes separated by a colon:

```
>>> lang
['C', 'C++', 'Python 3', 'Java']
>>> lang[1:3]
['C++', 'Python 3']
```

In the above example, the slice contains the elements starting at index 1 up to *but not including* index 3. The slice itself is a *new* list: it contains a copy of the values in the original list. So, we can modify the resulting slice without altering the original list. For example:

```
>>> lang
['C', 'C++', 'Python 3', 'Java']
>>> lst = lang[1:3]
>>> lst
['C++', 'Python 3']
>>> lst[0] = "C++14"
>>> lst
['C++14', 'Python 3']
>>> lang
['C', 'C++', 'Python 3', 'Java']
```

Note that, as with accessing individual elements, an index in a slice can be any expression that evaluates to an integer.

At times using a negative index as part of a slice can yield code that is easier to read. We can even mix and match negative and positive indices when we slice a list. For example, here are at two ways to extract a slice with all but the first and last elements of `lang`:

```
>>> lang[1:len(lang)-1]
['C++', 'Python 3']
>>> lang[1:-1]
['C++', 'Python 3']
```

Which version feels more natural to you? Here's another example: extract the last two elements of `lang`:

```
>>> lang[len(lang)-2:len(lang)]
['Python 3', 'Java']
>>> lang[-2:len(lang)]
['Python 3', 'Java']
```

When constructing a slice, one or both operands can be omitted. When first operand (that is, the value to the left of the colon) is omitted, its value defaults to 0:

```
>>> lang
['C', 'C++', 'Python 3', 'Java']
>>> lang[:2]
['C', 'C++']
```

When second operand (that is, the value to the right of the colon) is omitted, its value defaults to the length of the list:

```
>>> lang
['C', 'C++', 'Python 3', 'Java']
>>> lang[1:]
['C++', 'Python 3', 'Java']
>>> lang[-2:]
['Python 3', 'Java']
```

If both are omitted, the slice contains the entire list:

```
>>> lang
['C', 'C++', 'Python 3', 'Java']
```

(continues on next page)

(continued from previous page)

```
>>> lang[:]
['C', 'C++', 'Python 3', 'Java']
```

Remember that each slice itself is a new list, so the default slice notation (`[:]`) is commonly used to create a copy of the list.

```
>>> lang
['C', 'C++', 'Python 3', 'Java']
>>> lang2 = lang[:]
>>> lang2[0] = "Scheme"
>>> lang2[-1] = "Haskell"
>>> lang2
['Scheme', 'C++', 'Python 3', 'Haskell']
>>> lang
['C', 'C++', 'Python 3', 'Java']
```

Finally, we can specify an arbitrary *step* through the values in the slice by adding a second colon followed by the step size. For example, if we have a slice from index 1 up to but not including index 7 with a step of 2, we get the values at indexes 1, 3, and 5 (but not 7). For example:

```
>>> tens = [0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
>>> tens[1:7:2]
[10, 30, 50]
```

When the step is not specified, it defaults to one (i.e., we step through the elements of the list one by one).

The step *can* be negative:

```
>>> tens[7:1:-1]
[70, 60, 50, 40, 30, 20]
```

The bounds work as before: Python starts at the first bound (`7`) and goes up to but does not include the second bound (`1`) using the specified step size (`-1`).

Negative steps are most often used to yield a *new* list that contains the values of an existing list in reverse order:

```
>>> tens[::-1]
[100, 90, 80, 70, 60, 50, 40, 30, 20, 10, 0]
```

Omitted indices work differently when the step is negative. If we omit the first bound, the missing value defaults to the index of the last element in the list. A missing second bound defaults to a value immediately to the left of the first element in the list (thereby including the index of the first element in the range of legal values). We can use `None` as the bound and get the same behavior:

```
>>> tens[len(tens)-1:None:-1]
[100, 90, 80, 70, 60, 50, 40, 30, 20, 10, 0]
```

Unfortunately, there is no numeric value that we can specify that behaves in exactly the same way.

Degenerate slices

Perhaps surprisingly, *degenerate* slices, or slices with out-of-bounds indexes, do not fail at runtime; Python handles them gracefully instead. If you specify a bound that is too small, then Python uses the smallest legal value in its place (*e.g.*, `-len(lang)` for the list `lang`). If you specify a bound that is too large, Python uses the length of the list as the bound instead. Notice, for example, that evaluating `lang[-10:10]` and `lst[-len(lang):len(lang)]` yield the same four element list:

```
>>> lang[-10:10]
['C', 'C++', 'Python 3', 'Java']
>>> lang[-len(lang):len(lang)]
['C', 'C++', 'Python 3', 'Java']
```

One consequence of this design is that slicing an empty list is legal and simply yields a (new) empty list:

```
>>> [][0:2]
[]
```

It is also possible to generate an empty list when slicing a non-empty list. Here are two examples of this phenomenon, one that uses legal indexes and another that uses out-of-bounds indexes:

```
>>> lst[1:1]
[]
>>> lst[5:10]
[]
```

While this language design choice may seem at odds with your expectations, not having to worry about out-of-bounds indexes when constructing a slice is quite convenient.

Updating lists using slicing

Slices are typically used to extract a copy of part of a list, but they can be used on the left side of an assignment statement to specify a range of elements to be updated in an existing list as well. For example, the following code replaces the first two elements of `lang`:

```
>>> lang[0:2] = ["Ruby", "Scala"]
>>> lang
['Ruby', 'Scala', 'Python 3', 'Java']
```

This usage is much less common and should be used sparingly.

8.3 Iterating over a list

Often, you will need to perform some action for each element of a list. Processing a list element by element is usually called *iterating* over the list, and as we saw in *Control Flow Statements*, we can use a `for` loop to do this work:

```
prices = [10, 25, 5, 70, 10]

for p in prices:
    tax = 0.10 * p
    total = p + tax
    print("The price (with tax) is", total)
```

```
The price (with tax) is 11.0
The price (with tax) is 27.5
The price (with tax) is 5.5
The price (with tax) is 77.0
The price (with tax) is 11.0
```

Notice that the loop starts at the beginning of the list and proceeds through the list elements in order. In each iteration of the loop, the loop variable `p` contains the value of the list element that is being processed. The loop ends once it has iterated over all of the values in the list. If the list is empty, there are no values to process and so the body of the loop is never executed.

Alternatively, we could also use the `range` function, which we also discussed in *Control Flow Statements*, to iterate over the indexes of the list, and then use those to access the list:

```
prices = [10, 25, 5, 70, 10]

for i in range(len(prices)):
    tax = 0.10 * prices[i]
    total = prices[i] + tax
    print("The price (with tax) is", total)
```

```
The price (with tax) is 11.0
The price (with tax) is 27.5
The price (with tax) is 5.5
The price (with tax) is 77.0
The price (with tax) is 11.0
```

The above code produces the exact same result as the previous version, but it is considered poor style (you will often see this style referred to as "not Pythonic"). First of all, it is harder to read: instead of "iterating over a list of prices" we are "iterating over all valid indexes of a list of prices". It is also more error-prone: in the first version, `p` contains the price we are processing in the current iteration; in the second version, `i` contains an index that we could easily use by mistake as the price in place of `prices[i]`, the appropriate expression for accessing the `i`-th price in the list.

In other programming languages, iterating over the indexes is often the only way of iterating over a list. So, if you are already familiar with a language that uses this approach, remember that in Python, iterating over a range of indexes is typically considered poor style.

Occassionally, a task requires iterating over both the values in a list and their indices. Python provides a built-in function, named `enumerate`, that allows us to iterate over both the current value being processed in a loop and its index. Instead of specifying one name for the loop variable, we supply two–one for the index and one for the value–separated by a comma.

Here's a variant of the task above that uses both the index (named `i`) and the value (named `price`):

```
prices = [10, 25, 5, 70, 10]

for i, price in enumerate(prices):
    tax = 0.10 * price
    total = price + tax
    print("The price (with tax) of element", i, "is", total)
```

```
The price (with tax) of element 0 is 11.0
The price (with tax) of element 1 is 27.5
The price (with tax) of element 2 is 5.5
The price (with tax) of element 3 is 77.0
The price (with tax) of element 4 is 11.0
```

Tip

Keep in mind that cases where you need to know the index of each value as you iterate through a list are typically few and far between. So, when you are writing a loop to process a list, your first instinct should be to iterate over the values of the list. Only if you find that you need both the value and its index, should you use `enumerate`.

If you find yourself using `range` with the length of a list, you may want to rethink your implementation choices.

The line between values and indices can be a bit blurry. For example, it can be convenient to use the values in one list as indices into another list. Let's say we want to print the values in `lang` in an order that reflects one person's language preferences. We could represent those preferences as a list of integers between zero and three inclusive. The list `[2, 0, 3]`, for example, would represent the preferences of a person who prefers Python to C and C to Java and declined to rank C++. To print the languages in the preferred order we could write:

```
>>> preferences = [2, 0, 3]
>>> for p in preferences:
...     print(lang[p])
...
Python 3
Ruby
Java
```

Notice that `p`, which iterates over the preferences, is used as an index into the list `lang`.

8.4 Modifying a list

So far, we have seen how to create a list with a number of elements, and then access or modify those elements. We are not limited to working with just the elements specified when a list is created: lists can grow and shrink dynamically.

For example, lists have an `append` method that we can use to add a new element to the end of an existing list. A *method* is a special type of function that operates on the variable on which it is called. We call the `append` method like this:

```
lang.append("Scheme")
```

This code should be understood as follows: the `append` method will perform an operation on `lang`, and will do so with a parameter (`"Scheme"`), the value we want to add to the end of the list. For example:

```
>>> lang
['Ruby', 'Scala', 'Python 3', 'Java']
>>> len(lang)
4
>>> lang.append("Scheme")
>>> lang
['Ruby', 'Scala', 'Python 3', 'Java', 'Scheme']
>>> len(lang)
5
```

This operation modifies the list *in-place*. In other words, the `lang` list itself is modified, as opposed to having `append` return a new list with the appended value. We will see that many functions and methods modify lists in-place.

A Common Pitfall

A common mistake is to treat the append method as if it returns the updated list. It does not: it updates its list operand and returns None. As a result, a statement like this:

```
>>> lang = lang.append("Go")
```

Sets the value of lang to None, which is likely not what the programmer has in mind.

To avoid this mistake, never use the append method in an assignment statement or as part of a more complex expression.

Functions and methods

In *Introduction to Functions*, we introduced functions as an abstraction mechanism. Defining a function allows us to parameterize a task, assign it a name, and then use it by name in multiple places in our code. Methods also allow us to parameterize a task, assign it a name, and use it in multiple places. The difference between the two mechanisms, as you will see in more detail in *Classes and Objects*, is that methods are defined as part of a class and operate on (or are *applied to*) a specific value of that class (known as an *instance* of the class). The value is placed to the left of the . and the method name and parameters appear on the right of the .

In this code, for example,,

lang is a list (an instance of the list class) and append is the name of a list method that takes a single argument ("Scheme").

We will discuss how to define your own classes and methods in detail in a few chapters. For now, just keep in mind that operations on values, such as lists, are often defined as methods and are used with the . syntax.

Similar to append, we can use the extend method to extend an existing list with the contents of another list:

```
>>> lang = ["C", "C++", "Python", "Java"]
>>> lang2 = ["Pascal", "FORTRAN"]
>>> lang.extend(lang2)
>>> lang
['C', 'C++', 'Python', 'Java', 'Pascal', 'FORTRAN']
```

Like append, extend modifies the list in-place.

The insert method inserts new elements in a specific position and takes two parameters: the position where the new element will be inserted and the value to be inserted:

```
>>> lang
['C', 'C++', 'Python', 'Java', 'Pascal', 'FORTRAN']
>>> lang.insert(2, "Haskell")
>>> lang
['C', 'C++', 'Haskell', 'Python', 'Java', 'Pascal', 'FORTRAN']
```

Notice that insert doesn't replace the existing value in position 2. Instead, it inserts the new value at position 2 and shifts all the original values starting at index 2 to the right by one.

Finally, the pop method removes a value in a specific position. The pop method takes one parameter: the position of the item we want to remove.

```
>>> lang
['C', 'C++', 'Haskell', 'Python', 'Java', 'Pascal', 'FORTRAN']
>>> lang.pop(2)
```

(continues on next page)

(continued from previous page)

```
'Haskell'
>>> lang
['C', 'C++', 'Python', 'Java', 'Pascal', 'FORTRAN']
```

The pop method also returns the value that was removed from the list.

If we omit the parameter to pop, it will remove and return the last element in the list by default:

```
>>> lang
['C', 'C++', 'Python', 'Java', 'Pascal', 'FORTRAN']
>>> lang.pop()
'FORTRAN'
>>> lang
['C', 'C++', 'Python', 'Java', 'Pascal']
```

We can also remove elements using the `del` operator, which uses a different syntax. It is not a function or method, but a built-in operator, and it must be followed by the item we want to remove written using the list indexing syntax:

```
>>> lang
['C', 'C++', 'Python', 'Java', 'Pascal']
>>> del lang[3]
>>> lang
['C', 'C++', 'Python', 'Pascal']
```

Notice that, unlike pop, `del` doesn't return the value that is removed.

Tip

Do not modify lists as you iterate over them. Doing so may yield unexpected or surprising result. This loop, for example, may never terminate:

```
lst = [1, 2, 3, 4]
for val in lst:
    if val % 2 == 0:
        # val is even
        lst.append(val)
```

The next example terminates, but the desired result is not at all clear:

```
>>> l = [1, 2, 4, 5]
>>> for i, val in enumerate(l):
...     if i < len(l) - 1 and val % 2 == 0:
...         # remove the value that follows an
...         # even number.
...         del l[i+1]
...
```

Are we expecting to get `[1]` or `[1, 2, 5]`?

8.5 Creating a list based on another list

Now that we've seen how to iterate over a list as well as how to use the append method, we can discuss another common way of creating a list: by taking the values from an existing list, transforming them in some way, and creating a new list with the transformed values. For example, let's say we have a list of prices:

```
prices = [100.0, 59.99, 7.00, 15.00]
```

We may be interested in producing a new list that contains the same prices after a 10% discount is applied. We start by creating an empty list:

```
discounted_prices = []
```

Then, we use a for loop to iterate over the values in the original list. For each element, we transform the value by multiplying it by 0.9 and use the append method to add the transformed value to the new list:

```
for price in prices:
    new_price = price * 0.9
    discounted_prices.append(new_price)
```

The discounted_prices list now contains the discounted prices:

```
>>> discounted_prices
[90.0, 53.991, 6.3, 13.5]
```

This example shows the four key components used in this common computational pattern:

- a list to iterate over (prices, in the example above),
- a variable that holds the individual values over the course of the computation (price),
- an expression that computes a new value from the old one (price * 0.9), and
- a name for the resulting list (discounted_prices).

We can use the angle bracket notation that we introduced in *Control Flow Statements* to describe this pattern more formally as follows:

```
<list name> = []

for <variable name> in <list expression>:
    new_val = <transformation expression>
    <list name>.append(new_val)
```

Recall that words in **bold** represent keywords that are part of the language itself, whereas anything delimited with angle brackets means "substitute this for …" (i.e., you do not write the < and > characters themselves).

Python includes a special syntax, called a *list comprehension* for expressing these frequently-performed types of computations:

```
<list name> = [ <transformation expression> for <variable name>
                            in <list expression> ]
```

Here's the example shown above rewritten using this compact syntax:

```
>>> discounted_prices = [price * 0.9 for price in prices]
```

We are not required to assign the list comprehension to a variable. We can use it in any place where a list would be valid. For example:

```
>>> for n in [x**2 for x in range(1,11)]:
...     print(n)
...
1
4
9
16
25
36
49
64
81
100
```

The transformation expression can be any legal Python expression and is not required to use the values in the original list. For example, the list comprehension:

```
>>> [0 for i in range(10)]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

yields a list of ten zeros.

List comprehensions also allow us to filter which values from the original list will be included in the computation of the new list. For example, suppose we wanted to create a `discounted_prices` list, but only wanted to include prices (before the discount) that are greater than 10. If we were using a `for` loop, we might write something like this:

```
discounted_prices = []

for price in prices:
    if price > 10:
        new_price = price * 0.9
        discounted_prices.append(new_price)

print(discounted_prices)
```

```
[90.0, 53.991, 13.5]
```

With list comprehensions, we can just write this code instead:

```
>>> discounted_prices = [price * 0.9 for price in prices if price > 10]
>>> discounted_prices
[90.0, 53.991, 13.5]
```

So, the general syntax for list comprehensions becomes:

```
[ <transformation expression> for <variable name> in <list expression> if <boolean␣
↪expression> ]
```

where the boolean expression is evaluated for each value in the original list. If the boolean expression is true, the transformation expression is evaluated and the result is added to the new list.

In other words, this more general list comprehension expands into this code:

```
<list name> = []

for <variable name> in <list expression>:
    if <boolean expression>:
        new_val = <transformational expression >
        <list name>.append(new_val)
```

We will return to this topic again Section *List comprehensions revisited* to discuss an alternate way to think about the computation captured by the list comprehension shorthand.

8.6 Other operations on lists

Lists are a very versatile data structure in Python and we have a large number of operators, functions, and methods at our disposal to operate on them. We will only mention some of the more useful ones here; you can see the complete list in the official Python documentation, or by writing `help(list)` in the Python interpreter.

Given a list of integers:

```
>>> lst = [17, 47, 23, 101, 107, 5, 23]
```

We can find the minimum and maximum value with the built-in `min` and `max` functions:

```
>>> min(lst)
5
>>> max(lst)
107
>>> min([1, "abc", 3.14])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'str' and 'int'
```

The third example illustrates an important aspect of these functions: they will succeed only if there is a natural ordering on the elements of the list.

For lists containing numbers (either integers or floats), we can add up all the values in the list with the built-in `sum` function:

```
>>> sum(lst)
323
```

We can use the `count` method to count the number of occurrences of a given value:

```
>>> lst.count(47)
1
>>> lst.count(23)
2
>>> lst.count(29)
0
```

Note that while we could use this method to determine whether a list contains a specific value:

```
>>> lst.count(47) > 0
True
>>> lst.count(25) > 0
False
```

Python provides a more direct way to do this test, namely, the `in` operator:

```
>>> 47 in lst
True
>>> 25 in lst
False
```

We can reverse the list *in-place* with the `reverse` method:

```
>>> lst
[17, 47, 23, 101, 107, 5, 23]
>>> lst.reverse()
>>> lst
[23, 5, 107, 101, 23, 47, 17]
```

As noted earlier, we can produce a *new* list that contains the values of an existing list in reverse order using slicing:

```
>>> lst_reversed = lst[::-1]
>>> lst_reversed
[17, 47, 23, 101, 107, 5, 23]
>>> lst
[23, 5, 107, 101, 23, 47, 17]
```

We can sort the values in the list in ascending order using the `sort` method. Like `reverse`, this method will modify the list in-place:

```
>>> lst
[23, 5, 107, 101, 23, 47, 17]
>>> lst.sort()
>>> lst
[5, 17, 23, 23, 47, 101, 107]
```

If we want to create a sorted copy of the list without modifying the original list, we can use the built-in `sorted` function:

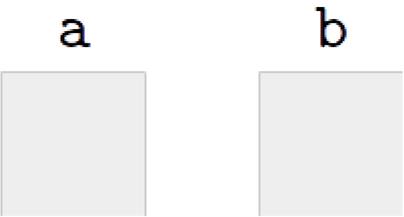
```
>>> lst = [17, 47, 23, 101, 107, 5, 23]
>>> lst2 = sorted(lst)
>>> lst2
[5, 17, 23, 23, 47, 101, 107]
>>> lst
[17, 47, 23, 101, 107, 5, 23]
```

By default, both the list `sort` method and the `sorted` function sort the values in a list in increasing order. Both have an optional keyword parameter, `reverse`, that can be set to `True` to indicate that the data should be sorted in decreasing order, instead. Notice, for example, the order of the elements in the result from this call to `sorted`:

```
>>> sorted(lst, reverse=True)
[107, 101, 47, 23, 23, 17, 5]
```

8.7 Variables revisited

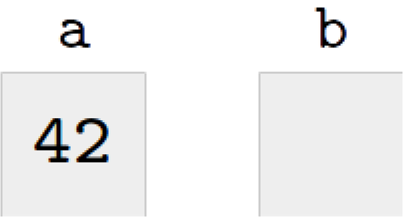
In *Programming Basics* we described variables as symbolic names representing locations in the computer's memory. When we assign a value to a variable, we are storing that value in the associated location in memory. So, say we have two variables a and b:



If we assign a value to variable a:

```
>>> a = 42
```

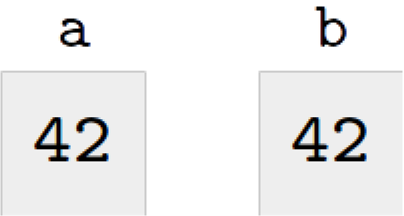
Then the position in memory represented by variable a will now contain the value 42:



If we now assign the value of variable a to b:

```
>>> b = a
```

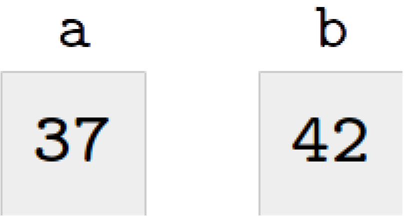
Then b will contain the same value as a:



However, each variable will still represent distinct positions in memory, which means that if we modify a:

```
>>> a = 37
```

The value of b is not affected:



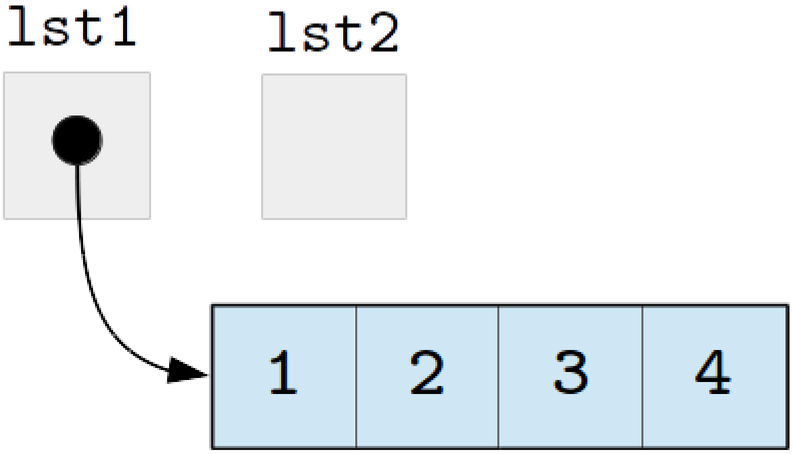
```
>>> a
37
>>> b
42
```

To be more specific, Python *copied* the value of variable `a` and stored the copy in `b` during the `b = a` assignment.

Lists, on the other hand, behave differently. Let's say we have two variables `lst1` and `lst2` and we assign a list to `lst1`:

```
>>> lst1 = [1, 2, 3, 4]
```

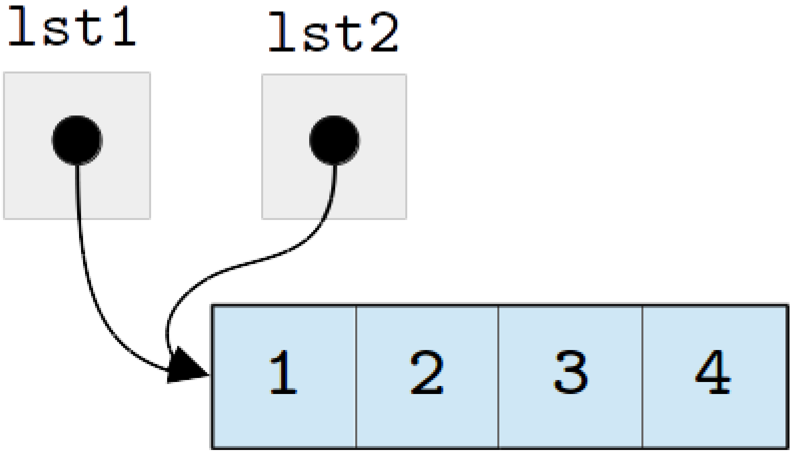
Unlike simple types like integers, the `lst1` variable (or, more specifically, the location in memory it represents) doesn't contain the list directly. Instead, it contains a *reference* to another location in memory that contains the list. We represent this behavior pictorially with an arrow going from `lst1` to the list:



If we now assign `lst1` to `lst2`:

```
>>> lst2 = lst1
```

`lst2` will point to the *same* list as `lst1`. In other words, assigning one list to another *does not* create a copy of the list; it creates a copy of the *reference* to the list, so that both variables point to the same list in memory:

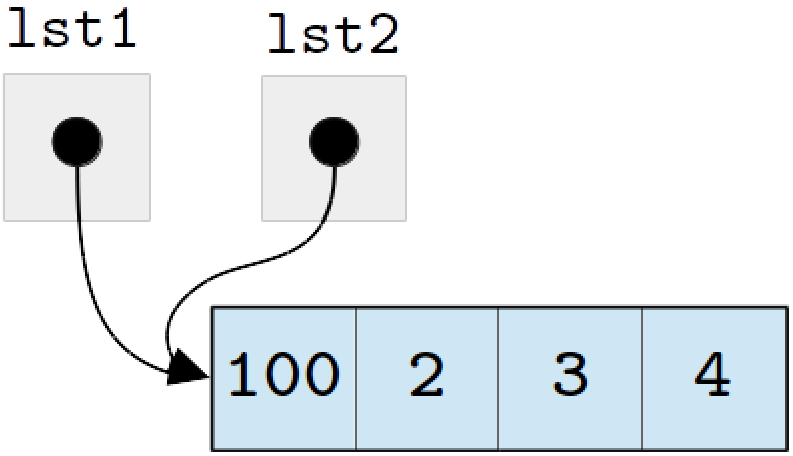


We refer to two (or more) variables that reference the same list as *aliases* of each other. The above assignment, for example, makes lst1 and lst2 aliases.

As a result of this sharing, changes made to a list through one variable *will* affect the other variable. For example, if we modify the first element of lst2:

```
>>> lst2[0] = 100
>>> lst2
[100, 2, 3, 4]
```

We are modifying both the list referenced by lst2 and the list referenced by lst1, because lst1 and lst2 refer to the same list.



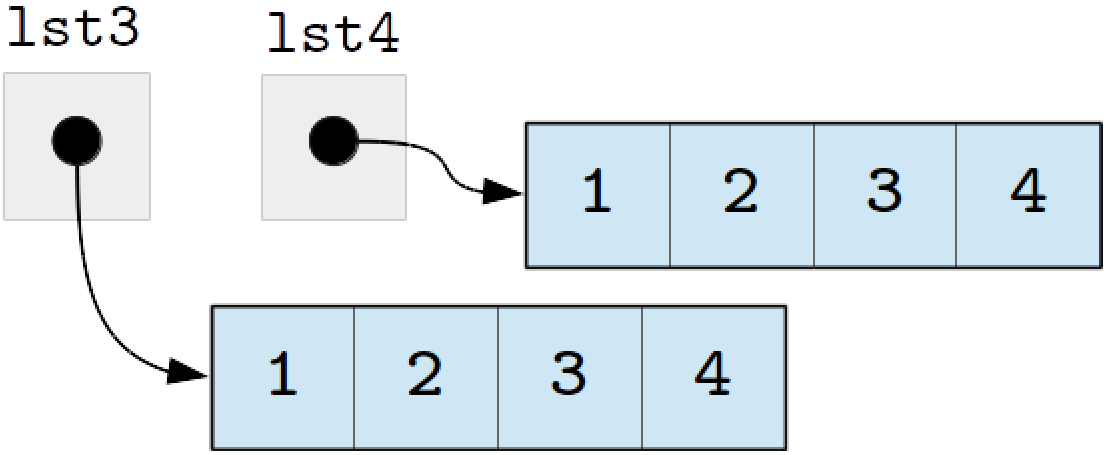
We can see that the change to lst2 also changed lst1:

```
>>> lst1
[100, 2, 3, 4]
```

This design makes comparing lists a bit tricky. Let's say we have the following statements:

```
>>> lst3 = [1, 2, 3, 4]
>>> lst4 = [1, 2, 3, 4]
```

After these assignments, lst3 and lst4 refer to separate lists, which just happen to contain the same values:



The equality operator (==) compares the contents of the lists, not whether the variables contain references to the same list in memory, and so, `lst3` and `lst4` are considered equal:

```
>>> lst3 == lst4
True
```

The `lst3` and `lst4` variables themselves do contain different values (since they point to different locations in memory, which just happen to contain lists with the same values). We can actually see these values (the location of the lists in memory) using the built-in `id` function:

```
>>> id(lst3)
140320075053376
>>> id(lst4)
140320075051840
>>> id(lst3) == id(lst4)
False
```

Note: if you run this code yourself, you will very likely not see the same `id` values shown above. The `id` value for `lst3`, however, should not be the same as the one for `lst4`.

In contrast, the `lst1` and `lst2` variables we created earlier do have the same `id` value, since they both refer to the same list in memory:

```
>>> id(lst1)
140320074973376
>>> id(lst2)
140320074973376
>>> id(lst1) == id(lst2)
True
```

In some languages, such as C, the equality operator (==) determines whether two variables refer to the same location in memory and not whether the values of the two variables are the same. This type of equality is known as *reference equality*. A test that determines whether two variables have the same value is known as *value equality*. Python provides both types of equality. The standard equality operator (==) performs value equality, while the `is` operator performs reference equality. Using the `is` operator, we could rewrite the above `id` examples as:

```
>>> lst3 is lst4
False
>>> lst1 is lst2
True
```

Finally, in some cases, we may actually want to assign a *copy* of a list to another variable. As noted earlier, we can just use the slicing operator, since it will always return a new list. More specifically, we specify the `[:]` slice (remember: when we omit the starting and ending indexes, they default to zero and the length of the list respectively). For example:

```
>>> lst5 = [1, 2, 3, 4]
>>> lst6 = lst5[:]
>>> lst5
[1, 2, 3, 4]
>>> lst6
[1, 2, 3, 4]
>>> lst5 == lst6
True
>>> id(lst5) == id(lst6)
False
```

(continues on next page)

(continued from previous page)

```
>>> lst5 is lst6
False
```

Notice that `lst5` and `lst6` end up pointing to *different* locations in memory (which happen to contain lists with the same values). So, unlike `lst1` and `lst2`, if we were to modify an element in `lst6`, `lst5` would not be affected:

```
>>> lst5
[1, 2, 3, 4]
>>> lst6
[1, 2, 3, 4]
>>> lst6[0] = 100
>>> lst6
[100, 2, 3, 4]
>>> lst5
[1, 2, 3, 4]
```

8.8 The heap

In the previous section, we explained that variables refer to lists and that more than one variable can refer to the same list. Where do the lists actually reside in memory? The answer is that lists (and other compound data structures that we'll discuss in subsequent chapters) reside in a part of memory known as the *heap*, which is managed behind the scenes for you by a part of the Python runtime system called the *garbage collector*.

The garbage collector allocates space in the heap when you create a new list or add values to an existing list and it is responsible for reclaiming space when it is no longer needed.

8.9 List parameters

In *Introduction to Functions*, we saw that Python functions use a type of parameter passing called *call-by-value* where the arguments to a function are evaluated *before* the function is called and the resulting *values* are used to initialize fresh copies of the formal parameters.

However, we just saw that, when it comes to lists, the value of a list variable is actually a *reference* to the list elsewhere in memory. This detail means that when we call a function with a list parameter, it is this reference (and not a copy of the list) that is used to initialize the formal parameters.

For example, suppose we want to write a function `scale` that takes a list `lst` and modifies it in-place so that each value in the list is scaled by a given `factor`. We could write the function like this:

```
def scale(lst, factor):
    '''
    Multiplies the values in a list in-place by a specified factor.

    Inputs:
        lst: a list
        factor: the factor to multiply by

    Returns: None
    '''
```

(continues on next page)

(continued from previous page)

```
    for i in range(len(lst)):
        lst[i] = lst[i] * factor
```

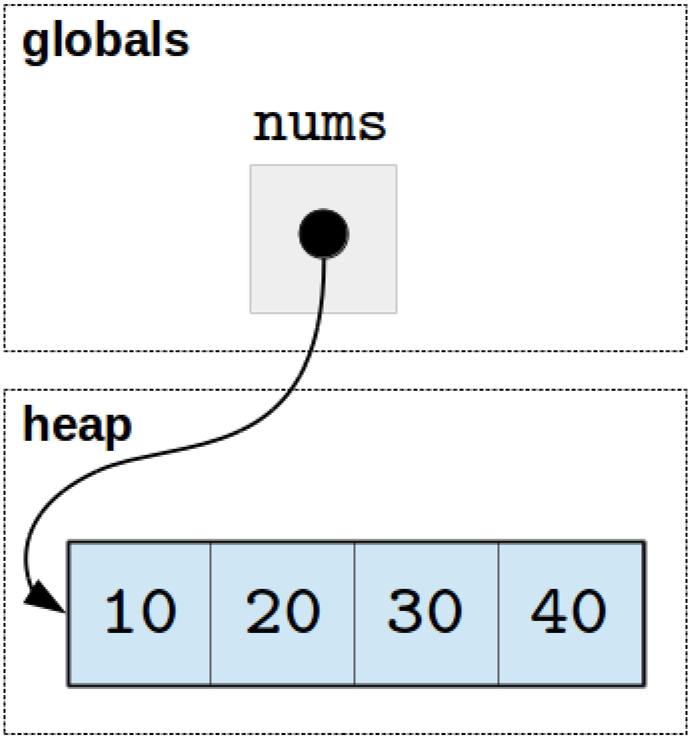
Earlier, we said that iterating over the range of indexes is considered "not Pythonic", but this is one case where it is justified, as we need to use the indexes of the list to update it (just iterating over the values would not allow us to do so). Later in the chapter we will see that there is a built-in function called `enumerate` that is generally preferred whenever we need to iterate over both the indexes and values of a list.

Let's try out this function:

```
>>> nums = [10, 20, 30, 40]
>>> scale(nums, 0.5)
>>> nums
[5.0, 10.0, 15.0, 20.0]
```

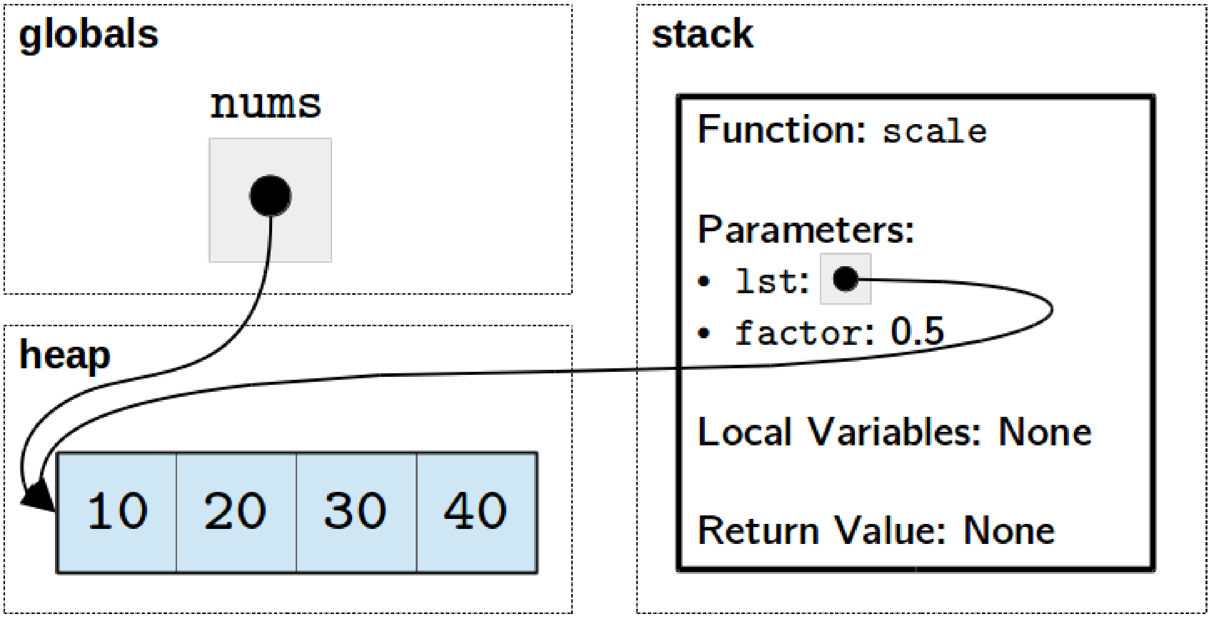
The `nums` list *was* modified by the function `scale`, which seems to go against our previous warning that functions cannot modify variables outside their scope. This seeming anomaly can best understood by looking at what happens in memory.

Before the call to `scale`, memory looks like this:



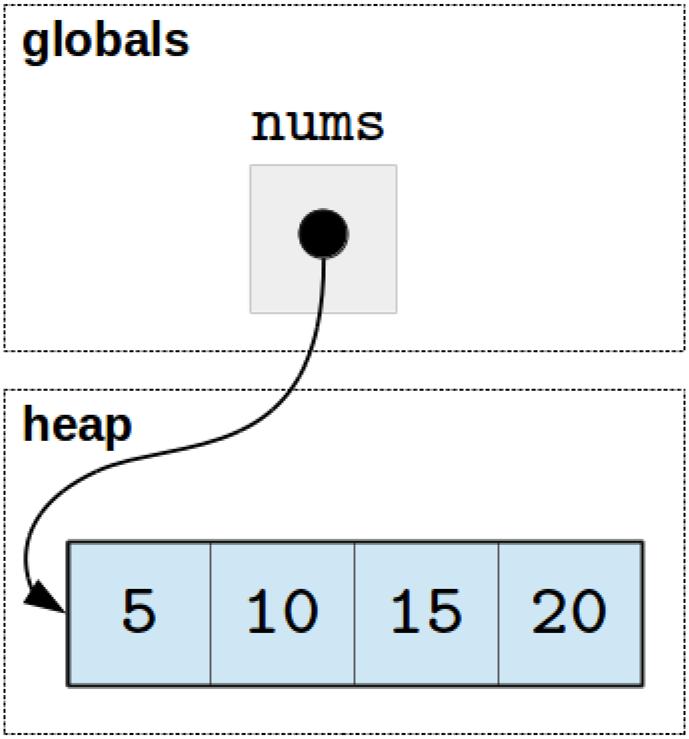
The variable `nums` resides in the space set aside for globals and the list resides in the heap.

Once the call to `scale` is made, memory will look like this:



Notice that the `lst` parameter in the stack frame for `scale` refers to the same list (in the heap) as the global `nums`. (If you skipped the *The function call stack* section of *Introduction to Functions*, you may want to revisit it now.)

Once the computation is finished and `scale` returns, the stack frame goes away and we are left with `nums` and the modified list.



As the client of a function, you need to be mindful that when you pass a list as a parameter to a function any changes made inside the function will affect the original list. If you write a function that, like `scale`, modifies a list in-place, make sure this behavior is properly documented so that anyone using the function is aware that the list that they are passing to the function will be modified. In general, it is considered poor style to modify a data structure as a side effect of a function unless that is the stated purpose of the function.

Next, since `scale` modifies the list in-place, it doesn't need to return anything. However, we could write a similar version of `scale` that, instead of modifying the list in place, creates and returns a new list, leaving the original list intact:

```python
def scale(lst, factor):
    '''
    Multiplies the values in a list by a specified factor.

    Inputs:
        lst: a list
        factor: the factor to multiply by

    Returns: a list of the same type as the input
    '''

    new_lst = []
    for x in lst:
        new_lst.append(x * factor)
    return new_lst
```

```
>>> nums = [10, 20, 30, 40]
>>> scaled_nums = scale(nums, 0.5)
>>> nums
[10, 20, 30, 40]
>>> scaled_nums
[5.0, 10.0, 15.0, 20.0]
```

This version of `scale` uses a `for`-loop and `append` to construct the new list. We can also write the function using list comprehension:

```python
def scale(lst, factor):
    '''
    Multiplies the values in a list by a specified factor.

    Inputs:
        lst: a list
        factor: the factor to multiply by

    Returns: a list of the same type as the input
    '''

    return [x * factor for x in lst]
```

In both versions, a list is constructed in the heap and a reference to the newly constructed list is returned.

This approach to passing and returning lists is efficient because the value handed around is just a memory address. We do not need to make a full copy of the list every time it is passed as an argument or returned as a result. Of course, this efficiency comes at a cost: it is easy to introduce hard-to-find bugs by unexpectedly modifying a list passed as a parameter.

When taking lists as parameters to functions, you should:

- Be mindful that any changes you make to the list *will* have an effect outside the function's scope;

- Think carefully about whether you want to modify the list in-place or return a new list with modified values; and finally,
- Record your choice in your function's docstring!

8.9.1 Aliases revisited

Earlier in this chapter, we mentioned that it is possible to introduce aliases–a second name for a data structure–using an assignment statement. For example, in this code:

```
lst1 = [1, 2, 3, 4]
lst2 = lst1
lst1 is lst2
```

lst1 and lst2 are aliases: they both refer to the same list, which contains the values 1, 2, etc.

Function parameters provide another way to introduce aliases.

For example, here's a function, remove_vals, that takes two lists and removes that values from lst2 that appear in lst1:

```
def remove_vals(lst1, lst2):
    """
    Remove the values that appear in lst2 from lst1.

    Inputs:
      lst1: a list of unique values
      lst2: a list of unique values

    Modifies: lst1
    """

    for val in lst2:
        if val in lst1:
            lst1.remove(val)
```

Notice that the docstring states that the inputs to this function are expected to be lists of unique values, that is, any given value appears at most once in a list. This restriction is included because the remove method removes the first occurrence of a value in a list, not all of the occurrences of a value in a list. If we wanted to allow a value to appear (and be removed) multiple times, we'd need to add an inner loop.

Let's first look at a call to this function that uses two different lists:

```
lstA = [1, 2, 3, 4]
lstB = [2, 4]
remove_vals(lstA, lstB)
```

During the call to remove_vals, memory will look like this:

After the call, it will look like this:

Notice that lstA has fewer elements: the two values that appeared in lstB have been removed as expected.

Now let's look at a call to this function that uses the *same* list for both arguments:

```
lst3 = [10, 20, 30, 40]
remove_vals(lst3, lst3)
print("lst3 after the call to remove_vals(lst3, lst3):", lst3)
```

```
lst3 after the call to remove_vals(lst3, lst3): [20, 40]
```

It is natural to expect the result of the function to be the empty list (`[]`), since the all the elements in the first parameter occur in the second parameter. But since `lst1` and `lst2` inside the function are aliases for the same list, the result will be unpredictable. We

8.10 Lists of lists

The lists we have seen so far contain simple values like integers or strings, but lists can, themselves, contain other lists. For example, the following list contains three elements, each of which is a four-integer list:

```
>>> m = [ [1,2,3,4], [5,6,7,8], [9,10,11,12] ]
```

Notice that, if we iterate over the list, we iterate over the three lists that constitute the elements of the outer list (not the twelve integers contained across the sublists).

