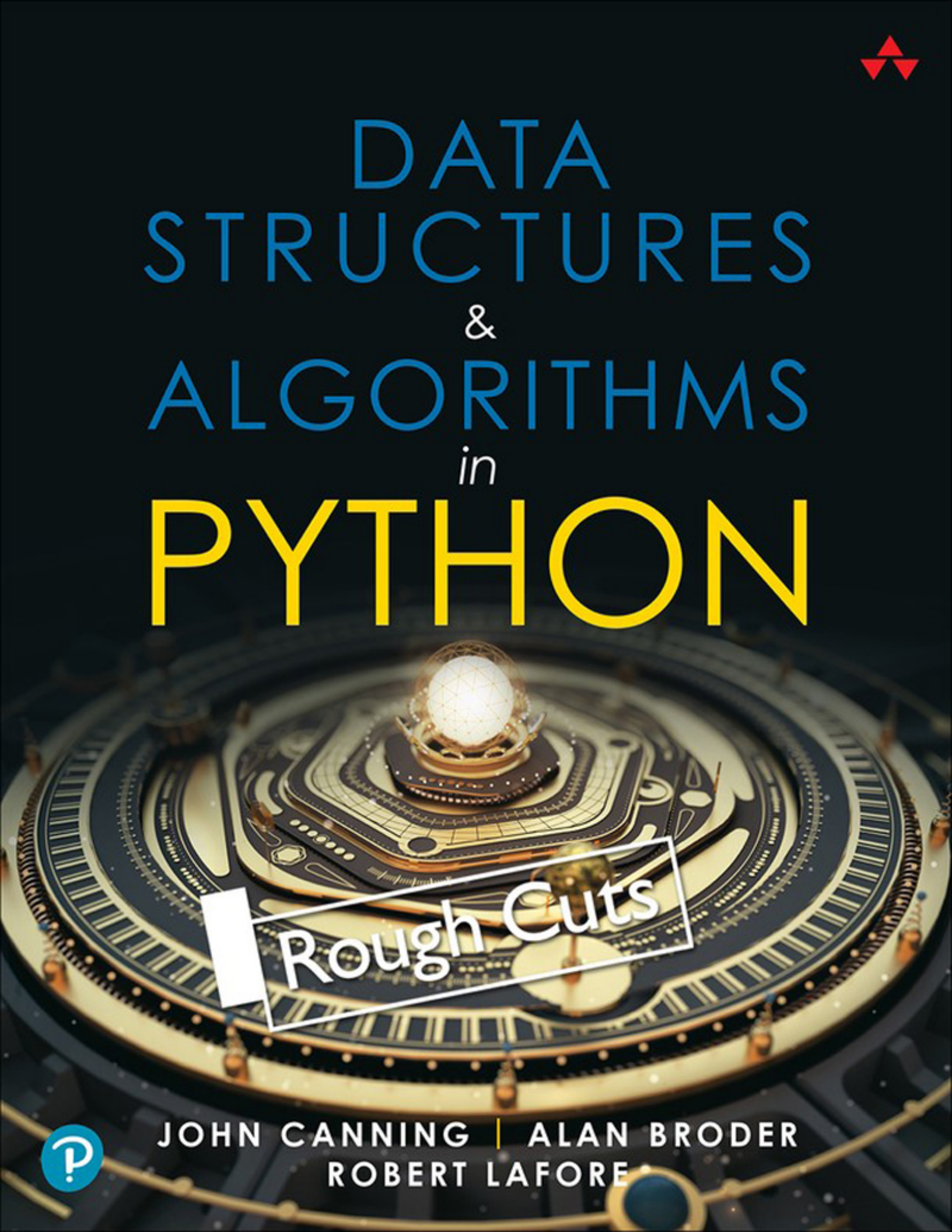

DATA
STRUCTURES
&
ALGORITHMS
in
PYTHON
Rough Cuts
JOHN CANNING | ALAN BRODER
ROBERT LAFORE

# Data Structures & Algorithms in Python

# Data Structures & Algorithms in Python

**John Canning**
**Alan Broder**
**Robert Lafore**

Addison-Wesley

**Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town**
**Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City**
**São Paulo • Sidney • Hong Kong • Seoul • Singapore • Taipei • Tokyo**

Library of Congress Control Number: 2022910068



ISBN-13: 978-0-13-485568-4
ISBN-10: 0-13-485568-X

**ScoutAutomatedPrintCode**

Editor-in-Chief

Mark Taub

Director, ITP Product Management

Brett Bartow

Acquisitions Editor

Kim Spenceley

Development Editor

Chris Zahn

Managing Editor

Sandra Schroeder

Project Editor

Mandie Frank

Copy Editor

Chuck Hutchinson
Indexer
Proofreader
Editorial Assistant
Cindy Teeters
Designer
Chuti Prasertsith
Compositor
codeMantra

# Pearson's Commitment to Diversity, Equity, and Inclusion

Pearson is dedicated to creating bias-free content that reflects the diversity of all learners. We embrace the many dimensions of diversity, including but not limited to race, ethnicity, gender, socioeconomic status, ability, age, sexual orientation, and religious or political beliefs.

Education is a powerful force for equity and change in our world. It has the potential to deliver opportunities that improve lives and enable economic mobility. As we work with authors to create content for every product and service, we acknowledge our responsibility to demonstrate inclusivity and incorporate diverse scholarship so that everyone can achieve their potential through learning. As the world's leading learning company, we have a duty to help drive change and live up to our purpose to help more people create a better life for themselves and to create a better world.

Our ambition is to purposefully contribute to a world where

- Everyone has an equitable and lifelong opportunity to succeed through learning
- Our educational products and services are inclusive and represent the rich diversity of learners
- Our educational content accurately reflects the histories and experiences of the learners we serve
- Our educational content prompts deeper discussions with learners and motivates them to expand their own learning (and worldview)

While we work hard to present unbiased content, we want to hear from you about any concerns or needs with this Pearson product so that we can

investigate and address them.

Please contact us with concerns about any potential bias at https://www.pearson.com/report-bias.html.

To my mother, who gave me a thirst for knowledge, to my father, who taught me the joys of engineering, and to June, who made it possible to pursue both.

*John Canning*

For my father Sol Broder, a computer science pioneer, who inspired me to follow in his footsteps.

To my mother Marilyn Broder, for showing me the satisfaction of teaching others.

To Fran, for making my life complete.

*Alan Broder*

# Contents

# Table of Contents

Register your copy of *Data Structures & Algorithms in Python* at informit.com for convenient access to downloads, updates, and corrections as they become available. To start the registration process, go to informit.com/register and log in or create an account. Enter the product ISBN 9780134855684 and click Submit. Once the process is complete, you will find any available bonus content under “Registered Products.”

# Acknowledgments

*From John Canning and Alan Broder*

Robert Lafore's Java-based version of this book has been a mainstay in Data Structures courses and professionals' reference shelves around the world for many years. When Alan's Data Structures course at Stern College for Women of Yeshiva University moved on to Python, the inability to use Lafore's book in the course was a real loss. We're thus especially happy to bring this new and revised edition to the world of Python programmers and students.

We'd like to thank the many students at Stern who contributed to this book either directly or indirectly over the past several years. Initial Python versions of Lafore's Java implementations were central to Alan's Python-based courses, and Stern student feedback helped improve the code's clarity, enhanced its performance, and sometimes even identified and fixed bugs!

For their valuable feedback and recommendations on early drafts of this new edition, we are grateful to many students in Alan's Data Structures courses, including Estee Brooks, Adina Bruce, Julia Chase, Hanna Fischer, Limor Kohanim, Elisheva Kohn, Shira Orlian, Shira Pahmer, Jennie Peled, Alexandra Roffe, Avigail Royzenberg, Batia Segal, Penina Waghalter, and Esther Werblowsky. Our apologies if we've omitted anyone's name.

An open-source package of data structure visualizations is available to enhance your study of this book, and Stern students played an active role in the development of the visualization software. John and Alan extend many thanks to the Stern student pioneers and leaders of this project, including Ilana Radinsky, Elana Apfelbaum, Ayliana Teitelbaum, and Lily Polonetsky, as well as the following past and present Stern student contributors: Zoe Abboudi, Ayelet Aharon, Lara Amar, Natania Birnbaum, Adina Bruce, Chani Dubin, Sarah Engel, Sarah Graff, Avigayil Helman, Michal

Kaufman, Sarina Kofman, Rachel Leiser, Talia Leitner, Shani Lewis, Rina Melincoff, Atara Neugroschl, Shira Pahmer, Miriam Rabinovich, Etta Rapp, Shira Sassoon, Mazal Schoenwald, Shira Schneider, Shira Smith, Riva Tropp, Alexandra Volchek, and Esther Werblowsky. Our apologies if we have left anyone off this list.

Many thanks go to Professor David Matuszek of the University of Pennsylvania for his early contributions of ideas and PowerPoint slides when Alan first started teaching Data Structures at Stern. Many of the slides available in the Instructors Resources section have their origin in his clear and well-designed slides. Also, we are grateful to Professor Marian Gidea of the Department of Mathematics of Yeshiva University for his insights into spherical trigonometry.

Finally, we owe a great debt to the talented editors at Pearson who made this book a reality: Mark Taber, Kim Spenceley, Mandie Frank, and Chris Zahn. Without their many talents and patient help, this project would just be an odd collection of text files, drawings, and source code.

*From Robert Lafore for the Java-based versions of the book*

# Acknowledgments to the First Edition

My gratitude for the following people (and many others) cannot be fully expressed in this short acknowledgment. As always, Mitch Waite had the Java thing figured out before anyone else. He also let me bounce the applets off him until they did the job, and extracted the overall form of the project from a miasma of speculation. My editor, Kurt Stephan, found great reviewers, made sure everyone was on the same page, kept the ball rolling, and gently but firmly ensured that I did what I was supposed to do. Harry Henderson provided a skilled appraisal of the first draft, along with many valuable suggestions. Richard S. Wright, Jr., as technical editor, corrected numerous problems with his keen eye for detail. Jaime Niño, Ph.D., of the University of New Orleans, attempted to save me from myself and occasionally succeeded, but should bear no responsibility for my approach or coding details. Susan Walton has been a staunch and much-appreciated supporter in helping to convey the essence of the project to the nontechnical. Carmela Carvajal was invaluable in extending our contacts

with the academic world. Dan Scherf not only put the CD-ROM together, but was tireless in keeping me up to date on rapidly evolving software changes. Finally, Cecile Kaufman ably shepherded the book through its transition from the editing to the production process.

## Acknowledgments to the Second Edition

My thanks to the following people at Sams Publishing for their competence, effort, and patience in the development of this second edition. Acquisitions Editor Carol Ackerman and Development Editor Songlin Qiu ably guided this edition through the complex production process. Project Editor Matt Purcell corrected a semi-infinite number of grammatical errors and made sure everything made sense. Tech Editor Mike Kopak reviewed the programs and saved me from several problems. Last but not least, Dan Scherf, an old friend from a previous era, provides skilled management of my code and applets on the Sams website.

# About the Author

**Dr. John Canning** is an engineer, computer scientist, and researcher. He earned an S.B. degree in electrical engineering from the Massachusetts Institute of Technology and a Ph.D. in Computer Science from the University of Maryland at College Park. His varied professions include being a professor of computer science, a researcher and software engineer in industry, and a company vice president. He now is president of Shakumant Software.

**Alan Broder** is clinical professor and chair of the Department of Computer Science at Stern College for Women of Yeshiva University in New York City. He teaches introductory and advanced courses in Python programming, data structures, and data science. Before joining Stern College, he was a software engineer, designing and building large-scale data analysis systems. He founded and led White Oak Technologies, Inc. as its CEO, and later served as the chairman and fellow of its successor company, Novetta, in Fairfax, Virginia.

# Introduction

What's in this book? This book is designed to be a practical introduction to data structures and algorithms for students who have just begun to write computer programs. This introduction will tell you more about the book, how it is organized, what experience we expect readers will have before starting the book, and what knowledge you will get by reading it and doing the exercises.

## Who This Book Is For

Data structures and algorithms are the core of computer science. If you've ever wanted to understand what computers can do, how they do it, and what they can't do, then you need a deep understanding of both (it's probably better to say "what computers have difficulty doing" instead of what they can't do). This book may be used as a text in a data structures and/or algorithms course, frequently taught in the second year of a university computer science curriculum. The text, however, is also designed for professional programmers, for high school students, and for anyone else who needs to take the next step up from merely knowing a programming language. Because it's easy to understand, it is also appropriate as a supplemental text to a more formal course. It is loaded with examples, exercises, and supplemental materials, so it can be used for self-study outside of a classroom setting.

Our approach in writing this book is to make it easy for readers to understand how data structures operate and how to apply them in practice. That's different from some other texts that emphasize the mathematical theory, or how those structures are implemented in a particular language or software library. We've selected examples with real-world applications and avoid using math-only or obscure examples. We use figures and visualization programs to help communicate key ideas. We still cover the

complexity of the algorithms and the math needed to show how complexity impacts performance.

## What You Need to Know Before You Read This Book

The prerequisites for using this book are: knowledge of some programming language and some mathematics. Although the sample code is written in Python, you don't need to know Python to follow what's happening. Python is not hard to understand, if you've done some procedural and/or object-oriented programming. We've kept the syntax in the examples as general as possible,

More specifically, we use Python version 3 syntax. This version differs somewhat from Python 2, but not greatly. Python is a rich language with many built-in data types and libraries that extend its capabilities. Our examples, however, use the more basic constructs for two reasons: it makes them easier to understand for programmers familiar with other languages, and it illustrates the details of the data structures more explicitly. In later chapters, we do make use of some Python features not found in other languages such as generators and list comprehensions. We explain what these are and how they benefit the programmer.

Of course, it will help if you're already familiar with Python (version 2 or 3). Perhaps you've used some of Python's many data structures and are curious about how they are implemented. We review Python syntax in Chapter 1, "Overview," for those who need an introduction or refresher. If you've programmed in languages like Java, C++, C#, JavaScript, or Perl, many of the constructs should be familiar. If you've only programmed using functional or domain-specific languages, you may need to spend more time becoming familiar with basic elements of Python. Beyond this text, there are many resources available for novice Python programmers, including many tutorials on the Internet.

Besides a programming language, what should every programmer know? A good knowledge of math from arithmetic through algebra is essential. Computer programming is symbol manipulation. Just like algebra, there are ways of transforming expressions to rearrange terms, put them in different

forms, and make certain parts more prominent, all while preserving the same meaning. It's also critical to understand exponentials in math. Much of computer science is based on knowing what raising one number to a power of another means. Beyond math, a good sense of organization is also beneficial for all programming. Knowing how to organize items in different ways (by time, by function, by size, by complexity, and so on) is crucial to making programs efficient and maintainable. When we talk about efficiency and maintainability, they have particular meanings in computer science. *Efficiency* is mostly about how much time it takes to compute things but can also be about the amount of space it takes. *Maintainability* refers to the ease of understanding and modifying your programs by other programmers as well as yourself.