```
>>> for row in m:
...     print(row)
...
[1, 2, 3, 4]
[5, 6, 7, 8]
[9, 10, 11, 12]
```

Lists-of-lists are often used to represent matrices. For example, list `m` could represent this matrix:

1	2	3	4
5	6	7	8
9	10	11	12

To access individual elements, we use the square brackets twice: once to specify the row, and again to specify the column:

```
>>> m[1][2]
7
>>> m[2][0]
9
```

Similarly, we can assign values to individual elements using the square bracket notation:

```
>>> m[1][1] = 0
>>> m
[[1, 2, 3, 4], [5, 0, 7, 8], [9, 10, 11, 12]]
>>> for row in m:
...     print(row)
...
[1, 2, 3, 4]
[5, 0, 7, 8]
[9, 10, 11, 12]
```

We can also nest lists even further. For example, we could use a "list-of-lists-of-lists" to represent a three-dimensional matrix:

```
>>> m3d = [ [ [1,2], [3,4] ], [ [5,6], [7,8] ], [ [9,10], [11,12] ] ]
>>> for submatrix in m3d:
...     for row in submatrix:
...         print(row)
...     print()
...
[1, 2]
[3, 4]

[5, 6]
[7, 8]

[9, 10]
[11, 12]
```

In the above example, m3d is a 3x2x2 matrix.

It is important, however, to understand that, even though we can use lists-of-lists to manipulate data in a matrix-like way, the data is not stored internally as a matrix: it is stored as a list with references to other lists, meaning that we can still access each individual list:

```
>>> m
[[1, 2, 3, 4], [5, 0, 7, 8], [9, 10, 11, 12]]
>>> m[2]
[9, 10, 11, 12]
```

Or modify these lists in a way that turns the data into something that is *not* a matrix:

```
>>> m[2] = "Foobar"
>>> m
[[1, 2, 3, 4], [5, 0, 7, 8], 'Foobar']
>>> for row in m:
...     print(row)
...
[1, 2, 3, 4]
[5, 0, 7, 8]
Foobar
```

This representation can cause problems when we initialize a list of lists. For example, we want to create a 5x5 matrix, with all elements initialized to 0. For example, we might be tempted to use the * operator to create a row of five zeroes with [0]*5 and then repeat that five times, like this:

```
>>> m = [ [0]*5 ] * 5
```

At first, it may seem like this approach worked:

```
>>> m
[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
>>> for row in m:
...     print(row)
...
```

(continues on next page)

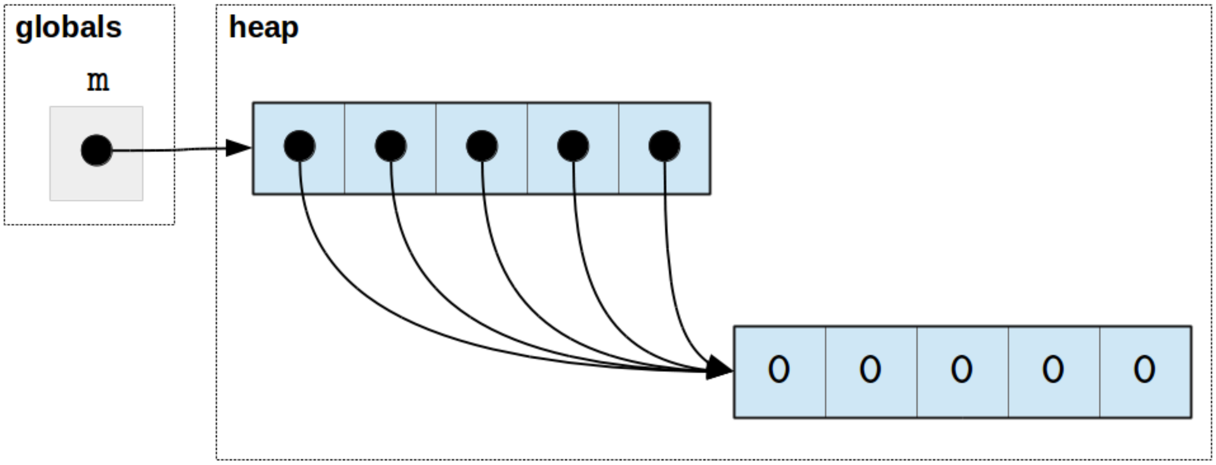
(continued from previous page)

```
[0, 0, 0, 0, 0]
[0, 0, 0, 0, 0]
[0, 0, 0, 0, 0]
[0, 0, 0, 0, 0]
[0, 0, 0, 0, 0]
```

However, something odd will happen if we try to modify an element of the matrix:

```
>>> m[2][3] = 10
>>> m
[[0, 0, 0, 10, 0], [0, 0, 0, 10, 0], [0, 0, 0, 10, 0], [0, 0, 0, 10, 0], [0, 0, 0, 10,␣
↪0]]
>>> for row in m:
...     print(row)
...
[0, 0, 0, 10, 0]
[0, 0, 0, 10, 0]
[0, 0, 0, 10, 0]
[0, 0, 0, 10, 0]
[0, 0, 0, 10, 0]
```

What happened here? When we created the list `m`, we actually initialized all five positions of `m` to point to the same list (`[0, 0, 0, 0, 0]`). Basically, the expression `[0]*5` produced one list, but then using `*` again didn't create five copies of this list: it created five copies of the *reference* to the list, which we can depict as follows:

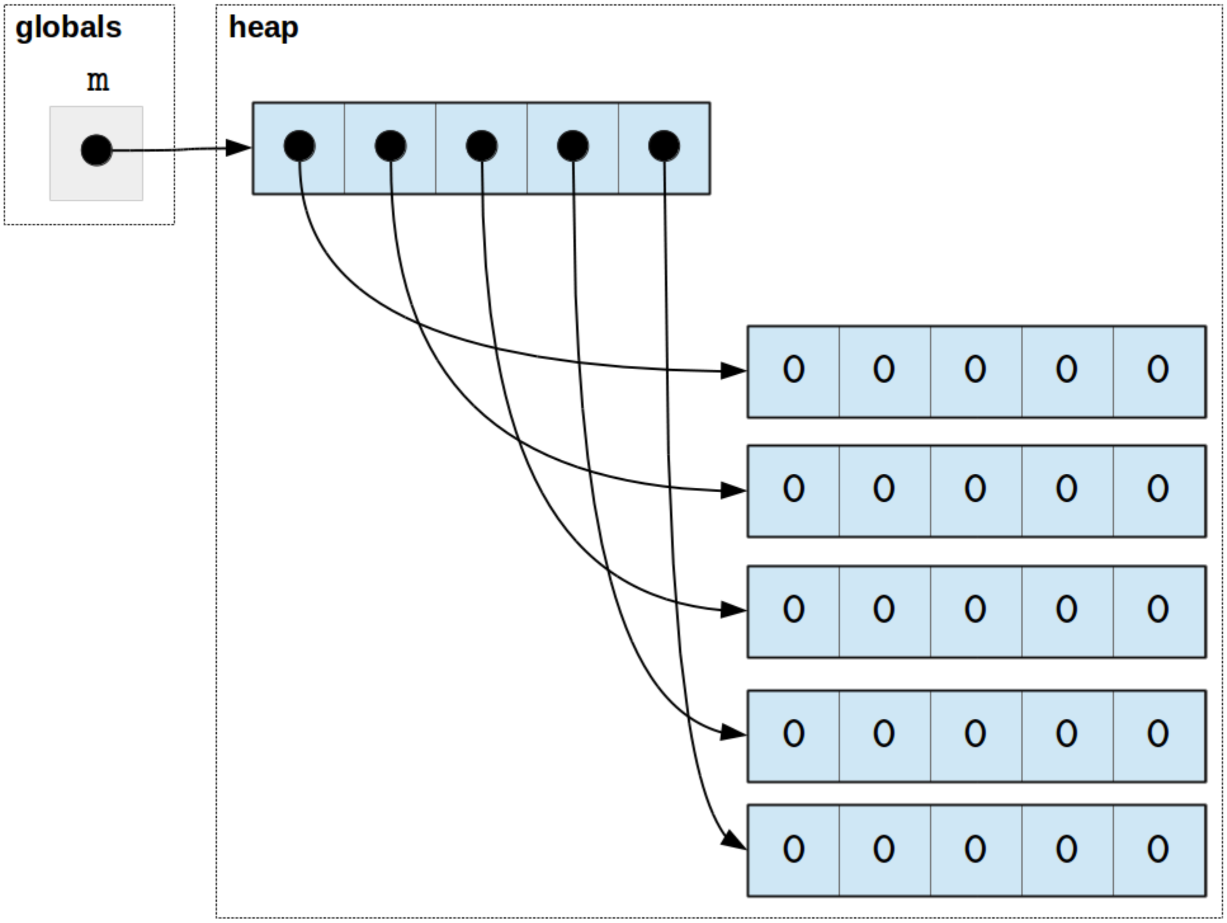


As a result, updating one "row" of the matrix actually updates them all! If this explanation of why the rows of `M` all refer to the same list is confusing, please re-read the *Variables revisited* section.

Instead of using multiplication on a nested list to initialize the matrix, we will use a loop:

```
>>> m = []
>>> for i in range(5):
...    m.append([0]*5)
...
```

Because the expression `[0]*5` is reevaluated in each iteration of the loop, a new list of five zeros is constructed and then appended to `m` on every iteration. The new version of `m` can be depicted as follows:



As we can see, modifying one row of m won't affect any other row now:

```
>>> m[2][3] = 10
>>> for row in m:
...     print(row)
...
[0, 0, 0, 0, 0]
[0, 0, 0, 0, 0]
[0, 0, 0, 10, 0]
[0, 0, 0, 0, 0]
[0, 0, 0, 0, 0]
```

In sum, while lists-of-lists can be used to manipulate matrix-like data, they must constructed used with care.

The way Python stores the lists internally means that many matrix operations (like matrix multiplication) can be quite inefficient. If you find yourself needing to manipulate matrix-like data extensively in your code (specially if you have to use higher-dimensional matrices), you may want to consider using a library like NumPy, which is specifically optimized for manipulating matrices and vectors, and which we will cover later in the book.

8.11 Shallow copy versus deep copy

Earlier in the section, we saw how to use slicing to make a copy of a list:

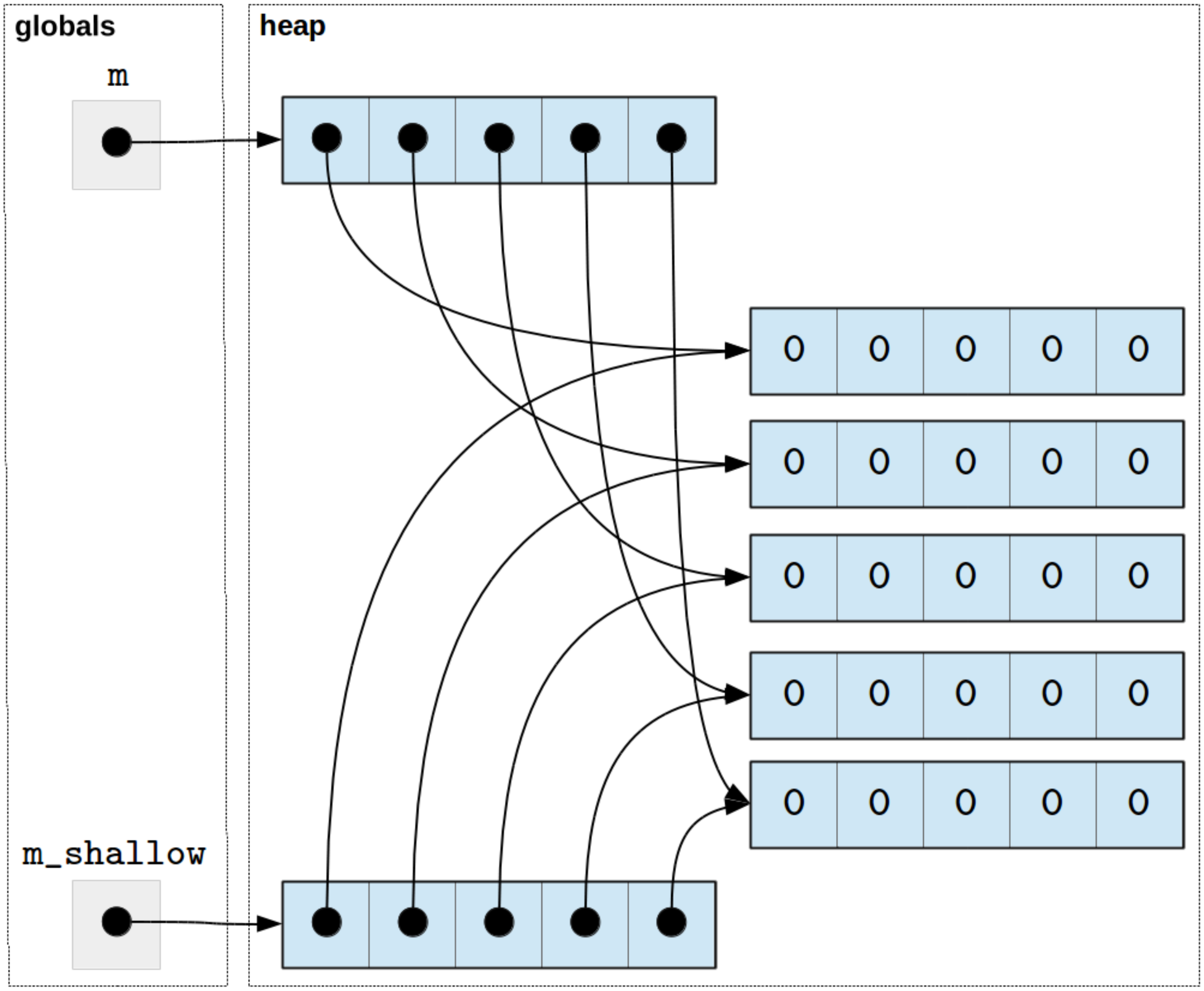
```
>>> lst = [1, 2, 3]
>>> lst_copy = lst[:]
>>> lst == lst_copy
True
>>> lst is lst_copy
False
```

This mechanism is very useful, but only does a shallow copy of the list. That is, it copies the values in the list. If the copied value is a reference to a list, the reference is copied, not the list itself.

This design is efficient, but can lead to unexpected sharing. For example, let's recreate our matrix of zeros and then make a shallow copy:

```
>>> m = []
>>> for i in range(5):
...    m.append([0]*5)
...
...
>>> m_shallow = m[:]
```

The result can be depicted as follows:



Notice that while the top-level level lists are different, they refer to the same sub-lists. We can verify this behavior using the `is` operator, which us whether two elements refer to the *same* location in the heap.

```
>>> m is m_shallow
False
>>> for i in range(len(m)):
...     print(i, m[i] is m_shallow[i])
...
0 True
1 True
2 True
3 True
4 True
```

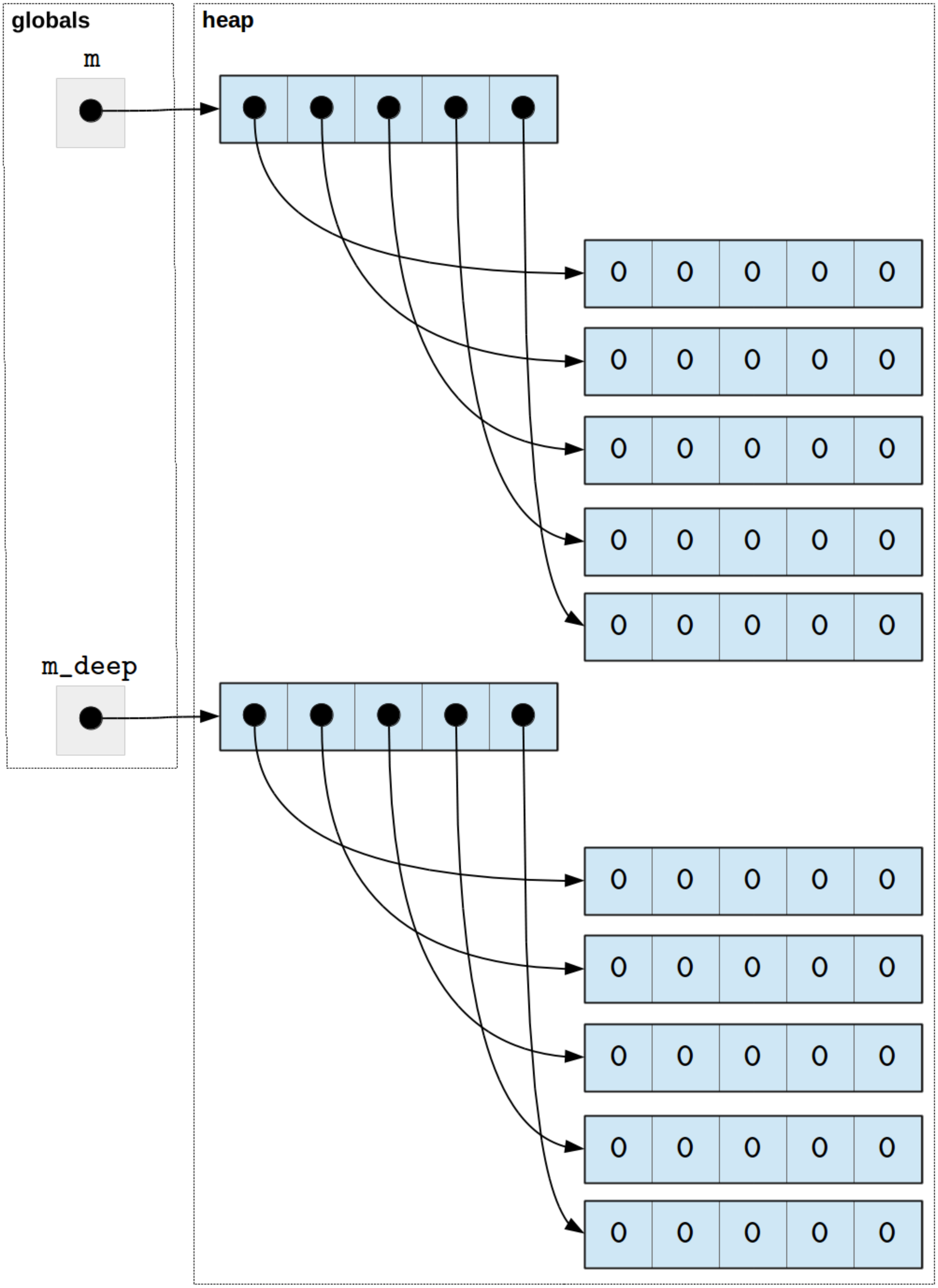
The `copy` library provides a function called `deepcopy` that can be used when your application needs to avoid sharing between the original and the copy.

```
>>> import copy
>>> m_deep = copy.deepcopy(m)
>>> for row in m_deep:
```

(continues on next page)

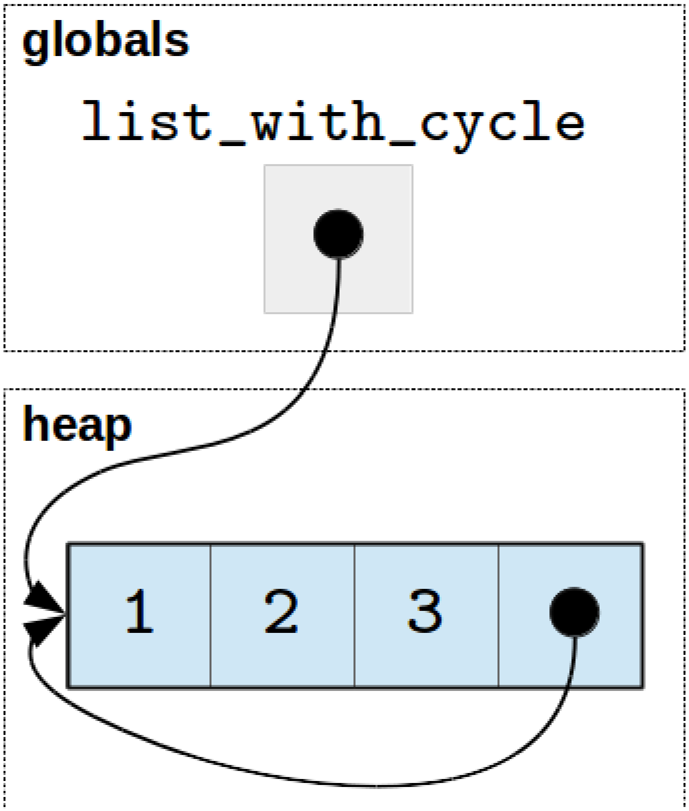
(continued from previous page)

```
...     print(row)
...
...
[0, 0, 0, 0, 0]
[0, 0, 0, 0, 0]
[0, 0, 0, 0, 0]
[0, 0, 0, 0, 0]
[0, 0, 0, 0, 0]
>>> m is m_deep
False
>>> for i in range(len(m)):
...     print(i, m[i] is m_deep[i])
...
0 False
1 False
2 False
3 False
4 False
```


globals
heap
m
m_deep
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0

The deep copy function needs to be used with care; it will get stuck in an infinite computation, if it is called on a data structure that loops back onto itself, such as:

```
>>> list_with_cycle = [1, 2, 3]
>>> list_with_cycle.append(list_with_cycle)
```



8.12 Tuples

Tuples, another data structure available in Python, are very similar to lists. We can use them to store sequences of values and can create tuple literals in the same way we create list literals, except using parentheses instead of square brackets:

```
>>> t = (100, 200, 300)
>>> t
(100, 200, 300)
```

We include a trailing comma in tuples of length one to to distinguish tuples from expressions that happen to be surrounded by parenthesis. Note, for example, the difference in the type of `t` versus the type of `e` in the following code:

```
>>> t = (1,)
>>> type(t)
<class 'tuple'>
>>> e = (1)
>>> type(e)
<class 'int'>
```

Many of the operations available on lists are also available on tuples:

```
>>> t = (100, 200, 300)
>>> len(t)
3
>>> t[2]
300
>>> max(t)
300
```

However, tuples differ from lists in one crucial way: they are *immutable*, which means that, once a tuple is created, its contents cannot be modified.

For example, we cannot assign a new value to an element of a tuple:

```
>>> t[1] = 42
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Nor can we append to a tuple, delete from a tuple, or carry out any operations that would modify a tuple in-place:

```
>>> t.append(42)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
>>> del t[1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object doesn't support item deletion
>>> t.sort()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'sort'
```

Despite this limitation, tuples are commonly used, especially to return multiple values from a function and to represent related data with different types, such as a person's name and salary (`("Alice", 5000)`). Here's an example of using a tuple to return multiple values:

```
import math
def compute_basic_stats(nums):
    '''

    Compute the mean and standard deviation of a list of numbers.

    Inputs:
      lst: list of numbers

    Returns: tuple with the mean and standard deviation.
    '''

    mean = sum(nums) / len(nums)
    sqdiffs = [(x - mean) ** 2 for x in nums]
    stdev = math.sqrt(sum(sqdiffs) / len(nums))
    return (mean, stdev)
```

We can call this function with a list of numbers, assign the result to a variable, and then extract the results by indexing:

```
quiz_scores = [9.9, 10.0, 7.6, 6.6, 12.0, 7.8, 11.0, 7.3, 7.4, 9.2]
stats = compute_basic_stats(quiz_scores)
print("Mean:", stats[0])
print("Standard deviation:", stats[1])
```

```
Mean: 8.88
Standard deviation: 1.7121915780659593
```

While this approach works, the resulting code is a bit cryptic. We could clean it up a bit by introducing named constants, such as `MEAN_SLOT` and `STDEV_SLOT`, for the different slots in the tuple and using them in place of the slot numbers.

```
MEAN_SLOT = 0
STDEV_SLOT = 1
stats = compute_basic_stats(quiz_scores)
print("Mean:", stats[MEAN_SLOT])
print("Standard deviation:", stats[STDEV_SLOT])
```

```
Mean: 8.88
Standard deviation: 1.7121915780659593
```

Better yet, we can give names to the individual values directly using Python's tuple *unpacking* mechanism, which allows programmers to specify multiple names, separated by commas, on the left side of an assignment statement and an expression that yields a tuple on the right side. Here's a sample call to `compute_basic_stats` that uses this mechanism to give names to the components of the return value:

```
>>> quiz_mean, quiz_stdev = compute_basic_stats(quiz_scores)
```

When unpacking a tuple, the number of names must match the number of values in the tuple. Python will throw a `ValueError` exception if the number of names does not match the length of the tuple:

```
>>> a, b = (1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: too many values to unpack (expected 2)
>>> a, b, c = (1, 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: not enough values to unpack (expected 3, got 2)
```

Tuple unpacking can also be used in `for` loops. For example, suppose we want to represent salary information for a small company. While we could represent this data using a list of lists, it is more common in Python to use a tuples to represent related data with different types:

```
>>> salaries = [ ("Alice", 5000), ("John", 4000), ("Carol", 4500) ]
```

We could iterate over the list in the usual way and use indices to extract values as needed in the body of the loop:

```
>>> for item in salaries:
...     name = item[0]
...     salary = item[1]
...     print(name, "has a salary of", salary)
...
```

(continues on next page)

(continued from previous page)

```
Alice has a salary of 5000
John has a salary of 4000
Carol has a salary of 4500
```

We can also use tuple unpacking in the for statement itself for this purpose:

```
>>> for name, salary in salaries:
...     print(name, "has a salary of", salary)
...
Alice has a salary of 5000
John has a salary of 4000
Carol has a salary of 4500
```

In the example above, the for loop will iterate over each element of salaries and, because we specified multiple variable names before the in, Python will assume that each element of salaries contains a tuple (or list) with two elements and will assign the first element to name and the second element to salary.

Note that this loop will fail if we have even a single tuple in the list with a different number of elements:

```
>>> salaries = [ ("Alice", 5000), ("John", "Smith", 4000), ("Carol", 4500) ]
>>> for name, salary in salaries:
...     print(name, "has a salary of", salary)
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: too many values to unpack (expected 2)
```

Another use of tuple unpacking in for loops arises when we need to use both the values and their indices when iterating over a list. You can now see that our earlier introduction of enumerate (shown again below) made use of tuple unpacking: enumerate yields a list of (index, value) tuples and the loop header supplies two names (i and price) to hold the components of the tuple as it is processed in the body of the loop.

```
prices = [10, 25, 5, 70, 10]

for i, price in enumerate(prices):
    tax = 0.10 * price
    total = price + tax
    print("The price (with tax) of element", i, "is", total)
```

```
The price (with tax) of element 0 is 11.0
The price (with tax) of element 1 is 27.5
The price (with tax) of element 2 is 5.5
The price (with tax) of element 3 is 77.0
The price (with tax) of element 4 is 11.0
```

8.13 Strings

Back in *Programming Basics* we introduced one of the basic data types in Python: strings. Strings allow us to store "text" or, more concretely, sequences (or "strings") of characters:

```
>>> s1 = "foobar"
>>> s2 = 'foobar'
```

As it turns out, string have some similarities to lists. Most notably, individual characters of a string can be accessed in the same way as elements in a list, using square brackets:

```
>>> s1[0]
'f'
```

We can also extract slices of a string:

```
>>> s1[1:4]
'oob'
```

Strings, like tuples, are immutable, so we cannot modify individual characters in a string:

```
>>> s1[0] = 'F'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

8.13.1 String formatting

So far, whenever we had to print a combination of strings and other values, we would just use the `print` function, which prints its parameters separated by spaces:

```
>>> a = 37
>>> print("Here is a number:", a)
Here is a number: 37
```

Strings also have a `format` method that allows us to combine values into a string in many different ways. When using this method, we include curly braces (`{}`) in any place where we want to embed another value. For example:

```
>>> s = "Here is a number: {}"
```

Then, we call the `format` method, and pass the value (or values) we want to embed in the string as a parameter (or parameters):

```
>>> s.format(a)
'Here is a number: 37'
>>> s.format(100)
'Here is a number: 100'
>>> s.format(3.1415)
'Here is a number: 3.1415'
>>> s.format("Not a number, but this still works")
'Here is a number: Not a number, but this still works'
```

Note how `format` returns a new string; it does not modify the format string itself. If we want to print the resulting string, we can pass the return value of `format` to `print`:

```
>>> print(s.format(a))
Here is a number: 37
```

In practice, it is common to call `format` directly on a string literal. This usage may look weird at first, but it is no different from calling `format` on a string variable:

```
>>> "Here is a number: {}".format(42)
'Here is a number: 42'
```

We can also embed multiple values into the string:

```
>>> a = 5
>>> b = 7
>>> print("{} times {} is {}".format(a, b, a*b))
5 times 7 is 35
```

And we can also control the way in which the embedded values are formatted. For example, when embedding a float, we can write `{:.Nf}`, substituting the number of decimal places we want to print for `N`:

```
>>> PI = 3.141592653589793
>>> print("The value of pi is {:.2f}".format(PI))
The value of pi is 3.14
>>> print("The value of pi is {:.4f}".format(PI))
The value of pi is 3.1416
```

There are many more ways to format strings with `format`. You can see the full description of how to specify formats for different types of values in the Python documentation for this method

8.13.2 Other operations on strings

Like lists, Python provides a large number of functions and methods to operate on strings. For example, we can use the `in` operator to determine whether a given string is contained within another string:

```
>>> s1 = "foobar"
>>> "oo" in s1
True
>>> "baz" in s1
False
```

If we want to find the location of a substring in a string, we can use the `find` method:

```
>>> s1 = "foobar"
>>> s1.find("oo")
1
>>> s1.find("baz")
-1
```

If the substring provided to `find` is found in the string, the method will return the index of the first occurrence of that substring. If the string does not contain the provided substring, then the method returns `-1`. Since `-1` is a legal index, it is important to check the return value of this method before you use it!

We also have methods that transform the strings in various ways:

```
>>> "FOOBAR".lower()
'foobar'
>>> "foobar".upper()
'FOOBAR'
>>> "hello world".capitalize()
'Hello world'
>>> "1.000.000".replace(".", ",")
'1,000,000'
>>> "   hello   world \t  \n".strip()
'hello   world'
```

While most of these functions are self-explanatory, the last one may not be: it constructs a new string that omits any leading or trailing white space (i.e., spaces, tabs, newlines, etc) from the original.

Two methods that will come in handy as we start to work with more data are the `split` and `join` methods. The `split` method takes a string that contains several values separated by a delimiter, such as a comma or a space, and returns a list of strings containing the values from the original string with the delimiters removed. For example:

```
>>> s = "foo,bar,baz"
>>> values = s.split(",")
>>> values
['foo', 'bar', 'baz']
```

The parameter to `split` is optional; if we omit it, `split` will assume that any whitespace characters (spaces, tabs, etc.) are the separator. For example:

```
>>> phrase = "The quick   brown  fox     jumps  over   the  lazy            dog"
>>> words = phrase.split()
>>> words
['The', 'quick', 'brown', 'fox', 'jumps', 'over', 'the', 'lazy', 'dog']
```

Note that `split` will consider any amount of whitespace to be a single delimiter. This behavior happens only when we call `split` without a parameter; when we specify an explicit parameter, multiple consecutive delimiters become "empty" values (strings of length zero):

```
>>> s = "foo,bar,,baz"
>>> values2 = s.split(",")
>>> values2
['foo', 'bar', '', 'baz']
```

A similar method, `join`, allows us to take a list of strings and concatenate them using a given delimiter. For example:

```
>>> values
['foo', 'bar', 'baz']
>>> sep = "|"
>>> sep.join(values)
'foo|bar|baz'
```

Note that, like the `format` method, we can call `join` directly on a string literal:

```
>>> "|".join(values)
'foo|bar|baz'
```

This section describes a small sample of what you can do with strings. To see a full list of methods, see the Python documentation or run `help(str)` from the Python interpreter.

CHAPTER

NINE

DICTIONARIES AND SETS

In the previous chapter, we saw that lists allow us to store multiple values in a single data structure, but they do so in a very specific way: the values are stored in a sequence. Python makes it very easy to iterate over that sequence, as well as access items at specific positions of the list. This design makes lists a great data structure for some use cases but not for others.

For example, suppose we were working with data on contributions to political campaigns, with the following information for each contribution:

- First name of contributor
- Last name of contributor
- ZIP Code where contributor lives
- Campaign that received the contribution
- Contribution amount

We could represent an individual contribution like this:

```
c = ["John", "Doe", "60637", "Kang for President 2016", 27.50]
```

And a list of contributions like this:

```
contributions = [
    ["John", "Doe", "60637", "Kang for President 2016", 27.50],
    ["Jane", "Doe", "60637", "Kodos for President 2016", 100.00],
    ["James", "Roe", "07974", "Kang for President 2016", 50.00]
    ]
```

Accessing the individual values inside this list requires using their positions in the list, which can lead to code that is difficult to read. For example, suppose we wanted to write a function that adds up all the contributions made to a given campaign. The function would look something like this:

```
def total_contributions_candidate(contributions, campaign):
    '''
    Compute total contributions to a candidate's campaign

    Inputs:
      contributions (list of lists): one list per contribution
      campaign (string): the name of the campaign

    Returns (float): total amount contributed
    '''
```

(continues on next page)

(continued from previous page)

```
    total = 0
    for contribution in contributions:
        if campaign == contribution[3]:
            total += contribution[4]
    return total
```

```
>>> total_contributions_candidate(contributions, "Kang for President 2016")
77.5
>>> total_contributions_candidate(contributions, "Kodos for President 2016")
100.0
```

Using `contribution[3]` to access the name of the campaign and `contribution[4]` to access the contribution amount makes the code hard to read. There are ways to make it a bit more readable, such as defining variables to store the position of each value (e.g., we could define `CAMPAIGN_INDEX = 3` and the access the campaign by writing `contribution[CAMPAIGN_INDEX]`), but this approach is error-prone. We could easily use the wrong value for the constant or the format of the data might change over time. In the latter case, we would need to check the fields we are using carefully to ensure that we are still accessing the correct positions.

For this application, it would be better to use a data structure called a *dictionary*. Like lists, dictionaries can be used to store multiple values, but do so by associating each value with a unique *key* rather than with a position in a sequence. This arrangement is similar to a physical dictionary: the words are the "keys" and the definitions are the "values".

We could store an individual contribution in a dictionary like this:

```
d = {"first_name": "John",
     "last_name": "Doe",
     "zip_code": "60637",
     "campaign": "Kang for President 2016",
     "amount": 27.50}
```

Notice that we have a sequence of entries separated by commas where each entry is a key-value pair separated by a colon. Instead of accessing values by their positions in the data structure, we access them by their keys. For example, if we wanted to access the ZIP Code of this contribution, we would use the key `"zip_code"`:

```
>>> d["zip_code"]
'60637'
```

Because values are accessed by key rather than by position, dictionaries are also referred to in other programming languages as "associative arrays" or "maps" (in the sense that they associate or map a key to a value).

In this example, all of the keys are strings. Although this design is fairly common, it is not required. In fact, many different Python types can be used for keys in a dictionary. For now, we'll restrict ourselves to types with values that cannot be changed (i.e. *immutable* types), which includes strings, integers, booleans, tuples of strings, etc. We'll come back to the question of acceptable types for dictionary keys briefly after we introduce Python classes.

While the types used for dictionary keys are restricted, the values can have any type and different keys can have values with different types. Notice, for example, that not all of the keys in our example have values with the same type: the value for `"amount"` is a float, while the rest are strings.

Our list of contributions would now look like this:

```
contributions = [
   {"first_name": "John",
    "last_name": "Doe",
```

(continues on next page)

(continued from previous page)

```
    "zip_code": "60637",
    "campaign": "Kang for President 2016",
    "amount": 27.50},
   {"first_name": "Jane",
    "last_name": "Doe",
    "zip_code": "60637",
    "campaign": "Kodos for President 2016",
    "amount": 100.00},
   {"first_name": "James",
    "last_name": "Roe",
    "zip_code": "07974",
    "campaign": "Kang for President 2016",
    "amount": 50.00}
]
```

And our function to compute the total contributions now looks like this:

```
def total_contributions_candidate(contributions, campaign):
    '''
    Compute total contributions to a candidate's campaign

    Inputs:
      contributions (list of dictionaries): one dictionary per
        contribution
      campaign (string): the name of the campaign

    Returns (float): total amount contributed
    '''

    total = 0
    for contribution in contributions:
        if campaign == contribution["campaign"]:
            total += contribution["amount"]
    return total
```

```
>>> total_contributions_candidate(contributions, "Kang for President 2016")
77.5
>>> total_contributions_candidate(contributions, "Kodos for President 2016")
100.0
```

The new function looks very similar to the original, but it is easier to read and maintain.

Dictionaries do more than simply provide convenient syntax for accessing values by key instead of position. As we'll see later in the chapter, dictionaries can also perform certain operations much more efficiently than lists and allow us to implement certain algorithms more elegantly.

It is very common to use a dictionary to represent a single "object" (in this case, a single contribution to a campaign, with each attribute of the contribution represented by a key in the dictionary), especially when exchanging data between different systems. In fact, later in the book, we will see a data format, JSON, that works well with Python dictionaries and lists. Later on, we will also see how *object-oriented programming* provides a third way of working with "objects" in our programs.

A possible pitfall: floats as keys

Before we move on talking about useful dictionary operations, we’d like to stop and consider a simple example an example that may surprise you:

```
>>> d = {0.3: "found"}
>>> d.get(0.1 + 0.1 + 0.1, "not found")
'not found'
```

Using your understanding of real numbers, you might except `get` to return `"found"`, the value associated with `0.3`. Using floats, however, `0.1 + 0.1 + 0.1` does not equal `0.3` and so, `get` does not find the key in the dictionary and returns the default value `"not_found"`.

So, while you technically use floats as keys for a dictionary, you should only do so when you are using values that can be represented exactly with a power of two or as the sum of powers of two.

9.1 Useful operations with dictionaries

Earlier, we saw that we can define a dictionary like this:

```
d = {"first_name": "John",
     "last_name": "Doe",
     "zip_code": "60637",
     "campaign": "Kang for President 2016",
     "amount": 27.50}
```

And access individual values by their keys like this:

```
>>> d["zip_code"]
'60637'
```

Note that when we attempt to *access* values in the dictionary, specifying a key that doesn’t exist in the dictionary produces an error:

```
>>> d["affiliation"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'affiliation'
```

In addition to the square-bracket notation, we can access a value using the `get` method, which method returns `None` if the dictionary does not contain the specified key.

```
>>> st = d.get("first_name")
>>> print(st)
John
>>> st = d.get("affiliation")
>>> print(st)
None
```

The `get` method also takes an optional second parameter to specify a default value that should be returned if the key doesn’t exist in the dictionary:

```
>>> d.get("affiliation", "Unknown")
'Unknown'
```

We can check if a dictionary contains a given key using the in operator:

```
>>> "first_name" in d
True
>>> "affiliation" in d
False
```

Dictionaries are also mutable: we can update the value associated with a key in a dictionary and add new key-value pairs. We assign a new value to a key like this:

```
>>> d["zip_code"] = "94305"
>>> d["zip_code"]
'94305'
```

and add a new key-value pair using the same syntax:

```
>>> d["affiliation"] = "Kodosican"
>>> d
{'first_name': 'John', 'last_name': 'Doe', 'zip_code': '94305', 'campaign': 'Kang for␣
↪President 2016', 'amount': 27.5, 'affiliation': 'Kodosican'}
```

We can also create a dictionary by starting from an empty (or partial) dictionary and assigning values:

```
d = {}
d["first_name"] = "John"
d["last_name"] = "Doe"
d["zip_code"] = "60637"
d["campaign"] = "Kang for President 2016"
d["amount"] = 27.50
d["affiliation"] = "Kodosican"
```

```
>>> d
{'first_name': 'John', 'last_name': 'Doe', 'zip_code': '60637', 'campaign': 'Kang for␣
↪President 2016', 'amount': 27.5, 'affiliation': 'Kodosican'}
```

We can remove an entry in the dictionary using the del operator:

```
>>> d
{'first_name': 'John', 'last_name': 'Doe', 'zip_code': '60637', 'campaign': 'Kang for␣
↪President 2016', 'amount': 27.5, 'affiliation': 'Kodosican'}
>>> del d["affiliation"]
>>> d
{'first_name': 'John', 'last_name': 'Doe', 'zip_code': '60637', 'campaign': 'Kang for␣
↪President 2016', 'amount': 27.5}
```

We can iterate over the keys in a dictionary using the dictionary's keys method:

```
>>> for key in d.keys():
...     print(key)
...
first_name
last_name
zip_code
campaign
amount
```

This operation is sufficiently common that Python provides a shorthand: iterating over the dictionary itself is equivalent to iterating over the keys. That is, when you iterate over a dictionary, the loop variable will iterate over the *keys* of the dictionary:

```
>>> for key in d:
...     print(key)
...
first_name
last_name
zip_code
campaign
amount
```

We can also iterate over the values using the dictionary's `values` method:

```
>>> for value in d.values():
...     print(value)
...
John
Doe
60637
Kang for President 2016
27.5
```

And, finally, we can iterate over both the keys and the values using the dictionary's `items` method:

```
>>> for key, value in d.items():
...     print(key, value)
...
first_name John
last_name Doe
zip_code 60637
campaign Kang for President 2016
amount 27.5
```

Notice that the keys and values are not printed in any particular order. Most notably, the keys are not shown in the order in which they were added to the dictionary, nor are they printed in alphabetical order. This behavior is another big difference between dictionaries and lists: lists store values in a specific order, and iterating over a list will always yield those values in that order. There is no such guarantee with dictionaries: if we iterate over the contents of a dictionary, we cannot assume any specific order and, in fact, that order can even change from one `for` loop to another!

A common pitfall: changing the set of keys in a dictionary as you iterate over it

There is an algorithm that allows us to compute the K most frequently seen items in a stream of data. We are not going to present the whole algorithm here. Instead, we will focus on a specific task within that algorithm that is used to exclude keys that have not been seen recently: take a dictionary that maps strings to positive integers, decrement all the values by one and remove any key-value pair where the resulting value is zero.

You might be tempted to write the following **incorrect** function to do this task:

```
def decr_and_remove(d):
    '''
    Given a dictionary that maps keys to positive integers,
    decrement the values and remove any key-value pair in which
```

(continues on next page)

(continued from previous page)

```
    the decremented value becomes zero

    Input:
       d (dictionary): maps keys to positive integers
    '''

    for key, value in d.items():
        d[key] = value - 1
        if d[key] == 0:
            del d[key]
```

This code is not valid because you *cannot* modify a data structure as you iterate over it! Here's one alternative solution:

```
def decr_and_remove(d):
    '''
    Given a dictionary that maps keys to positive integers,
    decrement the values and remove any key-value pair in which
    the decremented value becomes zero


    Input:
       d (dictionary): maps keys to positive integers
    '''

    keys_to_remove = []
    for key, value in d.items():
        d[key] = value - 1
        if d[key] == 0:
            keys_to_remove.append(key)

    for key in keys_to_remove:
        del d[key]
```

Here's another that computes and returns a new dictionary with the desired keys rather than removing the keys from the input dictionary.

```
def decr_and_keep_pos(d):
    '''
    Given a dictionary that maps keys to positive integers,
    subtract one from each value and include in the result only
    those key-value pairs that still have positive values after
    the decrement.

    Input:
      d (dictionary): keys to positive integers

    Returns (dictionary): keys to positive integers
    '''

    rv = {}
    for key, value in d.items():
        if value > 1:
```

(continues on next page)

(continued from previous page)

```
            rv[key] = value - 1
    return rv
```

9.2 Other ways to construct dictionaries

As we have seen, we can construct dictionaries using dictionary literals or updating an empty dictionary with new key-value pairs. We can also construct dictionaries use the `dict` function and dictionary comprehensions.

The `dict` function allows us to construct a *new* dictionary from a variety of different types of values including a list of key-value pairs or another dictionary. For example, here are three different ways to construct the same dictionary: the first uses a dictionary literal, the second uses a call to `dict` with a list of pairs, and the third uses a call to `dict` with a dictionary as the argument.

```
d = {"first_name": "John",
     "last_name": "Doe",
     "zip_code": "60637",
     "campaign": "Kang for President 2016",
     "amount": 27.50}

keys_and_data = [("first_name", "John"),
                 ("last_name", "Doe"),
                 ("zip_code", "60637"),
                 ("campaign", "Kang for President 2016"),
                 ("amount", 27.50)]

d_from_list = dict(keys_and_data)

d_from_dict = dict(d)
```

Dictionary comprehensions are similar to list comprehensions. In place of one transformation expression, dictionary comprehensions require two: one that yields a key and another that yields a value. For example, here's another way to construct our sample dictionary:

```
>>> d = {key: value for key, value in keys_and_data}
```

This example uses very simple expressions for the key and the value, but in general, you can use arbitrarily complex expressions for either. You can also specify a boolean expression to use as a filter. For example, we could rewrite the `decr_and_keep_pos` function from the previous section using a dictionary comprehension with a filter:

```
def decr_and_keep_pos(d):
    '''
    Given a dictionary that maps keys to positive integers,
    subtract one from each value and include in the result only
    those key-value pairs that still have positive values after the
    decrement.

    Input:
      d (dictionary): maps keys to positive integers

    Returns (dictionary): maps keys to positive integers
```

(continues on next page)

(continued from previous page)

```
    '''

    return {key: value - 1 for key, value in d.items() if value > 1}
```

A key-value pair from the input dictionary will be used in the construction of the result only if the original value is greater than one.

9.3 Accumulating values with dictionaries

Dictionaries are often used to accumulate a value based on a key. For example, earlier we computed the total number of contributions for a specific campaign. What if we wanted to compute the total contributions for each campaign? We could call our previous function over and over again, once per campaign, but there is a better approach. A dictionary that maps a campaign's name to the total contributions to that campaign is a perfect tool for this task. We start with an empty dictionary and add new entries as we encounter new campaigns:

```
def total_contributions_by_campaign(contributions):
    '''
    Compute total contributions by campaign

    Input:
      contributions (list of dictionaries): one dictionary per
        contribution

    Returns (dictionary): maps campaign names to floats
    '''

    rv = {}
    for contribution in contributions:
        campaign = contribution["campaign"]
        if campaign not in rv:
            rv[campaign] = 0
        rv[campaign] += contribution["amount"]
    return rv
```

Here's a sample run of this function:

```
>>> total_contributions_by_campaign(contributions)
{'Kang for President 2016': 77.5, 'Kodos for President 2016': 100.0}
```

Notice that the result is a dictionary.

This implementation uses the `not in` operator to identify campaigns that are being seen for the first time. An alternative is to use `get` with a default value of zero for previously unseen campaigns:

```
def total_contributions_by_campaign(contributions):
    '''
    Compute total contributions by campaign

    Input:
      contributions (list of dictionaries): one dictionary per
        contribution

```

(continues on next page)

(continued from previous page)

```
    Returns (dictionary): maps campaign names to floats
    '''

    rv = {}
    for contribution in contributions:
        campaign = contribution["campaign"]
        rv[campaign] = rv.get(campaign, 0) + contribution["amount"]
    return rv
```

Accumulating values in this way is very common.

9.4 Nested dictionaries

Dictionaries can be nested, meaning that the value associated with a key can itself be a dictionary. For example, we might have a dictionary that maps each candidate to a dictionary that maps a ZIP Code as a string to the total contributions to that candidate from that ZIP Code. Here's a version of this dictionary constructed using the contributions listed above:

```
contributions_by_cand_by_zipcode = {
    "Kodos for President 2016": {"60637": 100.00},
    "Kang for President 2016": {"07974": 50.00,
                                "60637": 27.50}
}
```

Note that we chose to represent ZIP Codes as strings, not integers, to make sure ZIP Codes like 07974 don't lose their leading zero and become 7974.

We can extract the the total amount of contributions from `60637` to the `Kang for President 2016` campaign using this expression:

```
>>> contributions_by_cand_by_zipcode["Kang for President 2016"]["60637"]
27.5
```

The first set of square brackets extracts the sub-dictionary associated with `"Kang for President 2016"` in `contributions_by_cand_by_zipcode`, while the second retrieves the value associated with `"60637"` from that sub-dictionary.

The code to compute this dictionary is not that much more complex than the code for accumulating total contributions by candidate.

```
def total_contributions_by_campaign_by_zip(contributions):
    '''
    Compute the total contributions from each ZIP Code for each
    campaign

    Input:
      contributions (list of dictionaries): one dictionary per
        contribution

    Returns (dictionary): maps a campaign name to a sub-dictionary
      that maps ZIP Codes (as strings) to floats
    '''
```

(continues on next page)

(continued from previous page)

```
    rv = {}
    for contribution in contributions:
        campaign = contribution["campaign"]
        zipcode = contribution["zip_code"]
        if campaign not in rv:
            rv[campaign] = {}
        rv[campaign][zipcode] = rv[campaign].get(zipcode, 0) + contribution["amount"]
    return rv
```

For each contribution, we first ensure that the campaign is initialized properly in the return value. We then use the get method to retrieve the total for contributions seen thus far from this zipcode for the campaign, which will be zero for the first contribution from this zipcode. Once we have that value , we can just add in the amount of the current contribution and update the dictionary. Here's a sample run of the function:

```
>>> total_contributions_by_campaign_by_zip(contributions)
{'Kang for President 2016': {'60637': 27.5, '07974': 50.0}, 'Kodos for President 2016': {
↪'60637': 100.0}}
```

Notice that, as expected, the result contains nested dictionaries.

9.5 Data structures and their complexity

The complexity of a computational solution to a problem, and how well that solution performs, depends not only on the algorithm we use, but also on our choice of data structures for that solution. Now that you know about two data structures, lists and dictionaries, it is important to understand the implications of using one or the other. This decision will not be just a matter of personal preference: your choice of data structure can have a considerable impact on how well (or how poorly) your code performs.

For example, our political contribution data only has postal ZIP Codes for the contributions, but not full addresses. One thing we might want to do is compute the total contributions in each state and, to do so, we need some way to map ZIP Codes to states. For the purposes of this example, let's assume we only care about three ZIP Codes: 60637 in Illinois, 07974 in New Jersey, and 94043 in California.

We could represent this data easily enough with a dictionary:

```
zip_codes_dict = {"60637": "IL",
                  "07974": "NJ",
                  "94305": "CA"}
```

But we could also store this mapping from ZIP Codes to states in a list of tuples, where each tuple represents a mapping from a ZIP Code to that ZIP Code's state:

```
zip_codes_list = [("60637", "IL"), ("07974", "NJ"), ("94305", "CA")]
```

Of course, the actual list of all the ZIP Codes would be much larger, with more than 40,000 entries.

Given a ZIP Code, obtaining that ZIP Code's state would involve iterating through the list until we find the tuple that contains the mapping from that ZIP Code to the corresponding state. We can write a simple function to perform this operation:

```
def get_state_from_zip(zip_code, zip_codes_list):
    '''
```

(continues on next page)

(continued from previous page)

```
    Find the state associated with a given ZIP Code

    Inputs:
      zip_code (string): a ZIP Code
      zip_codes_list (list): pairs of ZIP Codes and state abbreviations

    Returns (string): state abbreviation or None, if the ZIP Code
      does not appear in the list
    '''

    for zc, st in zip_codes_list:
        if zc == zip_code:
            return st

    return None
```

```
>>> get_state_from_zip("60637", zip_codes_list)
'IL'
>>> get_state_from_zip("90210", zip_codes_list)
```

Notice that, if there is no corresponding ZIP Code in our list, the function simply returns `None`.

So, at this point we have two solutions that, essentially, accomplish the same task:

```
>>> zip_codes_dict
{'60637': 'IL', '07974': 'NJ', '94305': 'CA'}
>>> zip_codes_list
[('60637', 'IL'), ('07974', 'NJ'), ('94305', 'CA')]
>>> zip_codes_dict.get("60637")
'IL'
>>> get_state_from_zip("60637", zip_codes_list)
'IL'
>>> zip_codes_dict.get("07974")
'NJ'
>>> get_state_from_zip("07974", zip_codes_list)
'NJ'
>>> zip_codes_dict.get("94305")
'CA'
>>> get_state_from_zip("94305", zip_codes_list)
'CA'
>>> zip_codes_dict.get("11111")
>>> get_state_from_zip("11111", zip_codes_list)
```

Finding a given ZIP Code's state in the list, however, takes an amount of time that is proportional to the size of the list. If we had 40,000 ZIP Codes, and the ZIP Code we're looking for is at the end of the list, we would have to traverse the entire list to find that value. While it's true that sometimes we'll search for a value that is towards the start of the list, the time to find a value will *on average* be proportional to the number of elements in the list. If we had 20,000,000 values in the list, the average time to find a value would be larger than if we have 40,000 values in the list.

Dictionaries, on the other hand, are implemented internally using a data structure called a *hash table* that is optimized to access key-value mappings very quickly. In fact, the amount of time it takes to access a key-value mapping is *not* proportional to the size of the input (in this case, the ZIP Codes). So, finding a value in a dictionary with 40,000 values will (roughly) take the same time as finding it in a dictionary with 20,000,000 values.

Big-O notation

The performance or *complexity* of an algorithm or, in this case, of an operation on a data structure, is typically specified using *big-O notation*.

While big-O notation has a formal definition, it is not essential to understand the concept. In a nutshell, if n is the size of the input to a problem (e.g., in this example, n would be the number of ZIP Codes), then saying that something runs in $O(n^2)$ means that its running time is *roughly proportional* to n^2 (i.e., as *n* gets bigger, the running time grows quadratically). On the other hand, if an algorithm runs in $O(\log n)$, its running time grows logarithmically.

Big-O notation helps us compare the performance of algorithms or individual operations on data structures. If one data structure can perform an operation in $O(n)$ and another data structure can perform that same operation in $O(n^2)$, we know that, on average, the first data structure performs that operation faster than the second data structure.

However, there is no "golden data structure" that beats every other data structure in every possible operation. In fact, when a data structure provides good performance in one operation, there will usually be a trade-off: other operations may be less efficient, or even not supported by the data structure. So, choosing the right data structure often involves asking yourself what operations you will be using the most, and whether each data structure can perform those operations efficiently.

In this case, we are looking at just one operation: finding a value in a list versus finding a value (through its key) in a dictionary. If n is the number of values, this operation can be done on lists in $O(n)$ (the running time is *linearly* proportional to n), while this operation can be done on dictionaries in $O(1)$, meaning that the running time is *not* proportional to n (this is often referred to as "constant time"; strictly speaking, this operation is done in something called "amortized constant time", but you don't need to concern yourself with this distinction). So, in this case, a dictionary would be a better choice.

However, you should not jump to the conclusion that we should now use dictionaries for everything. If we were evaluating a different operation, it could turn out that lists would be a better choice. We explore this proposition in more detail below.

We can observe this difference in performance empirically by running our code with the full ZIP Code database. We will show you the result of doing this experiment but, if you would like to follow along, you can use the following files provided in our *example code*: the `data_structures/dictionaries/zipcodes.py` module, as well as the ZIP Code database, `data_structures/dictionaries/us_postal_codes.csv`. You will also need to use IPython, not the regular Python interpreter (make sure to run it in the same directory that contains the `zipcodes.py` and `us_postal_codes.csv` files.

From the IPython interpreter, run the following:

```
In [1]: run zipcodes.py
```

This command will run the code in `zipcodes.py`, which reads the ZIP Code database and loads it into both a list (`zip_codes_list`) and a dictionary (`zip_codes_dict`). It will also define the `get_state_from_zip` function we wrote earlier, as well as a function called `gen_random_zip` that generates a random ZIP Code. We can see that they work as expected:

```
In [2]: get_state_from_zip("60637", zip_codes_list)
Out[2]: 'IL'

In [3]: get_state_from_zip("90210", zip_codes_list)
Out[3]: 'CA'

In [4]: zip_codes_dict.get("60637")
Out[4]: 'IL'

```

(continues on next page)

(continued from previous page)

```
In [5]: zip_codes_dict.get("90210")
Out[5]: 'CA'

In [6]: gen_random_zip()
Out[6]: '48344'

In [7]: gen_random_zip()
Out[7]: '22219'
```

Note: You will very likely get different ZIP Codes when you call `gen_random_zip`.

When you make these individual calls, they may seem to be returning nearly instantaneously. Unfortunately, testing code performance informally in the interpreter obscures what will happen when you write a program that involves calling a function thousands or even millions of times.

IPython includes a handy `%timeit` command that will allow us to see how a piece of code performs when run many times. As it turns out, the dictionary version is nearly 200 times faster than the list-based version!

```
In [8]: %timeit zip_codes_dict.get(gen_random_zip())
100000 loops, best of 3: 4.9 µs per loop

In [9]: %timeit get_state_from_zip(gen_random_zip(), zip_codes_list)
1000 loops, best of 3: 955 µs per loop
```

Note: You will likely get different running times, but the order of magnitude between the two times should be roughly the same.

Things get more interesting if we actually look at how these running times change depending on the size of the dataset. Our `zipcodes.py` file also defines a `small_zip_codes_list` and a `small_zip_codes_dict` with just 500 ZIP Codes, and a `medium_zip_codes_list` and a `medium_zip_codes_dict` with just 2000 ZIP Codes. If we use the `%timeit` command to time our list-based and dictionary-based implementations with these datasets, we get the following times:

	n = 5000	n = 20000	n = 43624
Lists	0.142 ms	0.493 ms	1.01 ms
Dictionaries	0.00477 ms	0.00478 ms	0.0048 ms

So, not only are dictionaries faster than lists in an absolute sense, their performance is independent of the size of the input. On the other hand, notice that as the dataset grows the list-based implementation takes more and more time.

This result occurs because, as we saw earlier, finding the mapping from a ZIP Code to a state in a list takes $O(n)$ time, while finding a mapping in a dictionary takes $O(1)$ time. If n is the number of ZIP Codes in our dataset, then the running time of a $O(n)$ solution will increase (roughly) linearly as our dataset gets bigger, while the running time of a $O(1)$ operation will remain (roughly) constant.

So, based on all this, it sounds like we should just use dictionaries everywhere instead of lists, right? Not so fast! As we mentioned earlier, there is no such thing as a “golden data structure” that performs every possible operation in an optimal fashion. There is, however, “the right data structure for this job” and, if “this job” is finding a mapping between a key and a value, then dictionaries are definitely the right data structure.

There are other operations, however, for which dictionaries are not so great. Most notably, as we saw earlier, dictionaries are *unordered* data structures, which means that iterating over a dictionary is not guaranteed to yield the keys in any particular order (and that order can even change from one `for` loop to another). On top of that, even if we don’t care about the order in which the values are processed, a `for` loop over a dictionary will usually be slower than a `for` loop over a list containing the same values. So, if the order of the data matters, or if we are going to process it in sequence often, then using a list will probably be a better choice.

Note

Why did we use random ZIP Codes instead of just testing the code with a fixed ZIP Code? If we used a fixed ZIP Code, the results could be skewed depending on where that ZIP Code is located in the list. For example, if we used 00210 (the first ZIP Code in the list), our list implementation would always find that ZIP Code very quickly (because it doesn't have to iterate through the rest of the list), making it look like the list version is just as good as the dictionary version:

```
In [10]: %timeit zip_codes_dict.get("00210")
The slowest run took 21.25 times longer than the fastest. This␣
↪could mean that an intermediate result is being cached
10000000 loops, best of 3: 63.3 ns per loop

In [11]: %timeit get_state_from_zip("00210", zip_codes_list)
The slowest run took 17.80 times longer than the fastest. This␣
↪could mean that an intermediate result is being cached
10000000 loops, best of 3: 127 ns per loop
```

In this case, the dictionary version is only two times faster than the list version. Notice that we also get a message about "an intermediate result being cached". This phrase refers to the fact that computers are usually smart about detecting when a location in memory is being accessed frequently, and making sure it is "cached" in higher-speed memory in the computer's processor (instead of having to go to the computer's main memory, which is slower). Testing our implementation with the same inputs over and over again could show performance gains that are due to this caching mechanism, not the efficiency of the data structure itself.

9.6 Sets

Dictionaries are great for associating values to unique keys, but sometimes we may simply want to have a collection of unique keys that we can access as efficiently as a dictionary, but without associating a value with each key. To do this, we can simply use a *set* data structure, which allows us to store an unordered collection of distinct elements.

Typical set operations include:

- *membership* ($e \in S$): which tests whether a given element, e, appears in the set S,
- *subset* ($S \subseteq T$): which tests whether every element in S also appears as an element in T.
- *union* ($S \cup T$): which yields a new set that contains the elements e where $e \in S$ and/or $e \in T$,
- *intersection* ($S \cap T$): which yields a new set that contains the elements e where $e \in S$ *and* $e \in T$, and
- *difference* ($S - T$): which yields a new set that contains the elements e where $e \in S$ and $e \notin T$.

Sets are sufficiently useful that Python provides a built-in `set` data structure.

We can define a set literal by surrounding the initial elements of the set with curly braces or using the built-in `set` function with a sequence (list, string, etc.) as the parameter:

```
>>> zipcodes = {"60637", "07974"}
>>> zipcodes
{'07974', '60637'}
>>> zipcodes = set(["60637", "07974"])
```

(continues on next page)

(continued from previous page)

```
>>> zipcodes
{'07974', '60637'}
>>> vowels = set("aeiou")
>>> vowels
{'u', 'o', 'e', 'a', 'i'}
```

To construct an empty set, we use the `set` function:

```
>>> empty_set = set()
```

We cannot use curly braces to create an empty set, because Python interprets `{}` as an empty dictionary.

We can determine the number of elements in a set using the `len` function:

```
>>> len(empty_set)
0
>>> len(zipcodes)
2
>>> len(vowels)
5
```

and we can test for set membership using the `in` operator:

```
>>> "14850" in zipcodes
False
>>> "60637" in zipcodes
True
```

We can determine whether one set is a subset of another set using the `issubset` method. Specifically, a call of the form `S.issubset(T)` tests whether the set `S` is a subset of the set `T`. For example:

```
>>> {"07974"}.issubset(zipcodes)
True
>>> {"60637", "14850"}.issubset(zipcodes)
False
```

Sets are mutable: we can add and remove elements. We can add elements to a set using the `add` method:

```
>>> zipcodes.add("14850")
>>> zipcodes.add("60637")
>>> zipcodes
{'14850', '07974', '60637'}
```

The second example illustrates an important point about sets: adding an element that is already a member of the set has no effect.

We can take elements out of a set using either the `remove` method or the `discard` method. The `remove` method removes a value from a set, if it is a member, and throws a `KeyError` exception if it is not. The `discard` method, in contrast, removes the value if it is a member of the set and does nothing if the value is not a member of the set.

```
>>> zipcodes.remove("60637")
>>> zipcodes.remove("60615")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

(continues on next page)

(continued from previous page)

```
KeyError: '60615'
>>> zipcodes
{'14850', '07974'}
>>> zipcodes.discard("14850")
>>> zipcodes.discard("60615")
>>> zipcodes
{'07974'}
```

Using the add method and len function, we can easily write a function to determine how many different ZIP Codes had at least one contribution.

```
def compute_num_zipcodes(contributions):
    '''
    Compute the number of zipcodes with at least one contribution

    Inputs:
      contributions (list of dictionaries): one dictionary per
        contribution

    Returns (int): the number of zipcodes with at least one
      contribution.
    '''

    zipcodes_seen = set()
    for contribution in contributions:
        zipcodes_seen.add(contribution["zip_code"])
    return len(zipcodes_seen)
```

```
>>> compute_num_zipcodes(contributions)
2
```

Alternatively, we can use set comprehensions, which are like dictionary comprehensions with only keys, to do the same computation:

```
def compute_num_zipcodes(contributions):
    '''
    Compute the number of zipcodes with at least one contribution

    Inputs:
      contributions (list of dictionaries): one dictionary per
        contribution

    Returns (int): the number of zipcodes with at least one
      contribution
    '''

    zipcodes_seen = {c["zip_code"] for c in contributions}
    return len(zipcodes_seen)
```

```
>>> compute_num_zipcodes(contributions)
2
```

Sets come with a variety of useful operations. We can compute a *new* set that is the union of two sets using the |

operator or the `union` method, the intersection of two sets using the `&` operator or the `intersection` method, and the difference of two sets using the `-` operator or the `difference` method:

```
>>> zipcodes = {"60637", "07974"}
>>> zipcodes | {"60637", "60615"}
{'07974', '60615', '60637'}
>>> zipcodes.union({"60637", "60615"})
{'07974', '60615', '60637'}
>>> zipcodes & {"60637", "60615"}
{'60637'}
>>> zipcodes.intersection({"60637", "60615"})
{'60637'}
>>> zipcodes - {"60637", "60615"}
{'07974'}
>>> zipcodes.difference({"60637", "60615"})
{'07974'}
```

Finally, since sets are collections, we can iterate over the elements in a set:

```
>>> for zc in zipcodes:
...     print(zc)
...
07974
60637
```

As with dictionaries, sets are unordered and so the elements will be printed in an arbitrary order.

CHAPTER

TEN

IMPLEMENTING A DATA STRUCTURE: STACKS AND QUEUES

Up to this point, we have been using data types (integers, strings, etc.) and data structures (lists and dictionaries) that are part of Python. While there is plenty we can do with them, sometimes we will need to implement our own custom data structures. In this chapter, we will present two new data structures: stacks and queues. While Python actually already provides implementations of these data structures, we will implement our own versions from scratch, so we can understand the process involved in implementing a data structure. As we'll see, a key aspect of this process is thinking about the *interface* of our data structure, or how other programmers (not us) will interact with that data structure.

10.1 Interfaces and APIs

When we interact with lists and dictionaries in Python, we are blissfully unaware of all their internal details. For example, take this simple dictionary code:

```
>>> d = {}
>>> d["A"] = 4.0
>>> "A" in d
True
```

In this code, we can think about the dictionary in fairly high-level terms: it contains a collection of mappings between keys and values, and the code above adds one such mapping and also queries whether a given mapping exists or not.

However, dictionaries internally use a data structure called a *hash table*. Adding even a single mapping to the hash table involves a series of potentially complicated steps, with a number of scenarios the implementer must handle: what if the key already exists in the hash table? What if it doesn't? What if the hash table doesn't have enough memory allocated to add more keys?

Fortunately, all those details are abstracted away for us, because Python allows us to interact with the dictionary using a simple *interface*. When we add a new mapping in the dictionary, we don't have to think in terms of how we must manipulate the internal hash table; we just do this:

```
d["A"] = 4.0
```

As users of a data structure, we only need to concern ourselves with the interface of that data structure. A term you'll hear often is *API*, which stands for *Application Programming Interface*. An API is a more general term, that can refer to the interface of a data structure (e.g., "the dictionary API"), but also to any collection of functions or protocols that allow us to interact with a software library or system, while abstracting away all of its internal details.

For example, when we import Python's `random` module, we are gaining access to a software library for working with random numbers.

```
>>> import random
>>> random.randint(1,100)
28
```

The `random` module has a well-specified API, and we use that API to interact with `random`. In the above piece of code, we don't need to know the exact mechanism by which `random.randint` generates a random number: we just need to know that we can call that function with a lower bound and an upper bound, and it will return a random number within those bounds.

APIs can also refer to interfaces that allow us to access certain services via the Internet, and specifically in a way that we can easily incorporate in a program. For example, imagine we wanted to find out the top Twitter users who use hashtag `#programming` (or any given hashtag). We could go to the Twitter website, enter the hashtag into their search tag, and then copy-paste the results to a file. However, this is not something a program can do easily.

Instead, Twitter provides an API for programs to access their data more conveniently. This API is protocol-focused: it specifies the messages that a program can send to their web server over the Internet to obtain certain data. For example, we can write a program that will send a message to the Twitter API asking for all the tweets that meet certain conditions (like including a given hashtag), and it will return a JSON-encoded list of those tweets, which we can then easily process from a program. A popular API of this form will often spawn competing libraries that are easier to use by abstracting away the details of the actual protocol.

So, when writing a complex program or system, you rarely have to implement the entire program completely from scratch, and it is common to use multiple APIs as building blocks. Even when writing simple Python programs, we often `import` existing libraries like `random` (so we don't have to implement our own functions for generating random numbers).

In this chapter, we will implement a stack data structure and a queue data structure, and we will focus not just on their internal implementation, but also on what interface (or API) we will provide for those data structures. In particular, we will provide an interface that is function-based: we expect anyone who uses the data structure to interact with it through a series of functions that we will write. Later in the book, we will introduce *classes* and *objects*, which will allow us to provide a different type of interface: an object-oriented interface.

10.2 Stacks

A stack is essentially a list of values, but with a more constrained set of operations available on that list. Most notably, we can only access and manipulate one element in the data structure: the last element that was inserted into the stack (which, in stacks, is known as the *top* of the stack).

A common way of representing a stack is (non-surprisingly) as values stacked on top of each other. For example:

```
  TOP

|  10 |
|  56 |
| 105 |
|  42 |
|  5  |
-------

 BOTTOM
```

In the above stack, we can only interact with the value `10`: we can see its value, we can remove it from the stack, or we can stack another value on top of it. However, we cannot interact with the other values in the stack without first removing the values above them (unless, for example, we remove value `10`, and then `56` becomes the top of the stack).

More specifically, these are all the operations we can do with a stack:

- Creating an empty stack
- *Pushing* a value into the stack. If a stack was empty, then the stack will contain just the pushed value. If the stack already had values, we "stack" the new value on the top of stack.
- *Popping* a value from the stack. This operation takes the value at the top of the stack and removes it (we also get to see what that value is)
- *Peeking* at the value at the top of the stack. This operation tells us what the value at the top of the stack is but unlike popping does not remove it from the stack.
- Checking whether a stack is empty.

It may seem like we're unnecessarily limiting ourselves: why would we want to use a data structure like this when we already have lists, which allow us to do so much more? One reason is that there are many algorithms that can implemented in a straightforward way if we use a stack data structure (or, rather, if we write the algorithm in terms of stack operations like pushing and popping). For example, here are some applications that use stacks:

- Undo algorithms in a word processor.
- Expression evaluation in compilers and interpreters.
- Functional call evaluation, which is the origin of the *function call stack*.

Later on, we'll see that another reason why limiting ourselves to a constrained set of operations can be beneficial from and interface/API standpoint.

However, let's first see how we can implement a stack. Since a stack is a sequence of values, we can use a list to implement one. For example, the stack we showed above could be represented as:

```
s = [ 5, 42, 105, 56, 10 ]
```

Since we only interact with the top of the stack, we will store the elements in the stack from bottom to top (i.e., the top of the stack will be the last element in the list). This way, we can "push" onto the stack using the list's `append` method:

```
>>> s
[5, 42, 105, 56, 10]
>>> s.append(37)
>>> s
[5, 42, 105, 56, 10, 37]
```

And popping can be done simply by calling the appropriately named `pop` method:

```
>>> s
[5, 42, 105, 56, 10, 37]
>>> s.pop()
37
>>> s
[5, 42, 105, 56, 10]
```

The complexity of list operations

Why do we choose to store the stack in a list from bottom to top, instead of from top to bottom? If we stored it from top to bottom, the stack would look like this:

```
s2 = [ 10, 56, 105, 42, 5 ]
```

And we could push using the `insert` method:

```
>>> s2
[10, 56, 105, 42, 5]
>>> s2.insert(0, 37)
>>> s2
[37, 10, 56, 105, 42, 5]
```

And pop using the pop method (but specifying that we want to pop the 0th element of the list):

```
>>> s2
[37, 10, 56, 105, 42, 5]
>>> s2.pop(0)
37
>>> s2
[10, 56, 105, 42, 5]
```

The results are seemingly the same as the bottom-to-top implementation, but the performance of the operations is not. In Python, appending to the end of the list and removing the last element in the list can be done very efficiently in $O(1)$ time.

Inserting and removing at the start of a Python list, on the other hand, requires $O(n)$ time, because all the elements in the list need to be shifted forward or backward by one position. This analysis is not universally true of all list data structures in all languages and, in fact, there are list implementations (including a deque data structure included in Python's collections module) that allows insertion and removal of the first element in $O(1)$ time. These types of differences give us another reason why, depending on how we intend to use a data structure, we may need to be aware of the complexity of its operations.

The complexity of many data structure operations in Python can be found here: https://wiki.python.org/moin/TimeComplexity

---

At this point, we know how to use a list in a stack-like manner, but nothing is stopping us from performing non-stack operations on that list. For example, we can easily modify non-top entries in the stack, which is not allowed in a stack:

```
>>> s[2] = 37
>>> s
[5, 42, 37, 56, 10]
```

This observation leads us to the other reason why limiting ourselves to a constrained set of operations can be beneficial: to ensure that the data structure is manipulated only in acceptable ways.

For example, earlier we discussed how dictionaries are implemented internally as hash tables, but we interact with dictionaries only through a limited set of operations provided by that data structure's interface or API. This API abstracts away the internal details of how dictionaries work, making our lives easier as programmers, but it also prevents us from wreaking havoc on the data structure by directly manipulating the internal hash table. Another way of seeing this is that dictionaries only allow us to interact with them on their own terms: the only way of manipulating them is through their API (which the programmers who implemented dictionaries have control over).

So, when implementing our stack data structure, we want to make sure that the programmer who uses that stack *cannot* manipulate it in non-stack ways (like modifying anything other than the top element of the stack). For now, we will accomplish this by defining an API as a collection of functions. Later on in the book, we will see how to define this same API but using an object-oriented approach.

So, we will need functions for the operations we described earlier. Creating an empty stack is simple enough:

```
def stack_create():
    return []
```

```
>>> s = stack_create()
```

Notice how the function returns a list but, conceptually, it returns a *stack*. The programmer who uses our stack doesn't need to know that s is actually a list (even though in Python this is easy enough to find out). In fact, a very important principle of API design is that it should be possible for the data type developer to change the internal implementation without affecting the users of our data type (who should be treating the value returned by create as an *opaque type*).

For example, let's go back to dictionaries again. When we create a new dictionary (e.g., d = {}) we are blissfully unaware that variable d actually refers to a hash table and, not just that, if the Python developers decided to switch to a different internal representation, we would keep using dictionaries the same way.

Pushing, popping, peeking, and checking emptiness are similarly straightforward:

```
def stack_push(stack, value):
    stack.append(value)

def stack_pop(stack):
    return stack.pop()

def stack_top(stack):
    return stack[-1]

def stack_is_empty(stack):
    return len(stack) == 0
```

Finally, let's also add a function that creates a string representation of the stack:

```
def stack_to_string(stack):
    s  = " TOP OF THE STACK\n"
    s += "--------------------\n"

    for v in reversed(stack):
        s += str(v).center(20) + "\n"

    s += "--------------------\n"
    s += "BOTTOM OF THE STACK\n"
    return s
```

Now, we can work with stacks using only these functions:

```
>>> s = stack_create()
>>> stack_push(s, 10)
>>> stack_push(s, 27)
>>> stack_push(s, 5)
>>> stack_push(s, 9)
>>> stack_push(s, 7)
>>> print(stack_to_string(s))
 TOP OF THE STACK
--------------------
         7
         9
         5
         27
         10
```

(continues on next page)

(continued from previous page)

```
-------------------
BOTTOM OF THE STACK
```

```
>>> stack_pop(s)
7
>>> print(stack_to_string(s))
 TOP OF THE STACK
-------------------
         9
         5
         27
         10
-------------------
BOTTOM OF THE STACK
```

Of course, because `stack_create` returns a list (which we're conceptually manipulating as a stack), nothing stops a user from doing this:

```
>>> s
[10, 27, 5, 9]
>>> s[2] = 37
>>> s
[10, 27, 37, 9]
```

This lack of control over access to the underlying representation is one of the limitations of using a function-based API in Python: it is still relatively easy to perform forbidden operations on the data structure. When we discuss object orientation later in the book, we will see that the object oriented paradigm allows us to define APIs in a way that more strongly protects the internal data of a data structure.

10.3 Queues

Like stacks, queues represent a sequence of values, but with a different set of allowed operations on that sequence of values. In particular, the start of that sequence of values is known as the *back* of the queue, and the end of the sequence is known as the *front* of the queue. For example:

```
       ------------------
BACK   10  56  105  42  5   FRONT
       ------------------
```

A queue is like the typical queue you encounter when waiting in line for something: elements can only enter the queue through the back of the queue and can only exit the queue through the front. More specifically, the only allowed operations with a queue are:

- Creating an empty queue
- *Enqueueing* a value at the back of the queue
- *Dequeueing* a value at the front of the queue
- *Peeking* at the value at the front of the queue
- Checking the size of queue, including whether it is empty

Assuming we only use these operations, no element in the queue can "skip the line". If an element enters the queue at the back of the queue, it will not leave the queue until enough elements (in front of it) are dequeued, so it reached the front of the queue itself.

Like stacks, we will use a list to implement our queue. With stacks, using the end of the list as the top of the stack was the best choice from a performance standpoint. For queues, we have to choose whether to use the start of the list as the back and the end of the list as the front, or vice versa.

As it turns out, we can choose either option. If we make the end of the list be the front of the queue, Enqueueing is $O(n)$ (because we're inserting at the start of the list) but dequeuing is $O(1)$ (because we're deleting at the end). If we make the end of the list be the back of the queue, the complexities are swapped (enqueueing is $O(1)$ and dequeuing is $O(n)$).

Assuming that the number of enqueue/dequeue operations is roughly the same (because in many applications, what gets enqueued eventually gets dequeued), there's really no difference between using the start or end of the list as the front (and vice versa for the back of the queue). These are the kind of issues that data structure implementors have to deal with, but which the programmers that use the data structures *should not care about* (except that, sometimes, the documentation will tell you the complexity of certain operations).

So, our queue operations can be implemented with the following functions:

```python
def queue_create():
    return []

def queue_is_empty(queue):
    return len(queue) == 0

def queue_length(queue):
    return len(queue)

def queue_enqueue(queue, value):
    queue.append(value)

def queue_dequeue(queue):
    return queue.pop(0)

def queue_front(queue):
    return queue[0]

def queue_to_string(queue):
    s  = "FRONT OF THE QUEUE\n"
    s += "------------------\n"

    for v in queue:
        s += str(v).center(19) + "\n"

    s += "------------------\n"
    s += "BACK OF THE QUEUE \n"
    return s
```

And we can now work with queues by calling those functions:

```python
>>> q1 = queue_create()
>>> queue_enqueue(q1, 10)
>>> queue_enqueue(q1, 27)
>>> queue_enqueue(q1, 5)
```

(continues on next page)

(continued from previous page)

```
>>> queue_enqueue(q1, 9)
>>> queue_enqueue(q1, 7)
>>> print(queue_to_string(q1))
FRONT OF THE QUEUE
------------------
         10
         27
         5
         9
         7
------------------
BACK OF THE QUEUE
```

```
>>> queue_front(q1)
10
>>> print(queue_to_string(q1))
FRONT OF THE QUEUE
------------------
         10
         27
         5
         9
         7
------------------
BACK OF THE QUEUE

>>> queue_dequeue(q1)
10
>>> print(queue_to_string(q1))
FRONT OF THE QUEUE
------------------
         27
         5
         9
         7
------------------
BACK OF THE QUEUE
```

CHAPTER

ELEVEN

CLASSES AND OBJECTS

In the previous chapter, we saw that we can define new data types by creating an API of functions to operate on that data type (like we did for stacks and queues). However, this approach has two main limitations.

First of all, when implementing stacks and queues, we decided to use a list to store information about our stack or queue. For example, the `stack_create` function returns an empty list:

```
def stack_create():
    return []
```

```
>>> s = stack_create()
```

Once we have that variable `s`, we can pass it to the other stack functions (`stack_pop`, `stack_push`, etc.) and essentially manipulate it as a stack, even though it actually contains a list.

However, nothing prevents us from manipulating that list in non-stack ways: we can still use any of Python's list operations directly on it. In other words, our data type has information that should be *private* to the data type, and that users of that data type cannot manipulate. We want those users to access only those operations and data that we choose to make *public*. Using a function-based API, we can't fully enforce this separation between the public and private parts of a data type.

Second, our stack/queue implementations allowed the stack/queue to be represented with a single variable (a list), which meant we only had to pass that as a parameter to the other functions. However, not all data types can be implemented like that. For example, I may want to define a data type with multiple *attributes*, such as a "student" data type that includes a first name, a last name, a student ID number, and other attributes of the student.

One way to address this second issue is to simply encapsulate those multiple attributes in a list or dictionary. In fact, back in *Dictionaries and Sets* we saw that this was a common use case of dictionaries. We could store information about a single student in a dictionary like this:

```
student = {
           "first_name": "Johnny",
           "last_name": "Coder",
           "student_id": "313370"
          }
```

In fact, this dictionary could, in turn, include other dictionaries that represent other data types, like courses or majors:

```
student = {
           "first_name": "Johnny",
           "last_name": "Coder",
           "student_id": "313370",
           "courses": [{"code": "CMSC 12100",
                        "name": "Computer Science with Applications I"},
```

(continues on next page)

(continued from previous page)

```
                       {"code": "CMSC 12200",
                        "name": "Computer Science with Applications II"}],
           "major": {"name": "Economics",
                     "short_name": "Econ"}
          }
```

This way, if we defined a function that operates on a student, all we need to do is pass a single student dictionary to that function (instead of one parameter for each of the attributes of the student). However, we can still manipulate that dictionary in ways that may not be allowed (like changing the student's ID number), and we could potentially use attribute names that make no sense (e.g., if we made a typo like writing `firstname` instead of `first_name`).

The *object-oriented paradigm* (or OO for short) addresses many of these issues by providing a mechanism to define new data types that encapsulate some attributes as well as operations that are allowed on the data type. OO also allows for a cleaner separation between the *public* attributes and operations that any programmer can access, and the *private* ones that only the data type implementor should be allowed to modify.

11.1 Defining a new class

In OO, a *class* is the definition of a new type, while *objects* are specific instances of that data type. One of way to think about this distinction is that a class is like the blueprint for a car, while objects are the actual individual cars. There is only one blueprint, and that blueprint specifies certain characteristics of the car that may vary from car to car, like its color or engine type. There can be multiple cars, all built according to the same blueprint, but with each individual car having different values for a given attribute (e.g., one car could be red and another blue).

So, let's say we want to define a new data type to represent a location on Earth. A location will have only two attributes: latitude and longitude. So, while we could implement this data type by using a list of two numbers, and creating a function-based API like we did for stacks and queues, we will see that defining a location class will have a number of advantages.

Below is the full specification of a new `Location` data type using an OO approach. For now, don't worry about the individual details of this code. We will deconstruct it piece by piece soon.

```
import math

class Location(object):
    def __init__(self, latitude, longitude):
        self.latitude = latitude
        self.longitude = longitude

    def to_string(self):
        if (self.latitude < 0.0):
            lat = "S"
        else:
            lat = "N"

        if (self.longitude < 0.0):
            lon = "W"
        else:
            lon = "E"

        return "({:.3f} {}, {:.3f} {})".format(abs(self.latitude),
                                               lat,
```

(continues on next page)

(continued from previous page)

```
                                               abs(self.longitude),
                                               lon)

    def distance_to(self, other):
        diffLatitude = math.radians(other.latitude - self.latitude)
        diffLongitude = math.radians(other.longitude - self.longitude)

        a = math.sin(diffLatitude/2) * math.sin(diffLatitude/2) + \
            math.cos(math.radians(self.latitude)) * \
            math.cos(math.radians(other.latitude)) * \
            math.sin(diffLongitude/2) * math.sin(diffLongitude/2)
        d = 2 * math.asin(math.sqrt(a))

        return 6371000.0 * d

    def __repr__(self):
        return self.to_string()
```

The above code defines a new `Location` data type that we can now use in Python to manipulate geographic locations (specified with a latitude and longitude):

```
>>> chicago_loc = Location( 41.8337329, -87.7321555 )
>>> newyork_loc = Location( 40.7056308, -73.9780035 )
>>> chicago_loc.to_string()
'(41.834 N, 87.732 W)'
>>> newyork_loc.to_string()
'(40.706 N, 73.978 W)'
>>> chicago_loc.distance_to(newyork_loc)
1155076.7723381526
```

Notice how we can create two separate locations, `chicago_loc` and `newyork_loc` that both conform to the `Location` "blueprint" (they both have their own values for `latitude` and `longitude`). We can also perform a useful operation on these locations: computing the distance from one location to another (in this case, the distance from Chicago to New York is 1,155,076 meters, or 1,115.07 kilometers).

Let's now take a closer look at how we have defined the `Location` class.

Our definition starts with this line:

```
class Location(object):
```

This line starts the definition of a `Location` class: notice how everything else is indented below the `class` statement.

Inside the class, we have what look like functions: `__init__`, `to_string`, `distance_to`, and `__repr__`. These definitions are referred to as *methods*. The first one is special: it is known as the *constructor*, and is always called `__init__`. The constructor is used to *initialize* a new object.

Remember that a class is like a blueprint, and an object is a specific *instance* of that class. So, when we do this:

```
chicago_loc = Location( 41.8337329, -87.7321555 )
```

What we're doing is taking the `Location` "blueprint" and constructing a location with latitude equal to 41.8337329 and longitude equal to -87.7321555. Internally, Python builds this instance by calling the `__init__` method, which is in charge of *initializing* the coordinates. However, notice how `Location`'s constructor actually has three parameters:

```
def __init__(self, latitude, longitude):
```

But, when we create the new `Location` object we only provide two parameters. When implementing a class, all the methods will have a `self` parameter, which represents the object the method is operating on. In th case of `__init__`, `self` is the new `Location` object we are constructing.

The `__init__` method is commonly used to initialize the object's *attributes*. In this case, `Location` has two attributes (latitude and longitude), so we need to set their values. We do so with the `latitude` and `longitude` parameters provided to the `__init__` method:

```
def __init__(self, latitude, longitude):
    self.latitude = latitude
    self.longitude = longitude
```

In the above code, `self.latitude` can be read as "`latitude of self`", i.e., the `latitude` attribute of the `self` object (which is the object we are constructing), while `latitude` (by itself, without `self.`) is just a parameter to the constructor.

So, to recap, when we evaluate this statement:

```
>>> chicago_loc = Location( 41.8337329, -87.7321555 )
```

A new `Location` object is created with two attributes, `latitude` and `longitude`, with their values initialized to `41.8337329` and `-87.7321555`, respectively. The object is then stored in the `chicago_loc` variable. We can actually access the `latitude` and `longitude` attributes by using the dot operator:

```
>>> chicago_loc.latitude
41.8337329
>>> chicago_loc.longitude
-87.7321555
```

Once we have created a new object, we can invoke other methods defined in the `Location` class. For example, the class includes the following method:

```
def to_string(self):
    if (self.latitude < 0.0):
        lat = "S"
    else:
        lat = "N"

    if (self.longitude < 0.0):
        lon = "W"
    else:
        lon = "E"

    return "({:.3f} {}, {:.3f} {})".format(abs(self.latitude),
                                            lat,
                                            abs(self.longitude),
                                            lon)
```

The purpose of this method is to produce a string representation of the coordinates. While we could simply print the object's latitude and longitude like this:

```
>>> print(chicago_loc.latitude, chicago_loc.longitude)
41.8337329 -87.7321555
```

This method produces a more compact representation that also indicates the hemisphere of each coordinate:

```
>>> chicago_loc.to_string()
'(41.834 N, 87.732 W)'
```

This method may look a bit odd because, if we were writing a *function* to do this, it would look like this:

```
def to_string(latitude, longitude):
    if (latitude < 0.0):
        lat = "S"
    else:
        lat = "N"

    if (longitude < 0.0):
        lon = "W"
    else:
        lon = "E"

    return "({:.3f} {}, {:.3f} {})".format(abs(latitude),
                                           lat,
                                           abs(longitude),
                                           lon)
```

And we would call it like this:

```
>>> to_string(41.8337329, -87.7321555)
'(41.834 N, 87.732 W)'
```

But, instead, when using objects, we call it like this:

```
>>> chicago_loc.to_string()
'(41.834 N, 87.732 W)'
```

Remember that the `self` parameter contains the object a method is operating on. Furthermore, this parameter is *implied* whenever we call a method. So, a call like `chicago_loc.to_string()` internally becomes something like this:

```
Location.to_string(chicago_loc)
```

Then, inside the method, an expression like `self.latitude` actually evaluates to “attribute `latitude` of object `chicago_loc`”. In fact, notice how calling `to_string` on different objects produces different results:

```
>>> regenstein_loc = Location( 41.79218,   -87.599934)
>>> ryerson_loc =    Location( 41.7902836, -87.5991959)
>>> regenstein_loc.to_string()
'(41.792 N, 87.600 W)'
>>> ryerson_loc.to_string()
'(41.790 N, 87.599 W)'
```

This is because the first call to `to_string` is done using the `regenstein_loc` object (which has its `latitude` and `longitude` attributes set to 41.79218 and -87.599934) and the second call is done using the `ryerson_loc` object (which has its `latitude` and `longitude` attributes set to different values: 41.7902836 and -87.5991959).

The `Location` class also has a `__repr__` method that simply returns the value return by `to_string`. The `__repr__` method is a special method that is called any time Python needs a string representation of a given object. So, once we implement a `__repr__` method, we can actually get its string representation simply by printing the object or by writing an object variable name in the interpreter:

```
>>> print(chicago_loc)
(41.834 N, 87.732 W)
>>> chicago_loc
(41.834 N, 87.732 W)
```

This mechanism is more convenient than having to remember to call a to_string method every time we want a string representation. While we would usually just place the code from to_string directly inside __repr__, we separated the implementation for clarity, and to highlight how we can also call to_string directly.

Finally, the Location class has a distance_to method that takes one parameter besides self. In fact, the purpose of this method is to compute the distance between two locations, where one location is the Location object that distance is called on, and the second location is passed via a parameter. Like before, one way of thinking about this is that this call:

```
chicago_loc.distance_to(newyork_loc)
```

Internally becomes something like this:

```
Location.distance_to(chicago_loc, newyork_loc)
```

This distance_to method computes the distance between two geographic coordinates This method computes the distance between two coordinates (i.e., two longitude/latitude pairs) using the haversine formula:

$$2r \arcsin\left(\sqrt{\sin^2\left(\frac{x_{lat}-y_{lat}}{2}\right)+\cos(y_{lat})\cos(x_{lat})\sin^2\left(\frac{x_{long}-y_{long}}{2}\right)}\right)$$

where:

- x_{lat}, x_{long} is the latitude and longitude *in radians* of point x.
- y_{lat}, y_{long} is the latitude and longitude *in radians* of point y.
- r is the radius of the sphere (in this case, Earth's average radius: 6,371km)

Encapsulating the use of this complex formula is yet another example of how functions (or, in this case, methods) can abstract away complex details about an operation. Instead of having to worry about the complicated formula above, we are able to manipulate locations with relatively simple operations:

```
>>> chicago_loc = Location( 41.8337329, -87.7321555 )
>>> newyork_loc = Location( 40.7056308, -73.9780035 )
>>> chicago_loc.distance_to(newyork_loc)
1155076.7723381526
```

11.2 Composition

So far, we have defined a new data type, Location, that encapsulates two float values. However, attributes are not limited to the built-in types in Python: we can define classes with attributes that are, themselves, objects. This mechanism is called *composition* and it is very powerful when defining new data types.

To further illustrate this point, we are going to model information about Chicago's Divvy bike share system. More specifically, we are going to use the data from the 2013 Divvy Data Challenge), which includes (anonymized) data on all the Divvy bicycle trips taken in 2013, as well as information about each Divvy station where bikes can be picked up and dropped off.

Ultimately, we will answer the following question: *What is the total duration and total distance of all the Divvy trips taken in 2013?* As we'll see, going through the effort of modeling the Divvy data using classes will make it relatively easy to answer not just this question, but also a number of other questions we did not set out to answer originally.

First of all, we need to model the Divvy stations. The dataset provides the following information about each station:

- `id`: A unique integer identifier.
- `name`: A string with descriptive name (e.g., `"State St & Harrison St"`)
- `latitude`: A float with the latitude of the station.
- `longitude`: A float with the longitude of the station.
- `dpcapacity`: The number of total docks at each station as of 2/7/2014
- `landmark`: An undocumented attribute (the Divvy Challenge never explained what this field meant)
- `online date`: A string with the date the station went live in the system (e.g., `"6/28/2013"`)

Given this information, we could define a new `DivvyStation` class as follows, with attributes corresponding to the information above (note: we use `stationID` instead of `id` because `id` is a reserved keyword in Python):

```
class DivvyStation(object):

    def __init__(self, stationID, name, latitude, longitude,
                 dpcapacity, landmark, online_date):

        self.stationID = stationID
        self.name = name
        self.latitude = latitude
        self.longitude = longitude
        self.dpcapacity = dpcapacity
        self.landmark = landmark
        self.online_date = online_date
```

However, we've already defined a `Location` class that encapsulates information about a geographical location *and* which implements some potentially useful operations on locations, such as the distance between two locations. Since we want to ultimately compute the total distance of all the Divvy trips, it seems like we may want to leverage that existing `Location` class.

So, instead of having a `latitude` and `longitude` attribute, we can define our class to have a `location` attribute that contains a `Location` object.

```
class DivvyStation(object):

    def __init__(self, stationID, name, latitude, longitude,
                 dpcapacity, landmark, online_date):

        self.stationID = stationID
        self.name = name
        self.location = Location(latitude, longitude)
        self.dpcapacity = dpcapacity
        self.landmark = landmark
        self.online_date = online_date
```

If we follow this approach, adding a method to compute the distance from one station to another becomes very simple:

```
class DivvyStation(object):

    def __init__(self, stationID, name, latitude, longitude,
                 dpcapacity, landmark, online_date):

        self.stationID = stationID
        self.name = name
        self.location = Location(latitude, longitude)
        self.dpcapacity = dpcapacity
        self.landmark = landmark
        self.online_date = online_date

    def distance_to(self, other_station):
        d = self.location.distance_to(other_station.location)
        return d
```

For example:

```
>>> s25 = DivvyStation(25, "Michigan Ave & Pearson St", 41.89766, -87.62351, 23, 34, "6/
↪28/2013")
>>> s44 = DivvyStation(44, "State St & Randolph St", 41.8847302, -87.62773357, 27, 2, "6/
↪28/2013")
>>> s25.distance_to(s44)
1479.6238912792467
>>> s25.distance_to(s44)
1479.6238912792467
```

Next, we need to model the information about a Divvy trip. The dataset provides the following information about each trip:

- trip_id: A unique integer identifier for the trip.
- starttime, and stoptime: The start and end time of the trip.
- bikeid: A unique integer identifier for the bike used in this trip.
- tripduration: The duration (in seconds) of the trip.
- from_station_id and to_station_id: The integer identifiers of the origin and destination stations.
- from_station_name and to_station_name: The names of the origin and destination stations.
- usertype: This field will be either Customer or Subscriber. A "customer" is a rider who purchased a 24-Hour Pass, and a "subscriber" is a rider who purchased an Annual Membership.
- gender: The gender of the rider. This field only has a value when the rider is a subscriber.
- birthday: The date of birth of the rider. This field only has a value when the rider is a subscriber.