You'll also need knowledge of how to find things on the Internet, download and install software, and run them on a computer. The instructions for downloading and running the visualization programs can be found in Appendix A of this book. The Internet has made it very easy to access a cornucopia of tools, including tools for learning programming and computer science. We expect readers to already know how to find useful resources and avoid sources that might provide malicious software.

# What You Can Learn from This Book

As you might expect from its title, this book can teach you about how data structures make programs (and programmers) more efficient in their work. You can learn how data organization and its coupling with appropriate algorithms greatly affect what can be computed with a given amount of computing resources. This book can give you a thorough understanding of how to implement the data structures, and that should enable you to implement them in any programming language. You can learn the process of deciding what data structure(s) and algorithms are the most appropriate to meet a particular programming request. Perhaps most importantly, you can learn when an algorithm and/or data structure will *fail* in a given use case. Understanding data structures and algorithms is the core of computer science, which is different from being a Python (or other language) programmer.

The book teaches the fundamental data structures that every programmer should know. Readers should understand that there are many more. These basic data structures work in a wide variety of situations. With the skills you develop in this book, you should be able to read a description of another data structure or algorithm and begin to analyze whether or not it will outperform or perform worse than the ones you've already learned in particular use cases.

This book explains some Python syntax and structure, but it will not teach you all its capabilities. The book uses a subset of Python's full capabilities to illustrate how more complex data structures are built from the simpler constructs. It is not designed to teach the basics of programming to someone who has never programmed. Python is a very high-level language with many built-in data structures. Using some of the more primitive types such as arrays of integers or record structures, as you might find in C or C++, is somewhat more difficult in Python. Because the book's focus is the implementation and analysis of data structures, our examples use approximations to these primitive types. Some Python programmers may find these examples unnecessarily complex, knowing about the more elegant constructs provided with the language in standard libraries. If you want to understand computer science, and in particular, the complexity of algorithms, you must understand the underlying operations on the primitives. When you use a data structure provided in a programming language or from one of its add-on modules, you will often have to know its complexity to know whether it will work well for your use case. Understanding the core data structures, their complexities, and trade-offs will help you understand the ones built on top of them.

All the data structures are developed using ***object-oriented programming*** (OOP). If that's a new concept for you, the review in Chapter 1 of how classes are defined and used in Python provides a basic introduction to OOP. You should not expect to learn the full power and benefits of OOP from this text. Instead, you will learn to implement each data structure as a ***class***. These classes are the *types of objects* in OOP and make it easier to develop software that can be reused by many different applications in a reliable way.

The book uses many examples, but this is not a book about a particular application area of computer science such as databases, user interfaces, or

artificial intelligence. The examples are chosen to illustrate typical applications of programs, but all programs are written in a particular context, and that changes over time. A database program written in 1970 may have appeared very advanced at that time, but it might seem very trivial today. The examples presented in this text are designed to teach how data structures are implemented, how they perform, and how to compare them when designing a new program. The examples should not be taken as the most comprehensive or best implementation possible of each data structure, nor as a thorough review of all the potential data structures that could be appropriate for a particular application area.

## Structure

Each chapter presents a particular group of data structures and associated algorithms. At the end of the chapters, we provide review questions covering the key points in the chapter and sometimes relationships to previous chapters. The answers for these can be found in Appendix C, "Answers to Questions." These questions are intended as a self-test for readers, to ensure you understood all the material.

Many chapters suggest *experiments* for readers to try. These can be individual thought experiments, team assignments, or exercises with the software tools provided with the book. These are designed to apply the knowledge just learned to some other area and help deepen your understanding.

Programming projects are longer, more challenging programming exercises. We provide a range of projects of different levels of difficulty. These projects might be used in classroom settings as homework assignments. Sample solutions to the programming projects are available to qualified instructors from the publisher.

## History

Mitchell Waite and Robert Lafore developed the first version of this book and titled it *Data Structures and Algorithms in Java*. The first edition was published in 1998, and the second edition, by Robert, came out in 2002.

John Canning and Alan Broder developed this version using Python due to its popularity in education and commercial and noncommercial software development. Java is widely used and an important language for computer scientists to know. With many schools adopting Python as a first programming language, the need for textbooks that introduce new concepts in an already familiar language drove the development of this book. We expanded the coverage of data structures and updated many of the examples.

We've tried to make the learning process as painless as possible. We hope this text makes the core, and frankly, the beauty of computer science accessible to all. Beyond just understanding, we hope you find learning these ideas fun. Enjoy yourself!

# 1. Overview

You have written some programs and learned enough to think that programming is fun, or at least interesting. Some parts are easy, and some parts are hard. You'd like to know more about how to make the process easier, get past the hard parts, and conquer more complex tasks. You are starting to study the heart of computer science, and that brings up many questions. This chapter sets the stage for learning how to make programs that work properly and fast. It explains a bunch of new terms and fills in background about the programming language that we use in the examples.

### In This Chapter



## What Are Data Structures and Algorithms?

**Data organizations** are ways data is arranged in the computer using its various storage media (such as random-access memory, or RAM, and disk) and how that data is interpreted to represent something. **Algorithms** are the procedures used to manipulate the data in these structures. The way data is arranged can simplify the algorithms and make algorithms run faster or slower. Together, the data organization and the algorithm form a **data structure**. The data structures

act like building blocks, with more complex data structures using other data structures as components with appropriate algorithms.

Does the way data is arranged and the algorithm used to manipulate it make a difference? The answer is a definite yes. From the perspective of nonprogrammers, it often seems as though computers can do anything and do it very fast. That's not really true. To see why, let's look at a nonprogramming example.

When you cook a meal, a collection of ingredients needs to be combined and manipulated in specific ways. There is a huge variety of ways that you could go about the individual steps needed to complete the meal. Some of those methods are going to be more efficient than others. Let's assume that you have a written recipe, but are working in an unfamiliar kitchen, perhaps while visiting a friend. One method for making the meal would be *Method A*:

1. Read through the full recipe noting all the ingredients it mentions, their quantities, and any equipment needed to process them.
2. Find the ingredients, measure out the quantity needed, and store them.
3. Get out all the equipment needed to complete the steps in the recipe.
4. Go through the steps of the recipe in the order specified.

Let's compare that to *Method B*:

1. Read the recipe until you identify the first set of ingredients or equipment that is needed to complete the first step.
2. Find the identified ingredients or equipment.
3. Measure any ingredients found.
4. Perform the first step with the equipment and ingredients already found.
5. Return to the beginning of this method and repeat the instructions replacing the word *first* with *next*. If there is no next step, then quit.

Both methods are *complete* in that that they should finish the complete recipe if all the ingredients and equipment are available. For simple recipes, they should take about the same amount of time too. The methods differ as the recipes get more complex. For example, what if you can't find the fifth ingredient? In method A, that issue is identified at the beginning before any other ingredients are combined. While neither method really explains what to do about

exceptions like a missing ingredient, you can still compare them under the assumption that you handle the exceptions the same way.

A missing ingredient could be handled in several ways: find a substitute ingredient, broaden the search for the ingredient (look in other rooms, ask a neighbor, go to the market), or ignore the ingredient (an optional garnish). Each of those remedies takes some time. If there is one missing ingredient and two cooks using the different methods handle it in the same way, both are delayed the same amount of time. If there are multiple missing ingredients, however, Method A should identify those earlier and allow for the possibility of getting all the missing ingredients in one visit to a neighbor or a market. The time savings of combining the tasks of replacing missing ingredients could be significant (imagine the market being far away or neighbors who want to talk for hours on every visit).

The order of performing the operations could have significant impact on the time needed to complete the meal. Another difference could be in the quality of the meal. For example, in Method B the cook would perform the first “step” and then move on to the next step. Let’s assume those use two different groups of ingredients or equipment. If finding or measuring the ingredients, or getting the equipment for the later steps takes significant time, then the results of the first step sit around for a significant time. That can have a bad effect on the quality of the meal. The cook might be able to overcome that effect in some circumstances by, say, putting the results of the first step in a freezer or refrigerator and then bringing them back to room temperature later. The cook would be preserving the quality of the food at the expense of the time needed to prepare it.

Would Method B ever be desirable if it takes longer or risks degrading the quality of the food? Perhaps. Imagine that the cook is preparing this meal in the unfamiliar kitchen of a family relative. The kitchen is full of family members, and each one is trying to make part of the meal. In this crowded situation, it could be difficult for each cook to get out all of their ingredients and equipment at once. There might not be enough counter space, or mixing bowls, or knives, for example, for each cook to have all their items assembled at the beginning. The cooks could be constrained to work on individual steps while waiting for equipment, space, or ingredients to become available. In this case, Method B could have advantages over asking all the cooks to work one at a time using Method A.

Coming back to programming, the algorithm specifies the sequence of operations that are to be performed, much like the steps in the recipe. The data organizations are somewhat analogous to how the ingredients are stored, laid out in the kitchen, and their proximity to other ingredients and equipment. For example, having the ingredient in the kitchen makes the process much faster than if the ingredient needs to be retrieved from a neighbor or the market. You can think of the amount of space taken up by spreading out the ingredients in various locations as the amount of space needed for the algorithm. Even if all the ingredients are in the kitchen where the cook is, there are ways of setting up the ingredients to make the cooking tasks go faster. Having them compactly arranged in the order they are needed minimizes the amount of moving around the cook must do. The organization of the ingredients can also help if a cook must be replaced by another cook in the middle of the preparation; understanding where each of the ingredients fits in to the recipe is faster if the layout is organized. This is another reason why good data organization is important. It also reinforces that concept that the algorithm and the data organization work together to make the data structure.

Data structures are important not just for speed but also to properly model the meaning of the data. Let's say there's an event that many people want to attend and they need to submit their phone number to have a chance to get tickets. Each person can request multiple tickets. If there are fewer tickets than the number of people who want to get them, some method needs to be applied to decide which phone numbers to contact first and determine the number of tickets they will receive. One method would be to go through the phone numbers one at a time and total up the number of tickets until all are given out, then go through the remaining numbers to let them know the tickets are gone. That might be a fair method if the numbers were put in an ordered list defined in a way that potential recipients understood— for example, a chronological list of phone numbers submitted by people interested in the tickets. If the tickets are to be awarded as a sort of lottery, then going through them sequentially means following any bias that is implicit in the order of the list. Randomly choosing a number from the list, contacting the buyer, and then removing the number would be fairer in a lottery system.

Data structures *model systems by assigning specific meanings to each of the pieces and how they interact*. The "systems" are real-world things like first-come, first-served ticket sales, or a lottery giveaway, or how roads connect cities. For the list of phone numbers with ticket requests, a first-come, first-served system needs the list in chronological order, some pointer to where in

the list the next number should be taken, and a pointer to where any newly arriving number should be added (after all previous list entries). The lottery system would need a different organization, and modeling the map of roads and cities needs another. In this book we examine a lot of different data structures. Each one has its strengths and weaknesses and is applicable to different kinds of real-world problems. It's important to understand how each one operates, whether it correctly models the behavior needed by a particular problem area, whether it will operate efficiently to perform the operations, and whether it can "scale" well. We say a data structure or algorithm scales well if it will perform as efficiently as possible as the amount of data grows.

## Overview of Data Structures

As we've discussed, not every data structure models every type of problem. Or perhaps a better way to put it is that the structures model the problem awkwardly or inefficiently. You can generalize the data structures somewhat by looking at the common operations that you are likely to do across all of them. For example, to manage the requests for tickets, you need to

- **Add** a new phone number (for someone who wants one or more tickets)
- **Remove** a phone number (for someone who later decides they don't want tickets)
- **Find** a particular phone number (the next one to get a ticket by some method, or to look up one by its characteristics)
- **List** all the phone numbers (show all the phone numbers exactly once, that is, without repeats except, perhaps, for cases where multiple identical entries were made)

These four operations are needed for almost every data structure that manages a large collection of similar items. We call them **insertion**, **deletion**, **search**, and **traversal**.

Here's a list of the data structures covered in this book and some of their advantages and disadvantages with respect to the four operations. Table 1-1 shows a very high-level view of the structures; we look at them in much more detail in the following chapters. One aspect mentioned in this table is the data structure's **complexity**. In this context, we're referring to the structure's ability

to be understood easily by programmers, not how quickly it can be manipulated by the computer.

**Table 1-1** *Comparison of Different Data Types*

| Data Structure | Advantages | Disadvantages |
|---|---|---|
| Array | Simple. Fast insertion as long as index is known. Fast traversal. | Slow search, size must be known at beginning. Slow to grow. |
| Ordered array | Same as array but quicker search for items if looking by sort key. | Same as array. |
| Stack | Simple. Fast for last-in, first-out (LIFO) insertion and deletion. Fast traversal. | Slow search for items other than last-in, first-out. |
| Queue | Moderately complex. Fast for first-in, first-out (FIFO) insertion and deletion. Fast traversal. | Slow search for items other than first-in, first-out. |
| Linked List | Moderately complex. Fast insertion and deletion for a known position in list. Easy to grow and shrink. Fast traversal. | Slow search. |
| Binary tree | Moderately complex. Quick search, insertion, and deletion when tree is balanced. Easy to grow and shrink. Fast traversal. | Deletion algorithm is complex and balancing can be time-consuming. |
| Red-Black tree | Quick search, insertion, and deletion. Tree always balanced. Fast traversal. | Complex. |
| 2-3-4 tree | Quick search, insertion, and deletion. Tree always balanced. Similar trees are good for disk storage. Easy to grow and shrink. Fast traversal. | Complex. |
| Quadtree | Quick search, insertion, and deletion by 2 dimensional coordinates. Easy to grow and shrink. Fast traversal. | Complex. |
| Hash table | Fast search. Fast insertion and quick deletion in most cases. | Complex. Can take more space than other data structures. Traversal may be slightly slower than others. |
| Heap | Moderately complex. Quick insertion and quick deletion of items in sorted order. Fast traversal. | Slow search for items other than minimum or maximum. |
| Graph | Fast insertion and unordered traversal of nodes and edges | Complex. Slow search (by label) and deletion. Traversal can be slow depending on path constraints. Can take more space than other data structures. |

# Overview of Algorithms

**Algorithms** are ways to implement an operation using a data structure or group of structures. A single algorithm can sometimes be applied to multiple data structures with each data structure needing some variations in the algorithm. For example, a depth first search algorithm applies to all the tree data structures, perhaps the graph, and maybe even stacks and queues (consider a stack as a tree with branching factor of 1). In most cases, however, algorithms are intimately tied to particular data structures and don't generalize easily to others. For example, the way to insert new items or search for the presence of an item is very specific to each data structure. We examine the algorithms for insertion, search, deletion, and traversal for all the data structures. That illustrates how much they vary by data structure and the complexity involved in those structures.