Since we already defined a DivvyStation class, we will use it to represent the information about the origin and destination stations. So, when defining a DivvyTrip class, instead of having from_station_id and to_station_id attributes, we will instead have a from_station attribute that will contain a DivvyStation object (we will also have a similar to_station attribute). So, the class will look like this:

```
class DivvyTrip(object):
    def __init__(self, trip_id, starttime, stoptime, bikeid,
                 tripduration, from_station, to_station,
                 usertype, gender, birthyear):
```

(continues on next page)

(continued from previous page)

```
        self.trip_id = trip_id
        self.starttime = starttime
        self.stoptime = stoptime
        self.bikeid = bikeid
        self.tripduration = tripduration
        self.from_station = from_station
        self.to_station = to_station
        self.usertype = usertype
        self.gender = gender
        self.birthyear = birthyear
```

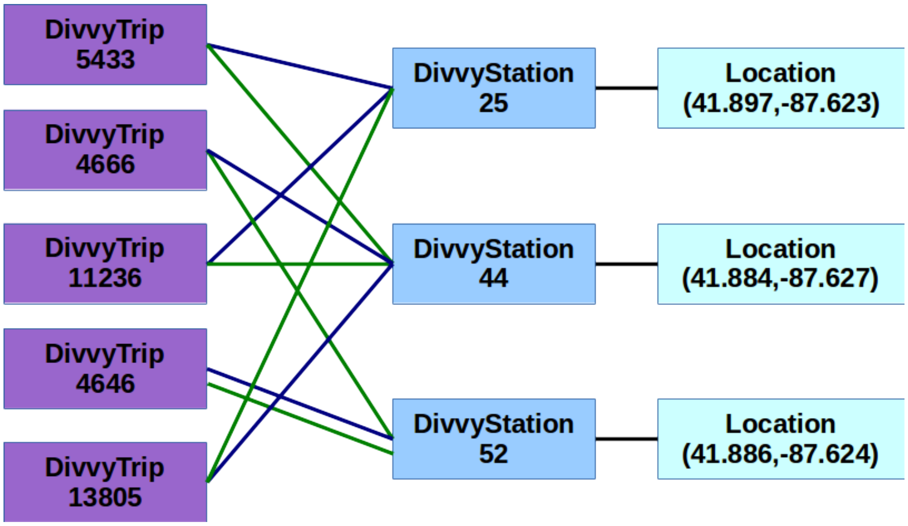
Now, let's say we created three stations:

```
>>> s25 = DivvyStation(25, "Michigan Ave & Pearson St", 41.89766, -87.62351, 23, 34, "6/
↪28/2013")
>>> s44 = DivvyStation(44, "State St & Randolph St", 41.8847302, -87.62773357, 27, 2, "6/
↪28/2013")
>>> s52 = DivvyStation(52, "Michigan Ave & Lake St", 41.88605812, -87.62428934, 23, 43,
↪"6/28/2013")
```

When creating the `DivvyTrip` objects, we would use the above `DivvyStation` objects as values for the `from_station` and `to_station` parameters to the constructor:

```
>>> trip5433 = DivvyTrip(5433,   "2013-06-28 10:43", "2013-06-28 11:03", 218, 1214,
...                      s25, s44, "Customer", None, None)
>>> trip4666 = DivvyTrip(4666,   "2013-06-27 20:33", "2013-06-27 21:22", 242, 2936,
...                      s44, s52, "Customer", None, None)
>>> trip11236 = DivvyTrip(11236, "2013-06-30 15:41", "2013-06-30 15:58", 906, 1023,
...                       s25, s44, "Customer", None, None)
>>> trip4646 = DivvyTrip(4646,   "2013-06-27 20:22", "2013-06-27 20:39", 477, 996,
...                      s52, s52, "Customer", None, None)
>>> trip13805 = DivvyTrip(13805, "2013-07-01 13:21", "2013-07-01 13:35", 469, 858,
...                       s44, s25, "Customer", None, None)
```

Notice how we can reuse the `DivvyStation` objects across `DivvyTrip` objects. For example, `s25` is the origin station in `trip5433` and `trip11236`, as well as the destination station in `trip13805`. This is yet another example of how we can *compose* multiple classes together to form complex data structures. The following figure shows these composition relationships between the objects we have created so far (blue lines represent an "origin station" relation and green lines represent a "destination station" relation):



Notice how the DivvyStation objects are “shared” between the various DivvyTrip objects, and how it is even possible for a trip to have the same DivvyStation object as its origin *and* its destination.

Another advantage of this composition is that we can conveniently access information about the origin and destination stations through the from_station and to_station attributes:

```
>>> trip5433.from_station.name
'Michigan Ave & Pearson St'
>>> trip4646.to_station.location
(41.886 N, 87.624 W)
>>> trip13805.to_station.stationID
25
```

This also means that adding a get_distance method to the DivvyTrip class, to compute the distance from the origin station to the destination station, becomes very simple:

```
class DivvyTrip(object):
    def __init__(self, trip_id, starttime, stoptime, bikeid,
                 tripduration, from_station, to_station,
                 usertype, gender, birthyear):
        self.trip_id = trip_id
        self.starttime = starttime
        self.stoptime = stoptime
        self.bikeid = bikeid
        self.tripduration = tripduration
        self.from_station = from_station
        self.to_station = to_station
        self.usertype = usertype
        self.gender = gender
        self.birthyear = birthyear

    def get_distance(self):
        return self.from_station.distance_to(self.to_station)
```

It seems like we’re getting closer to the point where we can answer the question we originally posed: *What is the total duration and total distance of all the Divvy trips taken in 2013?* However, so far, we’ve been creating DivvyStation and DivvyTrip objects manually. To answer this question, we will need to load the entire Divvy dataset. To do this, we have written a DivvyData class that encapsulates all interactions with the full dataset. We will not discuss the internal details of DivvyData here, but we encourage you to look at its source code.

We can create a DivvyData object as follows:

```
data = DivvyData(stations_filename="data/divvy_2013_stations.csv",
                 trips_filename="data/divvy_2013_trips.csv")
```

This object has two attributes: stations, with a list of DivvyStation objects corresponding to the stations in the dataset, and trips with a list of DivvyTrips objects.

```
>>> len(data.stations)
300
>>> len(data.trips)
759788
```

Computing the total distance of all the trips now involves just a simple for loop:

```
>>> total_distance = 0.0
>>> for trip in data.trips:
...     total_distance += trip.get_distance()
...
>>> total_distance
1557441209.2794986
```

Notice how that single call to get_distance abstracts away a lot of the details we've described above. get_distance takes the origin and destination stations, calls the distance_to method in DivvyStation which, in turn, accesses the location attributes of the stations, and calls the distance_to method in Location which implements a fairly elaborate formula that we are blissfully unaware of in the above piece of code.

Similarly, the total duration of all the trips involves an equally simple for loop:

```
>>> total_duration = 0.0
>>> for trip in data.trips:
...     total_duration += trip.tripduration
...
>>> total_duration
941734778.0
```

Now that we've organized our data in this way, and created convenient abstractions around stations and trips, it becomes easy to perform other computations. For example, by using Python's collections.Counter class (which itself provides a convenient abstraction around "counting things"), we can easily find out what are the top 10 origin stations in the Divvy system:

```
>>> from collections import Counter
>>> origin_stations = []
>>> for t in data.trips:
...     origin_stations.append(t.from_station)
>>> c = Counter(origin_stations)
>>> for station, count in c.most_common(10):
...     print("{:30} {}".format(station.name, count))
...
Millennium Park                17272
Streeter Dr & Illinois St      16710
Lake Shore Dr & Monroe St      15673
Clinton St & Washington Blvd   14967
```

(continues on next page)

(continued from previous page)

```
Michigan Ave & Oak St          13274
Museum Campus                  12676
McClurg Ct & Illinois St       10847
Michigan Ave & Lake St         10760
Canal St & Jackson Blvd        10141
Franklin St & Jackson Blvd     9784
```

11.3 Public vs private attributes

As you may have noticed, once we create a `Location` object, we can access its attributes freely:

```
>>> chicago_loc = Location( 41.8337329, -87.7321555 )
>>> chicago_loc.latitude
41.8337329
>>> chicago_loc.longitude
-87.7321555
```

We can also *modify* these attributes:

```
>>> chicago_loc
(41.834 N, 87.732 W)
>>> chicago_loc.latitude = -70.67
>>> chicago_loc
(70.670 S, 87.732 W)
```

However, allowing unfettered access to the attributes like this could make our `Location` class fail in some cases. For example, let's say we create the following locations:

```
>>> chicago_loc = Location( 41.8337329, -87.7321555 )
>>> newyork_loc = Location( 40.7056308, -73.9780035 )
```

We can now obtain the distance between the two locations like this:

```
>>> chicago_loc.distance_to(newyork_loc)
1155076.7723381526
```

But, if we now change the `latitude` coordinate of point `newyork_loc` to an invalid value, like a string, it will make the `distance_to` method fail, because it performs operations on `latitude` that assume that the attribute will contain a number:

```
>>> newyork_loc.latitude = "foobar"
>>> chicago_loc.distance_to(newyork_loc)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 25, in distance_to
TypeError: unsupported operand type(s) for -: 'str' and 'float'
```

The problem here is that `latitude` and `longitude` are *public* attributes, meaning that the user of the `Location` class can read and modify their values. Ideally, we would like to make these attributes *private*, and provide a controlled mechanism for access the values of `latitude` and `longitude`.

To explain how to make attributes private, we will use two new classes, `Point` and `Line`, each of which are originally implemented with public attributes. `Point` represent a point in two-dimensional space:

```
class Point(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return "({}, {})".format(self.x, self.y)

    def to_polar(self):
        r = math.sqrt( self.x**2 + self.y**2 )
        theta = math.degrees( math.atan( self.y / self.x ) )
        return r, theta

    def distance(self, other):
        return math.sqrt((self.x - other.x)**2 + (self.y - other.y)**2)
```

Line represents a line in two-dimensional spaces, as specified by two points:

```
class Line(object):
    def __init__(self, p0, p1):
        self.p0 = p0
        self.p1 = p1

    def is_vertical(self):
        return self.p0.x == self.p1.x

    def get_slope(self):
        if self.is_vertical():
            return float('inf')
        else:
            return (self.p1.y - self.p0.y) / (self.p1.x - self.p0.x)

    def get_y_intercept(self):
        if self.is_vertical():
            if self.p0.x == 0:
                return 0.0
            else:
                return float('NaN')
        else:
            return self.p0.y - self.get_slope()*self.p0.x

    def __repr__(self):
        if self.is_vertical():
            return "x = {}".format(self.p0.x)
        else:
            return "y = {}*x + {}".format(self.get_slope(), self.get_y_intercept())
```

Notice how this is yet another example of composition: a Line object will be composed of two Point objects:

```
>>> p = Point(0, 7)
>>> q = Point(1, 12)
>>> l = Line(p, q)
>>> l.get_slope()
```

(continues on next page)

(continued from previous page)

```
5.0
>>> l.get_y_intercept()
7.0
>>> l
y = 5.0*x + 7.0
```

We will focus first on the Point class. Its two attributes are currently public, which means we can modify them freely and, as with the Location class, this design can cause errors if we set the value of x or y to an invalid value:

```
>>> l.get_slope()
5.0
>>> p.x = "foobar"
>>> l.get_slope()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 13, in get_slope
TypeError: unsupported operand type(s) for -: 'int' and 'str'
```

To make x and y private, we will rename them to _x and _y:

```
class Point(object):
    def __init__(self, x, y):
        self._x = x
        self._y = y
```

Adding a single underscore before an attribute name is a common Python convention to indicate that an attribute shouldn't be directly accessed by the users of the class. i.e., they are intended to be *private* attributes that only the developers of the class should be able to manipulate.

Of course, we may want to give users of the Point class the ability to read and modify these attributes, but we want to do so in a controlled manner: we want to make sure they value of these attributes is always a number.

One way of doing this is by adding *getter* and *setter* method. These are methods whose purpose is to *get* or *set* the value of an attribute. For example, the getter and setter for _x would look like this:

```
def get_x(self):
    return self._x

def set_x(self, x):
    if not isinstance(x, (int, float)):
        raise ValueError("Not a number")
    self._x = x
```

Notice how set_x first checks whether the provided parameter x is an int or a float and, if not, raises a ValueError exception. We have not seen exceptions in detail yet, but this will basically make the program fail if we try to set x to be anything but a number.

The complete implementation of the Point class with getters and setters would look like this. Notice how the __init__ method also calls the setters, to ensure that the values of the attributes provided to the constructor are also correct:

```
class Point(object):
    def __init__(self, x, y):
        self.set_x(x)
        self.set_y(y)
```

(continues on next page)

(continued from previous page)

```
    def get_x(self):
        return self._x

    def set_x(self, x):
        if not isinstance(x, (int, float)):
            raise ValueError("Not a number")
        self._x = x

    def get_y(self):
        return self._y

    def set_y(self, y):
        if not isinstance(y, (int, float)):
            raise ValueError("Nor a number")
        self._y = y

    def __repr__(self):
        return "({}, {})".format(self._x, self._y)

    def to_polar(self):
        r = math.sqrt( self._x**2 + self._y**2 )
        theta = math.degrees( math.atan( self._y / self._x ) )
        return r, theta

    def distance(self, other):
        return math.sqrt((self._x - other._x)**2 + (self._y - other._y)**2)
```

Now, we can use the Point class as before, except we use the getters and setters to access the _x and _y attributes. If we try to set either of them to an invalid value, we will get an error:

```
>>> p = Point(2,3)
>>> p.get_x()
2
>>> p.set_x(3.14)
>>> p
(3.14, 3)
>>> p.set_x("foobar")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 11, in set_x
ValueError: Not a number
```

Notice how to_polar and distance *don't* use the getters and setters. The getters and setters are part of the class's *public* interface, so the users of our data type must use them to access the x and y coordinates. However, in the internal implementation of the class, we are still free to use the _x and _y attributes directly.

However, it is worth reiterating that adding a single underscore before an attribute name only makes the attribute private *by convention*, meaning that other programmers should know not to use any attribute that starts with an underscore, but Python doesn't actually protect these attributes:

```
>>> p._x
3.14
```

(continues on next page)

(continued from previous page)

```
>>> p._x = 5
>>> p
(5, 3)
```

If we want to make an attribute private, we can use two underscores before the attribute name (although, technically, this action doesn't make the attribute truly private; there are still ways of accessing that attribute, but they are slightly obfuscated, and not as simple as just using the name of the attribute with two underscores). However, we should be careful when doing this: if we are implementing *multiple* classes, we may want one class to access the attributes of another class. Using a single underscore will allow us to do that, while signalling to programmers that use the classes that they should not use those attributes directly. On the other hand, using two underscores will prevent other classes from accessing those attributes.

11.4 Properties

Switching to private attributes, and adding getters/setters, gives us greater control on how users of the Point class interact with its attributes. However, it also means that accessing or modifying an attribute now requires writing code like this:

```
p.get_x()
p.set_x(42)
```

Instead of this, which is arguably more readable:

```
p.x
p.x = 42
```

On top of that, if we originally wrote a class with public attributes, and then decide we need private attributes and getters/setters, we need to rewrite all the code that depended on those attributes. In fact, that's what we would need to do with the Line class, which will now be broken because our Point class now has an _x attribute instead of an x attribute:

```
>>> p = Point(0, 7)
>>> q = Point(1, 12)
>>> l = Line(p, q)
>>> l.get_slope()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 10, in get_slope
  File "<stdin>", line 7, in is_vertical
AttributeError: 'Point' object has no attribute 'x'
```

Fortunately, instead of having to re-write the Line class to use the getters and setters, we can modify the Point class to provide *managed attributes*. A managed attribute is a publicly accessible attribute, but where access to the attribute is *managed* by a getter and/or setter (without requiring the user of the class to explicitly call the getter or setter).

For example, in the Point class, we can define private _x and _y attributes, and then define public *managed* attributes called x and y which, when accessed, actually translate internally to a call to the getter/setter for _x and _y. In other words, when we do something like this:

```
p.x
p.x = 42
```

Python will translate the above to calls to the following:

```
p.get_x()
p.set_x(42)
```

In Python, this is accomplished by using *properties*. A property is defined inside the class like this:

```
<property_name> = property(<getter>, <setter>)
```

For example, in the Point class, we would just add the following to our class definition below the definitions of the get and set methods:

```
x = property(get_x, set_x)
y = property(get_y, set_y)
```

The complete code for the class would now look like this:

```
class Point(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def get_x(self):
        return self._x

    def set_x(self, x):
        if not isinstance(x, (int, float)):
            raise ValueError("Not a number")
        self._x = x

    def get_y(self):
        return self._y

    def set_y(self, y):
        if not isinstance(y, (int, float)):
            raise ValueError("Nor a number")
        self._y = y

    # Property definitions. Must be done *after* we've defined the getters/setters.
    x = property(get_x, set_x)
    y = property(get_y, set_y)

    def __repr__(self):
        return "({}, {})".format(self.x, self.y)

    def to_polar(self):
        r = math.sqrt( self._x**2 + self._y**2 )
        theta = math.degrees( math.atan( self._y / self._x ) )

    def distance(self, other):
        return math.sqrt((self.x - other.x)**2 + (self.y - other.y)**2)
```

Now, whenever we access x and y (which are not actual attributes, they just behave like attributes), Python will translate the use into a call to the corresponding getter or setter. This behavior is specially evident if we try to set the value of x to an invalid value:

```
>>> p = Point(2,3)
>>> p.x
2
>>> p.x = 6
>>> p
(6, 3)
>>> p.x = "foo"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 11, in set_x
ValueError: Not a number
```

Notice how we also access the attributes in this manner from inside the constructor, instead of manipulating the private _x and _y attributes:

```
def __init__(self, x, y):
    self.x = x
    self.y = y
```

This ensures that we are not able to create a Point object with invalid values for x or y:

```
>>> p = Point(100, "foobar")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in __init__
  File "<stdin>", line 19, in set_y
ValueError: Nor a number
```

Take into account that properties also allow us to define immutable attributes. All we have to do is *not* provide a setter when creating the property. For example:

```
x = property(get_x)
y = property(get_y)
```

Notice how, after making these changes, our implementation of Line works again:

```
>>> p = Point(0, 7)
>>> q = Point(1, 12)
>>> l = Line(p, q)
>>> l.get_slope()
5.0
```

Finally, we can also define properties using *function decorators*, which allow us to annotate the getters/setters directly. For example, this is how our Point class would look if we used function decorators to annotate the getters and setters:

```
class Point(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    # Notice how the name of the method is not "get_x" but simply "x"
    # the name of the attribute.
    @property
    def x(self):
```

(continues on next page)

(continued from previous page)

```
        return self._x

    # Now we define the setter, which has the same name, but an
    # additional parameter (the value being set). Notice how the
    # decorator isn't "@property", it's the name the attribute and .setter
    @x.setter
    def x(self, x):
       if not isinstance(x, (int, float)):
           raise ValueError("Not a number")
       self._x = x

    @property
    def y(self):
        return self._y

    @y.setter
    def y(self, y):
        if not isinstance(y, (int, float)):
            raise ValueError("Nor a number")
        self._y = y

    def __repr__(self):
        return "({}, {})".format(self.x, self.y)

    def distance(self, other):
        return math.sqrt((self.x - other.x)**2 + (self.y - other.y)**2)
```

When to use public attributes, private attributes, and properties

Using private attributes with getter and setters (with or without properties) does come at a cost: instead of accessing attributes directly (which is a very efficient operation), we now have to incur in the cost of a function call every time we interact with an attribute. Even if the performance hit is small, it still means we have to define a getter and possibly a setter for every attribute. This can resulted in a very bloated implementation if, say, we have a class with 50 attributes.

A good strategy is to start by implementing your code using public attributes. If the classes you are writing are going to be used only by you, or if you are just prototyping some code, there is probably not immediate advantage to using private attributes and getters/setters. However, if there comes a point when you need to manage access to the attributes, you can switch to using properties, which will not break any existing code that relied on those attributes.

In any case, if you are writing classes that will be used by other programmers, it is generally a good idea to very clearly separate the private and public data of your class, and making sure that access to private attributes is either managed through the use of getters/setters or properties or hidden from the client all together.

11.5 Class/static attributes vs instance attributes

In the Point class, x and y are *instance* attributes. This means that each Point object (or *instance* of the Point class) will have their own values for x and y:

```
>>> p1 = Point(2,3)
>>> p2 = Point(5,6)
>>> p1.x
2
>>> p2.x
5
```

We can also define *class attributes* which belong to the class as a whole, not to individual instances. These are also referred to as *static attributes*.

A common use case for this mechanism is to define *constants* that are relevant to that class. This would make sense in our Location class, where the distance_to method hardcoded the average radius of Earth (6371000.0 below):

```
def distance_to(self, other):
    diffLatitude = math.radians(other.latitude - self.latitude)
    diffLongitude = math.radians(other.longitude - self.longitude)

    a = math.sin(diffLatitude/2) * math.sin(diffLatitude/2) + \
        math.cos(math.radians(self.latitude)) * \
        math.cos(math.radians(other.latitude)) * \
        math.sin(diffLongitude/2) * math.sin(diffLongitude/2)
    d = 2 * math.asin(math.sqrt(a))

    return 6371000.0 * d
```

While we could define this as a global constant outside the Location class:

```
EARTH_RADIUS = 6371000.0

class Location(object):
    ...
```

We can instead define that value *inside* the Location class:

```
class Location(object):
    EARTH_RADIUS = 6371000.0

    ...
```

So the return statement in distance_to would now be this:

```
return Location.EARTH_RADIUS * d
```

Notice how, to access the value of a class attribute, we need to qualify it with the name of the class.

Another common use case for class attributes is when we have some data that needs to be shared by all instances of a class. In essence, this attribute is like a global variable, but one where we restrict the scope of the variable to a specific class. For example, if we have a Client class and want to assign each new object an incrementing identifier, we could define our class like this:

```
class Client(object):
    next_identifier = 1

    def __init__(self, name):
        self.name = name
        self.identifier = Client.next_identifier
        Client.next_identifier += 1
```

This class defines a class attribute `next_identifier` that is initialized to 1 at the start of our program, and which is incremented every time we construct a new `Client` object:

```
>>> a = Client("Alice")
>>> b = Client("Bob")
>>> c = Client("Carol")
>>> a.identifier
1
>>> b.identifier
2
>>> c.identifier
3
>>> Client.next_identifier
4
```

Part III.

Functional Programming and Recursion

CHAPTER

TWELVE

FUNCTIONAL PROGRAMMING

Programming languages are often classified into one of several paradigms. For example, in *imperative* programming languages, computation is expressed as sequences of statements that update the state of the computation, including control flow statements like conditional and looping statements. In this context, "imperative" means "to give commands" (where statements are commands that are "given" in a program). Python is an imperative programming language, as are many others: C, C++, Java, etc.

We have also seen the *object-oriented (OO) paradigm*, where programs manipulate objects encapsulating some data (their attributes) and operations on the objects (their methods). Languages like Java are known as pure OO languages, because we *must* use classes and objects everywhere in our program. Python, along with C++ and others, is a multi-paradigm language: we can use OO features, but can stick with just imperative constructs if we like.

Another major paradigm of programming languages is the *functional* paradigm, in which functions are primarily used to compute values, rather than for their side effects (e.g., updating variables, printing). Functional programming languages include LISP, Scheme, Haskell, SML, and others. Python is not a pure functional language, but it does support a number of features commonly found in functional programming languages that are not traditionally found in languages like C, C++, Java, etc. (although some of these languages are starting to include features that are influenced by functional programming languages).

In this chapter, we will focus on two aspects of functional programming that Python supports: higher-order functions and anonymous functions.

12.1 Higher-order functions

Let's say we wanted to find the value of an integral:

$$\int_{10}^{20} x^2 \, dx$$

One way to do this is to apply integration rules, which tell us that the antiderivative of x^2 is $\frac{x^3}{3}$:

$$\int_{10}^{20} x^2 \, dx = \left[\frac{x^3}{3}\right]_{10}^{20} = \frac{20^3}{3} - \frac{10^3}{3} = \frac{8000}{3} - \frac{1000}{3} = \frac{7000}{3} = 2333.\overline{3}$$

While it is possible to write a program that applies integration rules to evaluate a given integral, programs more commonly evaluate integrals using *numerical methods*, or ways to approximate the value of the integral.

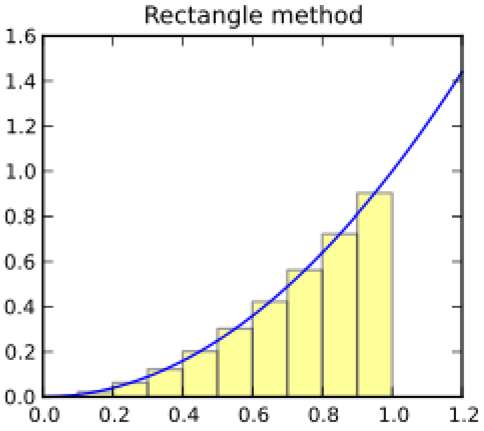
A common numerical method for computing integrals is the *rectangle method*, in which the area under the curve is approximated by N rectangles. If we are computing the integral between a and b, then the width of each rectangle will be:

$$\Delta = \frac{b-a}{N}$$

We can then approximate the integral as follows:

$$\int_a^b f(x)\,dx \approx \sum_{i=0}^{N-1} \Delta \cdot f\left(a + \Delta \cdot (i + 0.5)\right)$$

In other words, we divide the range between a and b into N subranges, all of width Δ. For each subrange, we compute the area of a rectangle, where the height of the rectangle is simply the value of the function at the midpoint of the subrange (using the start or end of the subrange is not desirable for reasons that are not worth getting into here) The sum of the areas of all those rectangles approximates the value of the integral between a and b.



We can easily write a function that implements this approach. Let's assume the function we're integrating is the square function:

```
def square(x):
    return x ** 2
```

Our integration function, which simply implements the above summation using a `for` loop, would look like this:

```
def integrate_square(lb, ub, N):
    '''
    Compute the integral of the square function between the
    specified bounds using the rectangle method.

    Inputs:
      lb (float): lower bound of the range
      ub (float): upper bound of the range
      N (int): the number of rectangles to use

    Returns (float): an approximation of the area under the curve
      between the specified bounds.
    '''

    delta = (ub - lb) / N
    sum = 0
    for i in range(N):
        sum += delta * square(lb + delta * (i + 0.5))
    return sum
```

If we run this function with `N` equal to 100, we get a reasonable approximation to the expected value of the integral ($2333.\overline{3}$)

```
>>> integrate_square(10.0, 20.0, 100)
2333.3250000000007
```

And, as we divide the area under the function into more rectangles, we get more accurate results, at the expense of more computation:

```
>>> integrate_square(10.0, 20.0, 1000)
2333.3332500000006
>>> integrate_square(10.0, 20.0, 10000)
2333.3333325000003
>>> integrate_square(10.0, 20.0, 100000)
2333.3333333249616
```

But what if we want to compute the integral of other functions? We would need to create functions that repeat essentially the same code! So, to integrate $f(x) = 2 \cdot x$ we would have to write the following:

```
def double(x):
    return 2 * x

def integrate_double(lb, ub, N):
    '''
    Compute the integral of the double function between the
    specified bounds using the rectangle method.

    Inputs:
      lb (float): lower bound of the range
      ub (float): upper bound of the range
      N (int): the number of rectangles to use

    Returns (float): an approximation of the area under the curve
      between the specified bounds
    '''

    delta = (ub - lb) / N
    sum = 0
    for i in range(N):
        sum += delta * double(lb + delta * (i + 0.5))
    return sum
```

```
>>> integrate_double(10.0, 20.0, 10000)
299.9999999999999
```

If we compare `integrate_square` and `integrate_double`, we can see that they are identical except for the call to the function that is being integrated. In Python, and in most functional programming languages, we address this code duplication by writing a general purpose `integrate` function that, instead of integrating a specific hard-coded function, takes a function *as a parameter*:

```
def integrate(fn, lb, ub, N):
    '''
    Compute the integral of the specified function between the
    specified lower and upper bounds using the rectangle method.

    Inputs:
```

(continues on next page)

(continued from previous page)

```
      fn (function): function to be integrated.  Must take a float
        and return a float
      lb (float): lower bound of the range
      ub (float): upper bound of the range
      N (int): the number of rectangles to use

    Returns (float): an approximation of the area under the curve
      (specified by the function parameter) between the bounds.
    '''


    delta = (ub - lb) / N
    sum = 0
    for i in range(N):
        sum += delta * fn(lb + delta * (i + 0.5))
    return sum
```

A call to this function looks like:

```
>>> integrate(square, 10.0, 20.0, 10000)
2333.3333325000003
```

Notice that `integrate` is nearly identical to both `integrate_square` and `integrate_double`, except we have added a parameter called `fn` that must be a function. When we call `integrate`, we are passing the `square` function itself (not the result of calling `square` with some specific values).

Because `integrate` takes a function parameter, we refer to it as a *higher-order function*. A higher-order function is any function that takes another function as a parameter or, as we'll see later on, returns a function as its result. Notice that we don't define higher-order functions in any special way; the syntax is the same as what we've seen so far.

So, since the function to be integrated is now a parameter to `integrate`, integrating a different function is as simple as calling `integrate` with the function we want to integrate:

```
>>> integrate(double, 10.0, 20.0, 10000)
299.9999999999999
```

We can also use functions included with Python:

```
>>> import math
>>> integrate(math.sqrt, 10.0, 20.0, 10000)
38.54662833413495
```

Note that `integrate` will only work if we pass it a function that takes a single float parameter and returns a float value. If we pass any other type of function, our code will fail:

```
def multiply(a, b):
   return a*b
```

```
>>> integrate(multiply, 10.0, 20.0, 10000)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 21, in integrate
TypeError: multiply() missing 1 required positional argument: 'b'
```

The exact type of failure will depend on the function. In this case, for example, we passed a function with two required parameters where a function of one parameter was needed.

12.2 Anonymous functions

The `integrate` function can take a function as a parameter, but it requires defining a separate function (like `square` and `double`) to pass as a parameter. Instead of defining a function just for the purposes of passing it as a parameter to another function, we can instead use *anonymous functions* (also known as *lambdas*). An anonymous function is basically a function we define on the fly without using the `def` statement to give it a name. For example, here is an anonymous function to compute the square of a single parameter `x`:

```
lambda x: x ** 2
```

And here is an anonymous function to add two parameters `x` and `y`:

```
lambda x, y: x + y
```

The general syntax of an anonymous function is the following:

`lambda` `<parameters>: <expression>`

Notice that anonymous functions contain a single expression. They don’t contain statements like a regular function (so you cannot use conditional statements or loops), nor do they have a `return` statement. The value the function will return is simply the result of the evaluating the expression specified in the `lambda` using the values supplied for the parameters.

And, since they’re anonymous, functions defined using `lambda` need to be given a name before they can be called. Anonymous functions are most commonly used as parameters to other functions. For example, the following piece of code is equivalent to passing the `square` function to `integrate`:

```
>>> integrate(lambda x: x ** 2, 10.0, 20.0, 10000)
2333.3333325000003
```

And the following piece of code is equivalent to passing the `double` function:

```
>>> integrate(lambda x: 2 * x, 10.0, 20.0, 10000)
299.9999999999999
```

12.3 map(), reduce(), filter()

Python provides a few very useful higher-order functions for working with lists.

Let’s say we had the following list:

```
lst = [1, 2, 3, 4, 5]
```

If we wanted to create a new list that contains the above values, but with each value incremented by one. We could do it like this:

```
lst2 = []
for x in lst:
    lst2.append(x + 1)
```

```
>>> print(lst2)
[2, 3, 4, 5, 6]
```

Functional programming languages provide a different mechanism for processing lists in this way. In fact, most functional programming languages lack looping constructs like `for` or `while` and, instead, provide mechanisms to apply functions to sequences of values in different ways (in the next chapter we will also see the general mechanism that functional programming languages use to repeat operations: *recursion*)

For example, if we wanted to increment each value of the list by one, we would define a function that increments a single value:

```
def incr(x):
    return x + 1
```

Then, we would call that function on each value of the list, and place the resulting values in a new list. However, instead of doing this task with a `for` loop, we can do it with a single call to the built-in `map` function

```
map(incr, lst)
```

The return value of this call is actually not a new list, but rather an iterable object (like the kind of value that we get from a call to `range`) that we can then use in a `for` loop or any other context, such as a call to `map`, that requires an iterable, or which we can convert to a new list:

```
>>> for x in map(incr, lst):
...     print(x)
...
2
3
4
5
6
>>> lst2 = list(map(incr, lst))
>>> lst2
[2, 3, 4, 5, 6]
```

As we saw with the `integrate` function, we don't need to define a new function before calling `map` and, instead, we can just use an anonymous function as the first parameter to `map`:

```
>>> lst
[1, 2, 3, 4, 5]
>>> list(map(lambda x: x * 2, lst))
[2, 4, 6, 8, 10]
>>> lst
[1, 2, 3, 4, 5]
>>> list(map(lambda x: x ** 2, lst))
[1, 4, 9, 16, 25]
>>> lst
[1, 2, 3, 4, 5]
```

Notice that `map` does not change the list it operates on.

If we would like to create a new list with elements copied from an input list *only* if they meet a given condition, we can instead use the built-in `filter` function. This function takes a function that returns a boolean result and a list as arguments. If, for a given value in the list, the supplied function returns `True`, then the value will be included in the resulting list. For example:

```
def is_odd(x):
    return x % 2 == 1
```

```
>>> lst
[1, 2, 3, 4, 5]
>>> list(filter(is_odd, lst))
[1, 3, 5]
```

Or, equivalently:

```
>>> list(filter(lambda x: x % 2 == 1, lst))
[1, 3, 5]
```

Another useful function found in many functional programming languages is the `reduce` function. Unlike `map` and `filter`, it does not apply a function to each value in the list; instead, it repeatedly applies the function to reduce the list to a single value. For example, say we had the following list:

```
lst = [1, 2, 3, 4]
```

If we wanted to multiply all the values in the list, we could write the following loop:

```
p = lst[0]
for x in lst[1:]:
    p = x * p

print(p)
```

```
24
```

Notice that this code boils down to evaluating this expression:

```
>>> ((lst[0] * lst[1]) * lst[2]) * lst[3]
24
```

If we define a `multiply` function, we could rewrite this expression as follows:

```
def multiply(x, y):
   return x * y
```

```
>>> multiply(multiply(multiply(lst[0], lst[1]), lst[2]), lst[3])
24
```

And this repeated application of a function is exactly what `reduce` does:

```
>>> from functools import reduce
>>> reduce(multiply, lst)
24
```

Notice that unlike `map` and `filter`, `reduce` needs to be imported from the `functools` library.

Or, using a `lambda`:

```
>>> reduce(lambda x, y: x * y, lst)
24
```

12.3.1 List comprehensions revisited

In *Lists, Tuples, and Strings*, we briefly described *list comprehensions* as shorthand notation for producing a new list based on an existing list. In particular, the following piece of code:

```
original_list = [1, -2, 3, 4, -5]
new_list = []

for x in original_list:
    if x > 0:
        new_list.append(x ** 2)

print(new_list)
```

```
[1, 9, 16]
```

Can be written more compactly using a list comprehension:

```
new_list = [x ** 2 for x in original_list if x > 0]
print(new_list)
```

```
[1, 9, 16]
```

Another way of thinking about list comprehensions is that they are a more readable notation for combining a `map` and `filter` on a list. The above list comprehension is equivalent to this code:

```
new_list = list(map(lambda x: x**2, filter(lambda x: x>0, original_list)))
print(new_list)
```

```
[1, 9, 16]
```

12.4 Returning a function

Recall that we defined higher-order functions as functions that can take other functions as arguments and return functions as results. So far, we've seen how higher-order functions can take a function as a parameter. In this section, we explain how to write functions that return new functions as results. For example, let's say that, given an arbitrary function, f that takes a single float and returns a float, we want to compute a new function to give us the value of the derivative of f at a given point. In other words, the end result would look something like this:

```
def square(x):
    return x ** 2
```

```
>>> square(9)
81
>>> square_prime = derivative(square)
>>> square_prime(9)
18
```

Notice that the `derivative` function returns a function that we can then call later in our code. `square_prime` is a variable, but one that refers to a new function (instead of containing a value like an integer, or referring to a list or dictionary). Furthermore, note that we want the `derivative` function to work for *any* arbitrary function, so we should also be able to write code like this:

```
>>> cube_prime = derivative(lambda x: x ** 3)
>>> cube_prime(3)
27
```

To explain how to do this, let's start by explaining how we can compute the derivative of a single function. As we did with integration, we will compute the value of the derivative *numerically*.

The derivative of a function f at point x is its slope at that point. We can approximate the slope by computing the value $f(x)$ and the value of the function at a nearby point $f(x + dx)$:

$$f'(x) = \frac{f(x + dx) - f(x)}{dx}$$

In calculus, you consider the limit as dx goes to zero. Here we will just assume dx is a small value like `0.00001`.

So, if we had the following function:

```
def cube(x):
    return x ** 3
```

We could follow the above definition to define its derivative like this:

```
def cube_slope(x, dx=0.00001):
    '''
    Compute the slope of the cube function at the specified point.

    Inputs:
      x (float): point at which to evaluate the slope
      dx (float): difference (default: 0.0001)

    Returns (float): approximation to the slope of the cube
      function at the input point.
    '''

    return (cube(x + dx) - cube(x)) / dx
```

```
>>> cube(10)
1000
>>> cube_slope(10)
300.0002999897333
```

The derivate of x^3 is $3x^2$, so we are correctly approximating its slope (which would be 300 at $x = 10$).

Now, if we wanted to compute the derivative for a different function, the resulting functions would look very similar:

```
def square(x):
    return x ** 2

def square_slope(x, dx=0.00001):
    '''
    Compute the slope of the square function at the specified point

    Inputs:
      x (float): point at which to evaluate the slope
      dx (float): difference (default: 0.0001)
```

(continues on next page)

(continued from previous page)

```
    Returns (float): approximation to the slope of the square
      function at the input point.
    '''

    return (square(x + dx) - square(x)) / dx
```

One first approach at generating this code could be to create a general purpose `slope` function, like we did with our integration function:

```
def slope(f, x, dx=0.00001):
    '''
    Compute the slope of the specified function at the specified
    point

    Inputs:
      f (function): function to take the slope of.  The function
        must a float and return a float
      x (float): point at which to evaluate the slope of f.
      dx (float): difference (default: 0.0001)

    Returns (float): approximation to the slope of the specified
      function function at the input point.
    '''

    return (f(x + dx) - f(x)) / dx
```

```
>>> slope(cube, 10)
300.0002999897333
>>> slope(square, 10)
20.00000999942131
```

However, we actually want to generate *new* functions (so we don't have to pass `cube` or `square` as a parameter every time we want to compute its derivative). We do this by defining one function *inside* another function, and having the outer function return the inner function:

```
def derivative(f):
    '''
    Create a function of one variable that computes the derivative
    of the specified function.

    Inputs:
      fn (function): function to be differentiated.  Must take a
        float and return a float

    Returns: a function that computes the derivative of the
      specified function at a given point with an optional
      parameter for dx.
    '''

    def slope(x, dx = 0.00001):
        ''' Compute the slope of f at the specified point '''
```

(continues on next page)

(continued from previous page)

```

        return (f(x + dx)-f(x)) / dx

    return slope
```

In this case `slope`, the inner function, is the function we want to generate. Notice that it does not need to take a function as a parameter. Instead, because it is in the same scope as the parameter `f`, it can access `f` directly. The outer function, `derivative`, takes a function parameter and will return a `slope` function that will use that function to compute the slope.

```
>>> cube_prime = derivative(cube)
>>> cube_prime(2)
12.000060000261213
>>> cube_prime(5)
75.00014999664018
>>> cube_prime(5, dx=0.0000001)
75.00000165805432
>>> quad_prime = derivative(lambda x: x ** 4)
>>> quad_prime(3)
108.0005400012851
```

Because the derivative of a function of one variable, is itself a function of one variable, we can apply our derivative function repeatedly to get functions that compute the second derivative, third derivative, etc.

```
>>> cube_prime = derivative(cube)
>>> cube_double_prime = derivative(cube_prime)
>>> cube_triple_prime = derivative(cube_double_prime)
>>> cube_prime(10)
300.0002999897333
>>> cube_double_prime(10)
59.99936547596007
>>> cube_triple_prime(10)
227.37367544323203
```

12.5 Scoping and inner functions

Before we move on to our next topic, recursion, let's revisit the topic of scoping. Recall that every variable is defined and accessible within a specific part of the program, known as the variable's scope. Our earlier discussion (see *Variable scope*) covered local variables, which have scope that is local to a specific function and global variables, which are defined outside the context of any function. Adding inner functions to the mix introduces in a third kind of scope: *nonlocal*. A local variable or parameter in an outer function, such as the parameter `f` in our `derivative` function, is visible as a nonlocal variable in an inner function as long as it is not *shadowed* by a variable or parameter of the same name in the inner function. Like globals, nonlocal variables are read-only within an inner function unless the programmer explicitly declares them as `nonlocal`.

CHAPTER THIRTEEN

RECURSION

As we have seen throughout the book, there are many cases where we must repeat operations and can do so with a `for` or `while` loop. This form of repetition is formally known as *iteration* and involves defining a loop condition and a block of statements to be repeated as long as that condition is true. Each repetition of the block of statements is called an *iteration* and an algorithm that uses this style of repetition is called an *iterative algorithm*.

In this chapter, we will discuss a different way of expressing repetition: *recursion*. Recursion is equally expressive as iteration, meaning that anything we can do with iteration can also be done with recursion (and vice versa). There are some algorithms, however, where a recursive solution will require less code and will result in cleaner and more intuitive code than the equivalent iterative solution. In this chapter, we will introduce recursion and work through several problems that lend themselves naturally to a recursive solution.

13.1 Factorials

Let's look at the factorial operation:

$$4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$$

One of the formal definitions of factorial is:

$$n! = \prod_{k=1}^{n} k$$

We can implement this definition using a `for` loop:

```
def factorial(n):
    '''  Compute n! iteratively '''

    rv = 1
    for k in range(1, n+1):
        rv = rv * k
    return rv
```

```
>>> factorial(4)
24
>>> factorial(5)
120
```

Factorials can also be defined as follows:

$$n! = \begin{cases} 1 & \text{if n=1} \\ n \cdot (n-1)! & \text{if n>1} \end{cases}$$

This is a *recursive definition*. Notice how there is no reference to loops or to repetition. Instead, factorials are defined in terms of themselves. At first, this approach may seem odd. Imagine finding a definition in the dictionary that looked like this:

> **Recursion**: See *recursion*.

You would keep coming back to the definition of "recursion" infinitely!

However, the recursive definition of factorial works because it is divided into two cases:

- *The base case*: In this case, the value of the factorial can be obtained immediately and trivially. In the case where $n = 1$, the value of the factorial is known immediately and without further computation: it is simply 1.
- *The recursive case*: In this case, we define the factorial in terms of itself. For factorials, we can define $n!$ as n times $(n - 1)!$

So, if we were computing $4!$, we would start in the recursive case, which tells us that:

$$4! = 4 \cdot 3!$$

To evaluate this formula, we need to find the value of $3!$ which, again, involves the recursive case:

$$3! = 3 \cdot 2!$$

Similarly for $2!$:

$$2! = 2 \cdot 1!$$

But, when we get to $1!$, we know that $1! = 1$, so we can plug 1 into the above formula, and we get:

$$2! = 2 \cdot 1 = 2$$

Now that we know the value of $2!$, we can plug that into this formula:

$$3! = 3 \cdot 2!$$

And we get:

$$3! = 3 \cdot 2 = 6$$

And, finally, now that we know the value of $3!$, plug it into the formula for $4!$:

$$4! = 4 \cdot 3!$$

And we get:

$$4! = 4 \cdot 6 = 24$$

Notice that the recursive case must be defined so that it gets us closer to the base case when we use it; otherwise, we would fall into infinite recursion.

We can implement our recursive definition of factorial in Python like this:

```python
def factorial_r(n):
    '''  Compute n! recursively '''

    if n == 1:
        return 1
    elif n > 1:
        return n * factorial_r(n-1)
```

```
>>> factorial_r(4)
24
>>> factorial_r(5)
120
```

Notice that our function `factorial_r` calls itself. We refer to such functions as *recursive functions*, and we refer to the point where the function calls itself as a *recursive call*. While the concept of recursion can be easy to understand at a high level (think, for example, of how easily we defined factorials recursively), writing recursive functions and understanding what happens during a recursive call often stumps beginning programmers.

So, we are going to spend some time dissecting exactly what happens in a recursive function. After that, we will work through several examples of recursive algorithms that will help us to understand how to design recursive functions, as well as when to use a recursive algorithm instead of an iterative solution.

13.2 The anatomy of a recursive function call

To show what happens in the `factorial_r` function, we have prepared a more verbose version that will do the same thing as the function shown earlier, but will print messages to help us understand exactly what's going on. Don't worry if you don't understand how we're formatting the output (especially how we indent the messages, which requires using an extra parameter). Focus instead on following what happens during each recursive call.

```
def factorial_r(n, indent=""):
    '''
    Compute n! recursively and print information about the
    computation along the way.

    Inputs:
      n (int): operand
      indent (string): spaces to use as a prefix when printing.

    Returns (int): n!
    '''

    if n == 1:
       print(indent + "factorial_r(1) -- BASE CASE -- The value of 1! is 1")
       print()
       return 1
    elif n > 1:
       print(indent + "factorial_r({}) -- START OF RECURSIVE CASE".format(n))
       print(indent + "                   The value of {}! is {}*{}!".format(n, n, n-1))
       print(indent + "                   I need to find out the value of {}!".format(n-
↪1))
       print()
       x = factorial_r(n-1, indent=indent+"    ")
       print(indent + "factorial_r({}) -- END OF RECURSIVE CASE".format(n))
       print(indent + "                   I now know that {}! is {}".format(n-1, x))
       print(indent + "                   Which means that {}! = {}*{}".format(n, n, x))
       print()
       return n * x
```

```
>>> factorial_r(4)
factorial_r(4) -- START OF RECURSIVE CASE
```

(continues on next page)

(continued from previous page)

```
                  The value of 4! is 4*3!
                  I need to find out the value of 3!

    factorial_r(3) -- START OF RECURSIVE CASE
                      The value of 3! is 3*2!
                      I need to find out the value of 2!

        factorial_r(2) -- START OF RECURSIVE CASE
                          The value of 2! is 2*1!
                          I need to find out the value of 1!

            factorial_r(1) -- BASE CASE -- The value of 1! is 1

        factorial_r(2) -- END OF RECURSIVE CASE
                          I now know that 1! is 1
                          Which means that 2! = 2*1

    factorial_r(3) -- END OF RECURSIVE CASE
                      I now know that 2! is 2
                      Which means that 3! = 3*2

factorial_r(4) -- END OF RECURSIVE CASE
                  I now know that 3! is 6
                  Which means that 4! = 4*6

24
```

Notice how, whenever we run into a recursive case, we put that function call on "hold" while we go deeper into the recursion rabbit hole until we get to a base case and can start "wrapping up" all of the recursive cases that we put on hold. Examining the function call stack (see *The function call stack* in chapter *Introduction to Functions*) can help explain what happens during a call to a recursive function. This process is a fairly low-level aspect of how recursive calls work, and so you can skip this discussion for now if you want. However, if you're the kind of person who understands concepts better if you know exactly what happens under the hood, read on.