Another core algorithm is **sorting**, where a collection of items is put in a particular order. Ordering the items makes searching for items faster. There are many ways to perform sort operations, and we devote Chapter 3, "Simple Sorting," to this topic, and revisit the problem in Chapter 7, "Advanced Sorting."

Algorithms are often defined **recursively**, where part of the algorithm refers to executing the algorithm again on some subset of the data. This very important concept can simplify the definition of algorithms and make it very easy to prove the correctness of an implementation. We study that topic in more detail in Chapter 6, "Recursion."

# Some Definitions

This section provides some definitions of key terms.

# Database

We use the term **database** to refer to the complete collection of data that's being processed in a particular situation. Using the example of people interested in tickets, the database could contain the phone numbers, the names, the desired number of tickets, and the tickets awarded. This is a broader definition than what's meant by a relational database or object-oriented database.

# Record

**Records** group related data and are the units into which a database is divided. They provide a format for storing information. In the ticket distribution example, a record could contain a person's name, a person's phone number, a desired number of tickets, and a number of awarded tickets. A record typically includes all the information about some entity, in a situation in which there are many such entities. A record might correspond to a user of a banking application, a car part in an auto supply inventory, or a stored video in a collection of videos.

# Field

Records are usually divided into several **fields**. Each field holds a particular kind of data. In the ticket distribution example, the fields could be as shown in Figure 1-1.


**Figure 1-1** *A record definition for ticket distribution*

The fields are named and have values. Figure 1-1 shows an empty box representing the storage space for the value. In many systems, the type of value is restricted to a single or small range of data types, just like variables are in many programming languages. For example, the desired number of tickets could be restricted to only integers, or only non-negative integers. In object-oriented systems, objects often represent records, and each object's **attributes** are the fields of that record. The terminology can be different in the various programming languages. Object attributes might be called members, fields, or variables.

# Key

When searching for records or sorting them, one of the fields is called the **key** (or **search key** or **sort key**). Search algorithms look for an exact match of the

key value to some target value and return the record containing it. The program calling the search routine can then access all the fields in the record. For example, in the ticket distribution system, you might search for a record by a particular phone number and then look at the number of desired tickets in that record. Another kind of search could use a different key. For example, you could search for a record using the desired tickets as search key and look for people who want three tickets. Note in this case that you could define the search to return the first such record it finds or a collection of all records where the desired number of tickets is three.

## Databases vs. Data Structures

The collection of records representing a database is going to require a data structure to implement it. Each record within the database may also be considered a data structure with its own data organization and algorithms. This decomposition of the data into smaller and smaller units goes on until you get to primitive data structures like integers, floating-point numbers, characters, and Boolean values. Not all data structures can be considered databases; they must support insertion, search, deletion, and traversal of records to implement a database.

## Programming in Python

Python is a programming language that debuted in 1991. It embraces object-oriented programming and introduced syntax that made many common operations very concise and elegant. One of the first things that programmers new to Python notice is that certain whitespace is significant to the meaning of the program. That means that when you edit Python programs, you should use an editor that recognizes its syntax and helps you create the program as you intend it to work. Many editors do this, and even editors that don't recognize the syntax by filename extension or the first few lines of text can often be configured to use Python syntax for a particular file.

## Interpreter

Python is an **interpreted language**, which means that even though there is a compiler, you can execute programs and individual expressions and statements by passing the text to an interpreter program. The compiler works by

translating the source code of a program into **bytecode** that is more easily read by the machine and more efficient to process. Many Python programmers never have to think about the compiler because the Python interpreter runs it automatically, when appropriate.

Interpreted languages have the great benefit of allowing you to try out parts of your code using an interactive command-line interpreter. There are often multiple ways to start a Python interpreter, depending on how Python was installed on the computer. If you use an Integrated Development Environment (IDE) such as IDLE, which comes with most Python distributions, there is a window that runs the command-line interpreter. The method for starting the interpreter differs between IDEs. When IDLE is launched, it automatically starts the command-line interpreter and calls it the *Shell*.

On computers that don't have a Python IDE installed, you can still launch the Python interpreter from a command-line interface (sometimes called a terminal window, or shell, or console). In that command-line interface, type `python` and then press the Return or Enter key. It should display the version of Python you are using along with some other information, and then wait for you to type some expression in Python. After reading the expression, the interpreter decides if it's complete, and if it is, computes the value of the expression and prints it. The example in Listing 1-1 shows using the Python interpreter to compute some math results.

**Listing 1-1** *Using the Python Interpreter to Do Math*

```
$ python
Python 3.6.0 (default, Dec 23 2016, 13:19:00)
Type "help", "copyright", "credits" or "license" for more information.
>>> 2019 - 1991
28
>>> 2**32 - 1
4294967295
>>> 10**27 + 1
1000000000000000000000000001
>>> 10**27 + 1.001
1e+27
>>>
```

In Listing 1-1, we've colored the text that you type in *blue italics*. The first dollar sign ($) is the prompt from command-line interpreter. The Python

interpreter prints out the rest of the text. The Python we use in this book is version 3. If you see `Python 2…` on the first line, then you have an older version of the Python interpreter. Try running `python3` in the command-line interface to see if Python version 3 is already installed on the computer. If not, either upgrade the version of Python or find a different computer that has `python3`. The differences between Python 2 and 3 can be subtle and difficult to understand for new programmers, so it's important to get the right version. There are also differences between every minor release version of Python, for example, between versions 3.8 and 3.9. Check the online documentation at https://docs.python.org to find the changes.

The interpreter continues prompting for Python expressions, evaluating them, and printing their values until you ask it to stop. The Python interpreter prompts for expressions using `>>>`. If you want to terminate the interpreter and you're using an IDE, you typically quit the IDE application. For interpreters launched in a command-line interface, you can press Ctrl-D or sometimes Ctrl-C to exit the Python interpreter. In this book, we show all of the Python examples being launched from a command line, with a command that starts with `$ python3`.

In Listing 1-1, you can see that simple arithmetic expressions produce results like other programming languages. What might be less obvious is that small integers and very large integers (bigger than what fits in 32 or 64 bits of data) can be calculated and used just like smaller integers. For example, look at the result of the expression 10**27 + 1. Note that these big integers are not the same as floating-point numbers. When adding integers and floating-point numbers as in 10**27 + 1.0001, the big integer is converted to floating-point representation. Because floating-point numbers only have enough precision for a fixed number of decimal places, the result is rounded to 1e+27 or $1 \times 10^{27}$.

Whitespace syntax is important even when using the Python interpreter interactively. Nested expressions *use indentation instead of a visible character* to enclose the expressions that are evaluated conditionally. For example, Python `if` statements demarcate the `then`-expression and the `else`-expression by indentation. In C++ and JavaScript, you could write

```
if (x / 2 == 1) {do_two(x)}
else {do_other(x)}
```

The curly braces enclose the two expressions. In Python, you would write

```
if x / 2 == 1:
   do_two(x)
else:
   do_other(x)
```

You must indent the two procedure call lines for the interpreter to recognize their relation to the line before it. You must be consistent in the indentation, using the same tabs or spaces, on each indented line for the interpreter to know the nested expressions are at the same level. Think of the indentation changes as replacements for the open curly brace and the close curly brace. When the indent increases, it's a left brace. When it decreases, it is a right brace.

When you enter the preceding expression interactively, the Python interpreter prompts for additional lines with the ellipsis prompt (…). These prompts continue until you enter an empty line to signal the end of the top-level expression. The transcript looks like this, assuming that x is 3 and the `do_other()` procedure prints a message:

```
>>> if x / 2 == 1:
...     do_two(x)
... else:
...     do_other(x)
...
Processing other value
>>>
```

Note, if you've only used Python 2 before, the preceding result might surprise you, and you should read the details of the differences between the two versions at https://docs.python.org. To get integer division in Python 3, use the double slash (`//`) operator.

Python requires that the indentation of **logical lines** be the same if they are at the same level of nesting. Logical lines are complete statements or expressions. A logical line might span multiple lines of text, such as the previous `if` statement. The next logical line to be executed after the `if` statement's `then` or `else` clause should start at the same indentation as the `if` statement does. The deeper indentation indicates statements that are to be executed later (as in a function definition), conditionally (as in an `else` clause), repeatedly (as in a loop), or as parts of larger construct (as in a class definition). If you have long expressions that you would prefer to split across multiple lines, they either

- Need to be inside parentheses or one of the other bracketed expression types (lists, tuples, sets, or dictionaries), or
- Need to terminate with the backslash character (\) in all but the last line of the expression

Inside of parentheses/brackets, the indentation can be whatever you like because the closing parenthesis/bracket determines where the expression ends. When the logical line containing the expression ends, the next logical line should be at the same level of indentation as the one just finished. The following example shows some unusual indentation to illustrate the idea:

```
>>> x = 9
>>> if (x %
...                 2 == 0):
...     if (x %
... 3 == 0):
...         'Divisible by 6'
...     else:
...         'Divisible by 2'
... else:
...     if (x %
...     3 == 0):
...         'Divisible by 3'
...     else:
...         'Not divisble by 2 or 3'
...
'Divisible by 3'
```

The tests of divisibility in the example occur within parentheses and are split across lines in an arbitrary way. Because the parentheses are balanced, the Python interpreter knows where the `if` test expressions end and doesn't complain about the odd indentation. The nested `if` statements, however, must have the same indentation to be recognized as being at equal levels within the conditional tests. The `else` clauses must be at the same indentation as the corresponding `if` statement for the interpreter to recognize their relationship. If the first `else` clause is omitted as in the following example,

```
>>> if (x %
...                 2 == 0):
...     if (x %
... 3 == 0):
```

```
...         'Divisible by 6'
... else:
...     if (x %
...     3 == 0):
...         'Divisible by 3'
...     else:
...         'Not divisble by 2 or 3'
...
'Divisible by 3'
```

then the indentation makes clear that the first `else` clause now belongs to the `if (x % 2 == 0)` and not the nested `if (x % 3 == 0)`. If `x` is 4, then the statement would evaluate to `None` because the `else` clause was omitted. The mandatory indentation makes the structure clearer, and mixing in unconventional indentation makes the program very hard to read!

Whitespace inside of strings is important and is preserved. Simple strings are enclosed in single (‘) or double (“) quote characters. They cannot span lines but may contain escaped whitespace such as newline (`\n`) or tab (`\t`) characters, e.g.,

```
>>> "Plank's constant:\n quantum of action:\t6.6e-34"
"Plank's constant:\n quantum of action:\t6.6e-34"
>>> print("Plank's constant:\n quantum of action:\t6.6e-34")
Plank's constant:
 quantum of action:    6.6e-34
```

The interpreter reads the double-quoted string from the input and shows it in *printed representation* form, essentially the same as the way it would be entered in source code with the backslashes used to escape the special whitespace. If that same double-quoted string is given to the `print` function, it prints the embedded whitespace in output form. To create long strings with many embedded newlines, you can enclose the string in triple quote characters (either single or double quotes).

```
>>> """Python
... enforces readability
... using structured
... indentation.
... """
'Python\nenforces readability\nusing structured\nindentation.\n'
```

Long, multiline strings are especially useful as documentation strings in function definitions.

You can add comments to the code by starting them with the pound symbol (#) and continuing to the end of the line. Multiline comments must each have their own pound symbol on the left. For example:

```
def within(x, lo, hi):    # Check if x is within the [lo, hi] range
   return lo <= x and x <= hi   # Include hi in the range
```

We've added some color highlights to the comments and reserved words used by Python like `def`, `return`, and `and`, to improve readability. We discuss the meaning of those terms shortly. Note that comments are visible in the source code files but not available in the runtime environment. The documentation strings mentioned previously are attached to objects in the code, like function definitions, and are available at runtime.

# Dynamic Typing

The next most noticeable difference between Python and some other languages is that it uses **dynamic typing**. That means that the data types of variables are determined at runtime, not declared at compile time. In fact, Python doesn't require any variable declarations at all; simply assigning a value to variable identifier creates the variable. You can change the value and type in later assignments. For example,

```
>>> x = 2
>>> x
2
>>> x = 2.71828
>>> x
2.71828
>>> x = 'two'
>>> x
'two'
```

The assignment statement itself doesn't return a value, so nothing is printed after each assignment statement (more precisely, Python's `None` value is returned, and the interpreter does not print anything when the typed expression evaluates to `None`). In Python 3.8, a new operator, `:=`, was introduced that

allows assignment in an expression that returns a value. For example, evaluating

```
(x := 2) ** 2 + (y := 3) ** 2
```

sets x to be 2 and y to be 3 and returns 13 as a value.

The type of a variable's value can be determined by the `type()` function or tested using the `isinstance()` function.

# Sequences

Arrays are different in Python than in other languages. The built-in data type that "looks like" an array is called a **list**. Python's `list` type is a hybrid of the arrays and linked lists found in other languages. As with variables, the elements of Python lists are dynamically typed, so they do not all have to be of the same type. The maximum number of elements does not need to be declared when creating a `list`, and `list`s can grow and shrink at runtime. What's quite different between Python `list`s and other linked list structures is that they can be indexed like arrays to retrieve any element at any position. There is no data type called array in the core of Python, but there is an `array` module that can be imported. The `array` module allows for construction of arrays of fixed-typed elements.

In this book, we make extensive use of the built-in `list` data type as if it were an array. This is for convenience of syntax, and because the underlying implementation of the `list` acts like arrays do in terms of indexed access to elements. Please note, however, that we do not use all of the features that the Python `list` type provides. The reason is that we want to show how fixed-type, fixed-length arrays behave in all computer languages. For new programmers, it's better to use the simpler syntax that comes with the built-in `list` for constructing arrays while learning how to manipulate their contents in an algorithm.

Python's built-in lists can be indexed using 0-relative indexes. For example:

```
>>> a = [1, 'a', False]
>>> a
[1, 'a', False]
>>> len(a)
```

```
3
>>> a[2]
False
>>> a[2] = True
>>> a
[1, 'a', True]
>>>
```

The three-element list in the example contains an integer, a string, and a Boolean value. The square brackets are used either to create a new list (as in the first assignment to `a`) or to enclose the index of an existing list (as in `a[2]`). The built-in `len()` function is used to determine the current size of its argument. Individual values in the list can be changed using the assignment statement. Strings can be treated like lists or arrays of characters, except that unlike lists, strings are immutable, meaning that after a string is created, the characters of the string cannot be changed. In Python 3, strings always contain Unicode characters, which means there can be multiple bytes per character.

```
>>> s = 'π = 3.14159'
>>> s
'π = 3.14159'
>>> len(s)
11
>>> π = 3.14159
>>> π
3.14159
```

In the preceding example, the string, `s`, contains the Greek letter π, which is counted as one character by the `len()` function, whereas the Unicode character takes two bytes of space. Unicode characters can also be used in variable names in Python 3 as shown by using π as a variable name.