When a call is made to `factorial_r(4)`, the following entry is added to the function call stack. For simplicity, we will omit the `indent` parameter, which is used purely for formatting purposes.

Function: `factorial_r`
Parameters:
- `n`: 4

Variables:
- `x`: *undefined*

Return value: None

When the function reaches the recursive call (`factorial_r(n-1)`), it will need to make a call to `factorial_r(3)` before it can set a value for `x`. So, we add an entry for `factorial_r(3)`:

Function: `factorial_r`
Parameters:

- `n`: 4

Variables:

- `x`: *undefined*

Return value: None

Function: `factorial_r`
Parameters:

- `n`: 3

Variables:

- `x`: *undefined*

Return value: None

Recall that our stacks grow down the page, so we added the frame for the call `factorial_r(3)` beneath the frame for the call `factorial_4`.

This process is repeated for every recursive call, until we reach the recursive call that triggers the base case:

Function: `factorial_r`
Parameters:

- `n`: 4

Variables:

- `x`: *undefined*

Return value: None

Function: `factorial_r`
Parameters:

- `n`: 3

Variables:

- `x`: *undefined*

Return value: None

Function: `factorial_r`
Parameters:

- `n`: 2

Variables:

- `x`: *undefined*

Return value: None

Function: `factorial_r`
Parameters:

- `n`: 1

Variables:

- `x`: *undefined*

Return value: None

At this point, we have a function call stack that is holding information for all the calls from `factorial_r(4)` to `factorial_r(1)`. When we reach the base case, the recursion starts to *unwind* because we have reached a point where `factorial_r` can return a value without having to make any more recursive calls. The last entry in the function call stack will return 1:

Function: `factorial_r`
Parameters:

- `n`: 1

Variables:

- `x`: *undefined*

Return value: 1

The function call represented by the previous entry in the function call stack can now assign a value to variable `x` because it has finished evaluating the function call `factorial_r(1)`, so the function call stack will look like this:

Function: `factorial_r`
Parameters:

- `n`: 4

Variables:

- `x`: *undefined*

Return value: None

Function: `factorial_r`
Parameters:

- `n`: 3

Variables:

- `x`: *undefined*

Return value: None

Function: `factorial_r`
Parameters:

- `n`: 2

Variables:

- `x`: 1

Return value: 2

Remember that once a function call ends, its entry is removed (or popped) from the function call stack, so we no longer have an entry for `factorial_r(1)`.

Now, the function call to `factorial_r(2)` will be able to return the value `2` (i.e., `n` multiplied by `x`), which will become the value for `x` in `factorial_r(3)`:

Function: `factorial_r`
Parameters:

- `n`: 4

Variables:

- `x`: *undefined*

Return value: None

Function: `factorial_r`
Parameters:

- `n`: 3

Variables:

- `x`: 2

Return value: 6

We repeat this process one final time, to obtain the return value of `factorial_r(4)`:

Function: `factorial_r`
Parameters:

- n: 4

Variables:

- x: 6

Return value: 24

Thinking about recursive calls in terms of their function call stack reinforces the notion that a fresh set of parameters and local variables is created for a *every* call to a function. Keeping this fact in mind can help avoid a common mistake: thinking that the recursive call simply "jumps" back to the beginning of the function. For example, in the `factorial_r` function:

```
def factorial_r(n):
    '''  Compute n! recursively '''

    if n == 1:
       return 1
    elif n > 1:
       x = factorial_r(n-1)
       return n * x
```

When we reach the `factorial_r(n-1)` call, we do not change the value of `n` and "jump" back up to the beginning of the function that is currently running. Instead, an entirely new entry in the function call stack is created, with its own values for `n` and `x`. If you find yourself struggling to understand how a recursive function works, try working through the function call stack step-by-step as we did above.

13.3 Recursion vs. iteration

At this point, we have seen that we can implement the factorial function using iteration:

```
def factorial(n):
    '''  Compute n! iteratively '''

    rv = 1
    for k in range(1, n+1):
        rv = rv * k
    return rv
```

And using recursion:

```
def factorial_r(n):
    '''  Compute n! recursively '''

    if n == 1:
       return 1
    elif n > 1:
       return n * factorial_r(n-1)
```

In this simple example, the amount of code we write for both functions is roughly the same, so why would we choose recursion over iteration? In general, there are a number of algorithms and data structures that lend themselves naturally to a recursive definition. In those cases, a recursive implementation will typically be simpler and more elegant than the equivalent iterative implementation. This will become more apparent as we see more recursive algorithms.

However, you should resist the urge to use recursion as just another way of performing repetition in your code. While it is true that recursion and iteration are equally expressive (every iterative algorithm can be converted to a recursive one, and vice versa), you will usually use recursion *only* when you are faced with problems that have an inherently recursive definition.

For example, let's say we want to create a list with all the numbers between a lower bound `lb` and `ub` (both inclusive). We can do this work very simply with iteration (for the sake of argument, let's assume we cannot use the `range` function):

```
def gen_nums(lb, ub):
    '''
    Generate a list of integers from a lower bound to an upper
    bound inclusive.

    Inputs:
      lb (int): lower bound
      ub (int): upper bound

    Returns (list of ints): list of integers from lb to ub
      inclusive
    '''

    l = []
    i = lb
    while i <= ub:
       l.append(i)
       i += 1

    return l
```

```
>>> gen_nums(1,5)
[1, 2, 3, 4, 5]
```

Although we can also write a recursive solution to this problem, it is arguably not as easy to understand as the equivalent iterative solution:

```
def gen_nums_r(lb, ub):
    '''
    Generate a list of integers from a lower bound to an upper
    bound inclusive.

    Inputs:
      lb (int): lower bound
      ub (int): upper bound

    Returns (list of ints): list of integers from lb to ub
      inclusive
    '''

    if lb > ub:
       return []
    else:
       return [lb] + gen_nums_r(lb+1, ub)
```

```
>>> gen_nums_r(1,5)
[1, 2, 3, 4, 5]
```

Notice that our base case is the case when the lower bound is greater than the upper bound. In this case, the list can be obtained trivially: it will simply be an empty list. In the recursive case, we create a list whose first element is the lower bound and the rest of the list (from lb+1 to ub) is obtained recursively (we're guaranteed to reach the base case eventually because we increment lb by one in each call and get closer to ub with each recursive call). As suggested earlier, you may want to work through the function call stack for a simple example call, such as gen_nums_r(2,5), if you have trouble understanding how this function works.

Choosing between iteration and recursion can also be a question of taste. Most functional programming languages depend on recursion to repeat operations and many of them do not include any iterative constructs at all, so someone who learned how to program using a functional programming language may find the recursive version of gen_nums_r to be more elegant and easier to understand. Programmers who have learned to program under the imperative paradigm (using loops for repetition), however, tend to use recursion only when dealing with problems that have an inherently recursive definition because it is easier to translate the problem into code using that definition rather than trying to figure out the equivalent iterative solution.

13.4 Permutations (or how to think recursively)

Generating permutations is a good example of a problem that has a simple recursive definition: given n elements, the permutations of those elements are all possible ways of arranging those elements.

For example, these are all of the possible permutations of (1,2,3):

```
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1
```

And these are all of the possible permutations of (1,2,3,4):

```
1 2 3 4
1 2 4 3
1 3 2 4
1 3 4 2
1 4 2 3
1 4 3 2

2 1 3 4
2 1 4 3
2 3 1 4
2 3 4 1
2 4 1 3
2 4 3 1

3 1 2 4
3 1 4 2
3 2 1 4
3 2 4 1
3 4 1 2
```

(continues on next page)

(continued from previous page)

```
3 4 2 1

4 1 2 3
4 1 3 2
4 2 1 3
4 2 3 1
4 3 1 2
4 3 2 1
```

Look closely at both sets of permutations. In particular, look at all the permutations of (1,2,3,4) that start with 4:

```
4 1 2 3
4 1 3 2
4 2 1 3
4 2 3 1
4 3 1 2
4 3 2 1
```

These permutations are generated by taking all of the permutations of (1,2,3) and adding 4 as the first element. In fact, the permutations of (1,2,3,4) are:

- All permutations of (2,3,4) with 1 as the first element;
- All permutations of (1,3,4) with 2 as the first element;
- All permutations of (1,2,4) with 3 as the first element; plus
- All permutations of (1,2,3) with 4 as the first element.

And the permutations of (1,2,3) are:

- All permutations of (2,3) with 1 as the first element;
- All permutations of (1,3) with 2 as the first element; plus
- All permutations of (1,2) with 3 as the first element.

And so on.

Intuitively, it looks like permutations are defined recursively, because we can define permutations of n elements in terms of permutations of $n - 1$ elements. To write a recursive solution, we will generally take the following three steps.

Step #1: Determine the input(s) and output(s) to the function. This step may seem obvious given that we do this work any time we write a function, but when solving a recursive problem, it is especially important to make these decisions first.

In this case, the input to our function will be a list of elements we want to generate permutations on. The output, or return value, will be a list of permutations, with each permutation being a list of elements. For example, if we call the function with the following list:

```
[1, 2, 3]
```

We would expect it to return the following:

```
[[1, 2, 3],
 [1, 3, 2],
 [2, 1, 3],
 [2, 3, 1],
```

(continues on next page)

(continued from previous page)

```
 [3, 1, 2],
 [3, 2, 1]]
```

The reason we need to figure the types out first is because we need to ensure that both our base case and our recursive case expect the same type of inputs and return the same type of outputs.

Step #2: Determine the base case(s) for the function. We need to think about the case (or cases) where solving the problem is trivial. With permutations, this case is when we're asked to produce all the permutations of one element. In this case, there is only one such permutation: the permutation with that element.

So, we can start writing our `permutations_r` function as follows:

```
def permutations_r(p):
    '''
    Compute all the permutations of the values in p.

    Inputs:
      p (list): list of values to permute

    Returns (list of lists): permutations of p
    '''

    if len(p) == 1:
        # Base case
        return p
    else:
        # TODO: Recursive case
        return []
```

However, the above code for the base case is not correct! Let's try calling the function in such a way that we hit the base case immediately:

```
>>> permutations_r([1])
[1]
```

The function returns `[1]`, but we decided that the return value of our function would be a *list* of permutations (and `[1]` represents a single permutation). What we really want to get from the function is `[ [1] ]`. So we re-write the function as follows:

```
def permutations_r(p):
    '''
    Compute all the permutations of the values in p.

    Inputs:
      p (list): list of values to permute

    Returns (list of lists): permutations of p
    '''

    if len(p) == 1:
        # Base case
        return [ p ]
    else:
```

(continues on next page)

(continued from previous page)

```
        # TODO: Recursive case
        return []
```

```
>>> permutations_r([1])
[[1]]
```

This is a common way to mess up the base case. While it may make intuitive sense for the function to return `[1]` in the trivial case because we only have one permutation, we must make sure that the code uses inputs and outputs consistently. Otherwise, the recursive case will not work.

Make sure that your function works correctly for the base case before moving onto the next step. Test it informally using the Python interpreter on inputs that will immediately hit the base case and make sure the return value is consistent with your desired type of output. Paying careful attention to the input and output types will save you a fair amount of trouble and debugging time farther down the road.

Step #3: Determine the recursive case(s) for the function. This step can be tricky because thinking recursively doesn't come naturally to many people. We suggest taking the following approach: if you are writing a function that takes some input x, write a solution that assumes that calls to that function with a *smaller value* for x will "just work".

For example, let's go back to the factorial function:

```
def factorial_r(n):
    '''  Compute n! recursively '''

    if n == 1:
       return 1
    elif n > 1:
       return n * factorial_r(n-1)
```

After designing our base case, we wrote a recursive case that assumes that the call to `factorial_r(n-1)` will "just work". If we actually write out the entire sequence of function calls and the function call stack we can understand *how* this works, but when starting to write a recursive solution , it is better to not go down the rabbit hole of trying to trace every recursive call all the way down to the base case.

Instead, we implement the recursive case *under the assumption* that recursive calls will "just work" as long as the parameters we pass to the recursive call move us closer to the base case. This is certainly true in the case of factorial: for any positive value of `n`, calling the function with `n-1` gets us closer to the base case of `n == 1`.

So, what is the recursive case for permutations? Our function is called with a list of elements we want to generate permutations on, and our base case is reached whenever that list contains a single element. So, if the function is called like this:

```
permutations_r([1,2,3,4])
```

We can *assume* that the following calls will "just work":

```
permutations_r([2,3,4])
permutations_r([1,3,4])
permutations_r([1,2,4])
permutations_r([1,2,3])
```

Or, more generally, if the function is called with a list of size `n`, we should implement our recursive case *under the assumption* that recursive calls with a list of size `n-1` will work. Notice how this approach will eventually get us to the base case of calling the function with a list of size `n == 1`.

Now, remember the example permutations we showed above. We remarked that the permutations of (1,2,3,4) are simply:

- All permutations of (2,3,4) with 1 as the first element;
- All permutations of (1,3,4) with 2 as the first element;
- All permutations of (1,2,4) with 3 as the first element; plus
- All permutations of (1,2,3) with 4 as the first element

So, let's implement this logic under the assumption that the recursive calls (with those smaller lists) will return the correct permutations. First of all, we said our function will return a list of permutations, so let's start there:

```
def permutations_r(p):
    '''
    Compute all the permutations of the values in p.

    Inputs:
      p (list): list of values to permute

    Returns (list of lists): permutations of p
    '''

    if len(p) == 1:
        # Base case
        return [ p ]
    else:
        # Recursive case
        rv = []

        # rv will contain the list of permutations

        return rv
```

Then, for each element of p, we want to obtain the permutations resulting from removing that element.

```
def permutations_r(p):
    '''
    Compute all the permutations of the values in p.

    Inputs:
      p (list): list of values to permute

    Returns (list of lists): permutations of p
    '''

    if len(p) == 1:
        # Base case
        return [ p ]
    else:
        # Recursive case
        rv = []

        for x in p:
           p_minus_x = [v for v in p if v != x]
```

(continues on next page)

(continued from previous page)

```
           perms_without_x = permutations_r(p_minus_x)

        return rv
```

p_minus_x is simply a copy of the list p but with x removed from it. Next, we make the recursive call with that list. If p is [1,2,3,4] and x is 4, then this recursive call would return the following list:

```
[[1, 2, 3],
 [1, 3, 2],
 [2, 1, 3],
 [2, 3, 1],
 [3, 1, 2],
 [3, 2, 1]]
```

Remember: avoid the urge to go down the rabbit hole of understanding exactly what happens in this recursive call! For now, we take the approach of writing the code under the assumption that the recursive call will return exactly what we expect (according to how we specified the inputs and outputs of our function).

However, our function is not yet finished. You should also resist the urge to try and test it as soon as you've written the recursive call. We said our function has to return a list of permutations, so we need to take the permutations returned by the recursive call, add x to each permutation, and then add it to our rv list. For example, in the specific case where p is [1,2,3,4] and x is 4, we want to add the following permutations to rv:

```
[[4, 1, 2, 3],
 [4, 1, 3, 2],
 [4, 2, 1, 3],
 [4, 2, 3, 1],
 [4, 3, 1, 2],
 [4, 3, 2, 1]]
```

So we add a loop that iterates over the permutations returned by the recursive call. For each permutation, we construct a new list containing x and the values from permutation and add it to the list we are going to return:

```
def permutations_r(p):
    '''
    Compute all the permutations of the values in p.

    Inputs:
      p (list): list of values to permute

    Returns (list of lists): permutations of p
    '''

    if len(p) == 1:
        # Base case
        return [ p ]
    else:
        # Recursive case
        rv = []

        for x in p:
           p_minus_x = [v for v in p if v!=x]
           perms_without_x = permutations_r(p_minus_x)
```

(continues on next page)

(continued from previous page)

```
           for perm in perms_without_x:
               pl = [x] + perm
               rv.append(pl)
        return rv
```

At this point, our function is finished:

```
>>> permutations_r([1])
[[1]]
>>> permutations_r([1,2,3])
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
```

Recursion and Induction

If you're familiar with inductive proofs, they can also provide a good framework to think about recursion. When doing an inductive proof, we prove a statement for one or more base cases (such as $n = 0$ and $n = 1$) and once we've done that, we take the *inductive step*: assuming the statement holds for n, prove it holds for $n + 1$.

Thinking recursively is similar to doing an inductive proof: once we've implemented the base case, we then implement the recursive case under the assumption that a recursive call (with parameters that get us closer to the base case) will work.

Like we did for the factorial function, here is a version of the permutations function that prints out messages explaining what happens at each recursive call. We encourage you to play around with it so you can understand what happens when you make recursive calls. This function has several extra parameters: `n`, which is the desired number of elements in each permutation, `verbose`, which controls the printing of information about the computation, and `level`, which is used internally by the function to track the depth of the recursion for printing purposes. Notice that the code handles the fact that there might be more than one element in `p` when `n == 1`.

```
def permutations(p, n, verbose=False, level=0):
    '''
    Compute all the permutations of length n of the values in p.

    Inputs:
      p (list): list of values to permute
      n (int): desired size of the permutations
      verbose (boolean): indicates whether information about the
        computation should be printed (default: False)
      level (int): depth of the recursion (default: 0)

    Returns (list of lists): permutations of length n of values in p
    '''

    assert len(p) >= n

    if verbose: print(("    " * level) + "permutations({}, {})".format(p, n))
    if n == 1:
        rv = [[x] for x in p]
        if verbose: print(("    " * level) + "result: {}".format(rv))
        return rv
    elif len(p) == 1:
        if verbose: print(("    " * level) + "result: {}".format([p]))
```

(continues on next page)

(continued from previous page)

```
        return [p]
    else:
        rv = []
        for x in p:
            if verbose: print(("    " * level) + "{} with...".format(x))
            rem = [v for v in p if v != x]
            for perms in permutations(rem, n-1, verbose, level+1):
                rv.append([x] + perms)
        if verbose: print(("    " * level) + "result: {}".format(rv))
        if verbose: print()
        return rv
```

```
>>> ps = permutations([1,2,3], 3, verbose=True)
permutations([1, 2, 3], 3)
1 with...
    permutations([2, 3], 2)
    2 with...
        permutations([3], 1)
        result: [[3]]
    3 with...
        permutations([2], 1)
        result: [[2]]
    result: [[2, 3], [3, 2]]

2 with...
    permutations([1, 3], 2)
    1 with...
        permutations([3], 1)
        result: [[3]]
    3 with...
        permutations([1], 1)
        result: [[1]]
    result: [[1, 3], [3, 1]]

3 with...
    permutations([1, 2], 2)
    1 with...
        permutations([2], 1)
        result: [[2]]
    2 with...
        permutations([1], 1)
        result: [[1]]
    result: [[1, 2], [2, 1]]

result: [[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
```

13.5 Binary Search

Now that we've started to learn how to think recursively, we'll work through another example to highlight how a recursive algorithm can have multiple base cases and multiple recursive cases. This algorithm can be written very intuitively and elegantly using recursion, while the equivalent iterative solution is, arguably, not as intuitive.

Let's say we have the following sorted list of integers:

```
lst = [1, 3, 4, 9, 12, 13, 20, 25, 27, 31, 42, 43, 50, 51]
```

How can we determine whether a given value exists in the list? We could loop through the list (or use the `in` operator). This method is called a *linear* search. This algorithm is very simple, but its complexity is $O(n)$. For example: what if we want to check whether 51 is in the list? We have to iterate through the whole list before we get there.

However, we can exploit the fact that the numbers in the list are sorted. We can start by accessing the middle element of the list, and then constrain the rest of our search to either the first half or the second half of the list. Then, once we know which half to focus on, we can do the same thing again: choose the middle element of that sub-list, and focus our search on the first or the second half of *that* sub-list. And so on and so forth until we either find the value or run out of elements to check.

For example, let's say we want to find whether 12 is contained in the list. This list has 14 elements in it, so the index of its middle element will be 14 divided by 2: 7. The first thing we do is check whether that position contains 12 (if it does, then we're done). It, however, contains 25, which tells us that we can constrain our search to the sublist containing all the elements before 25:

```
lst2 = [1, 3, 4, 9, 12, 13, 20]
```

Now, we repeat this process. The middle element of this list is 9. We now know that we should constrain our search to all the elements that come after 9 in `lst2`:

```
lst3 = [12, 13, 20]
```

Now, the middle element is 13, so we only need to look at the elements before that element:

```
lst4 = [12]
```

The middle element of this list is 12, the value we're looking for, so we are done and the algorithm will return that 12 *is* contained in the list.

Now let's see what happens when we look for a value that *doesn't* exist in the list: 17. The first steps are similar to the ones above:

```
lst = [1, 3, 4, 9, 12, 13, 20, 25, 27, 31, 42, 43, 50, 51]
# Middle element is 25, we look at elements before it
lst2 = [1, 3, 4, 9, 12, 13, 20]
# Middle element is 9, we look at elements after it
lst3 = [12, 13, 20]
```

Now, the middle element is 13, so now we need to look at the elements after it:

```
lst4 = [20]
```

At this point, we could realize that the list contains a single element, and that this element is not the one we're looking for but, to be consistent with how we've done the other steps, we're going to look at the list of elements before the middle element, which will simply be the empty list:

```
lst5 = []
```

Notice that this feels like a recursive solution: we keep doing a search on progressively smaller lists until we reach a trivial case.

This algorithm is called *binary search*: we progressively divide the search space in half until we find the element we're seeking (or realize it doesn't exist in the list). Unlike linear search, which has a complexity of $O(n)$, the complexity of binary search is $O(\log_2 n)$ because we split the search space in half with each step.

Binary search is also an example of an algorithm with two base cases and two recursive cases. We will follow the same steps we followed in the permutations example to flesh out the exact algorithm.

Step #1: Determine the input(s) and output(s) to the function. The input to our function is going to be a list `lst` and a value `x`. We want to determine whether `x` is contained in `lst`, so it will be enough to return a boolean value (`True` or `False`). It would not be hard to modify the algorithm to return the *index* of `x` in `lst`, but we will stick with returning a boolean value for simplicity.

Step #2: Determine the base case(s) for the function. In the examples we worked through above, we identified *two* base cases:

1. If `lst` is the empty list, then we return `False`.
2. If the middle element of `lst` is `x`, then we return `True`.

We can write this in code like this:

```
def binary_search(lst, x):
    '''
    Does x occur in lst?

    Inputs:
      lst (list of ints): sorted list of values
      x (int): value to find

    Returns (boolean): return True if x occurs in lst, and False
      otherwise.
    '''

    if len(lst) == 0:
        return False
    else:
        middle = len(lst)//2

        if lst[middle] == x:
            return True
        else:
            # TODO: Recursive cases
            pass
```

Step #3: Determine the recursive case(s) for the function. If we do not hit one of the base cases, we need to do a binary search on a sublist of `lst`. If `x` is less than the middle element of `lst`, then we search the sublist containing all the elements to the left of `lst` (i.e., `lst[:middle]`). If `x` is greater than the middle element of `lst`, then we search the sublist containing all the elements to the right of `lst` (i.e., `lst[middle+1:]`).

As with the permutations example, we make the recursive call under the assumption that it will "just work" as long as we pass values that get us progressively closer to the base cases (which we do by passing progressively smaller lists in each recursive call).

Our code with the recursive cases will look like this:

```python
def binary_search(lst, x):
    '''
    Does x occur in lst?

    Inputs:
      lst (list of ints): sorted list of values
      x (int): value to find

    Returns (boolean): return True if x occurs in lst, and False
      otherwise.
    '''

    if len(lst) == 0:
        return False
    else:
        middle = len(lst)//2

        if lst[middle] == x:
            return True
        elif lst[middle] > x:
            return binary_search(lst[0:middle], x)
        elif lst[middle] < x:
            return binary_search(lst[middle+1:], x)
```

A common pitfall

A common pitfall when writing recursive functions is to make the recursive call correctly, but then not do anything with the return value of that recursive call. A common mistake when writing the above code would be to write the recursive cases like this:

```python
elif lst[middle] > x:
   binary_search(lst[0:middle], x)
elif lst[middle] < x:
   binary_search(lst[middle+1:], x)
```

The recursive calls are correct, but we are not using their return value. In the permutations example, we took the permutations returned by the recursive call to create more permutations, and then returned those permutations. Here, we directly return whatever the recursive call returns.

If we run this function, we can see it behaves as expected:

```
>>> lst = [1, 3, 4, 9, 12, 13, 20, 25, 27, 31, 42, 43, 50, 51]
>>> binary_search(lst, 12)
True
>>> binary_search(lst, 25)
True
>>> binary_search(lst, 9)
True
>>> binary_search(lst, 17)
False
>>> binary_search(lst, -10)
```

(continues on next page)

(continued from previous page)

```
False
>>> binary_search(lst, 100)
False
```

Earlier we said that the complexity of binary search is $O(\log_2 n)$, but that is actually not true of the implementation we have provided above. The reason is subtle: We slice the list in each recursive call, and slicing makes a copy of the slice, so we're using more space and time than necessary. These are things we need to be mindful of when writing code: no operation comes for free, and even seemingly trivial operations have costs that can add up.

In fact, if we use IPython to test the runtime of our algorithm, we'll see it actually take *longer* than a linear search!

```
In [1]: import random

In [2]: large_list = list(range(1000000))

In [3]: %timeit large_list.index(random.randint(0,1000000-1))
100 loops, best of 3: 4.01 ms per loop

In [4]: %timeit binary_search(large_list, random.randint(0,1000000-1))
100 loops, best of 3: 7.69 ms per loop
```

We can solve this problem by making sure that we pass the complete list in each recursive call (remember: this approach will pass a *reference* to the list, not a copy, so we don't incur the cost of making copies of the list) and adding two parameters, `lb` and `ub`, that specify the lower and upper bound of the sublist we will be searching within (`lb` inclusive, `ub` exclusive). We've also added a `verbose` parameter to print messages that can help us see how the recursion unfolds.

So, our function becomes this:

```
def binary_search_r(lst, x, lb, ub, verbose):
    '''

    Does x occur in lst between the indexes lb (inclusive) and ub
    (exclusive)?

    Inputs:
      lst (list of ints): sorted list of values
      x (int): value to find
      lb (int): lower bound (inclusive)
      ub (int): upper bound (exclusive)
      verbose (boolean): indicates whether information about the
        computation should be printed

    Returns (boolean): return True if x occurs in lst, and False
      otherwise.
    '''

    if verbose:
        print("binary_search_r(lst, {}, {}, {})".format(x, lb, ub))

    if (lb >= ub):
        # out of values to consider in the list
        return False
    else:
```

(continues on next page)

(continued from previous page)

```
        middle = (lb + ub)//2

        if verbose: print("    middle = {}".format(middle))

        if (lst[middle] == x):
            return True
        elif (lst[middle] > x):
            return binary_search_r(lst, x, lb, middle, verbose)
        elif (lst[middle] < x):
            return binary_search_r(lst, x, middle+1, ub, verbose)
```

Notice how some of the operations have been re-written in terms of `lb` and `ub`. For example, the first base case now checks whether the lower bound is greater than or equal to the upper bound (which would be equivalent to searching on an empty list). As written, this function would require passing `0` as the lower bound and `len(lst)` as the upper bound any time we wanted to search through the entire list. In such cases, we can write a *wrapper* function to make the initial call to the function:

```
def binary_search_alt(lst, x, verbose = False):
    '''
    Does x occur in lst?

    Inputs:
      lst (list of ints): sorted list of values
      x (int): value to find
      verbose (boolean): indicates whether information about the
        computation should be printed (default: False)

    Returns (boolean): return True if x occurs in lst, and False
      otherwise.
    '''

    return binary_search_r(lst, x, 0, len(lst), verbose)
```

Let's give the function a try:

```
>>> lst = [1, 3, 4, 9, 12, 13, 20, 25, 27, 31, 42, 43, 50, 51]
>>> binary_search_alt(lst, 12, verbose=True)
binary_search_r(lst, 12, 0, 14)
    middle = 7
binary_search_r(lst, 12, 0, 7)
    middle = 3
binary_search_r(lst, 12, 4, 7)
    middle = 5
binary_search_r(lst, 12, 4, 5)
    middle = 4
True
>>> binary_search_alt(lst, 17, verbose=True)
binary_search_r(lst, 17, 0, 14)
    middle = 7
binary_search_r(lst, 17, 0, 7)
    middle = 3
binary_search_r(lst, 17, 4, 7)
```

(continues on next page)

(continued from previous page)

```
    middle = 5
binary_search_r(lst, 17, 6, 7)
    middle = 6
binary_search_r(lst, 17, 6, 6)
False
```

A common pitfall

In the above implementation, the values of `lst`, `x`, and `verbose` are the same in all recursive calls, and we modify only one of `lb` or `ub` in each recursive call. While this may seem inefficient, the cost of passing these values as parameters is negligible compared to the rest of the algorithm. Resist the urge to make parameters like `lst`, `x`, and `verbose` global variables simply because their values will be the same in all the recursive calls.

Now, if we test the performance of this version, we'll see that we do get a notable running time improvement compared to linear search:

```
In [1]: %timeit large_list.index(random.randint(0,1000000-1))
100 loops, best of 3: 3.73 ms per loop

In [2]: %timeit binary_search_alt(large_list, random.randint(0,1000000-1))
100000 loops, best of 3: 5.58 µs per loop
```

Finally, let's look at the iterative version of binary search:

```
def binary_search_iter(lst, x):
    '''
    Does x occur in lst?

    Inputs:
      lst (list of ints): sorted list of values
      x (int): value to find

    Returns (boolean): return True if x occurs in lst, and False
      otherwise.
    '''

    lb = 0
    ub = len(lst)

    while lb < ub:
        middle = (lb + ub)//2

        if (lst[middle] == x):
            return True
        elif (lst[middle] > x):
            ub = middle
        elif (lst[middle] < x):
            lb = middle + 1

    return False
```

We can see that it works as expected:

```
>>> lst = [1, 3, 4, 9, 12, 13, 20, 25, 27, 31, 42, 43, 50, 51]
>>> binary_search_iter(lst, 12)
True
>>> binary_search_iter(lst, 25)
True
>>> binary_search_iter(lst, 9)
True
>>> binary_search_iter(lst, 17)
False
>>> binary_search_iter(lst, -10)
False
>>> binary_search_iter(lst, 100)
False
```

The amount of code is not substantially different from that of the recursive solution but it is arguably not as intuitive as the recursive solution. Because binary search is defined recursively, its recursive implementation more closely mirrors its actual definition. With iteration, we are simply taking the recursive implementation and shoehorning it into a while loop. In this case, we only required a single while loop but more complex recursive algorithms cannot be restated iteratively as easily.

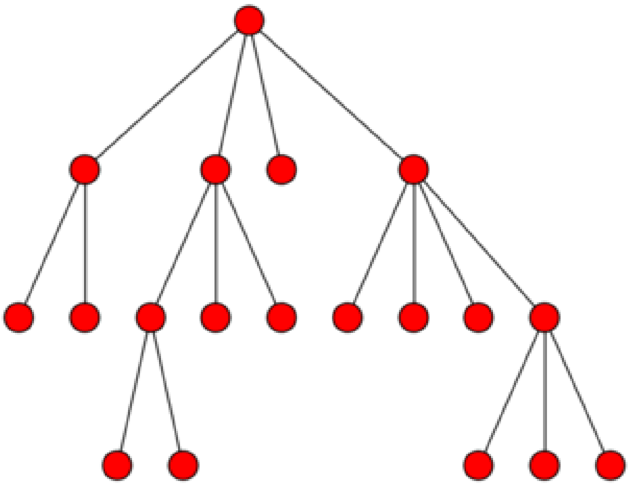
CHAPTER

FOURTEEN

TREES

In the previous chapter, we introduced recursion as a different way of repeating operations, allowing us to write *recursive algorithms* to solve certain problem. Some data structures, like the *tree* data structure we will discuss in this chapter, also rely on recursion and, specifically, the definition of the data structure itself is recursive.

A *tree* is a recursively-defined data structure that has multiple applications in computing. Informally, we can think of it as a root node that has multiple nodes as “branches”, and where each of these nodes, in turn, can have “branches” of its own. More formally, a tree is a rooted acyclic undirected graph. For example, this is a tree:



Notice how there is a root node at the top (unlike real trees, computer trees are drawn with the root at the top), and this node has four “child” nodes. The right-most child node, in turn, has four child nodes of its own. Etc.

But why is this a recursively-defined data structure? If we look just at the right-most child node of the root node, we could “prune” it from the root and it would still be a tree:

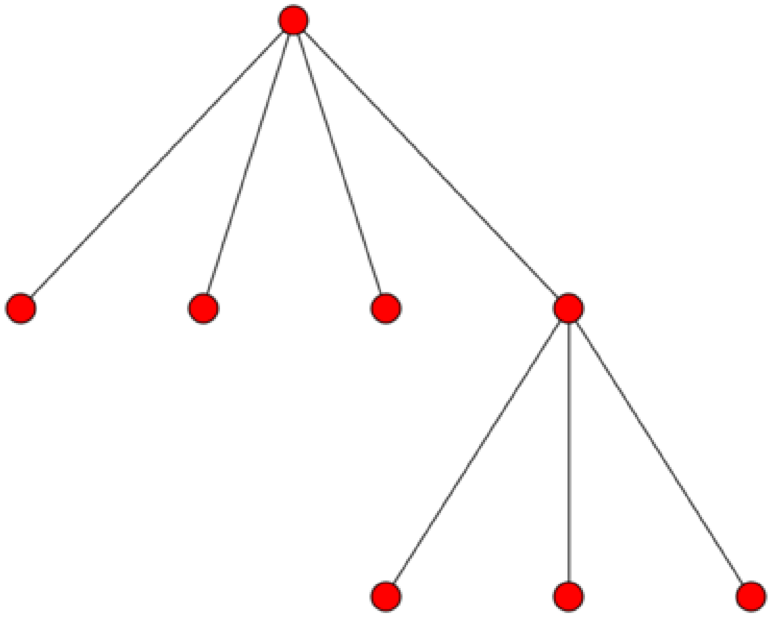
So, informally, a tree is a root node that is connected to other trees. More formally, we could define a tree as…

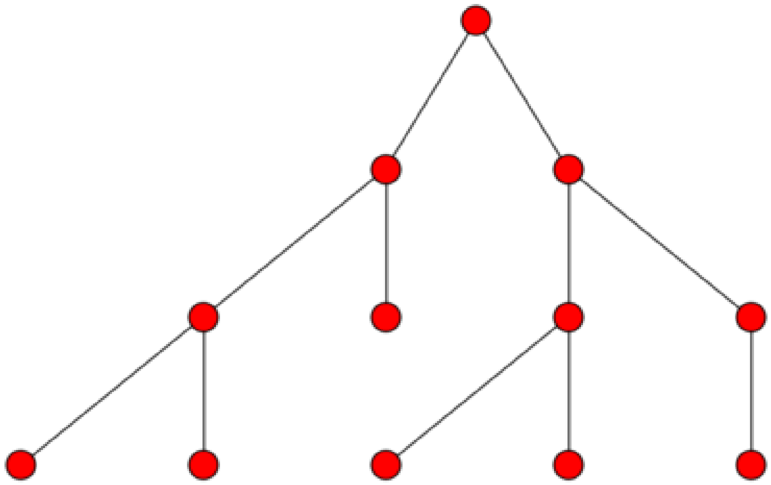
> A node with $0 \dots N$ subtrees (or *children*).

Like a recursive algorithm, we have a base case and a recursive case. The base case is when we encounter a node with 0 subtrees (which is called a *leaf node*), and the recursive case is when we encounter a node that has other trees as subtrees.

Under this definition, when $N = 2$, we have what is called a **binary tree** (we will talk about these in more detail later).

Finally, it is common for each node in a tree to have one or more associated values. We will specifically consider each node to have an associated key/value pair. As we’ll see later on, there are certain algorithms that rely on trees where each node’s key is unique (but the value is not).





14.1 Using Trees

We are going to start by presenting trees as opaque data structures: for now, we won't worry about their internal representation or how they work internally, and will interact with them through an API. We provide a `tree` module that contains a `Tree` class which we can use to work with trees. You can find this module in our *example code* under *functions/trees/tree.py*.

```
from tree import Tree
```

In particular, we will have access to the following methods:

```
# Create a tree with a root node with the given key/value pair,
# and no children. The key is required but the value is optional
# (if ommitted, it will just be None)
t = Tree(k,v)

# Properties to access the key and value of a tree (for reading and writing)
t.key
t.value

# Take an existing tree t2 and add it as a child to t
t.add_child(t2)

# Read-only property that allows us to iterate over the
# children of a tree
t.children

# Read-only property that returns the number of children a tree has
t.num_children

# Plot or print a tree
t.plot()
t.print()
```

Let's create a tree (for now, we'll leave the values blank by setting them to `None`):

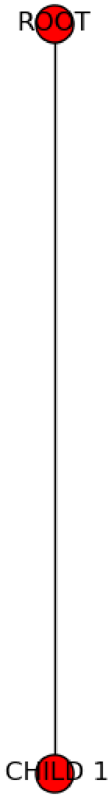
```
>>> t = Tree("ROOT")
```

Right now, this tree is just a single node with no subtrees:

We can add a child subtree using the `add_child` method:

```
>>> t1 = Tree("CHILD 1")
>>> t.add_child(t1)
```

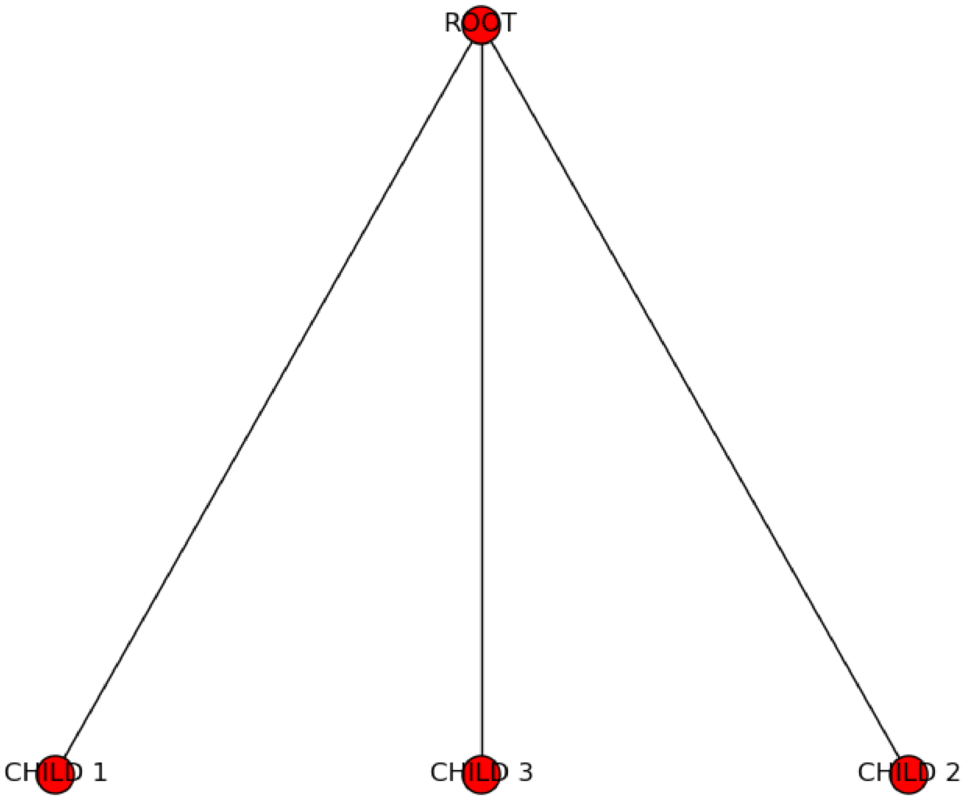
And now we have the “ROOT” node with a single “CHILD 1” child node:



If we add two more children to the root:

```
>>> t2 = Tree("CHILD 2")
>>> t3 = Tree("CHILD 3")
>>> t.add_child(t2)
>>> t.add_child(t3)
```

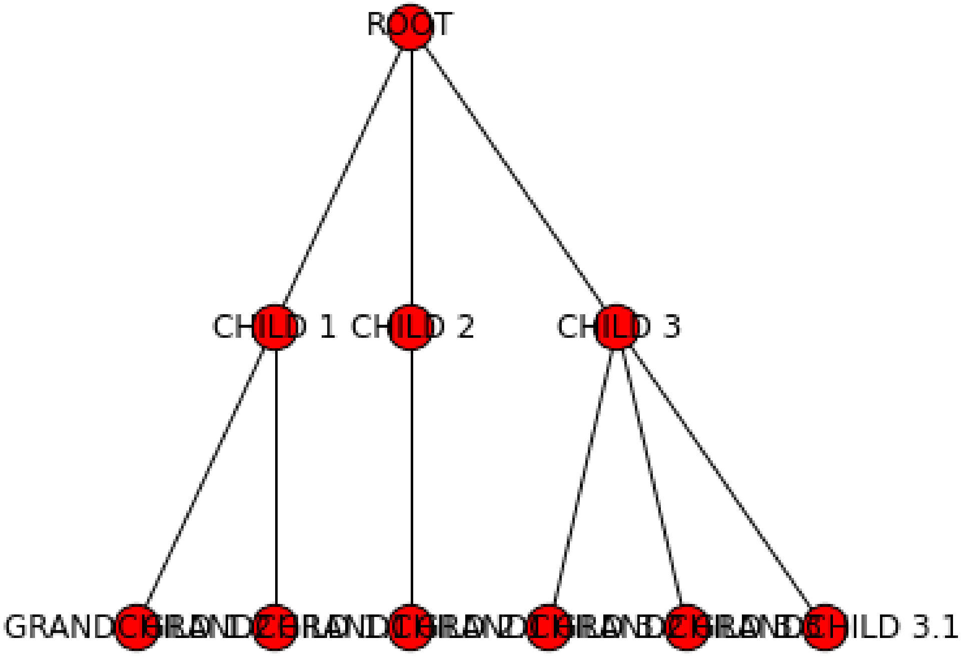
Then this starts looking more like a tree:



Finally, let's add more subtrees to the child trees we just created:

```
>>> t1.add_child( Tree("GRANDCHILD 1.1") )
>>> t1.add_child( Tree("GRANDCHILD 1.2") )
>>> t2.add_child( Tree("GRANDCHILD 2.1") )
>>> t3.add_child( Tree("GRANDCHILD 3.1") )
>>> t3.add_child( Tree("GRANDCHILD 3.2") )
>>> t3.add_child( Tree("GRANDCHILD 3.3") )
```

And the resulting tree looks like this:



We can use the `print` method to get a text representation of the tree:

```
>>> t.print()

ROOT: None
   │
   ├──CHILD 1: None
   │      │
   │      ├──GRANDCHILD 1.1: None
   │      │
   │      └──GRANDCHILD 1.2: None
   │
   ├──CHILD 2: None
   │      │
   │      └──GRANDCHILD 2.1: None
   │
   └──CHILD 3: None
          │
          ├──GRANDCHILD 3.1: None
          │
          ├──GRANDCHILD 3.2: None
          │
          └──GRANDCHILD 3.3: None
```

Notice how, if we call this method on `t1` (one of the children of the `t` tree), we get only the root of `t1` and its children:

```
>>> t1.print()

CHILD 1: None
  │
  ├──GRANDCHILD 1.1: None
  │
  └──GRANDCHILD 1.2: None
```

If we want to do this task for every child of `t`, we can use the `children` property to iterate over all of `t`'s child subtrees:

```
>>> for st in t.children:
...     st.print()
...     print()
...

CHILD 1: None
  │
  ├──GRANDCHILD 1.1: None
  │
  └──GRANDCHILD 1.2: None


CHILD 2: None
  │
  └──GRANDCHILD 2.1: None


CHILD 3: None
  │
  ├──GRANDCHILD 3.1: None
  │
  ├──GRANDCHILD 3.2: None
  │
  └──GRANDCHILD 3.3: None

```

Now, let's say we want to write a function to process the contents of a tree in some way. Because of their recursive nature, trees are easier to process using a recursive algorithm. For example, if we wanted to print the key of every node in the tree, we would use the following function:

```
def traverse(t):
    if t.num_children == 0:
        # BASE CASE: Leaf node. Print its key
        print(t.key)
    else:
       # RECURSIVE CASE: A node with at least one
       # subtree. Print the key of the tree's root
       # node, and then process the subtrees
       # recursively.
       print(t.key)
       for st in t.children:
           traverse(st)
```

```
>>> traverse(t)
ROOT
CHILD 1
GRANDCHILD 1.1
GRANDCHILD 1.2
CHILD 2
GRANDCHILD 2.1
CHILD 3
GRANDCHILD 3.1
GRANDCHILD 3.2
GRANDCHILD 3.3
```

In this case, we defined a base case and a recursive case that matches the recursive definition of the data structure: if we encounter a leaf node, we just print its key and we don't need to recurse further (because there are no subtrees) but, if we encounter a tree with subtrees, we print the root node's key, and then recursively process the tree's subtrees.

We can also rewrite this algorithm like this:

```
def traverse(t):
    print(t.key)

    for st in t.children:
        traverse(st)
```

```
>>> traverse(t)
ROOT
CHILD 1
GRANDCHILD 1.1
GRANDCHILD 1.2
CHILD 2
GRANDCHILD 2.1
CHILD 3
GRANDCHILD 3.1
GRANDCHILD 3.2
GRANDCHILD 3.3
```

These two implementations accomplish the exact same thing, but the second one doesn't explicitly separate the base case from the recursive case. Instead, if implicitly prevents a recursive call from happening when `t` is a leaf node, because `t.children` will return an empty list in this case (and `traverse` won't be called recursively, since the body of the `for` loop will never be run).