Python treats all the data types that can be indexed, such as lists, arrays, and strings, as **sequence data types**. The sequence data types can be **sliced** to form new sequences. Slicing means creating a subsequence of the data, which is equivalent to getting a substring for strings. Slices are specified by a start and end index, separated by a colon (:) character. Every element from the start index up to, but not including, the end index is copied into the new sequence. The start index defaults to 0, the beginning of the sequence, and the end index defaults to the length of sequence. You can use negative numbers for both array and slice indexes. Negative indices count backwards from the end of the

sequence; −1 means the last element, −2 means the second to last element, and so on. Here are some examples with a string:

```
>>> digits = '0123456789'
>>> digits[3]
'3'
>>> digits[-1]
'9'
>>> digits[-2]
'8'
>>> digits[3:6]
'345'
>>> digits[:-2]
'01234567'
```

Sequence data types can be **concatenated**, **multiplied**, **searched**, and **enumerated**. These operations sometimes require more function calls in other languages, but Python provides simple syntax to perform them. For example:

```
>>> [1, 2, 3] + ['a', 'b']
[1, 2, 3, 'a', 'b']
>>> '011' * 7
'011011011011011011011'
>>> '011' * 0
''
>>> 3 in [1, 2, 3]
True
>>> 'elm' in ['Elm', 'Asp', 'Oak']
False
```

The preceding example shows two lists **concatenated** with the plus (+) operator to form a longer list. Multiplying a string by an integer produces that many copies of the string, concatenated together. The `in` operator is a Boolean test that **searches** for an element in a sequence. It uses the == equality test to determine whether the element matches. These operations work with all sequence data types. This compact syntax hides some of the complexity of stepping through each sequence element and doing some operation such as equality testing or copying the value over to a new sequence.

# Looping and Iteration

Frequently we want to implement algorithms that process each of the elements of a sequence in order. For that, Python has several ways to **iterate** and **enumerate** sequences. For example:

```
>>> total = 0
>>> for x in [5, 4, 3, 2, 1]:
...   total += x
...
>>> total
15
```

The `for` *variable* `in` *sequence* syntax is the basic loop construct (or **iteration**) in Python. The nested expression is evaluated once for each value in the sequence with the variable bound to the value. There is no need to explicitly manipulate an index variable that points to the current element of the sequence; that's handled by the Python interpreter. One common mistake when trying to enter this expression in the interactive interpreter is to forget the empty line after the nested expression.

```
>>> total = 0
>>> for x in [5, 4, 3, 2, 1]:
...   total += x
... total
  File "<stdin>", line 3
    total
        ^
SyntaxError: invalid syntax
```

The reason this is so common is that the empty line is needed only for the interactive interpreter; the same Python expressions written in a file would not report this as an error. The interactive interpreter, however, waits for the empty line to signal the end of the `for` loop and begin evaluation of that full expression. When the interpreter finds a new expression starting at the same indent level as the `for` loop, it is dealing with two consecutive expressions and does not allow it. The interpreter expects to read one expression, evaluate it, and print the value, before starting to determine where the next expression begins and ends.

In some circumstances, having an explicit index variable is important. In those cases, there are a couple of convenient ways to perform the work. For example:

```
>>> height = [5, 4, 7, 2, 3]
>>> weightedsum = 0
>>> for i in range(len(height)):
...   weightedsum += i * height[i]
...
>>> weightedsum
36
>>> for i, h in enumerate(height):
...   weightedsum += i * h
...
>>> weightedsum
72
```

The example calculates a weighted sum where we multiply each value in the `height` list by the index of that value. The `range()` function can be thought of as a function that produces a list of integers starting at 0 and going up to, but not equal to, its argument. By passing `len(height)` as an argument, `range()` produces the list `[0, 1, 2, 3, 4]`. In the body of the first `for` loop, the `weightedsum` variable is incremented by the product of the index, `i`, and the value that `i` indexes in the `height` list. The second `for` loop repeats the same calculation using a slightly more concise form called **enumeration**. The `enumerate()` function can be thought of as taking a sequence as input and producing a sequence of pairs. The first element of each pair is an index, and the second is the corresponding value from its sequence argument. The second `for` loop has two variables separated by a comma, `i` and `h`, instead of just one in the previous loops. On each iteration of the enumerate loop, `i` is bound to the index and `h` is bound to the corresponding value from `height`. Python makes the common pattern of looping over a sequence very easy to write, both with or without an index variable.

The `range()` and `enumerate()` functions actually create *iterators*, which are complex data types that get called in each loop iteration to get the next value of the sequence. It doesn't actually produce a list in memory for the full sequence. We discuss how iterators can be used to represent very long sequences without taking up much memory in Chapter 5, "Linked Lists."

# Multivalued Assignment

The comma-separated list of variables can also be used in assignment statements to perform multiple assignments with a single equal sign (`=`)

operator. This makes the most sense when all the values being assigned are closely related, like coordinates of a vector. To illustrate:

```
>>> x, y, z = 3, 4, 5
>>> y
4
>>> (x, y, z) = [7, 8, 9]
>>> y
8
```

The sequences on both sides of the assignment operator must have the same length; otherwise, an error occurs. The sequences can be of different types. The second assignment here uses a **tuple** on the left and a list on the right side. Python's tuple data type is also a sequence data type very similar to a list, with the distinction that its elements cannot be modified. In this case, the tuple on the left must be a tuple of variable names. The comma is the operator that creates the tuple of the `x`, `y`, and `z` variables. Each of them is bound to the corresponding value in the list on the right. Looking back at the enumerate loop in the previous example, the iterations of the loop are performing the equivalent of

```
>>> i, h = (0, 5)
>>> weightedsum += i * h
>>> i, h = (1, 4)
>>> weightedsum += i * h
>>> i, h = (2, 7)
>>> weightedsum += i * h
>>> i, h = (3, 2)
>>> weightedsum += i * h
>>> i, h = (4, 3)
>>> weightedsum += i * h
```

Python's multivalued assignments can be used to swap or rotate values. That can be a surprise to programmers of other languages where an explicit temporary variable must be used to hold one or more of the values. For example:

```
>>> left, middle, right = 'Elm', 'Asp', 'Oak'
>>> left, middle, right
('Elm', 'Asp', 'Oak')
>>> left, middle, right = middle, right, left
```

```
>>> left, middle, right
('Asp', 'Oak', 'Elm')
```

The Python interpreter evaluates all the expressions on the right of the equal sign, puts the results in a tuple, and then makes the assignments to the variables on the left side from the tuple. The tuple holding the results is something like a hidden temporary variable. The parentheses surrounding the tuples on both sides of the equal sign operator are optional.

There is another kind of assignment statement that looks like the multivalued assignment but is different. You use it to assign the same value to multiple variables. For example:

```
>>> left = middle = right = 'Gum'
>>> left, middle, right
('Gum', 'Gum', 'Gum')
```

Several variables are all assigned the same value. The overall assignment statement, however, still evaluates as `None`.

# Importing Modules

Functions can return multiple values in the form of tuples, too. You can decide whether to store all these values as a single tuple, or in several variables, one for each of the tuple's elements, just like the multivalued assignment does. A good example of this is splitting pathname components. The `os` module is in Python's standard library and provides many tools to operate on the underlying operating system where Python is running. To access a module, you import it. After it is imported, you refer to its contents via the module name, followed by a period, and then the name of one of its functions or other definitions. For example:

```
>>> import math
>>> math.pi
3.141592653589793
>>> import os
>>> os.path.splitext('myfile.ext')
('myfile', '.ext')
>>> filename, extension = os.path.splitext('myfile.ext')
>>> extension
'.ext'
```

The import statement creates a **namespace** for all the objects it defines. That's important to avoid conflicts between a variable or function that you define and the one in the module. In the preceding example, the value for `pi` is stored in the `math` namespace. If you have a separate definition, say an approximation like `pi = 3.1`, your program can refer to each one without confusion. The `os` module has submodules, one of which is the `path` module. Nested submodules create nested namespaces with corresponding names separated by periods. The `os.path.splitext()` function splits a filename at the last period and returns the two parts as a tuple. The preceding example shows binding that result to two variables, `filename` and `extension`, and then shows the value of the extension. Occasionally, you may call a function that returns multiple values without needing to retain all of them. Some Python programmers like to use the simplest variable name, a single underscore (`_`), as a throwaway variable. For instance, if only the filename extension was needed, you could write

```
_, extension = os.path.splitext('myfile.ext')
```

# Functions and Subroutines

Of course, the core of all programming languages is the ability to define functions and subroutines. In Python, they are defined with the `def` statement. The parameters the function accepts are provided in a list of variables. Parameters can be mandatory or optional; the optional ones must have a default value to use when they are missing. The `def` statement has the name of the function, the parameter list in parentheses, a colon, followed by a nested expression that is the body of the routine. If it is a function, the body should contain one or more `return` statements for the value to be returned. Subroutines in other programming languages do not return values. In Python, all functions and subroutines return something, but if no explicit return value is provided in the body of a `def` statement, then the value returned is `None`.

Here's an implementation of a weighted sum function. It takes a sequence of weights and a sequence of values. If the `weights` sequence is shorter than that of the `values`, it uses the `missing` value as the default weight.

```
>>> def weightedsum(values, weights, missing=0):
...     sum = 0
...     for i, val in enumerate(values):
...         sum += val * (weights[i] if i < len(weights) else missing)
```

```
...     return sum
...
>>> weightedsum([4, 9, 16, 25], [2, 2])
26
>>> weightedsum([4, 9, 16, 25], [2, 2], 1)
67
```

The preceding example also illustrates Python's **conditional expression**, which has the following form:

```
expression if test_expression else expression
```

First the *test_expression* is evaluated. If it is true, then the leftmost *expression* is evaluated and returned. Otherwise, the rightmost *expression* is evaluated and returned.

We refer to the variables in the function's `def` statement as its **parameters**. The **arguments** are the values that the caller submits to the function. Most of the time, arguments are matched to their parameters by their position in the argument list. Keyword arguments, in contrast, can be in any order but must follow any positional arguments in the call. A function call can specify a parameter's value by preceding the value with the parameter's name and an equal sign (`=`). Function definitions can also treat a sequence of parameters as a list. These are marked in the function definition using an asterisk (`*`) before the parameter name. As an example, Python's `print` function accepts any number of objects to print along with multiple keyword parameters that control what separators it uses in printing and where and how it produces the output. Its definition looks like this:

```
def print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False):
```

Inside the body of the `print` subroutine, the `objects` parameter holds the list of things to print. The printed output is the string representation of each object value separated by a space character and terminated by a newline, by default. Normally, the output string is sent to the standard output stream and can be buffered. If, instead, you wanted to send the output to the standard error stream separated by tab characters, you could write

```
>>> import sys
>>> print(print, 'is a', type(print), file=sys.stderr, sep='\t')
```

```
<built-in function print>     is a  <class
'builtin_function_or_method'>
```

In this example, the call to `print` has three positional parameters that all get bound to the `objects` list. Note that the `file` and `sep` arguments are not in the same position as their corresponding parameters.

# List Comprehensions

Applying a calculation or function on each element of a list to produce a new list is such a useful concept that Python has a special syntax for it. The concept is called a **list comprehension**, and it is written as a little loop inside the brackets used to create lists:

```
[expression for variable in sequence]
```

The `for` *variable* `in` *sequence* part is the same syntax as for procedural loops. The *expression* at the beginning is the same as the loop body (except that it can't contain multiple statements; only a single expression is allowed). The Python interpreter goes through the *sequence* getting each element and binding the *variable* to that element. It evaluates the *expression* in the context of the bound *variable* to compute a value. That computed value is put in the output sequence in the same position as the element from the input *sequence*. Here are two equivalent ways to produce a list of squares of the values in another list:

```
>>> values = [7, 11, 13, 17, 19]
>>> squares = []
>>> for val in values:
...     squares.append(val * val)
...
>>> squares
[49, 121, 169, 289, 361]
>>> [ x * x for x in values ]
[49, 121, 169, 289, 361]
```

The first few lines in the example show a simple loop that appends the squares to an initially empty list. The last line in the example collapses all those lines (other than defining the values) into a single expression. The whitespace just inside the square brackets is optional, but many programmers put it in to make it clearer that this is a list comprehension instead of a list created by evaluating a series of comma-separated expressions.

The list comprehension is a very compact syntax for describing the incredibly useful concept of **mapping**. It hides all the implementation details of indices and loop exit conditions while still making the essence of the operation very clear. Mapping is used everywhere in programming. In many applications the same operation needs to be applied over huge amounts of data and collected into a similar structure to the input.

Let's look at a couple more comprehensions to get the idea. To get the cube of all the integers between 10 and 20, you can write

```
>>> [ x ** 3 for x in range(10, 21) ]
[1000, 1331, 1728, 2197, 2744, 3375, 4096, 4913, 5832, 6859, 8000]
```

To split hyphenated words, you could write

```
>>> [ w.split('-') for w in ['ape', 'ape-man', 'hand-me-down'] ]
[['ape'], ['ape', 'man'], ['hand', 'me', 'down']]
```

The loop can also have a filter condition. Only those elements satisfying the condition are put in the output sequence. The filter expression goes at the end, that is,

```
[expression for variable in sequence if filter_expression]
```

The filter expression should evaluate to `True` for those sequence elements you wish to keep in the output. For example, let's get the cube of all the integers between 10 and 20 that are not multiples of 3:

```
>>> [ x ** 3 for x in range(10, 21) if x % 3 != 0 ]
[1000, 1331, 2197, 2744, 4096, 4913, 6859, 8000]
```

And find all the characters in a string that are alphabetic:

```
>>> [ c for c in 'We, the People...' if c.isalpha() ]
['W', 'e', 't', 'h', 'e', 'P', 'e', 'o', 'p', 'l', 'e']
```

These compact forms make it easy to describe the core operation while hiding many of the looping details needed to implement basic operations.

# Exceptions

Python allows programs to define what happens when particular exception conditions occur. These exceptions aren't just for errors; Python uses exceptions to handle things like the end of iteration for loops, keyboard interrupts, and timer expiration, which are all expected events. Your programs can define new kinds of exceptions and ways to handle them if they occur. Exception handling can be defined within `try except` statements. The basic form is

```
try:
    <statements>
except:
    <statements>
```

The set of statements after the `try:` are executed from top to bottom, and if and only if an exception occurs during that execution, the statements after the `except:` are executed. If no exception occurs, then the next logical line after the `try except` statement is executed.

Programs can define different handlers for different kinds of exceptions. For example:

```
try:
    long_running_function()
except KeyboardInterrupt:
    print('Keyboard interrupted long running function')
except IndexError:
    print('Index out of range during long running function')
```

Each of the `except` clauses can specify a type (class) of exception. Exceptions of that type (or a subclass of the declared class) trigger the corresponding statements to be executed. When no exception class is specified in an `except` clause, as in the basic form example, any type of exception triggers the execution of the clause.