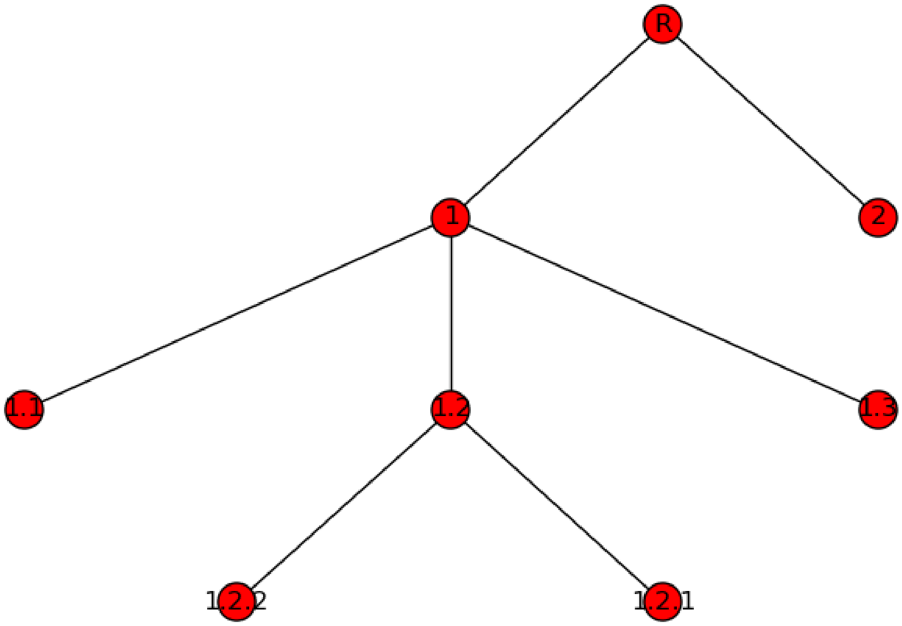
Similarly, we can recursively compute the *height* of a tree, which is the longest distance (in number of edges) from the root node to any leaf node. For example, the height of this tree is three:

```
>>> tt = Tree("R")
>>> t1 = Tree("1")
>>> t2 = Tree("2")
>>> tt.add_child(t2)
>>> tt.add_child(t1)
>>> t11 = Tree("1.1")
>>> t12 = Tree("1.2")
>>> t13 = Tree("1.3")
>>> t1.add_child(t11)
>>> t1.add_child(t12)
```

(continues on next page)

(continued from previous page)

```
>>> t1.add_child(t13)
>>> t121 = Tree("1.2.1")
>>> t122 = Tree("1.2.2")
>>> t12.add_child(t121)
>>> t12.add_child(t122)
```



If we wanted to compute the height of the tree, one approach we could take is to first find all the leaf nodes, compute the distance from each leaf node to the root, and then take the maximum of those distances.

However, we can solve this in a much more simple and elegant way by exploiting the recursive nature of the data structure:

1. Base case: A leaf node has a height of zero, because it is its own root, and thus there are no edges separating it and any other nodes.
2. Recursive case: If we have a tree with a root node and subtrees, then we know that the height must be *at least* one, because there is one edge separating the root node and each of its subtrees. We just need to find the heights of all the subtrees (by making a recursive call), find the maximum of all those heights, and add one to it.

So, we have the following:

```
def height(t):
    if t.num_children == 0:
        return 0
```

(continues on next page)

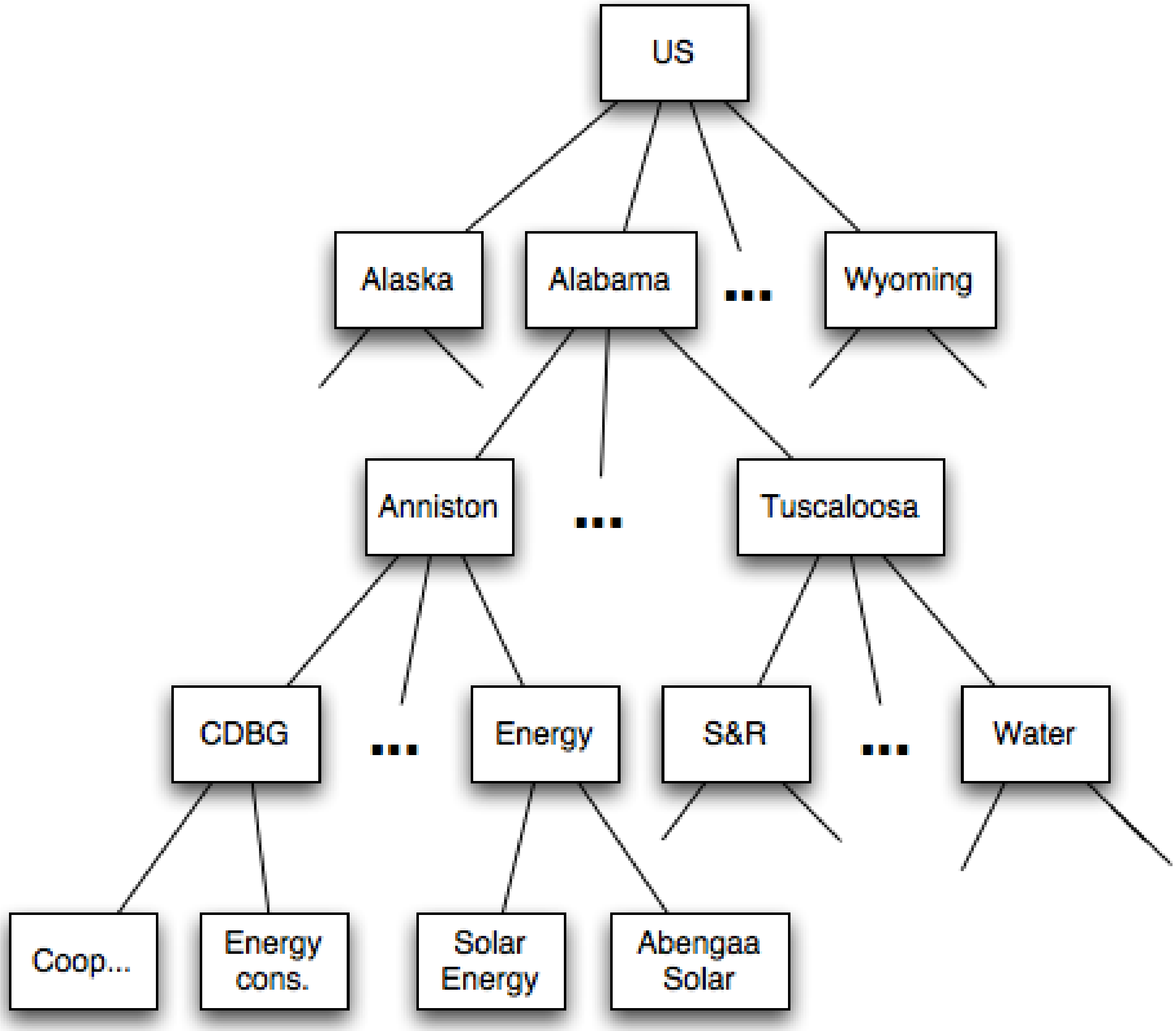
(continued from previous page)

```
    else:
        subtree_heights = [height(st) for st in t.children]
        return 1 + max(subtree_heights)
```

```
>>> height(t)
2
```

14.2 Stimulus Package data

Trees can be very useful when processing data that is structured hierarchically. For example, in February 2009, Congress passed the American Recovery and Reinvestment Act (aka “the Stimulus Package”), which provided funds for many different purposes, including infrastructure projects. The stimulus data can be viewed as a tree: the root of the tree represents the whole country, the interior nodes in the tree represent states, cities, and categories of projects, and the leaves represent specific projects.



The data about the funds allocated by the stimulus is provided in a CSV file with the following fields:

1. City
2. State
3. Project description
4. Jobs created by project
5. Project cost
6. Project program

For example, here are four entries in the file corresponding to Addison, IL:

```
Addison;IL;Resurfacing of approximately 10 miles of various industrial streets within␣
↪the village. Our industrial streets are typical 2-lanes, 44 back to back of curb and␣
↪gutter. It will be resurfaced with 1.5 inches  of bituminous asphalt, some storm␣
↪sewers,;60;12200000;Streets/Roads;[page]
Addison;IL;Resurfacing of approximately 10 miles of various residential streets within␣
↪the village. Our residential streets are typical 2-lanes, 34 back to back of curb and␣
↪gutter. It will be resurfaced with 1.5 inches  of bituminous asphalt, some storm␣
↪sewers,;60;4400000;Streets/Roads;[page]
Addison;IL;Resurfacing of Swift Road from US 20 to Collins Avenue, an approximate 4 mile␣
↪long village minor arterial roadway with a five lane cross section (the fifth lane␣
↪dedicated as a planted median strip or left turn lane) that carries about 20,000␣
↪vehicles;60;9245000;Streets/Roads;[page]
Addison;IL;Provide water to the 53 Estetes subdivision by the installation of eight␣
↪inches diameter watermains and stubs, valve vaults, fire hydrants, valve boxes,␣
↪bituminous patching and restoration of disturbed areas such as parkways, driveways,␣
↪ditches and draina;50;4400000;Water;[page]
```

However, instead of processing the data row by row, we can load it into a tree and process it hierarchically. As shown in the diagram above, the root of the tree will be the US as a whole, the children of the root will be the states, the children of the state nodes will be the cities, etc. The key of each node will be a description, and the associated value for the node will be a cost.

We provide a `stimulus` module that includes several functions to work with the stimulus data in a tree-like way:

```
import stimulus
```

You can find this module in our *example code* under *functions/trees/stimulus.py*.

In particular, the `read_stimulus_file` will return a `Tree` object. We'll start by loading just the Addison, IL data:

```
>>> stimt = stimulus.read_stimulus_file("data/Addison.txt")
```

We can now print the tree using the `Tree`'s `print` method. We use its `vformat` parameter to print the node values, which represent costs, as dollar amounts:

```
>>> stimt.print(vformat="${:,}")

US: $0
 │
 └──IL: $0
     │
     └──Addison: $0
```

(continues on next page)

(continued from previous page)

```
        │
        ├──Streets/Roads: $0
        │  │
        │  ├──Resurfacing of approximately 10 miles of various industrial
        │  │  streets within the village. Our industrial streets are typical
        │  │  2-lanes, 44 back to back of curb and gutter. It will be resurfaced
        │  │  with 1.5 inches  of bituminous asphalt, some storm sewers,:
        │  │  $12,200,000
        │  │
        │  ├──Resurfacing of approximately 10 miles of various residential
        │  │  streets within the village. Our residential streets are typical
        │  │  2-lanes, 34 back to back of curb and gutter. It will be resurfaced
        │  │  with 1.5 inches  of bituminous asphalt, some storm sewers,:
        │  │  $4,400,000
        │  │
        │  └──Resurfacing of Swift Road from US 20 to Collins Avenue, an
        │     approximate 4 mile long village minor arterial raodway with a five
        │     lane cross section (the fifth lane dedicated as a planted median
        │     strip or left turn lane) that carries about 20,000 vehicles:
        │     $9,245,000
        │
        └──Water: $0
           │
           └──Provide water to the 53 Estetes subdivision by the installation of
              eight inches diameter watermains and stubs, valve vaults, fire
              hydrants, valve boxes, bituminous patching and restoration of
              disturbed areas such as parkways, driveways, ditches and draina:
              $4,400,000
```

The module also includes a `print_stimulus_tree` function that will print the tree in the manner shown above.

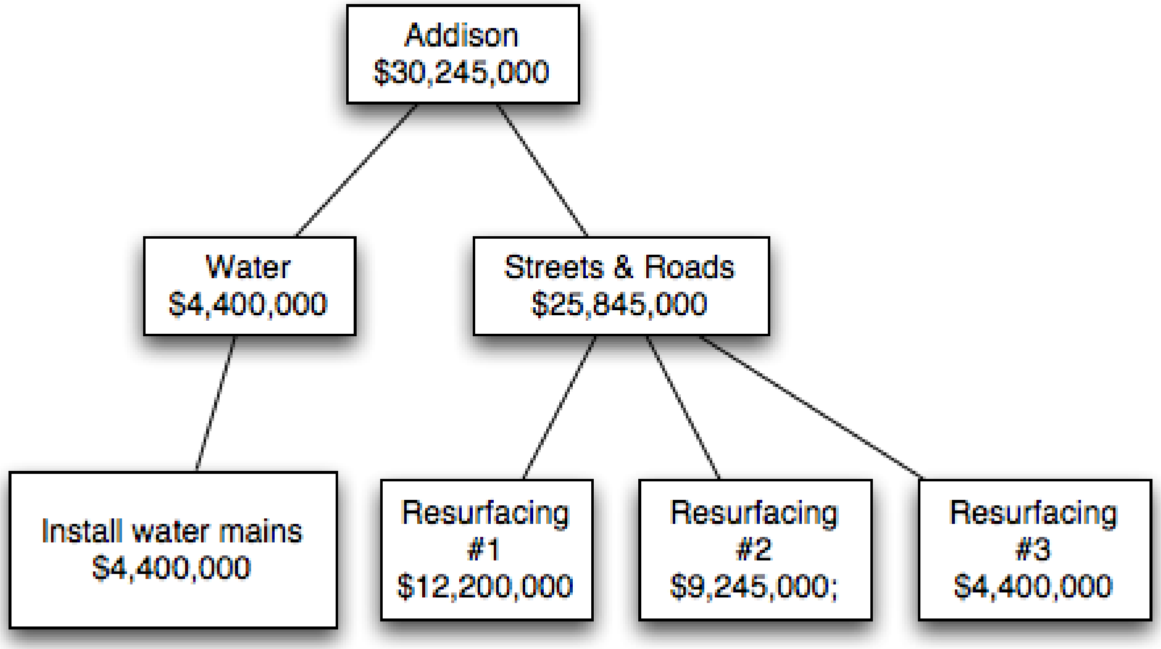
Notice how the tree includes the cost values for the individual projects, since that's the data contained in the file. It would be nice if we could aggregate these values so, for example, the `IL` node contained the sum of all the project nodes under it:

Because trees are a recursive data structure, we can easily implement this task with a recursive algorithm:

```python
def aggregate_stimulus_values(t):
    if t.num_children == 0:
        # We've reached a leaf node. The value of the
        # node will be the cost of a project
        return t.value
    else:
        # We're not at a leaf node yet, which means
        # we need to aggregate the values from all
        # the child trees
        total_cost = 0
        for st in t.children:
            total_cost += aggregate_stimulus_values(st)

        # Once we have the total cost, we modify the value of this
        # tree's root node to reflect that total cost.
        t.value = total_cost
```

(continues on next page)



(continued from previous page)

```
        return total_cost
```

```
>>> aggregate_stimulus_values(stimt)
30245000
>>> stimulus.print_stimulus_tree(stimt)

US: $30,245,000
  │
  └──IL: $30,245,000
     │
     └──Addison: $30,245,000
        │
        ├──Streets/Roads: $25,845,000
        │  │
        │  ├──Resurfacing of approximately 10 miles of various industrial
        │  │  streets within the village. Our industrial streets are typical
        │  │  2-lanes, 44 back to back of curb and gutter. It will be resurfaced
        │  │  with 1.5 inches  of bituminous asphalt, some storm sewers,:
        │  │  $12,200,000
        │  │
        │  ├──Resurfacing of approximately 10 miles of various residential
        │  │  streets within the village. Our residential streets are typical
        │  │  2-lanes, 34 back to back of curb and gutter. It will be resurfaced
        │  │  with 1.5 inches  of bituminous asphalt, some storm sewers,:
        │  │  $4,400,000
        │  │
```

(continues on next page)

(continued from previous page)

```
        │  └──Resurfacing of Swift Road from US 20 to Collins Avenue, an
        │     approximate 4 mile long village minor arterial raodway with a five
        │     lane cross section (the fifth lane dedicated as a planted median
        │     strip or left turn lane) that carries about 20,000 vehicles:
        │     $9,245,000
        │
        └──Water: $4,400,000
           │
           └──Provide water to the 53 Estetes subdivision by the installation of
              eight inches diameter watermains and stubs, valve vaults, fire
              hydrants, valve boxes, bituminous patching and restoration of
              disturbed areas such as parkways, driveways, ditches and draina:
              $4,400,000
```

Now, we can run this on the entire dataset:

```
>>> stimt = stimulus.read_stimulus_file("data/Stimulus.txt")
>>> stimulus.print_stimulus_tree(stimt, maxdepth=2)

US: $0
  │
  ├──AK: $0
  │
  ├──AL: $0
  │
  ├──AR: $0
  │
  ├──AZ: $0
  │
  ├──CA: $0
  │
  ├──CO: $0
  │
  ├──CT: $0
  │
  ├──DC: $0
  │
  ├──DE: $0
  │
  ├──FL: $0
  │
  ├──GA: $0
  │
  ├──HI: $0
  │
  ├──IA: $0
  │
  ├──ID: $0
  │
  ├──IL: $0
  │
  ├──IN: $0
```

(continues on next page)

(continued from previous page)

```
  │
  ├──KS: $0
  │
  ├──KY: $0
  │
  ├──LA: $0
  │
  ├──MA: $0
  │
  ├──MD: $0
  │
  ├──ME: $0
  │
  ├──MI: $0
  │
  ├──MN: $0
  │
  ├──MO: $0
  │
  ├──MS: $0
  │
  ├──MT: $0
  │
  ├──NC: $0
  │
  ├──ND: $0
  │
  ├──NE: $0
  │
  ├──NJ: $0
  │
  ├──NM: $0
  │
  ├──NV: $0
  │
  ├──NY: $0
  │
  ├──OH: $0
  │
  ├──OK: $0
  │
  ├──OR: $0
  │
  ├──PA: $0
  │
  ├──PR: $0
  │
  ├──RI: $0
  │
  ├──SC: $0
  │
  ├──SD: $0
```

(continues on next page)

(continued from previous page)

```
  │
  ├──TN: $0
  │
  ├──TX: $0
  │
  ├──UT: $0
  │
  ├──VA: $0
  │
  ├──VT: $0
  │
  ├──WA: $0
  │
  ├──WI: $0
  │
  ├──WV: $0
  │
  └──WY: $0
>>> aggregate_stimulus_values(stimt)
149758339820
>>> stimulus.print_stimulus_tree(stimt, maxdepth=2)

US: $149,758,339,820
  │
  ├──AK: $415,682,000
  │
  ├──AL: $3,675,416,025
  │
  ├──AR: $1,059,150,740
  │
  ├──AZ: $5,574,053,214
  │
  ├──CA: $23,194,447,835
  │
  ├──CO: $2,447,922,036
  │
  ├──CT: $2,650,247,958
  │
  ├──DC: $91,700,000
  │
  ├──DE: $52,000,000
  │
  ├──FL: $15,644,718,735
  │
  ├──GA: $2,622,606,973
  │
  ├──HI: $2,390,826,455
  │
  ├──IA: $185,815,080
  │
  ├──ID: $884,584,750
  │
```

(continues on next page)

(continued from previous page)

```
  ├──IL: $3,108,484,657
  │
  ├──IN: $2,598,965,295
  │
  ├──KS: $528,306,310
  │
  ├──KY: $1,519,552,354
  │
  ├──LA: $3,852,297,704
  │
  ├──MA: $1,072,920,450
  │
  ├──MD: $405,439,000
  │
  ├──ME: $219,461,480
  │
  ├──MI: $2,761,766,561
  │
  ├──MN: $983,556,849
  │
  ├──MO: $3,760,293,491
  │
  ├──MS: $2,433,551,064
  │
  ├──MT: $249,272,000
  │
  ├──NC: $1,976,159,270
  │
  ├──ND: $95,217,000
  │
  ├──NE: $380,458,530
  │
  ├──NJ: $2,685,299,405
  │
  ├──NM: $2,937,146,132
  │
  ├──NV: $1,521,987,313
  │
  ├──NY: $1,272,563,174
  │
  ├──OH: $4,233,069,611
  │
  ├──OK: $1,746,435,943
  │
  ├──OR: $909,352,610
  │
  ├──PA: $4,448,759,130
  │
  ├──PR: $22,093,053,760
  │
  ├──RI: $779,277,080
  │
```

(continues on next page)

(continued from previous page)

```
   ├──SC: $1,462,423,985
   │
   ├──SD: $471,900,000
   │
   ├──TN: $338,370,000
   │
   ├──TX: $10,775,423,039
   │
   ├──UT: $1,698,190,678
   │
   ├──VA: $2,305,689,266
   │
   ├──VT: $145,075,439
   │
   ├──WA: $1,713,748,676
   │
   ├──WI: $1,234,364,263
   │
   ├──WV: $700,000
   │
   └──WY: $150,636,500
```

14.3 Internal representation

So far, we have been working with trees by using a Tree class with a series of methods and properties that are useful when writing tree-based algorithms. But how do we represent trees internally? Earlier, we defined a tree as…

> A node with $0 \ldots N$ subtrees (or *children*).

We also said that we would consider each node in a tree to store a key/value pair. So, a Tree object t must contain the information about that tree's root node, and references to the other Tree objects that are t's child subtrees. We can do this with three private attributes:

- _k: The key of the root node
- _v: The value of the root node
- _children: The list of child subtrees.

Whenever we create a Tree object, we're creating a single node with no children (so the _children list will be empty). To build the rest of the tree, we add other Tree objects to the _children list by using the add_child method, which simply does the following:

```
def add_child(self, other_tree):
    self._children.append(other_tree)
```

Let's take a look at the tree we created earlier:

```
>>> t.print()

ROOT: None
   │
   ├──CHILD 1: None
```

(continues on next page)

(continued from previous page)

```
│   │
│   ├──GRANDCHILD 1.1: None
│   │
│   └──GRANDCHILD 1.2: None
│
├──CHILD 2: None
│   │
│   └──GRANDCHILD 2.1: None
│
└──CHILD 3: None
    │
    ├──GRANDCHILD 3.1: None
    │
    ├──GRANDCHILD 3.2: None
    │
    └──GRANDCHILD 3.3: None
```

We can see that its `_children` attribute is a list with three `Tree` objects (corresponding to `CHILD 1`, `CHILD 2`, `CHILD 3`):

```
>>> t._children
[<tree.Tree object at 0x7f9ed0052e50>, <tree.Tree object at 0x7f9e8989efa0>, <tree.Tree object at 0x7f9e898a91f0>]
```

If we look at its first child's `_children` attribute, we will see it is a list with two `Tree` objects (corresponding to `GRANDCHILD 1.1` and `GRANDCHILD 1.2`):

```
>>> t._children[0]._children
[<tree.Tree object at 0x7f9e898b8a60>, <tree.Tree object at 0x7f9e898b3a60>]
```

And if we look at the `_children` attribute of either of those trees, we will see they contain empty lists, because they are leaf nodes:

```
>>> t._children[0]._children[0]._children
[]
>>> t._children[0]._children[1]._children
[]
```

So, the recursive nature of the tree data structure is embodied in the `_children` attribute, which can contain other `Tree` objects which, in turn, can contain other `Tree` objects, etc. However, we do not access values in the tree by writing endless chains of `._children[]` expressions. Instead, we exploit the recursive nature of the data structure to write recursive algorithms.

While the above representation could be enough in many cases, it would not be enough to represent represent n-trees: trees where every node has exactly n subtrees. However, this would seem to prevent us from ever reaching a leaf node (which have *no* subtrees). An n-tree is actually defined as being:

1. The null tree, or
2. A node with n subtrees.

In this definition, a "null tree" is similar to having a null set: it is a tree composed of zero nodes. A leaf node is then defined as a single node where all n subtrees of the leaf node are null trees. Our implementation of the `Tree` class, a null tree is simply a `Tree` object with all its attributes (`_k`, `_v`, and `_children`) set to `None`.

Our implementation, however, does not require using n-trees. In fact, notice how none of the examples we have seen so far actually involved using null trees. However, if we wanted to use our `Tree` class to work with an n-tree, we would just need to make sure that every time we create a `Tree`, we initialize its list of children with null trees. We will see an example of this in the next section, where we will discuss a special kind of n-tree where $n = 2$: binary search trees.

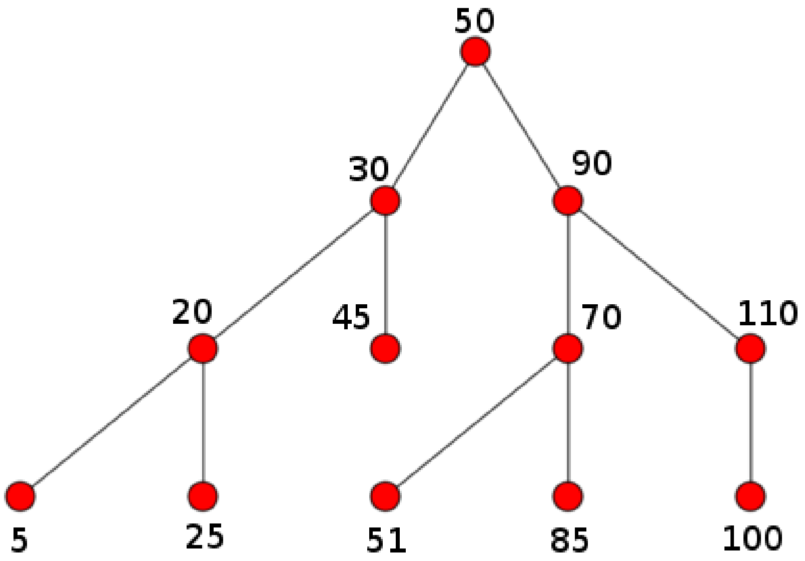
14.4 Binary Search Tree

A binary search tree, or BST, is a particular type of tree that has a number of interesting properties. It is a particular case of the general tree we've seen so far, but where every tree always has two child trees. We refer to these as the *left child* and the *right child*, and either or both of these child trees can be null.

Binary search trees also have the following property:

> Given a non-null tree with key k, all the nodes in the left child must have keys that are strictly less than k, and all the nodes in the right child must have keys that are strictly greater than k.

For example, this is a valid BST:



But this is not:

Why? Key `47` appears on the right child of `50`, which violates the BST property. Notice how, if we looked at the right child in isolation (i.e., a tree rooted at `90`), that tree *would* be a valid BST.

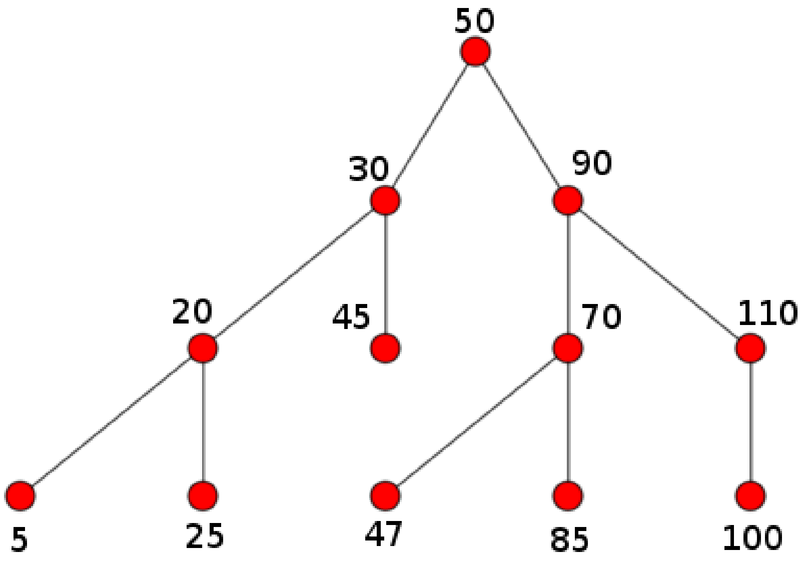
The operations on a BST are similar to those on a general tree, except we have to make sure the BST property is always maintained. So, for example, we can't create subtrees in arbitrary places like we did with the `add_child` method. Instead, we have a BST-specific `insert` method that inserts a key into the tree in such a way that the BST property is maintained. In the following examples, we will be using a `bst` module that you can find in our *example code* under *functions/trees/bst.py*.

```
from bst import BST

bt = BST()

bt.insert(50, "fifty")
bt.insert(30, "thirty")
bt.insert(5, "five")
```

(continues on next page)



(continued from previous page)

```
bt.insert(20, "twenty")
bt.insert(90, "ninety")
bt.insert(70, "seventy")
bt.insert(110, "one hundred and ten")
```

```
>>> bt.print()

50: fifty
  │
  ├──30: thirty
  │  │
  │  ├──5: five
  │  │  │
  │  │  ├──NULL
  │  │  │
  │  │  └──20: twenty
  │  │     │
  │  │     ├──NULL
  │  │     │
  │  │     └──NULL
  │  │
  │  └──NULL
  │
  └──90: ninety
     │
     ├──70: seventy
     │  │
     │  ├──NULL
     │  │
     │  └──NULL
     │
     └──110: one hundred and ten
        │
```

(continues on next page)

(continued from previous page)

```
         ├──NULL
         │
         └──NULL
```

Notice how all the leaf nodes have two null trees as children.

The BST property means that searching in trees is much more efficient than searching in lists, because we can progressively divide the search space in two, just like we did when using binary search. In fact, the algorithm to search a BST *is* binary search, except instead of splitting a list in two, we decide to look in either the left subtree or the right subtree.

The following is the implementation of the `find` method. The `__left` and `__right` properties in this code refer to the left and right subtree of the current tree.

```python
def find(self, k):
    """
    Finds a node with a given key in the tree.

    If such a key exists, it returns a tuple with
    True and the value associated with that key.

    Otherwise, it returns (False, None)
    """
    if self.is_null():
       return (False, None)

    if k < self.key:
       return self.__left.find(k)
    elif k > self.key:
       return self.__right.find(k)
    elif k == self.key:
       return (True, self.value)
```

```
>>> bt.find(90)
(True, 'ninety')
>>> bt.find(45)
(False, None)
>>> bt.find(51)
(False, None)
```

To compare the performance of a list versus a tree, let's let's create a list with 100,000 random integers, and then insert those integers into a BST:

```python
import random

MAXSIZE = 100000

nums = list(range(MAXSIZE))
random.shuffle(nums)

numst = BST()
for i in nums:
    numst.insert(i, None)
```

Now, we can use IPython's `%timeit` to compare how long it takes to find random values inside the list and inside the

BST:

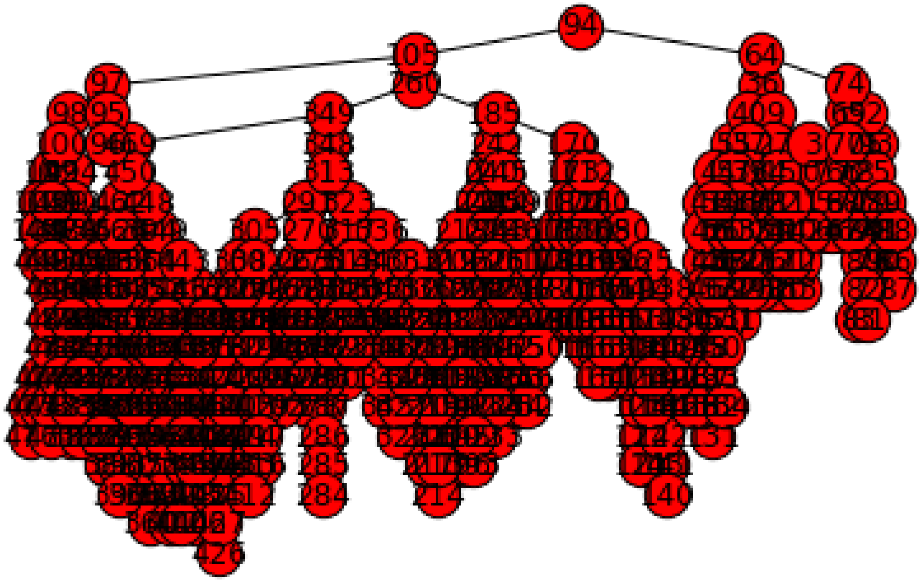
```
In [1]: %timeit nums.index(random.randint(0,MAXSIZE-1))
1000 loops, best of 3: 749 µs per loop

In [2]: %timeit numst.find(random.randint(0,MAXSIZE-1))
100000 loops, best of 3: 11.1 µs per loop
```

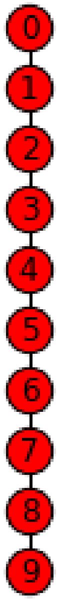
Finding in an unsorted list takes *linear* time: we may have to search all the way to the end of the list to find the value we're looking for. Finding in a tree takes *logarithmic* time, since we can use binary search to divide the search space in half in each step.

However, this requires loading all the data into the tree first. Inserting n values into a BST takes $O(n \cdot \log_2 n)$ time (each individual insertion takes $O(\log_2 n)$ time), while inserting n values into a list takes $O(n)$ time. Of course, if we're going to perform many search operations, the increased cost of loading the data into the tree (which is a one-time operation) may be worthwhile.

There is a caveat, though: finding an element in a tree takes $O(\log_2 n)$ if the tree is *balanced*. A tree is said to be balanced if the difference in the height of left and right subtree is not be greater than 1. The insertion algorithm we've implemented in our BST does not guarantee this property. In fact, our trees will usually look like this:



And, in the worse case scenario, if we insert all the values into the tree in order, we could end up with a tree that looks like this:



In the above tree, all the left subtrees are null, which means we've effectively ended up with a list. While our BST implementation makes no attempt to keep the tree balanced, there are other types of BSTs (such as AVL trees or Red-Black trees) that keep the tree balanced, at the expense of more complex insertion algorithms.

Part IV.

Working with Data

CHAPTER FIFTEEN

WORKING WITH FILES

A very common programming pattern proceeds as follows:

1. Read the contents of one or more files from disk and load the data into one or more data structures.
2. Manipulate the data in some way.
3. (Optional) Write the resulting data back to disk.

Programmers often connect multiple programs written using this pattern into data-processing pipelines.

In this chapter, we'll discuss how to work with several different file formats. We'll start with the most basic mechanisms and work our way up to higher-level tools and more complex formats.

15.1 Basic file I/O

Reading file input and writing file output is often referred to as I/O, or input/output. We'll start our discussion of basic file I/O with a simple example: load the contents of the text file `instructor-email.txt` into a sorted list. Here are the contents of the file:

```
amr@cs.uchicago.edu
borja@cs.uchicago.edu
yanjingl@cs.uchicago.edu
mwachs@cs.uchicago.edu
dupont@cs.uchicago.edu
```

and here's the desired result:

```
["amr@cs.uchicago.edu",
 "borja@cs.uchicago.edu",
 "dupont@cs.uchicago.edu",
 "mwachs@cs.uchicago.edu",
 "yanjingl@cs.uchicago.edu"]
```

To access the contents of a file, we first need to open it using the built-in function `open`:

```
>>> f = open("instructor-email.txt")
```

The `open` function returns a data structure, known as a *file pointer*, that we can use to work with the contents of the file. It is common to use the word *file* to refer to both a file on disk and a file pointer.

Once we have a file pointer, we can perform a number of operations on it. In a sense, text files are simply very long strings that are stored on disk. So, for example, we can read the entire contents of the file into a string in a single operation using the `read` method:

```
>>> s = f.read()
>>> s
'amr@cs.uchicago.edu\nborja@cs.uchicago.edu\nyanjingl@cs.uchicago.edu\nmwachs@cs.
↪uchicago.edu\ndupont@cs.uchicago.edu\n'
```

The characters `\n` are what is known as an *escape sequence*, which in this case encodes the newline character. Notice that if we print the value of `email_address`, all of the occurrences of the `\n` escape sequence are converted into newlines as expected:

```
>>> print(s)
amr@cs.uchicago.edu
borja@cs.uchicago.edu
yanjingl@cs.uchicago.edu
mwachs@cs.uchicago.edu
dupont@cs.uchicago.edu

```

We can convert the string into a list of email addresses using the string `split` method, which, by default, breaks the string into tokens using white space (that is, spaces, tabs, newlines, etc.) as a the delimiter.

```
>>> s.split()
['amr@cs.uchicago.edu', 'borja@cs.uchicago.edu', 'yanjingl@cs.uchicago.edu', 'mwachs@cs.
↪uchicago.edu', 'dupont@cs.uchicago.edu']
```

Finally, we can call the built-in function `sorted` to to get the desired result: a sorted list of email addresses:

```
>>> email_addresses = sorted(s.split())
```

When reading from a file, the operating system keeps track of the most recent position it has read. In this case, the file pointer has already reached the end-of-file (or EOF). So, if we call `read` again, we don't get the contents of the file, we get an empty string instead:

```
>>> data = f.read()
>>> data
''
```

Once we're done working with a file, we need to close the file pointer:

```
>>> f.close()
```

Following the close, you can no longer use that file pointer to access the file. (You'd have to reopen the file to use it again.) It is important to close files to free the associated resources and, as we'll see later on when writing to files, to ensure that all of your updates to the file are written to disk.

It is easy to forget to close a file, so it is common to use a `with` statement, which guarantees that the file will be closed no matter what happens in the body of the statement.

```
>>> with open("instructor-email.txt") as f:
...     s = f.read()
...     email_addresses = sorted(s.split())
...
>>> print(email_addresses)
['amr@cs.uchicago.edu', 'borja@cs.uchicago.edu', 'dupont@cs.uchicago.edu', 'mwachs@cs.
↪uchicago.edu', 'yanjingl@cs.uchicago.edu']
```

The with statement introduces a new name, in this case, f, that refers to the file pointer returned by the call to open. At the end of the with block, file f is closed automatically.

Instead of reading the file in one chunk, we can also read it line by line. One approach is to use a for loop that iterates over a text file line by line. For example, here's some code that reads and prints each line in the instructor-email.txt file:

```
>>> with open("instructor-email.txt") as f:
...     for line in f:
...         print(line)
...
amr@cs.uchicago.edu

borja@cs.uchicago.edu

yanjingl@cs.uchicago.edu

mwachs@cs.uchicago.edu

dupont@cs.uchicago.edu

```

This result may look a bit funny to you. Why the extra empty line? Each line from the file includes a newline at the end, and print adds a newline as well. We can see the actual representation of the string using the built-in repr function:

```
>>> with open("instructor-email.txt") as f:
...     for line in f:
...         print(repr(line))
...
'amr@cs.uchicago.edu\n'
'borja@cs.uchicago.edu\n'
'yanjingl@cs.uchicago.edu\n'
'mwachs@cs.uchicago.edu\n'
'dupont@cs.uchicago.edu\n'
```

When reading lines from a file, we can use the strip method from the string library to remove leading and trailing whitespace:

```
>>> with open("instructor-email.txt") as f:
...     for line in f:
...         print(line.strip())
...
amr@cs.uchicago.edu
borja@cs.uchicago.edu
yanjingl@cs.uchicago.edu
mwachs@cs.uchicago.edu
dupont@cs.uchicago.edu
```

To accomplish our goal of creating a sorted list of email addresses, we can combine a familiar pattern for constructing lists with a use of with and a call to the list sort method.

```
>>> email_addresses = []
>>> with open("instructor-email.txt") as f:
...     for line in f:
...         email = line.strip()
```

(continues on next page)

(continued from previous page)

```
...         email_addresses.append(email)
...
...
>>> email_addresses.sort()
```

15.2 Writing data to a file

To write to a file, we must open the file in write mode (note the use of "w" as a second parameter to open to specify that we're opening the file in write mode):

```
open("names2.txt", "w")
```

It is very important to understand that when you open an existing file in write mode, *all* of its existing contents will be wiped away! If you open a file that doesn't already exist in write mode, a new file will be created.

Once we have a writable file pointer, we can append a string to the file using write. For example, after we run this code:

```
with open("names.txt", "w") as f:
    f.write("Anne Rogers\n")
    f.write("Borja Sotomayor\n")
    f.write("Yanjing Li\n")
    f.write("Matthew Wachs\n")
    f.write("Todd Dupont\n")
```

The file names.txt will contain:

```
Anne Rogers
Borja Sotomayor
Yanjing Li
Matthew Wachs
Todd Dupont
```

We could also use the print method to generate this output, which has the advantage that it will add the newline automatically. The file keyword parameter allow us to specify a file pointer as the destination of call to print.

```
>>> with open("names2.txt", "w") as f:
...     print("Anne Rogers", file=f)
...     print("Borja Sotomayor", file=f)
...     print("Yanjing Li", file=f)
...     print("Matthew Wachs", file=f)
...     print("Todd Dupont", file=f)
...
```

Internally, writes to files are often stored in a buffer and then written out to disk in batches. When you close a file, you flush any buffered data to disk. If you do not close your file, the data from the last few writes you do may remain in the buffer and thus may not get written back to disk.

Let's put these pieces together to write a function that transforms a file with a list of email addresses into a new file with the domain name (that is, @cs.uchicago.edu) stripped off.

```
>>> def strip_domain(input_filename, output_filename):
...     '''
...
...     Strip the domain names off the email address from the input
...     file and write the resulting usernames to the output file.
...
...     Inputs:
...       input_filename: (string) name of a file with email addresses
...       output_filename: (string) name for the output file.
...     '''
...
...     # Load data into a data structure (a list of strings)
...     email_addresses = []
...     with open(input_filename) as f:
...         for line in f:
...             email = line.strip()
...             email_addresses.append(email)
...
...     # Transform the data
...     usernames = []
...     for email in email_addresses:
...         username, domain = email.split("@")
...         usernames.append(username)
...
...     # Write the data
...     with open(output_filename, "w") as f:
...         for username in usernames:
...             print(username, file=f)
...
```

In this case, the operation is simple enough that we could've produced the usernames during the input loop or during the output loop, but in general, it's good to separate input, transformation, and output into separate steps.

Python's basic file I/O functionality is sufficient for working with simple files, but sometimes we need work with more complex data files. In the next few sections, we'll introduce a few existing data formats and libraries that it make it easy to work with them. In general, it is better to use an existing format and its associated libraries, if you can, than to invent your own ad hoc format.

15.3 CSV

The acronym CSV stands for Comma Separated Values. CSV files contain values separated by commas (and sometimes by other delimiters) and are typically used to represent tabular data, that is, any data that can be organized into rows, each with the same columns (or *fields*).

Here are the contents of a CSV file named `instructors.csv`:

```
id,lname,fname,email
amr,Rogers,Anne,amr@cs.uchicago.edu
borja,Sotomayor,Borja,borja@cs.uchicago.edu
yanjingl,Li,Yanjing,yanjingl@cs.uchicago.edu
mwachs,Wachs,Matthew,mwachs@cs.uchicago.edu
dupont,Dupont,Todd,dupont@cs.uchicago.edu
```

The first line is the header row; it includes the names of the columns/fields (id, lname, fname, and email). The remaining lines contain the data.

With what we've seen so far, we could just use the existing file functions to read this file and generate some simple output:

```
>>> with open("instructors.csv") as f:
...     header = f.readline() # Skip the header row
...
...     for row in f:
...         fields = row.strip().split(",")
...
...         id = fields[0]
...         last_name = fields[1]
...         first_name = fields[2]
...         email = fields[3]
...
...         print("{} {}'s e-mail is {}".format(first_name, last_name, email))
...
Anne Rogers's e-mail is amr@cs.uchicago.edu
Borja Sotomayor's e-mail is borja@cs.uchicago.edu
Yanjing Li's e-mail is yanjingl@cs.uchicago.edu
Matthew Wachs's e-mail is mwachs@cs.uchicago.edu
Todd Dupont's e-mail is dupont@cs.uchicago.edu
```

We could similarly use the existing file functions to write a CSV file.

This process, however, can be very error-prone, and it doesn't account for a number of peculiarities in the CSV file format e.g., what if a column holds a string value with a comma in it? Such a value would be represented in double-quotes, like "Hello, world!", but the above code would fail. To make this concrete, let's look at what, would happen if we added this line to the file:

```
spade,"Spade, Jr",Sam,spade@cs.uchicago.edu
```

We'd get:

```
Jr" "Spade's e-mail is Sam
```

as the output for this line, because the split method is not designed to handle commas embedded in quoted sub-strings.

Fortunately, Python includes a csv module that allows us to work with CSV files more naturally.

```
>>> import csv
```

The DictReader class allows us to iterate over the rows in a CSV file, and access each field in a row via a dictionary (with the same field names specified in the header row or using the optional fieldnames parameter).

```
>>> with open("instructors.csv") as f:
...     reader = csv.DictReader(f)
...
...     for row in reader:
...         print("{} {}'s e-mail is {}".format(row["fname"], row["lname"], row["email
↪"]))
...
Anne Rogers's e-mail is amr@cs.uchicago.edu
Borja Sotomayor's e-mail is borja@cs.uchicago.edu
```

(continues on next page)

(continued from previous page)

```
Yanjing Li's e-mail is yanjingl@cs.uchicago.edu
Matthew Wachs's e-mail is mwachs@cs.uchicago.edu
Todd Dupont's e-mail is dupont@cs.uchicago.edu
```

We can use a similar class, `DictWriter`, to write CSV file.

```
fieldnames = ["id", "lname", "fname", "email"]

with open("instructors-122.csv", "w") as f:
    writer = csv.DictWriter(f, fieldnames=fieldnames)
    writer.writeheader()

    row = {"id": "amr",
           "lname": "Rogers",
           "fname": "Anne",
           "email": "amr@cs.uchicago.edu"}

    writer.writerow(row)

    row = {"id": "mwachs",
           "lname":"Wachs",
           "fname": "Matthew",
           "email": "mwachs@cs.uchicago.edu"}

    writer.writerow(row)
```

The `csv` module also has `reader` and `writer` classes that use lists to represent rows rather than dictionaries. This allows us to interact with the fields positionally instead of by name.

15.4 JSON

JSON (JavaScript Object Notation) is a *lightweight* data-interchange format that web services commonly use. It is also used as a data storage format. JSON supports a few different types:

- Object: looks like a python dictionary with string-value pairs (string:value) separated by commas,
- Array: empty list or list of values,
- Value: string, number, object, array, true, false, null

Note the nesting of object and array in the definition of value.