When a program doesn't specify an exception handler for the type of exception that occurs, Python's default exception handlers print out information about the exception and where it occurred. There are also optional `else` and `finally` clauses for `try except` statements, but we don't use them in this book.

You can specify exceptions in your programs by using the `raise` statement, which expects a single argument, an exception object, describing the condition. The statement

```
raise Exception('No more tickets')
```

raises a general exception with a short description. If the program has defined an exception handler for the general `Exception` class at the time this exception is raised, then it is used in place of Python's default handler. Otherwise, Python prints a stack trace and exits the program if you're not running in the interactive interpreter. We look more at exception objects in the next section.

# Object-Oriented Programming

Object-oriented programming developed as a way to organize code for data structures. The data managed in the structure is kept in an **object**. The object is also called an **instance** of a **class of objects**. For example, in distributing tickets for an event, you would want an object to hold the list of phone numbers for people desiring tickets. The class of the object could be a queue that makes it easier to implement a first-come, first-served distribution system (as described in Chapter 4, "Stacks and Queues"). Object classes define specific **methods** that implement operations on the object instances. For the ticketing system, there needs to be a method to add a new phone number for someone desiring tickets. There also needs to be methods to get the next phone number to call and to record how many tickets are assigned to each person (which might be stored in the same or different object). The methods are common to all objects in the class and operate on the data specific to each instance. If there were several events where tickets are being distributed, each event would need its own queue instance. The method for adding a phone number is common to each one of them and is **inherited** from the object class.

Python defines classes of objects with the `class` statement. These are organized in a hierarchy so that classes can inherit the definitions in other classes. The top of that hierarchy is Python's base class, `object`. The nested statements inside a `class` statement define the class; the `def` statements define methods and assignment statements define class attributes. The first parameter of each defined method should be `self`. The `self` parameter holds the object instance, allowing methods to call other instance methods and reference instance attributes. You reference an object instance's methods or attributes by

appending the method or attribute name to the variable holding the object, separated by a period as in `object.method()`.

To define the constructor for object instances, there should be a `def __init__()` statement inside the class definition. Like the other methods, `__init__()` should accept `self` as its first parameter. The `__init__()` method takes the empty instance and performs any needed initialization like creating instance variables and setting their values (and doesn't need to return `self` like constructors do in other languages). The `__init__()` method is unusual in that you will rarely see it called explicitly in a program (that is, in the form `variable.__init()`). Python has several special methods and other constructs that all have names beginning and ending with double underscores. We point out a few of them as we use them in examples in the text.

The short example in Listing 1-2 illustrates the basic elements of object-oriented programming in Python, along with some of its ability to handle fancy math concepts.

**Listing 1-2** *Object-Oriented Program Example, Object_Oriented_Client.py*

```
class Power(object):
    """A class that computes a specific power of other numbers.
    In other words, it raises numbers by a constant exponent.
    """

    default_exponent = 2

    def __init__(self, exponent=default_exponent):
        self.exponent = exponent

    def of(self, x):
        return x ** self.exponent

class RealPower(Power):  # A subclass of Power for real numbers

    def of(self, x):
        if isinstance(self.exponent, int) or x >= 0:
            return x ** self.exponent
        raise ValueError(
            'Fractional powers of negative numbers are imaginary')

print('Power:', Power)
print('Power.default_exponent:', Power.default_exponent)
```

```
square = Power()
root = Power(0.5)
print('square: ', square)
print('square.of(3) =', square.of(3))
print('root.of(3) =', root.of(3))
print('root.of(-3) =', root.of(-3))
real_root = RealPower(0.5)
print('real_root.of(3) =', real_root.of(3))
print('real_root.of(-3) =', real_root.of(-3))
print('Done.')
```

---

Listing 1-2 shows a file with two class definitions, three object instances, and some print statements to show how the objects behave. The purpose of the first class, `Power`, is to make objects that can be used to raise numbers to various exponents. That is explained in the optional document string that follows the `class` definition. Note that this is not a Python comment, which would have to start with a pound symbol (`#`) on the left.

Each object instance is created with its own exponent, so you can create `Power` objects to perform different power functions. The `Power` class has one class attribute, `default_exponent`. This is used by the constructor for the class to define what exponent to use if none is provided when an instance is created.

The constructor for the `Power` class simply stores the desired exponent as an **instance attribute**. This is somewhat subtle because there are three distinct kinds of storage: **class attributes**, **instance attributes**, and **local variables** in the methods. The `default_exponent` attribute is defined by the assignment in the top level of the `class` statement, so it is a **class attribute** and is shared among all instances. The `exponent` parameter in the `__init__()` method of `Power` is a **local variable** available only during the evaluation of the constructor. It has a default value that is supplied from the class attribute. When `__init__()` assigns a value to `self.exponent`, it creates an **instance attribute** for `self`. The instance attribute is unique to the object instance being created and is not shared with other objects of the class.

The `Power` class has one method called `of` that returns the result of raising a number to the exponent defined at instance creation. We use it after creating two instances, `square` and `root`, that can be called using `square.of(3)`, for example. To create the first instance, the program calls `Power(2)` and binds the result to the variable `square`. This behavior might be somewhat unexpected because there is no mention of `__init__()`. This is an example of Python's use of reserved names like `__init__()` to fill special roles such as object

constructors. When a class like `Power` is referenced using a function call syntax —followed by a parenthesized list of arguments—Python builds a new instance of the class and then invokes the `__init__()` method on it. The program in Listing 1-2 makes two calls to `Power()` to make one object that produces squares and one that produces square roots.

Underscores in Python names mean that the item being defined is either special to the interpreter or to be treated as private, visible only in the block where it's defined. Python uses several special names like `__init__` for customizing the behavior of objects and programs. The special names start and end with double underscores and enable the use of tools like using the class name as constructor call.

In the example of Listing 1-2, both the `default_exponent` class attribute and the `exponent` instance attribute are public because their names don't start with an underscore. If the name were changed to begin with an underscore, they are expected to be accessed only by the class and its methods and not to be accessed outside of the class or object. The restriction on access, however, is only a convention; the Python interpreter does not enforce the privacy of any attributes regardless of their name, nor does it provide other mechanisms to enforce privacy. Python does have a mechanism to mangle the names of class and object attributes that begin with a double underscore and end with at most one underscore. The name mangling is designed to make class-specific attributes that are not shared with subclasses, but they are still publicly accessible through the mangled names.

You can run the program by giving the filename as an argument to the Python interpreter program as follows (the colored text in *blue italics* is the text that you type):

```
$ python3 Object_Oriented_Client.py
Power: <class '__main__.Power'>
Power.default_exponent: 2
square:  <__main__.Power object at 0x10715ad50>
square.of(3) = 9
root.of(3) = 1.7320508075688772
root.of(-3) = (1.0605752387249068e-16+1.7320508075688772j)
real_root.of(3) = 1.7320508075688772
Traceback (most recent call last):
  File "01_code/Object_Oriented_Client.py", line 32, in <module>
    print('real_root.of(-3) =', real_root.of(-3))
  File "01_code/Object_Oriented_Client.py", line 20, in of
```

```
    'Fractional powers of negative numbers are imaginary')
ValueError: Fractional powers of negative numbers are imaginary
```

The transcript shows the output of the `print` statements. `Power` is a class, and `square` is an object instance of that class. The `square` instance is created by the `call` to `Power()` with no arguments provided, so the exponent defaults to 2. To create a similar way to compute square roots, the object created by `Power(0.5)` is assigned to `root`. The square of 3 printed by the program is the expected 9, and the square root of 3 is the expected 1.732.

The next `print` statement is for the square root of −3, which may be a less expected result. It returns a **complex number** where `j` stands for the square root of −1. While that may be interesting for some engineering applications, you might want a different behavior for other programs. The `RealPower` class is a subclass of the `Power` class that raises an exception when raising negative numbers to fractional powers.

In the `class` statement for `RealPower`, it is defined to inherit from the `Power` class. That means that `RealPower` will have the same `default_exponent` attribute and `__init__` constructor as `Power` does. In the class definition for `RealPower`, it replaces the `of` method with a new one that tests the values of the exponent and the numeric argument. If they fall in the category that produces imaginary numbers, it raises a `ValueError` exception.

The transcript shows how the Python interpreter handles exceptions. It prints a traceback showing the function and method calls that were being evaluated when the exception occurred. The traceback includes the line numbers and copies of the lines from the input file. After encountering the exception, the interpreter prints the traceback, quits, and does not evaluate the final `print` statement in the file.

# Summary

- Data can be arranged in the computer in different ways and using various storage media. The data is organized and interpreted to represent something.
- Algorithms are the procedures used to manipulate data.
- By coupling good data organization with appropriate algorithms, data structures provide the fundamental building block of all programs.

- Examples of data structures are stacks, lists, queues, trees, and graphs.
- Data structures are often compared by how efficiently they perform common operations.
- A database is a collection of many similar records, each of which describes some entity.
- The records of a database are composed of fields, each of which has a name and a value.
- A key field is used to search for and sort records.
- Data structures that act like databases support four core operations: insertion, search, deletion, and traversal.
- Data structures are implemented as classes using object-oriented programming.
- Python has rich support for object-oriented programming with classes to implement data structures.

# Questions

These questions are intended as a self-test for readers. The answers can be found in Appendix C.

1. Data structures that allow programs to _____ a record, _____ for a record, _____ a record, and _____ all the records are considered to be databases in this book.
2. Data structures are

   a. composed of names and values.

   b. built with fields that have methods attached to them.

   c. implemented with object classes.

   d. records that don't change in a database.
3. How can you tell if a `def` statement in Python defines a function or class method?
4. What algorithms can help make searching more efficient?

5. What are constructors used for? What special name does Python use for them?
6. What are some of the reasons for choosing one data structure over another?
7. What is a key field used for?
8. Good data organization can help with the speed of an algorithm, but what other benefits does it have?
9. For what purpose was object-oriented programming developed?
10. Which of the following are data structures used in programming?

    a. traceback

    b. heap

    c. list comprehension

    d. hash table

    e. recipe

    f. slices

    g. binary tree

# Experiments

Try the following experiments:

1-A Write a Python list comprehension that returns the individual characters of a string that are not whitespace characters. Apply it to the string `"4 and 20 blackbirds.\n”`

1-B Take a deck of playing cards, pull out the 13 spade cards, set aside the rest, and shuffle the spade cards. Devise an algorithm to sort them by number under the constraints:

a. All the cards must be held in one hand. This is the “first” hand.

b. Initially, the shuffled cards are all stacked with faces in one direction so that only one card is visible.

c. Initially, all the cards are held between the thumb and forefinger of the first hand.

d. The visible card in the stack can be pulled out using the other hand and placed in between any of the fingers of the first hand. It can only be placed at the front or the back of the cards in the stack of cards between those fingers.

e. The other hand can hold one card at a time and must place it somewhere in the first hand before picking out another visible card from one of the stacks.

f. The algorithm is done when all the cards are in sorted order in one stack in the hand.

Compare the efficiency of your algorithm with that of classmates or friends.

# 2. Arrays

Arrays are the most commonly used data structure for many reasons. They are straightforward to understand and match closely the underlying computer hardware. Almost all CPUs make it very fast to access data at known offsets from a base address. Almost every programming language supports them as part of the core data structures. We study them first for their simplicity and because many of the more complex data structures are built using them.

## In This Chapter



First, we look at the basics of how data is inserted, searched, and deleted from arrays. Then, we look at how we can improve it by examining a special kind of array, the ordered array, in which the data is stored in ascending (or descending) key order. This arrangement makes possible a fast way of searching for a data item: the binary search.

To improve a data structure's performance requires a way of measuring performance beyond just running it on sample data. Looking at examples of how it handles particular kinds of data makes it easier to understand the operations. We also take the first step to generalize the performance measure by looking at linear and binary searches, and introducing Big O notation, the most widely used measure of algorithm efficiency.

Suppose you're coaching kids-league soccer, and you want to keep track of which players are present at the practice field. What you need is an attendance-monitoring program for your computer—a program that maintains a database of the players who have shown up for practice. You can use a simple data structure to hold this data. There are several actions you would like to be able to perform:

- Insert a player into the data structure when the player arrives at the field.
- Check to see whether a particular player is present, by searching for the player's number in the structure.
- Delete a player from the data structure when that player leaves.
- List all the players present.

These four operations—insertion, searching, deletion, and enumeration (traversal)—are the fundamental ones in most of the data storage structures described in this book.

## The Array Visualization Tool

We often begin the discussion of a particular data structure by demonstrating it with a visualization tool—a program that animates the operations. This approach gives you a feeling for what the structure and its algorithms do, before we launch into a detailed explanation and demonstrate sample code. The visualization tool called Array shows how an array can be used to implement insertion, searching, and deletion.

Now start up the Array Visualization tool, as described in Appendix A, "Running the Visualizations." There are several ways to do this, as described in the appendix. You can start it up separately or in conjunction with all the visualizations. If you've downloaded Python and the source code to your computer, you can launch it from the command line using

```
python3 Array.py
```

Figure 2-1 shows the initial array with 10 elements, 9 of which have data items in them. You can think of these items as representing your players. Imagine that each player has been issued a team shirt with the player's number on the back. That's really helpful because you have just met most of these people and haven't learned all their names yet. To make things visually interesting, the shirts come in a variety of colors. You can see each player's number and shirt color in the array. The height of the colored rectangle is proportional to the number.

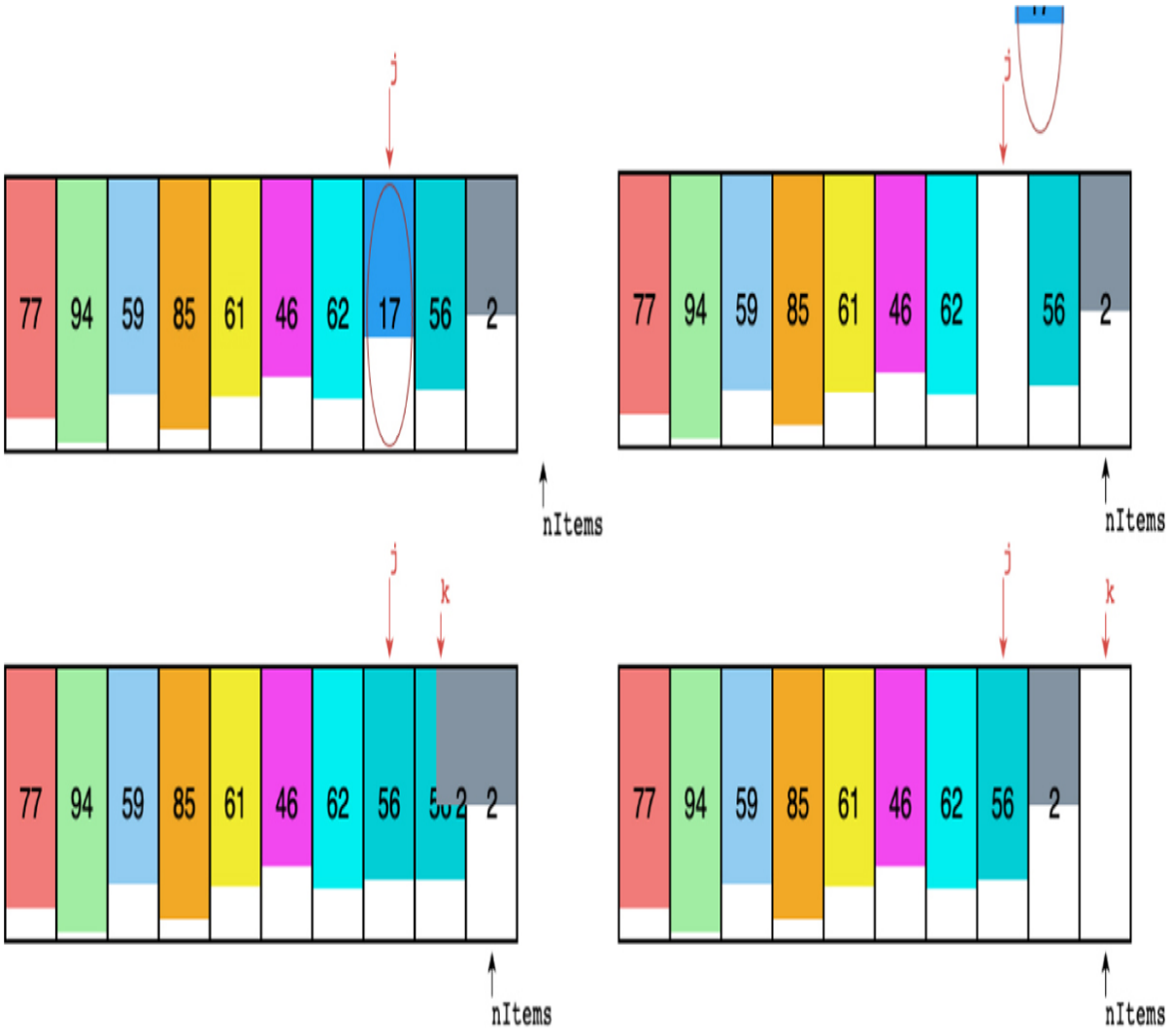



**Figure 2-1** *The Array Visualization tool*

This visualization demonstrates the four fundamental operations mentioned earlier:

- The Insert button inserts a new data item.
- The Search button searches for specified data item.
- The Delete button deletes a specified data item.
- The Traverse button lists all the items in the array.

The buttons for the first three of those operations are on the left, grayed out and disabled. The reason is that you haven't entered a number in the small box to their right. The hint below the box suggests that when there's a number to search, insert, or delete, the buttons will be enabled.

The Traverse button is on the right and enabled. That's because it doesn't require an argument to start traversing. We explore that and the other buttons on the right shortly.

On the left, there is also a button labeled New. It's used to create a new array of a given size. The size is taken from the text entry box like the other arguments. (The initial hint shown in Figure 2-1 also mentioned that you can enter the number of cells here.) Arrays must be created with a known size because they place the contents in adjacent memory cells, and that memory must be allocated for exclusive use by the array. We look at each of these operations in turn.

# Searching

Imagine that you just arrived at the playing field to start coaching, your assistant hands you the computer that's tracking attendance, and a player's parent asks if the goalies can start their special drills. You know that players 2, 5, and 17 are the ones who play as goalies, but are they all here? Although answering this question is trivial for a real coach with paper and pencil, let's look at the details needed to do it with the computer.

You want to search to see whether all the players are present. The array operations let you search for one item at a time. In the visualization tool, you can select the text entry box near the Search button, the hint disappears, and you enter the number `2`. The button becomes enabled, and you can select it to

start the search. The tool animates the search process, which starts off looking like Figure 2-2.


**Figure 2-2** *Starting a search for player 2*

The visualization tool now shows several new things. A box in the lower right shows a program being executed. Next to the array drawing, an arrow labeled with a `j` points to the first cell. The `j` advances to each cell in turn, checking to see whether it holds the number 2. When it reaches where the `nItems` arrow points on the right, all the cells have been searched. The `j` arrow disappears, and a message appears at the bottom: `Value 2 not found`.

This process mimics what humans would do—scan the list, perhaps with a finger dragged along the numbers, confirming whether any of them match the number being sought. The visualization tool shows how the computer represents those same activities. The tool allows you to pause the animation of the process if you want to look at the details. The three buttons at the bottom right of the operations area, ▶ ⏭ ■, control the animation. The leftmost one plays or pauses the animation. The middle button plays until the next code step, and the square shape on the right stops the operation. At the bottom, you can slide the animation speed control left or right to slow down or speed up the animation.

Try searching for the value 17 (or some other number present in the array). The visualization tool starts with the `j` arrow pointing at the leftmost array cell. After checking the value there, it moves to the next cell to the right. That process repeats seven times in the array shown in Figure 2-2, and then the value is circled. After a few more changes in the code box, the message `Value 17 Found` appears at the bottom (we return to the code in a moment).

What's important to notice here is that the search went through seven steps before finding the value. When you searched for 2, it went through all nine items before stopping. Let's call the number of items N (which is just a shorter version of the `nItems` shown in the visualization). When you search for an item that is in the array, it could take 1 to N steps before finding it. If there's an equal chance of being asked to search for each item, the average number of search steps is (1 + 2 + 3 + … + N−1 + N) / N which works out to (N + 1) / 2. Unsuccessful searches take N steps every time.

There's another simple but still important thing to notice. What if there were duplicate values in the array? That's not supposed to happen on the team, of course. Every player should have a distinct number. If one player lent one of their shirts to a teammate who forgot theirs, however, it would not be surprising to see the same number twice.

The search strategy must know whether to expect multiple copies of a number to exist. The visualization tool stops when it finds the first matching number. If multiple copies are allowed, *and* it's important to find all of them, then the search couldn't stop after finding the first one. It would have to go through all N items and identify how many matched. In that case, both successful and unsuccessful searches would take N steps.

## Insertion

We didn't find player 2 when asked before, but now that player has just arrived. You need to record that player 2 is at practice, so it's time to insert them in the array. Type `2` in the text entry box and select Insert. A new colored rectangle with 2 in it appears at the bottom and moves into position at the empty cell indicated by the `nItems` pointer. When it's in position, the `nItems` pointer moves to the right by one. It may now point beyond the last cell of the array.

The animation of the arrival of the new item takes a little time, but in terms of what the computer has to do, only two steps were needed: writing the new value in the array at the position indicated by `nItems` and then incrementing `nItems` by 1. It doesn't matter if there were two, three, or a hundred items already in the array; inserting the value always takes two steps. That makes the insertion operation quite different from the search operation and almost always faster.

More precisely, the number of steps for insertion doesn't depend on how many items are in the array as long as it is not full. If all the cells are filled, putting a value outside of the array is an error. The visualization tool won't let that happen and will produce an error message if you try. (In the history of programming, however, quite a few programmers did not put that check in in the code, leading to buffer overflows and security problems.)

The visualization tool lets you insert duplicate values (if there are available cells). It's up to you to avoid them, possibly by using the Search operation, if you don't want to allow it.

## Deletion

Player 17 has to leave (he wants to start on the homework assignment due tomorrow). To delete an item in the array, you must first find it. After you type in the number of the item to be deleted, a process like the search operation begins in the visualization tool. A `j` arrow appears starting at the leftmost cell and steps to the right as it checks values in the cells. When it finds and circles the value as shown in the top left of Figure 2-3, however, it does something different.


**Figure 2-3** *Deleting an item*

The visualization tool starts by reducing `nItems` by 1, moving it to point at the last item in the array. That behavior might seem odd at first, but the reason will become clear when we look at the code. The next thing it does is move the deleted value out of the array, as shown in the top right of Figure 2-3. That doesn't actually happen in the computer, but the empty space helps distinguish that cell visually during what happens next.

A new arrow labeled `k` appears pointing to the same cell as `j`. Each of the items to the right of the deleted item is copied into the cell to its left, and `k` is advanced. So, for example, item 56 is copied into cell `j`, and then item 2 is copied into where 56 was, as shown in the bottom left of the figure. When they are all moved, the `nItems` arrow (and `k`) points at the empty cell just after the last filled cell, as shown at the bottom right. That makes the array ready to accept the next item to insert in just two steps. (The visualization tool clears the last cell after copying its contents to the cell to its left. This behavior is not strictly necessary but helps the visualization.)

Implicit in the deletion algorithm is the assumption that holes are not allowed in the array. A **hole** is one or more empty cells that have filled cells above them (at higher index numbers). If holes are allowed, the algorithms for all the operations become more complicated because they must check to see whether a cell is empty before doing something with its contents. Also, the algorithms become less efficient because they waste time looking at unoccupied cells. For these reasons, occupied cells must be arranged contiguously: no holes allowed.

Try deleting another item while carefully watching the changes to the different arrows. You can slow down and pause the animation to see the details.

How many steps does each deletion take? Well, the `j` arrow had to move over a certain number of times to find the item being deleted, let's call that J. Then you had to shift the items to the right of `j`. There were N − J of those items. In total there were J + N − J steps, or simply N steps. The steps were different in character: checking values versus copying values (we consider the difference between making value comparisons and shifting items in memory later).

### Traversal

Arrays are simple to traverse. The data is already in a linear order as specified by the index to the array elements. The index is set to 0, and if it's less than the current number of items in the array, then the array item at index 0 is processed. In the Array visualization tool, the item is copied to an output box, which is similar to printing it. The index is incremented until it equals the current number of items in the array, at which point the traversal is complete. Each item with an index less than `nItems` is processed exactly once. It's very easy to traverse the array in reverse order by decrementing the index too.

## The Duplicates Issue

When you design a data storage structure, you need to decide whether items with duplicate keys will be allowed. If you're working with a personnel file and the key is an employee number, duplicates don't make much sense; there's no point in assigning the same number to two employees. On the other hand, a list of contacts might have several entries of people who have the same family name. Two entries might even have the same given and family names. Assuming the names are the key for looking up the contact, duplicate keys should be allowed in a simple contacts list. Another example would be a data structure designed to keep track of the food items in a pantry. There are likely to be several identical items, such as cans of beans or bottles of milk. The key could be the name of the item or the label code on its package. In this context, it's likely the program will not only want to search for the presence for an item but also count how many identical items are in the store.

If you're writing a data storage program in which duplicates are not allowed, you may need to guard against human error during an insertion by checking all the data items in the array to ensure that not one of them already has the same key value as the item being inserted. This check reduces the efficiency, however, by increasing the number of steps required for an insertion from one to N. For this reason, the visualization tool does not perform this check.

## Searching with Duplicates

Allowing duplicates complicates the search algorithm, as we noted. Even if the search finds a match, it must continue looking for possible additional matches until the last occupied cell. At least, this is one approach; you could also stop after the first match and perform subsequent searches after that. How you proceed depends on whether the question is "Find me everyone with the family name of Smith," "Find me someone with the family name of Smith," or the similar question "Find how many entries have the family name Smith."

Finding all items matching a search key is an **exhaustive search**. Exhaustive searches require N steps because the algorithm must go all the way to the last occupied cell, regardless of what is being sought.

## Insertion with Duplicates

Insertion is the same with duplicates allowed as when they're not: a single step inserts the new item. Remember, however, that if duplicates are prohibited, and

there's a possibility the user will attempt to input the same key twice, the algorithm must check every existing item before doing an insertion.

## Deletion with Duplicates

Deletion may be more complicated when duplicates are allowed, depending on exactly how "deletion" is defined. If it means to delete only the first item with a specified value, then, on the average, only N/2 comparisons and N/2 moves are necessary. This is the same as when no duplicates are allowed. This would be the desired way to handle deleting an item such as a can of beans from a kitchen pantry when it gets used. Any items with duplicate keys remain in the pantry.

If, however, deletion means to delete *every* item with a specified key value, the same operation may require multiple deletions. Such an operation requires checking N cells and (probably) moving more than N/2 cells. The average depends on how the duplicates are distributed throughout the array.

## Traversal with Duplicates

Traversal means processing each of the stored items exactly once. If there are duplicates, then each duplicate item is processed once. That means that the algorithm doesn't change if duplicates are present. Processing all the *unique* keys in the data store exactly once is a different operation.

Table 2-1 shows the average number of comparisons and moves for the four operations, first where no duplicates are allowed and then where they are allowed. N is the number of items in the array. Inserting a new item counts as one move.

**Table 2-1** *Duplicates OK Versus No Duplicates*

| | **No Duplicates** | **Duplicates OK** |
|---|---|---|
| *Search* | N/2 comparisons | N comparisons |
| *Insertion* | No comparisons, one move | No comparisons, one move |
| *Deletion* | N/2 comparisons, N/2 moves | N comparisons, more than N/2 moves |
| *Traversal* | N processing steps | N processing steps |

The difference between N and N/2 is not usually considered very significant, except when you're fine-tuning a program. Of more importance, as we discuss toward the end of this chapter, is whether an operation takes one step, N steps, or $N^2$ steps, which would be the case if you wanted to enumerate all the pairs of items stored in a list. When there are only a handful items, the differences are small, but as N gets bigger, the differences can become huge.

### Not Too Swift

One of the significant things to notice when you're using the Array Visualization tool is the slow and methodical nature of the algorithms. Apart from insertion, the algorithms involve stepping through some or all of the cells in the array performing comparisons, moves, or other operations. Different data structures offer much faster but slightly more complex algorithms. We examine one, the search on an ordered array, later in this chapter, and others throughout this book.

### Deleting the Rightmost Item

Deletion is the slowest of the four core operations in an array. If you don't care what item is deleted, however, it's easy to delete the last (rightmost) item in the array. All that task requires is reducing the count of the number of items, `nItems`, by one. It doesn't matter whether duplicates are allowed or not; deleting the last item doesn't affect whether duplicates are present. The Array Visualization tool provides a Delete Rightmost operation to see the effect of this fast—one "move" or assignment no comparison—operation.

## Using Python Lists to Implement the Array Class

The preceding section showed the primary algorithms used for arrays. Now let's look at how to write a Python class called `Array` that implements the array abstraction and its algorithms. But first we want to cover a few of the fundamentals of arrays in Python.

As mentioned in Chapter 1, "Overview," Python has a built-in data structure called `list` that has many of the characteristics of arrays in other languages. In the first few chapters of this book, we stick with the simple, built-in Python constructs for lists and use them as if they were arrays, while avoiding the use of more advanced features of Python lists that might obscure the details of

what's really happening in the code. In Chapter 5, "Linked Lists," we introduce linked lists and describe the differences. For those programmers who started with Python lists, it may seem odd to initialize the size of the `Array`'s list at the beginning, but this is a necessary step for true arrays in all programming languages. The memory to hold all the array elements must be allocated at the beginning so that the sequence of elements can be stored in a contiguous range of memory and any array element can be accessed in any order.