An application might receive a string in JSON format from another application, like a web service, or as a file. Python has a `json` module for handling JSON data in either form. We'll start by looking at the functions that operate on strings, but first, we need to import the library.

```
>>> import json
```

The `dumps` function takes a Python data structure and encodes it in JSON format:

```
>>> l = ['baz', None, 1.0, 2]
>>> json.dumps(l)
'["baz", null, 1.0, 2]'
```

Notice that the result is a string representation of the data, with some minor differences from Python (e.g., “null” instead of “None”)

JSON allows for nested data structures and, so, the `dumps` function handles them as well:

```
>>> data = [ 'foo', {'bar': l} ]
>>> json.dumps(data)
'["foo", {"bar": ["baz", null, 1.0, 2]}]'
```

The `loads` function takes a string containing data encoded in JSON format and decodes it, yielding the corresponding Python data structures:

```
>>> json.loads("42")
42
```

```
>>> json.loads("[1,2,3]")
[1, 2, 3]
```

```
>>> json.loads('["foo", {"bar": ["baz", null, 1.0, 2]}]')
['foo', {'bar': ['baz', None, 1.0, 2]}]
```

The `load` and `dump` functions perform analogous operations, but on files. The `dump` function takes two arguments—an appropriate data structure and a file pointer—and writes a JSON encoding of the data structure to the file. Here’s a sample call:

```
>>> with open("saved_data.json", "w") as f:
...     json.dump(data, f)
...
```

that yields a file named `saved_data.json` with contents of:

```
["foo", {"bar": ["baz", null, 1.0, 2]}]
```

For a large and complex data structure, the standard encoding can be hard to read. You can use the optional `indent` parameter, which allows you to specify the number of spaces to indent for each level of nesting, and the `sort_keys` parameter to generate output that is easier to read. This code, for example,

```
>>> with open("saved_data.json", "w") as f:
...     json.dump(data, f, indent=4, sort_keys=True)
...
```

yields a version of `saved_data.json` with the following contents:

```
[
    "foo",
    {
        "bar": [
            "baz",
            null,
            1.0,
            2
        ]
    }
]
```

The `load` function takes a file pointer, reads the data from the file, and returns the decoded data structure:

```
>>> saved_data = None
>>> with open("saved_data.json") as f:
...     saved_data = json.load(f)
...
>>> saved_data
['foo', {'bar': ['baz', None, 1.0, 2]}]
```

15.5 YAML

YAML (YAML Ain't Markup Language) is, like JSON, a lightweight data format that is intended to be both machine-readable and human-readable. It uses indentation rather than braces and brackets to represent nesting. We won't describe this format in detail, other than to point out that these files are easy to read and to process automatically.

YAML is often used for configuration files, as well as files that need to be processed by a program, but also need to be readable by a non-technical user. For example, the following `rubric.yml` file could be used to provide the results of grading a programming assignment:

```
Points:
    Tests:
        Points Possible: 50
        Points Obtained: 45

    Implementing foo():
        Points Possible: 20
        Points Obtained: 10

    Implementing bar():
        Points Possible: 20
        Points Obtained: 20

    Code Style:
        Points Possible: 10
        Points Obtained: 7.5

Penalties:
    Code comments are written in Old English: -5

Bonuses:
    Worked alone: 10

Total Points: 87.5 / 100

Comments: >
    Well done!
```

As with any library, we need to import the `yaml` library before we can use it:

```
>>> import yaml
```

We can load a YAML file using `yaml.load` with a file pointer:

```
>>> with open("rubric.yml") as f:
...     rubric = yaml.load(f)
...
```

The result will be a dictionary:

```
>>> rubric
{'Points': {'Tests': {'Points Possible': 50, 'Points Obtained': 45}, 'Implementing foo()
↪': {'Points Possible': 20, 'Points Obtained': 10}, 'Implementing bar()': {'Points␣
↪Possible': 20, 'Points Obtained': 20}, 'Code Style': {'Points Possible': 10, 'Points␣
↪Obtained': 7.5}}, 'Penalties': {'Code comments are written in Old English': -5},
↪'Bonuses': {'Worked alone': 10}, 'Total Points': '87.5 / 100', 'Comments': 'Well done!\
↪n'}
```

Notice that the nesting of the dictionary reflects the nesting of the indentation above.

CHAPTER

SIXTEEN

NUMPY

NumPy is a Python library that supplies two major features. First, it provides better support for large multi-dimensional arrays. Second, it provides high-level mathematical functions that operate on them.

Unlike many other languages, Python doesn't include a native type for arrays or matrices. Instead, we could use lists, lists-of-lists, lists-of-lists-of-lists, etc. Although these data structures can be used to represent multi-dimensional arrays, working with them can get pretty messy. As an example, let's look at some data from the National Institute of Diabetes and Digestive and Kidney Diseases. (The data is available at the UCI Machine Learning Repository) During this study, information was collected for a group of patients from the Pima Indian Tribe. We can represent this data with a matrix in which each row corresponds to the data collected for an individual patient and each column corresponds to a specific feature (e.g., diastolic blood pressure) for all the patients. Here is subset of the data that includes information for ten patients for four features (plasma glucose level, diastolic blood pressure, triceps skin foldthickness, and two-hour serum insulin):

```
[[89.0, 66.0, 23.0, 94.0],
 [137.0, 40.0, 35.0, 168.0],
 [78.0, 50.0, 32.0, 88.0],
 [197.0, 70.0, 45.0, 543.0],
 [189.0, 60.0, 23.0, 846.0],
 [166.0, 72.0, 19.0, 175.0],
 [118.0, 84.0, 47.0, 230.0],
 [103.0, 30.0, 38.0, 83.0],
 [115.0, 70.0, 30.0, 96.0],
 [126.0, 88.0, 41.0, 235.0]]
```

Before applying classification or prediction algorithms, statisticians often standardize data such that each feature (or variable) in the matrix for analysis has a mean of zero and standard deviation of one. Let's consider this common pre-processing step. Specifically, the values in the standardized matrix, m' should be computed using the following formulas:

$$\mu_j = 1/N * \sum_{i=0}^{N-1} m_{i,j}$$

$$\sigma_j = \sqrt{1/N * \sum_{i=0}^{N-1} (m_{i,j} - \mu_j)^2}$$

$$m'_{i,j} = (m_{i,j} - \mu_j)/\sigma_j$$

where $m_{i,j}$ is the value of the jth feature for patient i, μ_j is the mean of the jth feature, and σ_j is its standard deviation.

We can translate these formulas into code that uses for loops and list indexing to process data that is represented as a list of lists:

```python
def compute_feature_mean(data, j):
    '''
    Compute the mean of feature (column) j

    Inputs:
      data (list of list of floats)

    Returns (float): mean of feature j
    '''

    N = len(data)
    total = 0
    for i in range(N):
        total += data[i][j]
    return total/N


def compute_feature_stdev(data, j):
    '''
    Compute the standard deviation of feature (column) j

    Inputs:
      data (list of lists of floats)

    Returns (float): standard deviation of feature j
    '''

    N = len(data)
    mu = compute_feature_mean(data, j)
    total = 0
    for i in range(N):
        total = total + (data[i][j] - mu) ** 2
    return math.sqrt(1 / N * total)

def standardize_features(data):
    '''
    Standardize features to have mean 0.0 and standard deviation
    1.0.

    Inputs:
      data (list of list of floats): data to be standardized

    Returns (list of list of floats): standardized data
    '''

    N = len(data)
    M = len(data[0])

    # initialize the result w/ NxM list of lists of zeros.
    rv = []
    for i in range(N):
        rv.append([0] * M)
```

(continues on next page)

(continued from previous page)

```
    # for each feature
    for j in range(M):
        mu = compute_feature_mean(data, j)
        sigma = compute_feature_stdev(data, j)

        # standardized feature
        for i in range(N):
            rv[i][j] = (data[i][j] - mu)/sigma

    return rv
```

While this code is straightforward, it is nowhere near as compact as the mathematics and it is easy to get the indexing wrong. Also, the functions `compute_feature_mean` and `compute_feature_stdev` are not as general as one might like. For example, they are not particularly useful for computing the mean of a list or the standard deviation of a row in a list-of-lists.

We can use list comprehensions to make the helper functions more compact, but the resulting code is still not as general as one would like:

```
def compute_feature_mean(data, j):
    '''
    Compute the mean of feature (column) j.

    Inputs:
      data (list of lists of floats)

    Returns (float): mean of feature j
    '''

    N = len(data)
    return sum([data[i][j] for i in range(N)]) / N

def compute_feature_stdev(data, j):
    '''
    Compute the standard deviation of feature (column) j.

    Inputs:
      data (list of lists of floats)

    Returns (float): standard deviation of feature j
    '''

    N = len(data)
    mu = compute_feature_mean(data, j)
    return math.sqrt(1 / N * sum([(data[i][j] - mu) ** 2 for i in range(N)]))
```

List-of-lists do not provide an easy way to operate on a subset of the elements of the data structure, such as the elements of a column, as a unit. Consequently, our functions are awkward and unwieldy. As you will see later in this chapter, NumPy's array data structure supports operations that work on the elements of an array collectively and allows programmers to apply these operations to operands of different sizes and shapes. It also provides easy ways to read and update sub-arrays.

Using these mechanisms, we can write code that is substantially more compact and resembles the underlying mathematics. The function `standardize_features`, for example, can be written as:

```
def standardize_features(data):
    '''
    Standardize features to have mean 0.0 and standard deviation
    1.0.

    Inputs:
      data (2D NumPy array of floats): data to be standardized

    Returns (2D NumPy array of floats): standardized data
    '''

    mu = data.mean(axis=0)
    sigma = data.std(axis=0)
    return (data - mu) / sigma
```

Although we do not expect you to grasp the details of this function right now, you can see that this version is more compact. Once you have finished this chapter, we hope that you will find this version easier to understand and reproduce than the loops and lists version.

16.1 Importing NumPy

Before we can use NumPy, we need to import it. Traditionally, programmers use the `as` option with `import` to import NumPy and assign it a short alias (`np`):

```
>>> import numpy as np
```

16.2 Creating arrays

There are a variety of ways to create arrays. The easiest is to call the NumPy `array` method with a list as a parameter. We use as many levels of list nesting as desired dimensions in the array. Here's code, for example, that allocates sample arrays that have one (`a1d`), two (`a2d`), and three (`a3d`) dimensions and shows their values:

```
>>> a1d = np.array([1,2,3])
>>> a1d
array([1, 2, 3])
>>> a2d = np.array([ [1,2,3] , [4,5,6] ])
>>> a2d
array([[1, 2, 3],
       [4, 5, 6]])
>>> a3d = np.array([ [ [1,2,3] , [4,5,6] ] , [ [10,20,30] , [40,50,60] ] ])
>>> a3d
array([[[ 1,  2,  3],
        [ 4,  5,  6]],

       [[10, 20, 30],
        [40, 50, 60]]])
```

Unlike lists, all of the elements in an array must have the same type and the size of an array is fixed once it has been created. These limitations help enable operations on Numpy arrays to be more efficient than similar list operations.

Arrays have several useful properties: we can determine the array's number of dimensions using the `ndim` property and sizes of dimensions using the `shape` property, which evaluates to a tuple with one integer value per dimension:

```
>>> print("a3d has", a3d.ndim, "dimensions")
a3d has 3 dimensions
>>> print("a3's shape is:", a3d.shape)
a3's shape is: (2, 2, 3)
```

An array's `size` property yields the number of elements in the array, or the product of its shape.

```
>>> print("a3 has", a3d.size, "values")
a3 has 12 values
```

We can construct arrays of all zeros or all ones using the NumPy library routines `zeros` and `ones`. These methods will construct a one-dimensional array of length `N` if called with an integer argument, `N`, or an N-dimensional array if called with a tuple of integers of length `N`. Here are some examples:

```
>>> np.zeros(10)
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
>>> np.zeros((2, 2))
array([[0., 0.],
       [0., 0.]])
>>> np.ones((3, 2, 2))
array([[[1., 1.],
        [1., 1.]],

       [[1., 1.],
        [1., 1.]],

       [[1., 1.],
        [1., 1.]]])
```

NumPy also includes a couple of routines, `arange` and `linspace`, for constructing arrays that range over a set of values. Both functions are more versatile than Python's `range` function. The `arange` function creates arrays of values that range from a lower bound up to but not including an upper bound in specified increments. The lower bound, upper bound, and increment can all be floating point values.

```
>>> np.arange(1, 3, 0.5)
array([1. , 1.5, 2. , 2.5])
>>> np.arange(3, 1, -0.5)
array([3. , 2.5, 2. , 1.5])
```

As with the `range` function, the lower bound is optional and defaults to zero. The increment is also optional and defaults to one.

```
>>> np.arange(3)
array([0, 1, 2])
```

The `linspace` function is similar, but takes the desired number of values in the resulting array as an argument, instead of the interval between values, and the upper bound is included in the result. For example, if we wanted to create an array with seven equally-spaced values between 0 and 100 (inclusive), we would just use the following:

```
>>> np.linspace(0, 100, 7)
array([  0.        ,  16.66666667,  33.33333333,  50.        ,
        66.66666667,  83.33333333, 100.        ])
```

Finally, NumPy includes a function, `loadtxt`, for loading data from a file into an array. This function takes the name of the file as a required argument. Programmers can also specify the data type of the values, a number of header rows to skip (`skiprow`), the delimiter that is used to separate values in a row (`delimiter`), etc. We can load the data shown above from a file named `pima-indians-diabetes.csv` into an array named `data` with this call to the `loadtxt` function:

```
>>> data = np.loadtxt("pima-indians-diabetes.csv", skiprows=1, delimiter=",")
>>> data
array([[  6.   , 148.   ,  72.   , ...,   0.627,  50.   ,   1.   ],
       [  1.   ,  85.   ,  66.   , ...,   0.351,  31.   ,   0.   ],
       [  8.   , 183.   ,  64.   , ...,   0.672,  32.   ,   1.   ],
       ...,
       [  5.   , 121.   ,  72.   , ...,   0.245,  30.   ,   0.   ],
       [  1.   , 126.   ,  60.   , ...,   0.349,  47.   ,   1.   ],
       [  1.   ,  93.   ,  70.   , ...,   0.315,  23.   ,   0.   ]])
```

16.3 Array indexing and slicing

NumPy arrays support a variety of ways to access the stored data. The most familiar mechanism uses square brackets to index the array and looks like list indexing:

```
>>> a1d[2]
3
>>> a2d[1][2]
6
>>> a3d[1][1][2]
60
```

Experienced NumPy programmers, however, typically use an alternate format that accepts tuples as indexes. The first item in the tuple specifies the row index, which is the first dimension, the second item specifies the column index, which is the second dimension, and so on. Our first example above does not change because `a1d` is one-dimensional. The latter two examples, however, would be written by an experienced programmer as:

```
>>> a2d[1, 2]
6
>>> a3d[1, 1, 2]
60
```

Like lists, array elements are mutable, and can be updated using an array index on the left side of an assignment statement. For example, notice the change in the value of `a1d` shown before and after the assignment statement in the following code:

```
>>> a1d
array([1, 2, 3])
>>> a1d[0] = 7
>>> a1d
array([7, 2, 3])
```

Programmers can slice NumPy arrays using slicing notation familiar from lists. Let’s look at some examples of slicing using a one-dimensional array, named `a`.

```
>>> a = np.array([0,   1,   4,   9,  16,  25,  36,  49,  64,  81, 100, 121])
>>> a[1:7]
array([ 1,  4,  9, 16, 25, 36])
>>> a[3:10:2]
array([ 9, 25, 49, 81])
>>> a[:]
array([  0,   1,   4,   9,  16,  25,  36,  49,  64,  81, 100, 121])
>>> a[10:3:-1]
array([100,  81,  64,  49,  36,  25,  16])
```

Recall that the format for specifying a slice is: `X:Y:Z`, where `X` specifies an inclusive lower bound, `Y` specifies an exclusive upper bound, and `Z` specifies the increment. If omitted, the lower bound defaults to zero, the upper bound defaults to `N`, where `N` is the size of a one-dimensional array and the size of the corresponding dimension for a multi-dimensional array, and finally, the increment defaults to one. The second colon is typically omitted when the desired increment is one. Using a single colon (`:`) to specify a slice combines these defaults and is equivalent to specifying `0:N:1` as the slice.

Slicing one-dimensional arrays is not all that different from slicing lists. Things get more interesting when slicing multi-dimensional arrays. Here are a few examples:

```
>>> b = np.array([[0, 1, 4, 9],
...               [16, 25, 36, 49],
...               [64, 81, 100, 121],
...               [144, 169, 196, 225],
...               [256, 289, 324, 361],
...               [400, 441, 484, 529]])
>>> b[1:4, :3]
array([[ 16,  25,  36],
       [ 64,  81, 100],
       [144, 169, 196]])
>>> b[:, 2]
array([  4,  36, 100, 196, 324, 484])
```

The first example extracts the value of a 3x3 sub-array that consists of the first three columns (written as `0:3:1` in long form or `:3`, using the defaults) from rows one, two, and three (`1:4:1` or `1:4`) of `b`. The second extracts all the values in column two as a *one-dimensional* array.

Sub-arrays, like single elements, can be updated by specifying a slice on the left side of an assignment statement and an array with the same shape and type as the slice on the right side. (We'll see later that the same-shape requirement can be relaxed in some cases.)

```
>>> b[:, 2] = np.array([7, 7, 7, 7, 7, 7])
>>> b
array([[  0,   1,   7,   9],
       [ 16,  25,   7,  49],
       [ 64,  81,   7, 121],
       [144, 169,   7, 225],
       [256, 289,   7, 361],
       [400, 441,   7, 529]])
```

This ability to extract and modify columns and, more generally, sub-arrays, with ease explains, in part, the appeal of NumPy as a tool. In many cases, NumPy will allow us to perform, in just one or two lines of code, operations that would typically require using one or more loops with lists.

To make this idea concrete, let's return to our standardization example. We can replace the list comprehension in our second implementation of compute_feature_mean with a slice:

```
def compute_feature_mean(data, j):
    '''
    Compute the mean of feature (column) j.

    Inputs:
      data (2D NumPy array of floats)

    Returns (float): mean of feature j
    '''

    N = data.shape[0]
    return sum(data[:, j]) / N
```

As you might expect, the built-in sum function returns the sum of elements when it is called on an array.

16.4 Operations on arrays

NumPy supports a rich set of operations that behave quite differently than similar-looking operations on lists. In particular, many operations operate element-by-element rather than on the array as a unit. Let's use the array a2d defined earlier as an example:

```
>>> a2d
array([[1, 2, 3],
       [4, 5, 6]])
```

If we multiply a2d by the integer 2, we get back a new array in which element (i, j) has the value a2d[i, j] * 2.

```
>>> a2d * 2
array([[ 2,  4,  6],
       [ 8, 10, 12]])
```

The same operation on a list, in contrast, would perform repeated concatenation:

```
>>> l2 = [[1, 2, 3], [4, 5, 6]]
>>> l2 * 2
[[1, 2, 3], [4, 5, 6], [1, 2, 3], [4, 5, 6]]
```

Here are a few more examples:

```
>>> a2d + 2
array([[3, 4, 5],
       [6, 7, 8]])
>>> a2d > 2
array([[False, False,  True],
       [ True,  True,  True]])
>>> a2d == 2
array([[False,  True, False],
       [False, False, False]])
```

Notice that the second and third examples yield results that have the same shape as a2d, but that the elements are booleans rather than floats. These operations will turn out to be very useful when we discuss boolean indexing later in the chapter.

We can use these scalar operations plus slicing to rewrite our `compute_feature_stdev` function more compactly:

```
def compute_feature_stdev(data, j):
    '''
    Compute the standard deviation of feature (column) j.

    Inputs:
      data (2D NumPy array of floats)

    Returns (float): standard deviation of feature j
    '''

    N = data.shape[0]
    mu = compute_feature_mean(data, j)
    return math.sqrt(1 / N * sum((data[:, j] - mu) ** 2))
```

This version of the function uses slicing to extract the feature as a one-dimensional array. It then subtracts the mean (`mu`), which is a float, from the values in this array and compute the squares of the subtracted values. In both operations, one operand is a one-dimensional array and the other is a scalar (a float, in this case). The rest of the expression uses standard floating point operations and the square root function from the `math` library.

In addition to these scalar operations, NumPy also supports operations where both operands are arrays. In the simplest case, both operands have the same shape and the operation is performed element-by-element.

```
>>> x = np.array([[10, 20, 30],
...               [40, 50, 60]])
>>> a2d + x
array([[11, 22, 33],
       [44, 55, 66]])
>>> a2d / x
array([[0.1, 0.1, 0.1],
       [0.1, 0.1, 0.1]])
```

Using these element-wise operations, we can rewrite our code to standardize features more compactly:

```
def standardize_features(data):
    '''
    Standardize features to have mean 0.0 and standard deviation
    1.0.

    Inputs:
    data (2D NumPy array of floats): data to be standardized

    Returns (2D NumPy array of floats): standardized data
    '''

    N,M = data.shape

    mus = [compute_feature_mean(data, j) for j in range(M)]
    mu_vec = np.array(mus)
    sigmas = [compute_feature_stdev(data, j) for j in range(M)]
    sigma_vec = np.array(sigmas)

    # initialize the result w/ NxM list of lists of zeros.
```

(continues on next page)

(continued from previous page)

```
    rv = np.zeros(data.shape)

    # for each row
    for i in range(N):
        rv[i] = (data[i] - mu_vec) / sigma_vec

    return rv
```

This version constructs arrays with the means and standard deviations of features and then uses element-wise subtraction and division to standardize the rows of the data.

A common pitfall

What do you think is the result of using the * operator?

```
>>> d = np.array([[1, 2], [3, 4]])
>>> e = np.array([[1, 0], [0, 1]])
>>> d
array([[1, 2],
       [3, 4]])
>>> e
array([[1, 0],
       [0, 1]])
>>> d * e
array([[1, 0],
       [0, 4]])
```

It's element-wise multiplication, not matrix multiplication! We need to use the `dot` method to compute a matrix product.

```
>>> np.dot(d, e)
array([[1, 2],
       [3, 4]])
```

NumPy also supports a large number of mathematical functions, such as `np.sin`, that are applied element-wise:

```
>>> f = np.array([[1, -1], [np.pi, -np.pi]])
>>> f
array([[ 1.        , -1.        ],
       [ 3.14159265, -3.14159265]])
>>> np.cos(f)
array([[ 0.54030231,  0.54030231],
       [-1.        , -1.        ]])
```

16.5 Reshaping arrays

Before we discuss more complex operations on arrays, we must introduce the notion of reshaping an array. We can change an array's shape using the `reshape` method:

```
>>> a = np.arange(12)
>>> a
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
>>> ra = a.reshape(3, 4)
>>> ra
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

Be aware that the original array and the reshaped array share the same underlying data. This design has two consequences. First, the size (i.e., the number of elements) of the original array and the size of the reshaped array must be the same. So, this expression:

```
np.linspace(0, 100, 10).reshape(2, 5)
```

which creates a one-dimensional array with 10 elements and then resizes it into a two-dimensional array that spreads these ten values over two rows with five values each, is acceptable. On the other hand, this expression:

```
np.linspace(0, 100, 7).reshape(2, 5)
```

which tries to reshape an array with seven elements into one with ten elements, is not.

Second, if you reshape an array, updating either the original or the reshaped array updates both arrays! Notice in this code, for example, that both `a` and `ra` change as a result of the update to `a[0]`:

```
>>> a[0] = 7
>>> a
array([ 7,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
>>> ra
array([[ 7,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

16.6 Reductions

Reduction methods allow us to "reduce" an array to a single value. For example, we might want to compute the mean of all of the values, the standard deviation of all of the values, etc. Given an array `b`, for example:

```
>>> b = (np.arange(24) ** 2).reshape(6, 4)
>>> b
array([[  0,   1,   4,   9],
       [ 16,  25,  36,  49],
       [ 64,  81, 100, 121],
       [144, 169, 196, 225],
       [256, 289, 324, 361],
       [400, 441, 484, 529]])
```

we can compute the mean and standard deviation using the `mean` and `std` methods:

```
>>> print("The mean of b is:", b.mean())
The mean of b is: 180.16666666666666
>>> print("The standard deviation of b is:", b.std())
The standard deviation of b is: 164.84883648023995
```

These operations can also be applied along different dimensions to yield an array of values. For example, we might want to compute the means of the rows or the standard deviations of the values in each column. Such tasks can be accomplished by specifying an axis as an optional argument to the reduction method. For example, to compute row means, we could use the expression:

```
>>> b.mean(axis=1)
array([  3.5,  31.5,  91.5, 183.5, 307.5, 463.5])
```

and to compute the standard deviations of the columns, we could use the expression:

```
>>> b.std(axis=0)
array([142.33606555, 155.49776704, 168.73911488, 182.04273003])
```

If you are like us, your immediate reaction to these expressions is, “wait a minute, why are you specifying axis 1 to get the mean of the rows instead of axis 0?”

An axis specifies a family of arrays over which to compute some desired value. Let’s think about the two-dimensional case first and use `b` as an example. If the axis is `0` , the family will have four values: `b[:,  0]`, `b[:,  1]`, `b[:,  2]`, and `b[:,  3]`. The family is constructed by slicing `b` using a colon in the axis dimension and each of the possible index values in the non-axis dimension.

```
>>> col_means = np.array([b[:, 0].mean(),
...                       b[:, 1].mean(),
...                       b[:, 2].mean(),
...                       b[:, 3].mean()])
>>> b.mean(axis=0) == col_means
array([ True,  True,  True,  True])
```

If the axis is 1, the family is constructed by slicing `b` with a colon for dimension one, which picks up all the columns in a row, and each of the possible row indices for non-axis or row dimension.

```
>>> row_means = np.array([b[0, :].mean(),
...                       b[1, :].mean(),
...                       b[2, :].mean(),
...                       b[3, :].mean(),
...                       b[4, :].mean(),
...                       b[5, :].mean()])
>>> b.mean(axis=1) == row_means
array([ True,  True,  True,  True,  True,  True])
```

In general, if an array *D* has N dimensions and shape $(d_0, ..., d_{i-1}, d_i, d_{i+1}, ..., d_{N-1})$, the result of a reduction along axis i will have $N - 1$ dimensions and shape $(d_0, ..., d_{i-1}, d_{i+1}, ..., d_{N-1})$. The value at index $(j_0, ..., j_{i-1}, j_{i+1}, ..., j_{N-1})$ will be the result of applying the reduction operation to the slice $D[j_0, ..., j_{i-1}, :, j_{i+1}, ..., j_{N-1}]$.

To make this concrete, let’s reshape `b` into a three-dimensional array and take the sum along axis one:

```
>>> b2 = b.reshape((3, 2, 4))
>>> b2
array([[[  0,   1,   4,   9],
```

(continues on next page)

(continued from previous page)

```
        [ 16,  25,  36,  49]],

       [[ 64,  81, 100, 121],
        [144, 169, 196, 225]],

       [[256, 289, 324, 361],
        [400, 441, 484, 529]]])
>>> b2.sum(axis=1)
array([[ 16,  26,  40,  58],
       [208, 250, 296, 346],
       [656, 730, 808, 890]])
```

As expected, the resulting array has shape $(3, 4)$ and the resulting value at index `(1, 2)`, for example, is the sum of slice `b2[1, :, 2]`.

```
>>> b2[1, :, 2]
array([100, 196])
>>> sum(b2[1, :, 2])
296
```

Returning to our example, we can replace the code to compute the mean and standard deviation arrays with reductions along axis 0:

```
def standardize_features(data):
    '''
    Standardize features to have mean 0.0 and standard deviation
    1.0.

    Inputs:
      data (2D NumPy array of floats): data to be standardized

    Returns (2D NumPy array of floats): standardized data
    '''

    N,M = data.shape

    mu_vec = data.mean(axis=0)
    sigma_vec = data.std(axis=0)

    # initialize the result w/ NxM list of lists of zeros.
    rv = np.zeros(data.shape)

    # for each row
    for i in range(N):
        rv[i] = (data[i] - mu_vec) / sigma_vec

    return rv
```

16.7 Fancy indexing

NumPy supports fancier ways of indexing that are more powerful than those provided for regular Python lists. The simplest of these mechanisms allows us to specify the desired indexes with a list:

```
>>> a = np.arange(100, 112)
>>> a
array([100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111])
>>> a[ [1, 3, 6] ]
array([101, 103, 106])
```

In this case, the result is a one-dimensional array with three values: `a[1]`, `a[3]`, and `a[6]`.

If we use a multi-dimensional array as the index, the values at the specified indices in the data array are returned in an array of the same shape as the index array.

```
>>> a[ np.array([ [1,3] , [10, 7] ]) ]
array([[101, 103],
       [110, 107]])
```

Indexing with an array of booleans yields a *flattened*, that is, one-dimensional, array. A value from the data array is included in the result if the corresponding value in the index array has the value `True`.

```
>>> c = np.array([100, 200, 300])
>>> c[np.array([True, False, True])]
array([100, 300])
```

This indexing method is most useful in combination with relational operators:

```
>>> b = (np.arange(24) ** 2).reshape(6, 4)
>>> b
array([[  0,   1,   4,   9],
       [ 16,  25,  36,  49],
       [ 64,  81, 100, 121],
       [144, 169, 196, 225],
       [256, 289, 324, 361],
       [400, 441, 484, 529]])
>>> b > 100
array([[False, False, False, False],
       [False, False, False, False],
       [False, False, False,  True],
       [ True,  True,  True,  True],
       [ True,  True,  True,  True],
       [ True,  True,  True,  True]])
>>> b[b > 100]
array([121, 144, 169, 196, 225, 256, 289, 324, 361, 400, 441, 484, 529])
```

When we use this mechanism with assignments, the elements specified by the filter are updated. For example, here's a statement that sets all elements in `b` greater than 100 to zero:

```
>>> b[b > 100] = 0
>>> b
array([[  0,   1,   4,   9],
       [ 16,  25,  36,  49],
```

(continues on next page)

(continued from previous page)

```
       [ 64,  81, 100,   0],
       [  0,   0,   0,   0],
       [  0,   0,   0,   0],
       [  0,   0,   0,   0]])
```

Filters can be combined using the element-wise and (&), or (|), and xor (exclusive or) operations (^). Here, for example, is a statement that replaces the odd values greater than 100 with zeros in b:

```
>>> b = (np.arange(24) ** 2).reshape(6, 4)
>>> b[(b > 100) & (b % 2 == 1)] = 0
>>> b
array([[  0,   1,   4,   9],
       [ 16,  25,  36,  49],
       [ 64,  81, 100,   0],
       [144,   0, 196,   0],
       [256,   0, 324,   0],
       [400,   0, 484,   0]])
```

If we put all of the above together, we can do some pretty elaborate computations with arrays/matrices in just a few lines. For example, we might want to filter out all outliers in b that are more than one standard deviation away from the mean:

```
>>> b = (np.arange(24) ** 2).reshape(6, 4)
>>> b2 = b - b.mean()
>>> b[abs(b2 / b2.std()) < 1]
array([ 16,  25,  36,  49,  64,  81, 100, 121, 144, 169, 196, 225, 256,
       289, 324])
```

16.8 Advanced topics

This section covers advanced topics: broadcasting, the linear algebra library, and matrices. You can safely skip this part on your first read. Understanding broadcasting is helpful, but it is a complex topic that may be easier to grasp after you've had some experience with arrays.

16.8.1 Broadcasting

In the previous section, we described operations with one array and one scalar operand and operations on two arrays of the same shape. In this section, we will discuss the process of broadcasting, which makes it possible to perform operations on arrays that have compatible but not identical shapes. In these cases, NumPy *logically* constructs intermediate values that have the same shape using *broadcasting* before performing the element-by-element operations. For the sake of efficiency, NumPy does not actually construct these intermediate values in memory, but we'll describe the process as if it does because it makes broadcasting easier to understand.

We'll start our explanation by describing the broadcasting process using a pair of arrays that have the same number of dimensions and then discuss what to do when the arrays do not have the same number of dimensions.

Assume we have two arrays, D and E, with shapes $(d_0, d_1, ..., d_{N-1})$ and $(e_0, e_1, ..., e_{N-1})$ respectively. The arrays are *compatible* if $d_i = e_i$, $d_i = 1$, or $e_i = 1$ for $0 \leq i < N$. That is, two arrays are compatible if, for every dimension, the arrays either have the same size along that dimension or one of them has size one for that dimension.

Let's make this more concrete by looking at the compatibility of a few different combinations of sample arrays.

```
>>> a2by3 = np.array([[1, 2, 3],
...                   [4, 5, 6]])
>>> a2by3.shape
(2, 3)
>>> a1by3 = np.array([[4, 5, 6]])
>>> a1by3.shape
(1, 3)
>>> a3by3 = np.array([[4,  5,  6],
...                   [7,  8,  9],
...                   [10, 11, 12]])
>>> a3by3.shape
(3, 3)
>>> a2by1 = np.array([[1],
...                   [2]])
>>> a2by1.shape
(2, 1)
```

The arrays `a2by3` and `a1by3` are compatible because `a1by3` has size one for the first dimension and `a2by3` and `a1by3` both have the same size (3) in the second dimension. Similar reasoning explains that `a1by3` and `a2by1` are also compatible. The arrays `a2by3` and `a3by3`, in contrast, are not compatible because they have different sizes for the first dimension and those sizes are both greater than one. As a result, the expression `a2by3 + a3by3` will fail when evaluated.

```
>>> a2by3 + a3by3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast together with shapes (2,3) (3,3)
```

The next step is to determine the shape of the arrays created by broadcasting, which is computed as a function of the shapes the underlying arrays: $(max(d_0, e_0), max(d_1, e_1), ..., max(d_{N-1}, e_{N-1}))$. Returning to our examples, the broadcast shape for `a2by3 + a1by3` is $(max(2, 1), max(3, 3))$ or $(2, 3)$. Similarly, the broadcast shape for `a1by3 + a2by1` will be $(max(1, 2), max(3, 1))$ or $(2, 3)$.

To create an array of the correct shape, broadcasting replicates values along the dimensions where the size of the original array is one and the size of broadcast value is greater than one. Once this process is complete, NumPy can perform element-by-element operations on the intermediate arrays to produce a result.

Let's return to our example of `a2by3 + a1by3`. We know that the arrays are compatible and that the broadcast shape is $(2, 3)$. The array `a2by3` already has the right shape. The array `a1by3`, on the other hand, needs to be stretch from $(1, 3)$ to $(2, 3)$, which is accomplished by replicating along the row dimension (that is, dimension 0):

```
array([[4, 5, 6],
       [4, 5, 6]])
```

Once this intermediate value is computed, NumPy can add it to `a2by3` to get a final result of:

```
>>> a2by3 + a1by3
array([[ 5,  7,  9],
       [ 8, 10, 12]])
```

In this case, only one of the operands needed to be stretched to construct an intermediate value of the right shape. In other cases, both arrays need to be stretched. For example, as noted above `a1by3 + a2by1` will yield a value with a shape of $(2, 3)$, which requires NumPy to stretch `a1by3` from $(1, 3)$ to $(2, 3)$ as in the previous example, and to stretch `a2by1` from $(2, 1)$ to $(2, 3)$. In the latter case, the values are replicated along column because `a2by1` has size one in the second dimension:

```
array([[1, 1, 1],
       [2, 2, 2]])
```

Once both values are stretched, NumPy can construct the final result for `a1by3 + a2by1`:

```
>>> a1by3 + a2by1
array([[5, 6, 7],
       [6, 7, 8]])
```

Now, let's consider what happens when one of the arrays has fewer dimensions than the other. Since it does not matter for this computation which is smaller, let's say that E is smaller and has shape $(e_0, e_1, ..., e_M)$ where $M < N$. To start the broadcasting process, NumPy reshapes `E` into an array with `N` dimensions. The shape of that array is constructed by prepending $N - M$ ones to the original shape of `E`: $(1, ..., 1, e_0, e_1, ..., e_M)$. You can think of this transformation as being equivalent to wrapping one extra pair of square brackets around the list passed to `np.array` for each added dimension. Once the array with fewer dimensions has been reshaped, the broadcasting process can proceed as described above.

At the beginning of the previous section, we explained that NumPy supports operations with one array operand and one scalar operand. We can explain now that these operations are supported through broadcasting. A scalar can be thought of as a one dimension array of size 1. For example, consider what happens when we compute `a2by3 + 2`, which is equivalent to `a2by3 + np.array([2])`. Since `a2by3` has shape $(2, 3)$ and `np.array([2])` has shape $(1,)$, NumPy will logically reshape `np.array([2])` into `np.array([[2]])`, which has a shape of $(1, 1)$, as a first step. It will then determine that the broadcast shape for `a2by3 + 2` is $(2, 3)$ and will replicate `np.array([[2]])` along both dimensions to create an intermediate value of the right shape:

```
array([[2, 2, 2],
       [2, 2, 2]])
```

Finally, it will add `a2by3` to this intermediate value to yield:

```
array([[3, 4, 5],
       [6, 7, 8]])
```

Let's put all of these ideas together and look at what happens when we add `a1by3 + a7` where `a7` is defined as:

```
>>> a1by3by1 = np.array([[[1],
...                       [2],
...                       [3]]])
>>> a1by3by1.shape
(1, 3, 1)
```

Because `a1by3` and `a1by3by1` do not have the same number of dimensions, the first step will be for NumPy to logically construct a new array from `a1by3` that has the value:

```
array([ [ [ 4, 5, 6 ] ] ])
```

This value, which has the shape: $(1, 1, 3)$, has the same number of dimensions as `a1by3by1`. NumPy will then determine that the broadcast shape of `a1by3 + a1by3by1` is $(1, 3, 3)$ and it will construct intermediate results of the form:

```
array([[[ 4.,  5.,  6.],
        [ 4.,  5.,  6.],
        [ 4.,  5.,  6.]]])
```

and

```
array([[[ 1.,  1.,  1.],
        [ 2.,  2.,  2.],
        [ 3.,  3.,  3.]]])
```

and finally combine them to yield a result of:

```
array([[[5, 6, 7],
        [6, 7, 8],
        [7, 8, 9]]])
```

With NumPy, an assignment is legal as long as the array on the right side can be broadcast to the same shape as the array or sub-array on the left side. This allows us to relax the requirement for arrays on either side of an assignment to have the same shape. For example:

```
>>> b = (np.arange(24) ** 2).reshape(6, 4)
>>> b[:, 0:2] = np.array([10, 20])
>>> b
array([[ 10,  20,   4,   9],
       [ 10,  20,  36,  49],
       [ 10,  20, 100, 121],
       [ 10,  20, 196, 225],
       [ 10,  20, 324, 361],
       [ 10,  20, 484, 529]])
```

The slice referenced on the left side of the assignment statement has shape $(6, 2)$, while the array on the right-side has shape $(2,)$. To complete the assignment, NumPy broadcasts `np.array([10, 20])` into:

```
array([[10, 20],
       [10, 20],
       [10, 20],
       [10, 20],
       [10, 20],
       [10, 20]])
```

which has the expected shape of $(6, 2)$, and then performs the update.

16.8.2 Example: Standardizing features, revisited

We now have all the pieces necessary to understand the NumPy solution to the task of standardizing features that we presented at the start of the chapter.

```
def standardize_features(data):
    '''
    Standardize features to have mean 0.0 and standard deviation
    1.0.

    Inputs:
      data (2D NumPy array of floats): data to be standardized

    Returns (2D NumPy array of floats): standardized data
    '''
```

(continues on next page)

(continued from previous page)

```
    mu_vec = data.mean(axis=0)
    sigma_vec = data.std(axis=0)
    return (data - mu_vec) / sigma_vec
```

We discussed the first couple of lines, which use reductions to compute the means and standard deviations of the features, above. Only the last line is new. The expression `(data - mu_vec)` yields an array with shape `(N, M)`. Notice that we do not need a loop to do this computation. Instead, we rely on NumPy's broadcasting mechanism to convert `mu` from a vector of length `M` into an `N` by `M` array and use element-wise subtraction to do the computation. In the new array, the jth column holds `N` copies of the mean of the jth feature. Broadcasting also converts `sigma_vec` from `1` by `M` array into an `N` by `M` array, and allows us to do element-wise division to compute the desired `N` by `M` array.

16.8.3 Linear algebra

NumPy includes a linear algebra library (`numpy.linalg`), which is traditionally imported with the alias `la`:

```
>>> import numpy.linalg as la
```

This library includes functions for performing a few different decompositions (`cholesky`, `qr`, and `svd`), computing eigenvalues, eigenvectors, norms, and other values (e.g., `det`, `norm`, `matrix_rank`), and determining the inverse of a matrix.

As an example, let's use a few of these methods to solve the following system of equations:

$$1 \cdot x_0 + 3 \cdot x_1 = 11 \tag{16.1}$$

$$2 \cdot x_0 + 8 \cdot x_1 = 28 \tag{16.2}$$

We can represent this system of equations using two arrays. We can write them mathematically as:

$$\mathbf{X} = \begin{pmatrix} 1 & 3 \\ 2 & 8 \end{pmatrix} \quad \mathbf{Y} = \begin{pmatrix} 11 \\ 28 \end{pmatrix}$$

and construct them using NumPy as follows:

```
>>> X = np.array([ [1,3] , [2, 8] ])
>>> Y = np.array([ [11], [28] ])
```

One way to solve this system is to compute by the dot product of the inverse of `X` and `Y`:

```
>>> iX = la.inv(X)
>>> iX
array([[ 4. , -1.5],
       [-1. ,  0.5]])
>>> solution = np.dot(iX, Y)
>>> solution
array([[2.],
       [3.]])
```

Another is to use the `linalg.solve` function.

```
>>> solution = la.solve(X, Y)
>>> solution
array([[2.],
       [3.]])
```

In this case, both methods have similar performance. In general, `la.solve()` is preferred because it will exploit properties of `X`, such as symmetry, to increase efficiency, when appropriate.

16.8.4 NumPy's matrix type

NumPy also has a matrix type that is useful because some of the operators "make more sense" with matrices. e.g., * will do matrix multiplication, not element-wise multiplication:

```
>>> d = np.matrix([[1, 2], [3, 4]])
>>> e = np.matrix([[1, 0], [0, 1]])
>>> d * e
matrix([[1, 2],
        [3, 4]])
```

However, most NumPy developers recommend using the more general array type (anything you can do with a matrix, you can do with an array; e.g., matrix multiplication is just the `dot` method). The converse is not true: matrices are always two-dimensional.

CHAPTER

SEVENTEEN

THE PANDAS LIBRARY

Pandas is a data analysis toolkit for Python that makes it easy to work with certain types of data, specially:

- Tabular data (i.e., anything that can be expressed as labelled columns and rows)
- Time series data
- Matrix data
- Statistical datasets

In this chapter, we will focus on tabular data and organize our discussion of this library around an exploration of data from the 2015 New York City Street Tree Survey, which is freely available from the New York City open data portal (https://data.cityofnewyork.us). This survey was performed by the New York City Department of Parks and Recreation with help from more than 2000 volunteers. The goal of the survey is to catalog the trees planted on the City right-of-way, typically the space between the sidewalk and the curb, in the five boroughs of New York. We can use this data to answer questions such as:

1. How many different species are planted as street trees in New York?
2. What are the five most common street tree species in New York?
3. What is the most common street tree species in Brooklyn?
4. What percentage of the street trees in Queens are dead or in poor health?
5. How does street tree health differ by borough?

The survey data is stored in a CSV file that has 683,789 lines, one per street tree. (Hereafter we will refer to trees rather than street trees for simplicity.) The census takers record many different attributes for each tree, such as the common name of the species, the location of the tree, etc. Of these values, we will use the following:

- `boroname`: the name of the borough in which the tree resides;
- `health`: an estimate of the health of the tree: one of good, fair, or poor;
- `latitude` and `longitude` : the location of the tree using geographical coordinates;
- `spc_common`: the common, as opposed to Latin, name of the species;
- `status`: the status of the tree: one of alive, dead, or stump;
- `tree_id`: a unique identifier

Some fields are not populated for some trees. For example, the `health` field is not populated for dead trees and stumps and the species field (`spc_common`) is not populated for stumps and most dead trees.

Pandas supports a variety of data types. We'll talk about three—`DataFrame`, `Series` and `Categorical`—in detail. We also talk about how filtering, which was introduced in our discussion of Numpy, and grouping, a new concept, can be used to answer complex questions with little code. But before we get get started, we need to import Pandas. This step is traditionally done using the `as` option with `import` to introduce a short alias (`pd`) for the library.

```
>>> import pandas as pd
```

17.1 DataFrames

We'll start by loading the tree data into a data structure. Data frames, the main data structure in Pandas, were inspired by the data structure of the same name in R and are designed to represent tabular data. The library function `pd.read_csv` takes the name of a CSV file and loads it into a data frame. Let's use this function to load the tree data from a file named `2015StreetTreesCensus_TREES.csv`:

```
>>> trees = pd.read_csv("2015StreetTreesCensus_TREES.csv")
```

This file is available in our *example code*, in the `working_with_data/pandas/` directory. Please note that it is provided as a ZIP file called `tree-census-data.zip`, and you must un-zip it before using it.

The variable `trees` now refers to a Pandas `DataFrame`. Let's start by looking at some of the actual data. We'll explain the various ways to access data in detail later. For now, just keep in mind that the columns have names (for example, "Latitude", "longitude", "spc_common", etc) and leverage the intuition you've built up about indexing in other data structures.

Here, for example, are a few columns from the first ten rows of the dataset:

```
>>> trees10 = trees[:10]
>>> trees10[["Latitude", "longitude", "spc_common", "health", "boroname"]]
    Latitude  longitude        spc_common health       boroname
0  40.723092 -73.844215         red maple   Fair         Queens
1  40.794111 -73.818679           pin oak   Fair         Queens
2  40.717581 -73.936608       honeylocust   Good       Brooklyn
3  40.713537 -73.934456       honeylocust   Good       Brooklyn
4  40.666778 -73.975979   American linden   Good       Brooklyn
5  40.770046 -73.984950       honeylocust   Good      Manhattan
6  40.770210 -73.985338       honeylocust   Good      Manhattan
7  40.762724 -73.987297   American linden   Good      Manhattan
8  40.596579 -74.076255       honeylocust   Good  Staten Island
9  40.586357 -73.969744  London planetree   Fair       Brooklyn
```

Notice that the result looks very much like a table in which both the columns and the rows are labelled. In this case, the column labels came from the first row in the file and the rows are simply numbered starting at zero.

Here's the full first row of the dataset with all 41 attributes:

```
>>> trees.iloc[0]
created_at                                       08/27/2015
tree_id                                              180683
block_id                                             348711
the_geom      POINT (-73.84421521958048 40.723091773924274)
tree_dbh                                                  3
stump_diam                                                0
curb_loc                                             OnCurb
status                                                Alive
health                                                 Fair
spc_latin                                       Acer rubrum
spc_common                                        red maple
steward                                                None
```

(continues on next page)

(continued from previous page)

```
guards                                        None
sidewalk                                  NoDamage
user_type                          TreesCount Staff
problems                                      None
root_stone                                      No
root_grate                                      No
root_other                                      No
trnk_wire                                       No
trnk_light                                      No
trnk_other                                      No
brnch_ligh                                      No
brnch_shoe                                      No
brnch_othe                                      No
address                          108-005 70 AVENUE
zipcode                                      11375
zip_city                              Forest Hills
cb_num                                         406
borocode                                         4
boroname                                    Queens
cncldist                                        29
st_assem                                        28
st_senate                                       16
nta                                           QN17
nta_name                              Forest Hills
boro_ct                                    4073900
state                                     New York
Latitude                                   40.7231
longitude                                 -73.8442
x_sp                                   1.02743e+06
y_sp                                        202757
Name: 0, dtype: object
```

and here are a few specific values from that row:

```
>>> first_row = trees.iloc[0]
>>> first_row["Latitude"]
40.72309177
>>> first_row["longitude"]
-73.84421522
>>> first_row["boroname"]
'Queens'
```

Notice that the latitude and longitude values are floats, while the borough name is a string. Conveniently, `read_csv` analyzes each column and if possible, identifies the appropriate type for the data stored in the column. If this analysis cannot determine a more specific type, the data will be represented using strings.

We can also extract data for a specific column:

```
>>> trees10["boroname"]
0          Queens
1          Queens
2        Brooklyn
3        Brooklyn
```

(continues on next page)

(continued from previous page)

```
4         Brooklyn
5        Manhattan
6        Manhattan
7        Manhattan
8    Staten Island
9         Brooklyn
Name: boroname, dtype: object
```

and we can easily do useful things with the result, like count the number of times each unique value occurs:

```
>>> trees10["boroname"].value_counts()
Brooklyn         4
Manhattan        3
Queens           2
Staten Island    1
Name: boroname, dtype: int64
```

Now that you have a some feel for the data, we'll move on to discussing some useful attributes and methods provided by data frames. The `shape` attribute yields the number of rows and columns in the data frame:

```
>>> trees.shape
(683788, 42)
```

The data frame has fewer rows (683,788) than lines in the file (683,789), because the header row is used to construct the column labels and does not appear as a regular row in the data frame.

We can use the `columns` attribute to examine the column labels:

```
>>> trees.columns
Index(['created_at', 'tree_id', 'block_id', 'the_geom', 'tree_dbh',
       'stump_diam', 'curb_loc', 'status', 'health', 'spc_latin', 'spc_common',
       'steward', 'guards', 'sidewalk', 'user_type', 'problems', 'root_stone',
       'root_grate', 'root_other', 'trnk_wire', 'trnk_light', 'trnk_other',
       'brnch_ligh', 'brnch_shoe', 'brnch_othe', 'address', 'zipcode',
       'zip_city', 'cb_num', 'borocode', 'boroname', 'cncldist', 'st_assem',
       'st_senate', 'nta', 'nta_name', 'boro_ct', 'state', 'Latitude',
       'longitude', 'x_sp', 'y_sp'],
      dtype='object')
```

We noted earlier that the rows, like columns, have labels. Collectively, these values are known as the index. As currently constructed, the tree data set has an index that ranges from zero to 683,787. Often, data sets have a meaningful value that can be used to uniquely identify the rows. The trees in our data set, for example, have unique identifiers that can be used for this purpose. Let's re-load the data and specify the name of a column to use for the index using the `index_col` parameter:

```
>>> trees = pd.read_csv("2015StreetTreesCensus_TREES.csv",
...                     index_col="tree_id")
```

Now that we have a meaningful index, we can use the `index` attribute to find the names of the rows:

```
>>> trees.index
Int64Index([180683, 200540, 204026, 204337, 189565, 190422, 190426, 208649,
            209610, 192755,
```

(continues on next page)

(continued from previous page)

```
            ...
            200671, 193070, 195173, 155348, 184210, 155433, 183795, 166161,
            184028, 200607],
           dtype='int64', name='tree_id', length=683788)
```

Now let's look at accessing values in the sample data frame in a more systematic way. We can extract the data for a specific row using either the appropriate row label or the position of the row in the data frame. To index the data using the row label (`180683` for the row at position 0), we use the `loc` operator with square brackets.

```
>>> trees.loc[180683]
created_at                                       08/27/2015
block_id                                             348711
the_geom      POINT (-73.84421521958048 40.723091773924274)
tree_dbh                                                  3
stump_diam                                                0
curb_loc                                             OnCurb
status                                                Alive
health                                                 Fair
spc_latin                                       Acer rubrum
spc_common                                        red maple
steward                                                None
guards                                                 None
sidewalk                                           NoDamage
user_type                                  TreesCount Staff
problems                                               None
root_stone                                               No
root_grate                                               No
root_other                                               No
trnk_wire                                                No
trnk_light                                               No
trnk_other                                               No
brnch_ligh                                               No
brnch_shoe                                               No
brnch_othe                                               No
address                                   108-005 70 AVENUE
zipcode                                               11375
zip_city                                       Forest Hills
cb_num                                                  406
borocode                                                  4
boroname                                             Queens
cncldist                                                 29
st_assem                                                 28
st_senate                                                16
nta                                                    QN17
nta_name                                       Forest Hills
boro_ct                                             4073900
state                                              New York
Latitude                                            40.7231
longitude                                          -73.8442
x_sp                                            1.02743e+06
y_sp                                                 202757
Name: 180683, dtype: object
```

To access the row using the row number, that is, its position in the data frame, we use `iloc` operator and square brackets:

```
>>> trees.iloc[0]
created_at                                       08/27/2015
block_id                                             348711
the_geom      POINT (-73.84421521958048 40.723091773924274)
tree_dbh                                                  3
stump_diam                                                0
curb_loc                                             OnCurb
status                                                Alive
health                                                 Fair
spc_latin                                       Acer rubrum
spc_common                                        red maple
steward                                                None
guards                                                 None
sidewalk                                           NoDamage
user_type                                  TreesCount Staff
problems                                               None
root_stone                                               No
root_grate                                               No
root_other                                               No
trnk_wire                                                No
trnk_light                                               No
trnk_other                                               No
brnch_ligh                                               No
brnch_shoe                                               No
brnch_othe                                               No
address                                   108-005 70 AVENUE
zipcode                                               11375
zip_city                                       Forest Hills
cb_num                                                  406
borocode                                                  4
boroname                                             Queens
cncldist                                                 29
st_assem                                                 28
st_senate                                                16
nta                                                    QN17
nta_name                                       Forest Hills
boro_ct                                             4073900
state                                              New York
Latitude                                            40.7231
longitude                                          -73.8442
x_sp                                            1.02743e+06
y_sp                                                 202757
Name: 180683, dtype: object
```

In both cases the result of evaluating the expression has type Pandas `Series`:

```
>>> type(trees.iloc[0])
<class 'pandas.core.series.Series'>
```

A `Series` is defined as "a one-dimensional labeled array capable of holding any data type (integers, strings, floating point numbers, Python objects, etc.)." Briefly, we can think of a `Series` as an array and index it using integers, for example, `trees.iloc[0][0]` yields the first value in the first row ("08/27/2015"). We can also think of it as a dictionary and index it using the labels. We can, for example, extract the exact same value using the expression `trees.iloc[0]["created_at"]`.

As we saw earlier, we can use slicing to construct a new data frame with a subset of the rows of an existing data frame. For example, this statement from above:

```
>>> trees10 = trees[0:10]
```

constructs a data frame that contains the first ten rows (row 0 through row 9 inclusive) of the trees data set. One thing to keep in mind, the new data frame and the original data frame share the same underlying data, which means that updating one, updates the other!

We can extract the values in a specific column as a series using square brackets with the column name as the index:

```
>>> trees10["spc_common"]
tree_id
180683           red maple
200540             pin oak
204026         honeylocust
204337         honeylocust
189565     American linden
190422         honeylocust
190426         honeylocust
208649     American linden
209610         honeylocust
192755    London planetree
Name: spc_common, dtype: object
```

We can also use dot notation to access a column, if the corresponding label conforms to the rules for Python identifiers and does not conflict with the name of a `DataFrame` attribute or method:

```
>>> trees10.spc_common
tree_id
180683           red maple
200540             pin oak
204026         honeylocust
204337         honeylocust
189565     American linden
190422         honeylocust
190426         honeylocust
208649     American linden
209610         honeylocust
192755    London planetree
Name: spc_common, dtype: object
```

The tree dataset has many columns, most of which we will not be using to answer the questions posed at the beginning of the chapter. As we saw above, we can extract the desired columns using a list as the index:

```
>>> cols_to_keep = ['spc_common', 'status', 'health', 'boroname', 'Latitude', 'longitude
↪']
>>> trees_narrow = trees[cols_to_keep]
>>> trees_narrow.shape
(683788, 6)
```

This new data frame has the same number of rows and the same index as the original data frame, but only six columns instead of the original 41.

If we know in advance that we will be using only a subset of the columns, we can specify the names of the columns of interest to `pd.read_csv` and get the slimmer data frame to start. Here's a function that uses this approach to construct the desired data frame:

```
>>> def get_tree_data(filename):
...     '''
...     Read slim version of the tree data and clean up the labels.
...
...     Inputs:
...         filename: name of the file with the tree data
...
...     Returns: DataFrame
...     '''
...     cols_to_keep = ['tree_id', 'spc_common', 'status', 'health', 'boroname',
...                     'Latitude', 'longitude']
...     trees = pd.read_csv(filename, index_col="tree_id",
...                         usecols=cols_to_keep)
...     trees.rename(columns={"Latitude":"latitude"}, inplace=True)
...     return trees
...
...
>>> trees = get_tree_data("2015StreetTreesCensus_TREES.csv")
```

A few things to notice about this function: first, the index column, `tree_id`, needs to be included in the value passed with the `usecols` parameter. Second, we used the `rename` method to fix a quirk with the tree data: "Latitude" is the only column name that starts with a capital letter. We fixed this problem by supplying a dictionary that maps the old name of a label to a new name using the `columns` parameter. Finally, by default, `rename` constructs a new dataframe. Calling it with the `inplace` parameter set to `True`, causes frame updated in place, instead.

17.2 A note about missing values

As we noted when we described the tree data, some attributes are not included for some entries. The entries for stumps and dead trees, for example, do not include values for the health of the tree. We can see the impact of this missing data in row 630, which contains information about a dead tree in Queens:

```
>>> trees.iloc[630]
status           Dead
health            NaN
spc_common        NaN
boroname       Queens
latitude       40.738
longitude    -73.9216
Name: 192569, dtype: object
```

Notice that both the `health` and the `spc_common` fields have the value `NaN`, which stands for "not a number." This value is inserted in place of missing values by default by `pd.read_csv`. We'll talk about how Pandas handles these values as we explore the trees data set.

17.3 Series

Now that we have the data in a form suitable for computation, let's look at what is required to answer the first two questions: "How many different tree species are planted in New York?" and "What are the five most common tree species in New York?"

One approach would be to write code to iterate over tree species in the spc_common column, build a dictionary that maps these names to counts, and then process the dictionary to extract the answers to our questions:

```
>>> counts = {}
>>> for kind in trees.spc_common:
...     if pd.notnull(kind):
...         counts[kind] = counts.get(kind, 0) + 1
...
...
>>> print("Number of unique street tree species:", len(counts.keys()))
Number of unique street tree species: 132
>>> top_5 = sorted(counts.items(), key=lambda x: (x[1], x[0]), reverse=True)[:5]
>>> for kind, val in top_5:
...     print(kind)
...
London planetree
honeylocust
Callery pear
pin oak
Norway maple
```

Recall that the species is not specified for stumps and most dead trees and that missing values are represented in the data frame with the value NaN. We do not want to include these values in our count and so we'll check for NaN using pd.notnull before we update counts. The method pd.notnull can be used with a single value or with a compound value, such as a list, series, or a data frame. In this context, we are calling it with a single value and it will return True if kind is not NaN and False otherwise.

This code works, but thus far all we have gained from using Pandas in service to answering our question is the ability to extract and iterate over a column from the data frame, which is not very exciting. We can, in fact, answer these questions with very little code by using some of the many attributes and methods provided by the Series data type. Here for example, is code, to answer the first question using the nunique method:

```
>>> num_unique = trees.spc_common.nunique()
>>> print("Number of unique street tree species:", num_unique)
Number of unique street tree species: 132
```

which returns the number of unique values in a series. By default, it does not include NaN in the count. The unique method, which returns a Numpy array with the unique values in the series, is also useful.

We can also answer the second question with very little code. we'll use the value_counts method to construct a *new* series in which the index labels are the different species that occurred in spc_common and the values count the number of times each species occurred. Conveniently, the values are ordered in descending order by count, so the most frequent item is first and we can use slicing to extract the top five:

```
>>> vc = trees.spc_common.value_counts()
>>> vc[:5]
London planetree    87014
honeylocust         64264
Callery pear        58931
```

(continues on next page)

(continued from previous page)

```
pin oak             53185
Norway maple        34189
Name: spc_common, dtype: int64
```

The resulting series is not quite what we want: the names of the trees. Fortunately, we can extract these names from the slices series using the `index` attribute:

```
>>> for kind in vc.index[:5]:
...     print(kind)
...
London planetree
honeylocust
Callery pear
pin oak
Norway maple
```

In addition to the `index` attribute, the `Series` type includes other useful attributes, such as `size` which holds the number of elements in the series and `values`, which yields an array with the series' values. This type also comes with a large number of useful methods. We'll discuss a few of them here.

As we just saw, we can slice a series. We can also extract individual elements using the `loc` and `iloc` operators with a value's index and position respectively:

```
>>> vc.loc["London planetree"]
87014
>>> vc.iloc[0]
87014
```

We can also drop the `loc` operator and just use square brackets with the index:

```
>>> vc["London planetree"]
87014
```

Finally, if the index is a legal Python identifier and it does not conflict with a `Series` attribute or method name, we can use the dot notation. "London planetree" does not qualify because of the embedded space, but "mimosa" on the other hand, does:

```
>>> vc.mimosa
163
```

The series `describe` method computes a new series that contains summary information about the data and is very useful when you are exploring a new dataset. The result depends on the type of the values stored in the series. In the case of the common names of the tree species, the values are strings and so, `describe` tells us the number of entries (`count`), the number of unique values (`unique`), the most common value (`top`), and its frequency `freq`).

```
>>> trees.spc_common.describe()
count                   652169
unique                     132
top       London planetree
freq                     87014
Name: spc_common, dtype: object
```

By default, all of these values are computed excluding `NaN` values. This method provides an alternative way to answer our first question:

```
>>> num_unique = trees.spc_common.describe()["unique"]
>>> print("Number of unique street tree species:", num_unique)
Number of unique street tree species: 132
```

One thing to note about this code: we used the square bracket notation to access the `unique` value from the result of `describe`. Could we have used dot notation instead? No, because even though `unique` is a legal Python identifier, it conflicts with the name of a series method.

Given a series with numeric values `describe` computes the number of values in the series, the mean of those values, their standard deviation, and their quartiles. Latitude and longitude are the only values represented by floats in our sample dataset and it does not really make sense to compute quartiles for these values. So, we'll construct a simple series with an index that ranges from zero to ten to use an example using the `pd.Series` constructor and a list of values:

```
>>> sample_series = pd.Series([92, 74, 80, 60, 72, 90, 84, 74, 75, 70])
>>> sample_series
0    92
1    74
2    80
3    60
4    72
5    90
6    84
7    74
8    75
9    70
dtype: int64
>>> sample_series.describe()
count    10.000000
mean     77.100000
std       9.643075
min      60.000000
25%      72.500000
50%      74.500000
75%      83.000000
max      92.000000
dtype: float64
```

The `describe` method can also be applied to data frames.

As with NumPy arrays, operators are applied element-wise and Numpy-style broadcasting is used to construct values of the same shape prior to applying the operator. For example, we could compute a series with the percentage live trees, dead trees, and stumps using a call to `value_counts` and a couple of operators:

```
>>> trees.status.value_counts()/len(trees) * 100
Alive    95.376491
Stump     2.581794
Dead      2.041715
Name: status, dtype: float64
```

17.4 Filtering

Now let's move on to our third and fourth questions: "What is the most common street tree species in Brooklyn?" and "What percentage of the trees street in Queens are dead or in poor health?"

We could answer these questions by iterating over the data frame using the `iterrows` method, which yields a tuple with the label and corresponding value for each row in the data frame. In the body of the loop, we would identify the relevant rows, construct a dictionary, and calculate the most frequent species as we go:

```
>>> top_kind = ""
>>> top_count = 0
>>> brooklyn_tree_counts = {}
>>> for idx, tree in trees.iterrows():
...     if tree["boroname"] == "Brooklyn":
...         kind = tree["spc_common"]
...         if pd.notnull(kind):
...             brooklyn_tree_counts[kind] = brooklyn_tree_counts.get(kind, 0) + 1
...             if brooklyn_tree_counts[kind] > top_count:
...                 top_count = brooklyn_tree_counts[kind]
...                 top_kind = kind
...
>>> print("Most common tree in Brooklyn:", top_kind)
Most common tree in Brooklyn: London planetree
```

Alternatively, we could leverage Pandas a bit more by constructing a new data frame with the relevant rows and then using the `Series` `mode` method to find the most frequent non-null value in the `spc_common` column:

```
>>> brooklyn_trees = []
>>> for idx, tree in trees.iterrows():
...     if tree["boroname"] == "Brooklyn":
...         brooklyn_trees.append(tree)
...
>>> bt = pd.DataFrame(brooklyn_trees)
>>> print("Most common tree in Brooklyn:", bt.spc_common.mode().iloc[0])
Most common tree in Brooklyn: London planetree
```

The `mode` method returns returns a series of with the most frequent value or, in the case of a tie, values

```
>>> bt.spc_common.mode()
0    London planetree
dtype: object
```

To find the name of the species that occurs most often, we'll extract the first value from this series using `iloc[0]` Given a tie, we'll just use the first one.

Unfortunately, both of these approaches are quite slow, because iterating over a data frame one row at a time is an expensive operation. A better way to answer this question is to use filtering, which is similar to filtering in Numpy. Here's a statement that computes the same data frame with entries for the trees in Brooklyn much more efficiently:

```
>>> in_brooklyn = trees.boroname == "Brooklyn"
>>> bt = trees[in_brooklyn]
```

The variable `in_brooklyn` refers to a series with boolean values, one per tree, that are `True` for trees in Brooklyn and `False` otherwise. If we use a series of booleans to index into a data frame, the result will be a new data frame that includes the rows for which the corresponding boolean is `True`.

Using this approach, we can compute the most common tree in Brooklyn quite compactly and efficiently:

```
>>> bt = trees[trees.boroname == "Brooklyn"]
>>> print("Most common tree in Brooklyn:", bt.spc_common.mode().iloc[0])
Most common tree in Brooklyn: London planetree
```

Notice that this version specifies the filter expression directly as the index, rather than assigning an intermediate name to the series of booleans.

We can combine multiple conditions using the element-wise and (&) and element-wise or (|) operators. For example, here's code that constructs a data frame with the entries for trees in Queens that are that are either dead or in poor health:

```
>>> filter = (trees.boroname == "Queens") & \
...            ((trees.status == "Dead") | (trees.health == "Poor"))
>>> bad_trees_queens = trees[filter]
```

Note that the parentheses are necessary because element-wise and (&) has higher precedence than both equality (==) and element-wise or (|).

To answer our question "What percentage of the trees street in Queens are dead or in poor health?", we need both the number of trees in Queens (excluding stumps) and the number of bad trees in Queens, so we'll split the filtering into two pieces:

```
>>> not_stump_in_queens = (trees.boroname == "Queens") & (trees.status != "Stump")
>>> trees_in_queens = trees[not_stump_in_queens]
>>> bad_tree_filter = (trees_in_queens.status == "Dead") | \
...                   (trees_in_queens.health == "Poor")
>>> bad_trees_in_queens = trees_in_queens[bad_tree_filter]
>>> s = "Percentage of the trees street in Queens are dead or in poor health: {:.2f}%"
>>> print(s.format(len(bad_trees_in_queens)/len(trees_in_queens)*100))
Percentage of the trees street in Queens are dead or in poor health: 5.72%
```

The first two lines build a data frame for the trees in Queens (excluding stumps), while the second and third lines filter this new data frame further to find the dead trees and trees in poor health.

17.5 Grouping

To answer our last question— "How does tree health differ by borough?"—we will compute a data frame similar to Table X, which has one row per borough. Specifically, it contains data for the percentages of trees in the borough deemed to be in good, fair, or poor health and for dead trees and stumps.

Table 1: Stree Tree Health by Borough

Borough	Good	Fair	Poor	Dead	Stumps
Bronx	78.2%	12.8%	3.6%	3.0%	2.5%
Brooklyn	78.0%	14.1%	3.6%	1.9%	2.4%
Manhattan	72.4%	17.5%	5.5%	2.8%	1.8%
Queens	77.4%	13.8%	3.8%	1.8%	3.2%
Staten Island	78.5%	13.8%	4.0%	1.8%	1.9%

We'll work up to this task by answering some easier questions first:

1. How many entries does the data set have for each borough?

2. For each borough, how many entries are for live trees, dead trees, and stumps?
3. For each borough, what percentage of the trees are live, dead, or stumps?

To count the number of entries per borough, we could walk over the individual rows and update a dictionary that maps boroughs to entry counts. But as we learned in the last section, iterating over the rows individually is slow and is best to be avoided. Another way to compute this information would be to use filtering to create a data frame for each borough and then compute its length. Here's a function that uses this approach:

```
>>> def find_per_boro_count(trees):
...     counts = []
...     boros = sorted(trees.boroname.unique())
...     for boro in boros:
...         boro_df = trees[trees.boroname == boro]
...         counts.append(len(boro_df))
...
...     return pd.Series(counts, index=boros)
...
...
>>> find_per_boro_count(trees)
Bronx             85203
Brooklyn         177293
Manhattan         65423
Queens           250551
Staten Island    105318
dtype: int64
```

In each iteration of the loop, we identify the trees in a specific borough, count them using `len`, and then save the number on a list. Once this list is constructed, we use it and the list of borough names to create a series with the desired result.

The task of separating the rows of a data frame into groups based on one or more fields and then doing some work on the individual groups is very common and so, Pandas provides the `groupby` method to simplify the process. The groups can be specified in a variety of ways. We'll start with the most straightforward: using a column label or a tuple of column labels. This method returns a special `DataFrameGroupBy` object. When we iterate over an object of this type, we get a tuple that contains the name of the group and a data frame that contains the rows that belong to the group.

Here's a version of `find_per_boro_count` that uses `groupby` on the borough name rather than explicit filtering:

```
>>> def find_per_boro_count(trees):
...     counts = []
...     boros = []
...     for boro, boro_df in trees.groupby("boroname"):
...         counts.append(len(boro_df))
...         boros.append(boro)
...
...     return pd.Series(counts, index=boros)
...
...
>>> find_per_boro_count(trees)
Bronx             85203
Brooklyn         177293
Manhattan         65423
Queens           250551
Staten Island    105318
dtype: int64
```

Other than the use of groupby, the only difference between this function and the previous one is that we construct the list of borough names from the group names rather than applying unique to the boroname column.

This version runs faster than the previous version, but it turns out there is an even better way to compute this result. This particular task, grouping by a field or fields and then counting the number of items in each group, is so common that the DataFrameGroupBy class includes a method, named size, for this computation. Using this method, we can count the number of trees per borough in one line:

```
>>> trees.groupby("boroname").size()
boroname
Bronx             85203
Brooklyn         177293
Manhattan         65423
Queens           250551
Staten Island    105318
dtype: int64
```

How do we build on this code to compute the number of live trees, dead trees, and stumps for each borough? Recall that this information is available for each tree in the status field. We can group the data using this field along with the borough to get the information we want:

```
>>> status_per_boro = trees.groupby(["boroname", "status"]).size()
>>> status_per_boro
boroname       status
Bronx          Alive      80585
               Dead        2530
               Stump       2088
Brooklyn       Alive     169744
               Dead        3319
               Stump       4230
Manhattan      Alive      62427
               Dead        1802
               Stump       1194
Queens         Alive     237974
               Dead        4440
               Stump       8137
Staten Island  Alive     101443
               Dead        1870
               Stump       2005
dtype: int64
```

What might not be immediately clear from this output is that status_per_boro is a Series with a hierarchical index, not a data frame.

```
>>> status_per_boro.index
MultiIndex([(        'Bronx', 'Alive'),
            (        'Bronx',  'Dead'),
            (        'Bronx', 'Stump'),
            (     'Brooklyn', 'Alive'),
            (     'Brooklyn',  'Dead'),
            (     'Brooklyn', 'Stump'),
            (    'Manhattan', 'Alive'),
            (    'Manhattan',  'Dead'),
            (    'Manhattan', 'Stump'),
```

(continues on next page)

(continued from previous page)

```
            (       'Queens', 'Alive'),
            (       'Queens',  'Dead'),
            (       'Queens', 'Stump'),
            ('Staten Island', 'Alive'),
            ('Staten Island',  'Dead'),
            ('Staten Island', 'Stump')],
           names=['boroname', 'status'])
```

We can extract the information for a given borough as series using square brackets and the name of the borough:

```
>>> status_per_boro["Bronx"]
status
Alive    80585
Dead      2530
Stump     2088
dtype: int64
```

and we can extract a specific value, say, the number of live trees, for a specific borough using either two sets of square brackets or one set of square brackets and a tuple:

```
>>> status_per_boro["Bronx"]["Alive"]
80585
>>> status_per_boro["Bronx", "Alive"]
80585
```

Our desired result is a data frame, not a series with a hierarchical index, but before we convert the result to the right type, let’s add some code to calculate percentages rather than the counts:

```
>>> pct_per_boro = status_per_boro/trees.groupby("boroname").size()*100
>>> pct_per_boro
boroname       status
Bronx          Alive     94.580003
               Dead       2.969379
               Stump      2.450618
Brooklyn       Alive     95.742077
               Dead       1.872042
               Stump      2.385881
Manhattan      Alive     95.420571
               Dead       2.754383
               Stump      1.825046
Queens         Alive     94.980263
               Dead       1.772094
               Stump      3.247642
Staten Island  Alive     96.320667
               Dead       1.775575
               Stump      1.903758
dtype: float64
```

Notice that while the numerator of the division operation is a series with a hierarchical index, the denominator is a series with a flat index. Pandas uses Numpy-style broadcasting to convert the denominator into something like this value:

```
boroname       status
Bronx          Alive      85203
               Dead       85203
               Stump      85203
Brooklyn       Alive     177293
               Dead      177293
               Stump     177293
Manhattan      Alive      65423
               Dead       65423
               Stump      65423
Queens         Alive     250551
               Dead      250551
               Stump     250551
Staten Island  Alive     105318
               Dead      105318
               Stump     105318
dtype: float64
```

before performing the division. Similarly, the value 100 is broadcast into a series with the right shape before the multiplication is done.

We now have the right values in the wrong form. We need to convert the series into a data frame with the borough names as the index and the different values for status (`'Alive'`, `'Dead'`, and `'Stump'`) as the column labels. There are a couple of ways to accomplish this task. For now, we'll describe the most straight-forward approach: use the `unstack` method, which converts a series with a hierarchical index into a data frame. By default it uses the lowest level of the hierarchy as the column labels in the new data frame and the remaining levels of the hierarchical index as the index of the new data frame. In our example, `unstack` will use the `boroname` as the data frame index and the values of `status` as the column labels, which is exactly what we want:

```
>>> pct_per_boro.unstack()
status             Alive      Dead     Stump
boroname
Bronx          94.580003  2.969379  2.450618
Brooklyn       95.742077  1.872042  2.385881
Manhattan      95.420571  2.754383  1.825046
Queens         94.980263  1.772094  3.247642
Staten Island  96.320667  1.775575  1.903758
```

We've walked through several steps to get to this point, let's put them together before we move on:

```
>>> counts_per_boro = trees.groupby("boroname").size()
>>> status_per_boro = trees.groupby(["boroname", "status"]).size()
>>> pct_per_boro = status_per_boro/counts_per_boro * 100.0
>>> pct_per_boro_df = pct_per_boro.unstack()
>>> pct_per_boro_df
status             Alive      Dead     Stump
boroname
Bronx          94.580003  2.969379  2.450618
Brooklyn       95.742077  1.872042  2.385881
Manhattan      95.420571  2.754383  1.825046
Queens         94.980263  1.772094  3.247642
Staten Island  96.320667  1.775575  1.903758
```

We are now close to answering the question of how tree health differs by borough. The last step is to replace the `Alive` column in our current result with three new columns— Good, Fair, and Poor—using information in the health column.

A common way to do this type of task is to compute and then process a new column with the combined information—health for live trees and status for dead trees and stumps. We'll use the Pandas where method to compute the new column (combined) and then add it to the trees data frame using an assignment statement.

```
>>> trees["combined"] = trees.status.where(trees.status != "Alive", trees.health)
```

The where method will return a series with the same shape and index as trees.status. A given entry in this series will hold the same value as the corresponding entry in trees.status if the corresponding entry has a value other than "Alive" and the corresponding value from trees.health, if not. The where method uses the index to identify corresponding entries. In this example, every index value in trees.status appears in both the condition and in what is known as the *other* argument (tree.health, in our example). When a given index is missing from either the condition or from the other argument, where will insert a NaN into the result.

Here's some sample data from the result:

```
>>> trees[629:632]
        status health        spc_common  boroname   latitude  longitude combined
tree_id
179447   Alive   Good  northern red oak  Brooklyn  40.637491 -73.955789     Good
192569    Dead    NaN               NaN    Queens  40.738044 -73.921552     Dead
179766   Stump    NaN               NaN  Brooklyn  40.637379 -73.953814    Stump
```

Rows 629-631 happen to contain information about a live tree, a dead tree, and a stump. Notice that the new combined column contains the health of the first tree, which is live, but the status for the other two, which are not.

Once we have completed our computation, we can drop the combined column from the data frame using drop with axis=1. Putting it all together, we get this function, which computes a data frame with information about how tree health differs by borough:

```
>>> def tree_health_by_boro(trees):
...     trees["combined"] = trees.status.where(trees.status != "Alive",
...                                            trees.health)
...     num_per_boro = trees.groupby("boroname").size()
...     combined_per_boro = trees.groupby(["boroname","combined"]).size()
...     pct_per_boro = combined_per_boro/num_per_boro*100.0
...     pct_per_boro_df = pct_per_boro.unstack()
...     trees.drop("combined", axis=1)
...     return pct_per_boro_df[["Good", "Fair", "Poor", "Dead", "Stump"]]
...
...
>>> tree_health_by_boro(trees)
combined            Good       Fair      Poor      Dead     Stump
boroname
Bronx          78.169783  12.777719  3.632501  2.969379  2.450618
Brooklyn       77.956829  14.142126  3.643122  1.872042  2.385881
Manhattan      72.387387  17.516775  5.516409  2.754383  1.825046
Queens         77.432539  13.789209  3.758516  1.772094  3.247642
Staten Island  78.494654  13.801060  4.024003  1.775575  1.903758
```

By default, unstack will order the columns by value, which is not ideal for our purposes. Our function solves this problem by indexing the result of unstack with a list of the columns in the preferred order.

It seems a little silly to add a column and almost immediately remove it. And, in fact, we don't actually need to add the combined data to the data frame to use it in a call groupby. In addition to specifying the groups using one or more column names, we can also specify the groups using one or more series. We can even combine the two approaches: use one or more column names and one or more series.

Let's look at what happens when we use a single series to specify the groups:

```
>>> combined_status = trees.status.where(trees.status != "Alive",
...                                      trees.health)
>>> trees.groupby(combined_status).size()
status
Dead      13961
Fair      96504
Good     528850
Poor      26818
Stump     17654
dtype: int64
```

The groups are determined by the values `combined_status`. A row from `tree` is included in a given group, if `combined_status` contains an entry with the same index and the associated value matches the group's label. For example, `177922` appears as an index value in both `trees` and `combined_status`:

```
>>> combined_status.loc[177922]
'Stump'
>>> trees.loc[177922]
status                 Stump
health                   NaN
spc_common               NaN
boroname       Staten Island
latitude             40.5285
longitude           -74.1652
combined               Stump
Name: 177922, dtype: object
```

This tree will be in the `'Stump'` group, because that's the value of tree `177922` in `combined_status.loc`. In contrast, tree `180683` will be in group `'Fair'`, because that's its value in `combined_status`:

```
>>> combined_status.loc[180683]
'Fair'
>>> trees.loc[180683]
status              Alive
health               Fair
spc_common      red maple
boroname           Queens
latitude          40.7231
longitude        -73.8442
combined             Fair
Name: 180683, dtype: object
```

In this case, every index value in `trees` has a corresponding value in the index for `combined_status`. If `trees` had contained an index that did not occur in `combined_status`, then the corresponding row would not have appeared in any group. Similarly, an index that appeared in `combined_status` and not `trees` would not have an impact on the result. This process of matching up the indices and discarding entries that do not have mates is known as an *inner join*.

While this example used a single series, we can also use a list or tuple of series to specify the groups. In this case, a row from the data frame is included in the result if its index occurs in all the series in the list. The row's group is determined by creating a tuple using its index to extract the corresponding values from each of the series.

We can also mix column names and series. In this case, you can think of the column name as a proxy for the column as a series. That is, trees.groupby("boroname", "status") is the same as trees.groupby([trees.boroname, trees.status]).

Using this approach, we can skip adding the "combined" field to trees and just use the series directly in the groupby:

```
>>> def tree_health_by_boro(trees):
...     combined_status = trees.status.where(trees.status != "Alive",
...                                          trees.health)
...     num_per_boro = trees.groupby("boroname").size()
...     combined_per_boro = trees.groupby(["boroname",combined_status]).size()
...     pct_per_boro = combined_per_boro/num_per_boro*100.0
...     pct_per_boro_df = pct_per_boro.unstack()
...     return pct_per_boro_df[["Good", "Fair", "Poor", "Dead", "Stump"]]
...
...
>>> tree_health_by_boro(trees)
status              Good       Fair      Poor      Dead     Stump
boroname
Bronx          78.169783  12.777719  3.632501  2.969379  2.450618
Brooklyn       77.956829  14.142126  3.643122  1.872042  2.385881
Manhattan      72.387387  17.516775  5.516409  2.754383  1.825046
Queens         77.432539  13.789209  3.758516  1.772094  3.247642
Staten Island  78.494654  13.801060  4.024003  1.775575  1.903758
```

17.6 Pivoting

In the previous section, we used unstack to convert a series with a hierarchical index into a data frame with a flat index. In this section, we'll look at another way to handle the same task using the pct_per_boro series:

```
>>> pct_per_boro
boroname       status
Bronx          Alive     94.580003
               Dead       2.969379
               Stump      2.450618
Brooklyn       Alive     95.742077
               Dead       1.872042
               Stump      2.385881
Manhattan      Alive     95.420571
               Dead       2.754383
               Stump      1.825046
Queens         Alive     94.980263
               Dead       1.772094
               Stump      3.247642
Staten Island  Alive     96.320667
               Dead       1.775575
               Stump      1.903758
dtype: float64
```

As first step, we will convert the series into a data frame using the series' to_frame method:

```
>>> pct_per_boro_df = pct_per_boro.to_frame()
>>> pct_per_boro_df
                                0
boroname      status
Bronx         Alive   94.580003
              Dead     2.969379
              Stump    2.450618
Brooklyn      Alive   95.742077
              Dead     1.872042
              Stump    2.385881
Manhattan     Alive   95.420571
              Dead     2.754383
              Stump    1.825046
Queens        Alive   94.980263
              Dead     1.772094
              Stump    3.247642
Staten Island Alive   96.320667
              Dead     1.775575
              Stump    1.903758
```

The resulting data frame retains the hierarchical index from the series and has a single column with name `0`. Using `reset_index`, we can shift the hierarchical index values into columns and construct a new range index:

```
>>> pct_per_boro_df = pct_per_boro_df.reset_index()
>>> pct_per_boro_df
         boroname status          0
0           Bronx  Alive  94.580003
1           Bronx   Dead   2.969379
2           Bronx  Stump   2.450618
3        Brooklyn  Alive  95.742077
4        Brooklyn   Dead   1.872042
5        Brooklyn  Stump   2.385881
6       Manhattan  Alive  95.420571
7       Manhattan   Dead   2.754383
8       Manhattan  Stump   1.825046
9          Queens  Alive  94.980263
10         Queens   Dead   1.772094
11         Queens  Stump   3.247642
12  Staten Island  Alive  96.320667
13  Staten Island   Dead   1.775575
14  Staten Island  Stump   1.903758
```

You'll notice that this data frame is long and thin as opposed to short and wide, as is desired. Before we convert the data frame into the appropriate shape, let's rename the column labelled `0` into something more descriptive:

```
>>> pct_per_boro_df.rename(columns={0:"percentage"}, inplace=True)
```

Finally, we can use the data frame `pivot` method to convert a long and thin data frame into a short and wide data frame. This function takes three parameters: a column name to use as the index for the new data frame, a column to use to make the new data frames' column names, and the column to use for filling the new data frame. In our case, we'll use the borough name as the index, the `status` column to supply the column names, and the recently renamed `percentage` column to supply the values:

```
>>> pct_per_boro_pvt = pct_per_boro_df.pivot("boroname", "status", "percentage")
>>> pct_per_boro_pvt
status             Alive      Dead     Stump
boroname
Bronx          94.580003  2.969379  2.450618
Brooklyn       95.742077  1.872042  2.385881
Manhattan      95.420571  2.754383  1.825046
Queens         94.980263  1.772094  3.247642
Staten Island  96.320667  1.775575  1.903758
```

Because every borough has at least one tree of each status, values are available for all the entries in pivot result. If a combination is not available, `pivot` fills in a `NaN`. We can see an example of this by computing a pivot table for tree species per boro:

```
>>> species_per_boro = trees.groupby(["spc_common", "boroname"]).size()
>>> species_per_boro_df = species_per_boro.to_frame().reset_index()
>>> species_per_boro_pvt = species_per_boro_df.pivot("spc_common", "boroname", 0)
```

and then by examining the resulting value for a species named Atlas cedar:

```
>>> species_per_boro_pvt.loc["Atlas cedar"]
boroname
Bronx             6.0
Brooklyn         34.0
Manhattan         NaN
Queens           39.0
Staten Island     8.0
Name: Atlas cedar, dtype: float64
```

Notice that the entry for `Manhattan` is `NaN`.

In case you are curious, here's the code that we used to identify a tree to use as an example:

```
>>> species_per_boro_pvt[species_per_boro_pvt.isnull().any(axis=1)]
boroname        Bronx  Brooklyn  Manhattan  Queens  Staten Island
spc_common
Atlas cedar       6.0      34.0        NaN    39.0            8.0
Shantung maple    9.0      12.0        NaN    27.0           11.0
Virginia pine     1.0       3.0        3.0     3.0            NaN
false cypress    13.0      35.0        NaN    51.0            9.0
trident maple     7.0      12.0        NaN    68.0           23.0
```

Let's pull this expression apart. The expression `species_per_boro_pvt.isnull()` yields a data frame of booleans with the same shape as `species_per_boro_pvt`, that is 132 (species) by 5 (boroughs). A given entry is `True` if the corresponding entry in `species_per_boro_pvt` is `NaN`. We use a reduction to convert this value into a series of booleans, where the ith entry is `True` if the ith row contains at least one `True`. Recall from our discussion of reductions in the Numpy chapter, that we use an axis of one when to apply the reduction function to the rows. Also, recall that `any` returns `True` if at least one value in the input is `True`. Putting this together: the expression `species_per_boro_pvt.isnull().any(axis=1)` yields a series of booleans, one per tree type, that will be `True` if the corresponding entry in `species_per_boro_pvt` contains a `NaN` for at least one borough. Finally, we use this series as a filter to the original pivot table to extract the desired entries.

While we'll admit that this code computes a somewhat esoteric result—species that occur in some, but not all five boroughs—it is impressive how little code is required to extract this information from the data set.

As an aside, in this case, it makes sense to replace the null values with zeros. We can do this task easily using the `fillna` method:

```
>>> species_per_boro_pvt.fillna(0, inplace=True)
>>> species_per_boro_pvt.loc["Atlas cedar"]
boroname
Bronx             6.0
Brooklyn         34.0
Manhattan         0.0
Queens           39.0
Staten Island     8.0
Name: Atlas cedar, dtype: float64
```

As in other cases, adding `inplace=True` as a parameter instructs `fillna` to make the changes in place rather construct a new data frame.

17.7 Saving space with Categoricals

Our tree data set contains nearly 700,000 rows, so a natural question to ask is: how large is the memory footprint? We can answer this question using the `info` method for data frames. Like many Pandas methods, `info` has a variety of options. For our purposes, we'll use the `verbose=False` option, which reduces the amount of information generated by the method and `memory_usage="deep"` option, which tells `info` to include all the memory costs associated with the data frame. Before we run `info`, we'll reload the data from the file to clean up any extra columns left over from our earlier computation:

```
>>> trees = get_tree_data("2015StreetTreesCensus_TREES.csv")
>>> trees.info(verbose=False, memory_usage="deep")
<class 'pandas.core.frame.DataFrame'>
Int64Index: 683788 entries, 180683 to 200607
Columns: 6 entries, status to longitude
dtypes: float64(2), object(4)
memory usage: 181.1 MB
```

The last line of the output tells us that our `trees` data frame uses more than 180MB of space, which is non-trivial. One way to reduce this amount is to replace strings with categoricals, which are more space efficient. Categorical variables are used to represent features that can take on a small number of values, such as, borough names or the `status` and `health` fields of our tree data set. Though we can represent these values as strings, it is more space efficient to represent them using the Pandas `Categorical` class, which uses small integers as the underlying representation. This efficiency may not matter for a small dataset, but it can be significant for a large dataset.

We can construct a categorical variable from the values in a series using the `astype` method with `"category"` as the type. Here's some code that displays a slice with borough names for the first five entries in the `trees` data frame, constructs a new series from the `boroname` field using a categorical variable, and then shows the first five values of the new series:

```
>>> trees.boroname[:5]
tree_id
180683      Queens
200540      Queens
204026    Brooklyn
204337    Brooklyn
189565    Brooklyn
Name: boroname, dtype: object
```

(continues on next page)

(continued from previous page)

```
>>> boro_cat = trees.boroname.astype("category")
>>> boro_cat[:5]
tree_id
180683      Queens
200540      Queens
204026    Brooklyn
204337    Brooklyn
189565    Brooklyn
Name: boroname, dtype: category
Categories (5, object): ['Bronx', 'Brooklyn', 'Manhattan', 'Queens', 'Staten Island']
```

Notice that the `dtype` has changed from `object` to `category` and that the category has five values `Bronx`, `Brooklyn`, etc.

Using this simple mechanism dramatically reduces the memory footprint of the trees data. Specifically, we can reduce the amount of memory needed by a factor of 10 by converting the `boroname`, `health`, `spc_common`, and `status` fields from strings to categoricals:

```
>>> for col_name in ["boroname", "health", "spc_common", "status"]:
...     trees[col_name] = trees[col_name].astype("category")
...
...
>>> trees.info(verbose=False, memory_usage="deep")
<class 'pandas.core.frame.DataFrame'>
Int64Index: 683788 entries, 180683 to 200607
Columns: 6 entries, status to longitude
dtypes: category(4), float64(2)
memory usage: 18.9 MB
```

Another way to create a `Categorical` is to define a set of labelled bins and use them along with the method `pd.cut` to construct a categorical variable from a series with numeric values. Our tree data does not have a natural example for this type of categorical, so we'll use ten sample diastolic blood pressure values as an example:

```
>>> dbp_10 = pd.Series([92, 74, 80, 60, 72, 90, 84, 74, 75, 70])
```

We might want to label values below 80 as "normal", values between 80 and 90 as pre-hypertension, and values at 90 or above as high. To define the bin boundaries, we specify an ordered list of values:

```
>>> dbp_bins = [0.0, 80, 90, float("inf")]
```

The values are paired in sequence to create the bin boundaries : [0.0, 80], (80, 90], and (90, float("inf")]. By default, the right value in each pair is not included in the bin. Also, by default the first interval does not include the smallest value. Both of these defaults can be changed using optional parameters named `right` and `include_lowest` respectively. Somewhat counter-intuitively, `right` needs to be set to `False`, if you want to include the rightmost edge in the bin.

The bin labels are specified as a list:

```
>>> dbp_labels = ["normal", "pre-hypertension", "high"]
```

The number of labels should match the numbers of bins, which means this list is one shorter than the list of floats used to define the bin boundaries.

Give these bin boundaries and labels, we can create a categorical variable from the sample diastolic blood pressures using `pd.cut`:

```
>>> pd.cut(dbp_10, dbp_bins, labels=dbp_labels, right=False)
0                high
1              normal
2    pre-hypertension
3              normal
4              normal
5                high
6    pre-hypertension
7              normal
8              normal
9              normal
dtype: category
Categories (3, object): ['normal' < 'pre-hypertension' < 'high']
```

As expected from the description above, patients 3, 4, 7-9 are labelled as having “normal” diastolic blood pressure, patients 2 and 6 are labelled with “pre-hypertension” and patients 0 and 5 are labelled has having high diastolic blood pressure.

17.8 Summary

Pandas is very complex library and we have barely skimmed the surface of its many useful features. We strongly encourage you to look at the documentation to explore the wealth of options it provides.