# Creating an Array

As we noted in Chapter 1, Python lists are constructed by enclosing either a list of values or by using a list comprehension (loop) in square brackets. The list comprehension really builds a new `list` based on an existing list or sequence. To allocate lists with large numbers of values, you use either an iterator like `range()` inside a list comprehension or the multiplication operator. Here are some examples:

```
integerArray = [1, 1, 2, 3, 5]   # A list of 5 integers
charArray = ['a' for j in range(1000)]  # 1,000 letter 'a' characters
booleArray = [False] * 32768     # 32,768 binary False values
```

Each of these assignment statements creates a list with specific initial values. Python is dynamically typed, and the items of a list do not all have to be of the same type. This is one of the core differences between Python lists and the arrays in statically typed languages, where all items must be of the same type. Knowing the type of the items means that the amount of memory needed to represent each one is known, and the memory for the entire array can be allocated. In the case of `charArray` in the preceding example, Python runs a small loop 1,000 times to create the list. Although all three sample lists start with items of the same type, they could later be changed to hold values of any type, and their names would no longer be accurate descriptions of their contents.

Data structures are typically created empty, and then later insertions, updates, and deletions determine the exact contents. Primitive arrays are allocated with a given maximum size, but their contents could be anything initially. When you're using an array, some other variables must track which of the array elements have been initialized properly. This is typically managed by an integer that stores the current number of initialized elements.

In this book, we refer to the storage location in the array as an **element** or as a **cell**, and the value that is stored inside it as an **item**. In Python, you could write the initialization of an array like this:

```
maxSize = 10000
myArray = [None] * maxSize
myArraySize = 0
```

This code allocates a list of 10,000 elements each initialized to the special `None` value. The `myArraySize` variable is intended to hold the current number of inserted items, which is 0 at first. You might think that Python's built-in `len()` function would be useful to determine the current size, but this is where the implementation of an array using a Python list breaks down. The `len()` function returns the allocated size, not how many values have been inserted in `myArray`. In other words, `len(myArray)` would always be 10,000 (if you only change individual element values). We will track the number of items in our data structures using other variables such as `nItems` or `myArraySize`, in the same manner that must be done with other programming languages.

# Accessing List Elements

Elements of a list are accessed using an **integer index** in square brackets. This is similar to how other languages work:

```
temp = myArray[3]    # get contents of fourth element of array
myArray[7] = 66      # insert 66 into the eighth cell
```

Remember that in Python—as in Java, C, and C++—the first element is numbered 0, so the indices in an array of 10 elements run from 0 to 9. What's different in Python is that you can use negative indices to specify the count from the end of the list. You can think of that as if the current size of the array were added to the negative index. In the previous example, you could get the last item by writing `myArray[maxSize - 1]` or, more simply, `myArray[-1]`.

Python also supports **slicing** of lists, but that seemingly simple operation hides many details that are important to the understanding of an `Array` class's behavior. As a result, we don't use slices in showing how to implement an Array class using lists.

## Initialization

All the methods we've explored for creating lists in Python involve specifying the initial value for the elements. In other languages, it's easy to create or **allocate** an array without specifying any initial value. As long as the array elements have a known size and there is a known quantity of them, the memory needed to hold the whole array can be allocated. Because computers and their operating systems reuse memory that was released by other programs, the element values in a newly allocated array could be anything. Programmers must be careful not to write programs that use the array values before setting them to some desired value because the uninitialized values can cause errors and other unwanted behavior. Initializing Python list values to `None` or some other known constant avoids that problem.

## An Array Class Example

Let's look at some sample programs that show how a list can be used. We start with a basic, object-oriented implementation of an `Array` class that uses a Python list as its underlying storage. As we experiment with it, we will make changes to improve its performance and add features. Listing 2-1 shows the class definition, called `Array`. It's stored in a file called `BadArray.py`.

**Listing 2-1** *The `BadArray.py` Module*

```
# Implement an Array data structure as a simplified type of list.

class Array(object):

   def __init__(self, initialSize):     # Constructor
      self.__a = [None] * initialSize  # The array stored as a list
      self.nItems = 0     # No items in array initially

   def insert(self, item):       # Insert item at end
      self.__a[self.__nItems] = item   # Item goes at current end
      self.__nItems += 1         # Increment number of items

   def search(self, item):
      for j in range(self.nItems):      # Search among current
  if self.__a[j] == item:        # If found,
     return self.__a[j]  # then return item

      return None         # Not found -> None
```

```
   def delete(self, item):      # Delete first occurrence
      for j in range(self.nItems):     # of an item
  if self.__a[j] == item:       # Found item
      for k in range(j, self.nItems):  # Move items from
         self.__a[k] = self.__a[k+1]   # right over 1
      self.nItems -= 1     # One fewer in array now
      return True  # Return success flag

      return False     # Made it here, so couldn't find the item

   def traverse(self, function=print): # Traverse all items
      for j in range(self.nItems):     # and apply a function
  function(self.__a[j])
```

The `Array` class has a constructor that initializes a fixed length list to hold the array of items. The array items are stored in a private instance attribute, `__a`, and the number of items stored in the array is kept in the public instance attribute, `nItems`. The four methods define the four core operations.

Before we look at the implementation details, let's use a program to test each operation. A separate file, `BadArrayClient.py`, shown in Listing 2-2 uses the `Array` class in the `BadArray.py` module. This program imports the class definition, creates an `Array` called `arr` with a `maxSize` of 10, inserts 10 data items (integers, strings, and floating-point numbers) in it, displays the contents by traversing the `Array`, searches for a couple of items in it, tries to remove the items with values 0 and 17, and then displays the remaining items.

**Listing 2-2** *The `BadArrayClient.py` Program*

```
import BadArray
maxSize = 10       # Max size of the Array
arr = BadArray.Array(maxSize)    # Create an Array object

arr.insert(77)     # Insert 10 items
arr.insert(99)
arr.insert("foo")
arr.insert("bar")
arr.insert(44)
arr.insert(55)
arr.insert(12.34)
arr.insert(0)
arr.insert("baz")
arr.insert(-17)
```

```
print("Array containing", arr.nItems, "items")
arr.traverse()

print("Search for 12 returns", arr.search(12))

print("Search for 12.34 returns", arr.search(12.34))

print("Deleting 0 returns", arr.delete(0))
print("Deleting 17 returns", arr.delete(17))

print("Array after deletions has", arr.nItems, "items")
arr.traverse()
```

---

To run the program, you can use a command-line interpreter to navigate to a folder where both files are present and run the following:

```
$ python3 BadArrayClient.py
Array containing 10 items
77
99
foo
bar
44
55
12.34
0
baz
-17
Search for 12 returns None
Search for 12.34 returns 12.34
Traceback (most recent call last):
  File "BadArrayClient.py", line 23, in <module>
    print("Deleting 0 returns", arr.delete(0))
  File "/Users/canning/chapters/02 code/BadArray.py", line 24, in
delete
    self.__a[k] = self.__a[k+1]   # right over 1
IndexError: list index out of range
```

The results show that most of the methods work properly; this example illustrates the use of the public instance attribute, `nItems`, to provide the number of items in the `Array`. The Python traceback shows that there's a problem with the `delete()` method. The error is that the list index was out of range. That means either `k` or `k+1` was out of range in the line displayed in the traceback. Going back to the code in `BadArray.py`, you can see that `k` lies in

the range of `j` up to but not including `self.nItems`. The index `j` can't be out of bounds because the method already accessed `__a[j]` and found that it matched `item` in the line preceding the `k` loop. When `k` gets to be `self.nItems - 1`, then `k + 1` is `self.nItems`, and that *is outside of the bounds* of the maximum size of the `list` initially allocated. So, we need to adjust the range that `k` takes in the loop to move array items. Before fixing that, let's look more at the details of all the algorithms used.

## Insertion

Inserting an item into the `Array` is easy; we already know the position where the insertion should go because we have the number of current items that are stored. Listing 2-1 shows that the item is placed in the internal `list` at the `self.nItems` position. Afterward, the number of items attribute is increased so that subsequent operations will know that that element is now filled. Note that the method does not check whether the allocated space for the `list` is enough to accommodate the new item.

## Searching

The `item` variable holds the value being sought. The `search` method steps through only those indices of the internal `list` within the current number of items, comparing the `item` argument with each array item. If the loop variable `j` passes the last occupied element with no match being found, the value isn't in the `Array`. Appropriate messages are displayed by the `BadArrayClient.py` program: `Search for 12 returns None` or `Search for 12.34 returns 12.34.`

## Deletion

Deletion begins with a search for the specified item. If found, all the items with higher index values are moved down one element to fill in the hole in the `list` left by the deleted item. The method decrements the instance's `nItems` attribute, but you've already seen an error happen before that point. Another possibility to consider is what happens if the item isn't found. In the implementation of Listing 2-1, it returns `False`. Another approach would be to raise an exception in that case.

## Traversal

Traversing all the items is straightforward: We step through the `Array`, accessing each one via the private instance variable `__a[j]` and apply the `(print)` function to it.

### Observations

In addition to the bug discovered, the `BadArray.py` module does not provide methods for accessing or changing arbitrary items in the `Array`. That's a fundamental operation of arrays, so we need to include that. We will keep the code simple and focus attention on operations that will be common across many data structures.

The `Array` class demonstrates **encapsulation** (another aspect of object-oriented programming) by providing the four methods and keeping the underlying data stored in the `__a list` as private. Programs that use `Array` objects are able to access the data only through those methods. The `nItems` attribute is public, which makes it convenient for access by `Array` users. Being public, however, opens that attribute to being manipulated by `Array` users, which could cause errors to occur. We address these issues in the next version of the program.

## A Better Array Class Implementation

The next sample program shows an improved interface for the array storage structure class. The class is still called `Array`, and it's in a file called `Array.py`, as shown in Listing 2-3.

The constructor is almost the same except that the variable holding the number of items in the `Array` is renamed `__nItems`. The double underscore prefix marks this instance attribute as private. To make it easy to get the current number of items in `Array` instances, this example introduces a new method named `__len__()`. This is a special name to Python because objects that implement a `__len__()` method can be passed as arguments to the built-in Python `len()` function, like all other sequence types. By calling this function, programs that use the `Array` class can get the value stored in `__nItems` but are not allowed to set its value.

**Listing 2-3** *The `Array.py` Module*

```
# Implement an Array data structure as a simplified type of list.
```

```
class Array(object):
   def __init__(self, initialSize):     # Constructor
      self.__a = [None] * initialSize  # The array stored as a list
      self.__nItems = 0  # No items in array initially

   def __len__(self):     # Special def for len() func
      return self.__nItems       # Return number of items

   def get(self, n):      # Return the value at index n
      if 0 <= n and n < self.__nItems: # Check if n is in bounds, and
  return self.__a[n]      # only return item if in bounds

   def set(self, n, value):      # Set the value at index n
      if 0 <= n and n < self.__nItems: # Check if n is in bounds, and
  self.__a[n] = value     # only set item if in bounds

   def insert(self, item):       # Insert item at end
      self.__a[self.__nItems] = item   # Item goes at current end
      self.__nItems += 1         # Increment number of items

   def find(self, item):         # Find index for item
      for j in range(self.__nItems):   # Among current items
  if self.__a[j] == item:        # If found,
     return j      # then return index to item
      return -1    # Not found -> return -1

   def search(self, item):       # Search for item
      return self.get(self.find(item)) # and return item if found

   def delete(self, item):       # Delete first occurrence
      for j in range(self.__nItems):   # of an item
  if self.__a[j] == item:        # Found item
     self.__nItems -= 1  # One fewer at end
     for k in range(j, self.__nItems):  # Move items from
        self.__a[k] = self.__a[k+1]     # right over 1
     return True  # Return success flag

      return False      # Made it here, so couldn't find the item

   def traverse(self, function=print): # Traverse all items
      for j in range(self.__nItems):   # and apply a function
  function(self.__a[j])
```

It's important to know about Python's mechanisms to manage "private" attributes. The underscore prefix in a name indicates that the attribute *should*

*be treated as private but does not guarantee it*. Using a double underscore prefix in an attribute like `__nItems` causes Python to use **name mangling**, making it harder but not impossible to access the attribute. It also makes accessing that attribute in subclasses more complex. To make private attributes that can be easily accessed by the same name in subclasses, use a single underscore prefix. For this example, we choose to keep the double underscore name to illustrate how to control public access, such as using the `__len__()` method.

The example in Listing 2-3 also introduces `get()` and `set()` methods that allow `Array` users to read and write the values of individual elements based on an index. This is the basic functionality of arrays in all languages. The `get()` method checks that the desired index is within the current bounds and returns the value if it is. Note that if the index is out of bounds, there is no explicit return value. Python functions and methods return `None` if execution reaches the end of the function body. The `set()` method checks that the index is in bounds and sets that cell's value, if so, returning `None`.

The `insert()` method remains the same, but we change the `search()` method by breaking it up into two methods. We define a new `find()` method that finds the index to the item being sought. This method loops through the current items and returns the index of the item if it's found, or −1 if it isn't. We choose −1 for the return value because it cannot possibly be confused with a valid index value into the `list` and to guarantee the output of `find()` is always an integer (not `None`). The output of the `find()` method can thus be passed to the `get()` method to get the item after finding its index. If the item is not found, `find()` returns −1, and `get()` and `search()` still return `None`.

We alter the `delete()` method to fix the index out-of-bounds error in `BadArray.py` by moving the decrement of `__nItems` to occur *before* the loop that moves items to the left in the `list` (see Listing 2-3). The `traverse()` method remains the same.

**Listing 2-4** *The `ArrayClient.py` Program*

```
import Array
maxSize = 10       # Max size of the array
arr = Array.Array(maxSize)       # Create an array object

arr.insert(77)     # Insert 10 items
```

```
arr.insert(99)
arr.insert("foo")
arr.insert("bar")
arr.insert(44)
arr.insert(55)
arr.insert(12.34)
arr.insert(0)
arr.insert("baz")
arr.insert(-17)

print("Array containing", len(arr), "items")
arr.traverse()

print("Search for 12 returns", arr.search(12))

print("Search for 12.34 returns", arr.search(12.34))

print("Deleting 0 returns", arr.delete(0))
print("Deleting 17 returns", arr.delete(17))

print("Setting item at index 3 to 33")
arr.set(3, 33)

print("Array after deletions has", len(arr), "items")
arr.traverse()
```

---

A new client program, `ArrayClient.py`, exercises the `Array` class, as shown in Listing 2-4. This program is almost identical to `BadArrayClient.py` but uses the new module name, `Array`, and tests the new features of the interface by calling the `len()` function on the `Array` and the `set()` method. The tests that call `search()` are also testing the new `find()` and `get()` methods.

You can confirm that the bug is fixed and no new bugs have shown up by running `ArrayClient.py` to see the following:

```
$ python3 ArrayClient.py
Array containing 10 items
77
99
foo
bar
44
55
12.34
0
baz
```

```
-17
Search for 12 returns None
Search for 12.34 returns 12.34
Deleting 0 returns True
Deleting 17 returns False
Setting item at index 3 to 33
Array after deletions has 9 items
77
99
foo
33
44
55
12.34
baz
-17
```

We now have a functional `Array` class that implements the four core methods for a data storage object. The code shown in the Array visualization tool is that of the `Array` class in Listing 2-3. Try using the visualization tool to search for an item and follow the highlights in the code. You will see that it calls the `search()` method, which calls the `find()` method. Those both show up separated by a gray line. When the `find()` method finishes, its local variables are erased, and its source code disappears, leaving the `search()` method to use its result and try to get the item. The visualization does not show the execution of the `get()` method but does show a message indicating whether the item was found or not.

You can also try out the allocation of a new array. The "New" operation allocates an array of a size you provide (if the cells fit on the screen). If you ask for a large number of cells, it makes them smaller, and the numbers may be hidden. The code shown for the New operation is that of the `__init__()` constructor for the `Array` class. The Random Fill operation fills any empty cells of the current array with random keys. The Delete Rightmost removes the last item from the array. These aren't methods in the basic `Array` class, but they are helpful for the visualization.

# The Ordered Array Visualization Tool

Imagine an array in which the data items are arranged in order of ascending values—that is, with the smallest value at index 0, and each cell holding a value larger than the cell below. Such an array is called an **ordered array**.

When you insert an item into this array, the correct location must be found for the insertion: just above a smaller value and just below a larger one. Then all the larger values must be moved up to make room.

Why would you want to arrange data in order? One advantage is that you can speed up search times dramatically using a **binary search**. At the same time, you are making the insert operation more complex because it must find the proper location for each new item.

To get a feel for what these changes bring, start the Ordered Array Visualization tool, using the procedure described in Appendix A. You see an array; it's similar to the one in the Array Visualization tool, but the data is ordered. Figure 2-4 shows what this tool looks like when it starts.

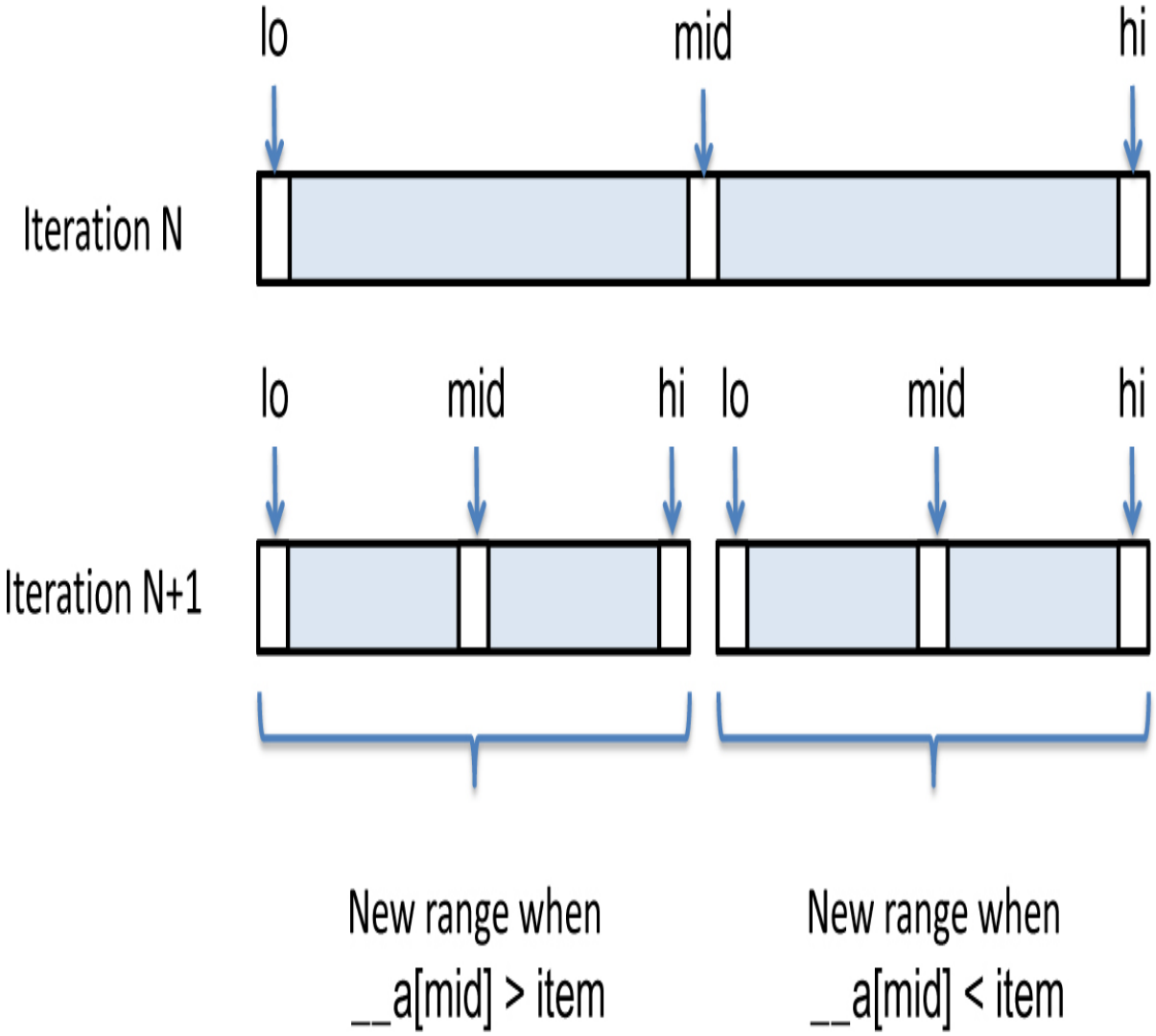



**Figure 2-4** *The Ordered Array Visualization tool*

# Linear Search

Before we describe how ordering the array helps, we need to elaborate on the kinds of searches we're discussing. The search algorithm used in the (unordered) Array Visualization tool is called a **linear search**. A linear search operates just like someone running a finger over a list of items to find a match. In the visualization, a brown arrow steps along, until it finds a match or reaches the `nItems` limit.

# Binary Search

The payoff for using an ordered array comes when you use a binary search. You use this for the Search operation because it is much faster than a linear search, especially for large arrays.

## The Guess-a-Number Game

Binary search is a classic approach to guessing games. In the Guess-a-Number game, a friend asks you to guess a number between 1 and 100 (Figure 2-5). When you guess a number, she tells you one of three things:

- Your guess is larger than the number she's thinking of, or
- It's smaller, or
- You guessed correctly.

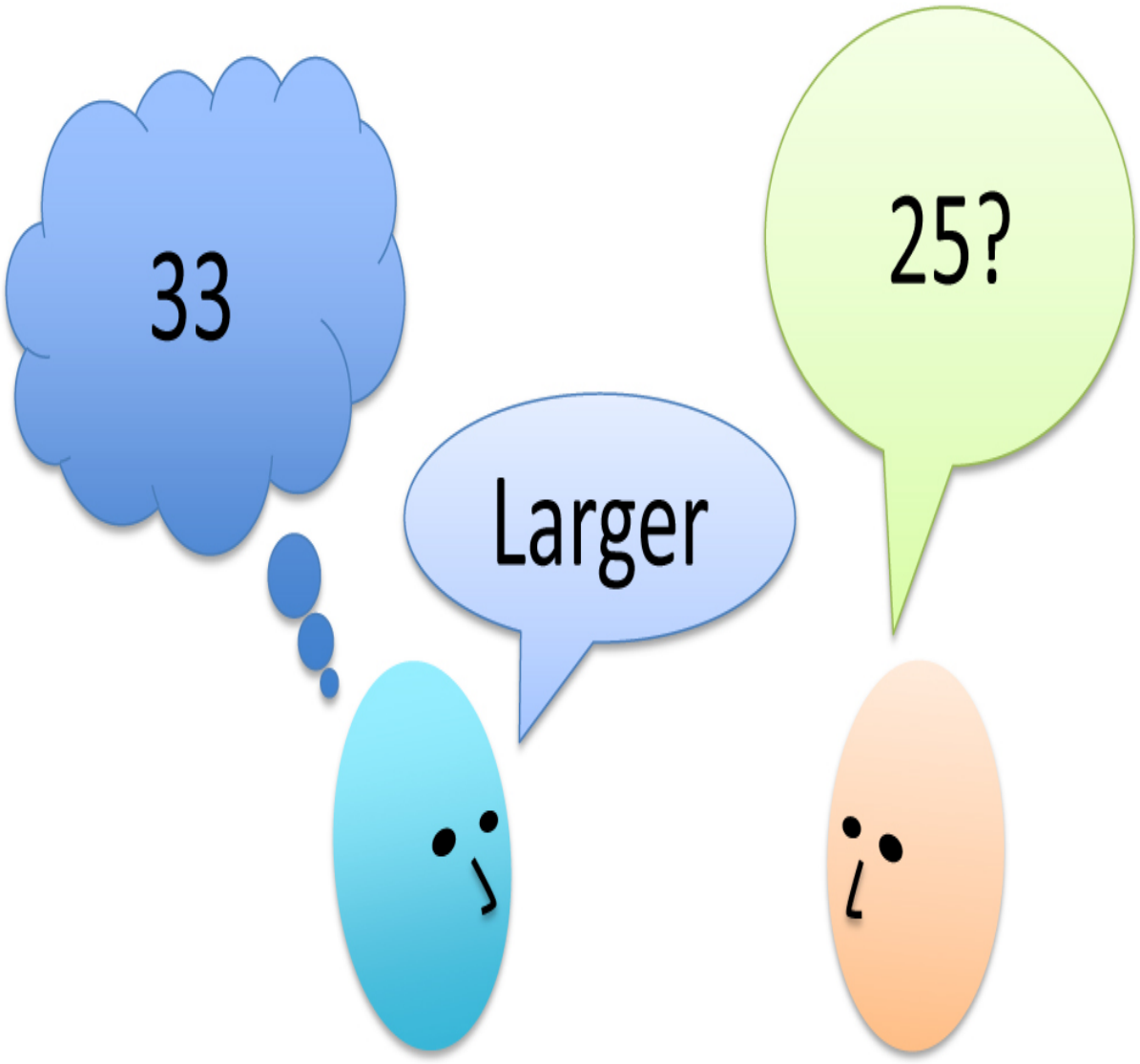



**Figure 2-5** *The Guess-a-Number Game*

To find the number in the fewest guesses, you should always start by guessing 50. If your friend says your guess is too low, you deduce the number is between 51 and 100, so your next guess should be 75 (halfway between 51 and 100). If she says it's too high, you deduce the number is between 1 and 49, so your next guess should be 25.

Each guess allows you to divide the range of possible values in half. Finally, if you haven't already found it on an early guess, the range is only one number long, and that's the answer.

Notice how few guesses are required to find the number. If you used a linear search, guessing first 1, then 2, then 3, and so on, finding the number would

take you, on the average, 50 guesses. In a binary search, each guess divides the range of possible values in half, so the number of guesses required is far fewer. Table 2-2 shows a game session when the number to be guessed is 33.

**Table 2-2** *Guessing a Number*

| Step Number | Number Guessed | Result | Range of Possible Values |
|---|---|---|---|
| 0 | | | 1–100 |
| 1 | 50 | Too high | 1–49 |
| 2 | 25 | Too low | 26–49 |
| 3 | 37 | Too high | 26–36 |
| 4 | 31 | Too low | 32–36 |
| 5 | 34 | Too high | 32–33 |
| 6 | 32 | Too low | 33–33 |
| 7 | 33 | Correct | |

The correct number is identified in only 7 guesses. This is the maximum number of guesses, regardless of the number chosen by your friend. You might get lucky and guess the number before you've worked your way all the way down to a range of one. This would happen if the number to be guessed was 50, for example, or 34. The most important thing to remember is that you will always find the number in 7 or fewer guesses, which compares well with the maximum of 100 guesses and average of 50 guesses if you search linearly, ignoring the too high and too low clues.

## Binary Search in the Ordered Visualization Tool

If you change the Guess-a-Number game into a Where-is-a-Number game, you can use the same strategy in searching the ordered array. It's a subtle shift, but the question is now asking, "What is the index of the cell holding number X?" The indices range from 0 to N−1, so there is the same kind of range to be searched. You can start with the middle array cell and then narrow the range based on what you find there.

Try searching for the newly inserted 55 in ordered array by typing `55` in the text entry box and selecting Search. The Ordered Array Visualization tool shows the array and adds three arrows labeled `lo`, `mid`, and `hi`. The `lo` and `hi` arrows point to the first and last cells of the array, respectively. The `mid` arrow is placed at the midpoint between them, as shown in the top part in Figure 2-6.


**Figure 2-6** *Initial ranges in a binary search*

When the range of `lo` to `hi` spans an odd number of cells, the midpoint is the same number of cells from either end, but when it's even, it must be closer to one or the other end. The visualization tool always chooses `mid` to be closer to `lo`, and you'll see why when looking at the code.

After comparing the value at `mid` (59) with the value you're trying to find, the algorithm determines that 55 must lie in the range to the left of `mid`. It moves the `hi` arrow to be one less than `mid` to narrow the range and leaves `lo` unchanged. Then it updates `mid` to be at the midpoint of the narrowed range. The bottom of Figure 2-6 shows this second step of the process.

Each step reduces the range by about half. With the initial 10-element array, the ranges go from 10 to 5, to 2, to 1 at the very most (in Figure 2-6, it goes from 10 to 4 to 2, before finding 55 at index 2). If `mid` happens to point at the goal item, the search can stop. Otherwise, it will continue until the range collapses to nothing.

Try a few searches to see how quickly the visualization tool finds values. Try a search for a value not in the array to see what happens. With a 10-element array, it will take at most four values for `mid` to determine whether the value is present in the array.

What about larger arrays? Use the New operation to find out. Select the text entry box, enter `35`, and then select New. If there's enough room in the tool window, it will draw 35 empty cells of a new array. Fill them with random values by selecting Random Fill. The cells show colored rectangles, but the numbers disappear when the cells are too skinny. You can try a variation of the Guess-a-Number game here by typing in a value, selecting Search, and seeing whether it ended up in the array. If you succeed, the tool will add an oval to the cell and provide a success message at the end. You can also select a colored rectangle with your pointer, and it will fill in its value in the text entry area.

Can you figure out how many steps the binary search algorithm will take to find a number based on the size of the array you're searching? We return to this question in the last section of this chapter.

## Duplicates in Ordered Arrays

We saw that the presence of duplicate values affected the number of comparisons and shifts needed in searches and deletes on unordered arrays. Does that change for ordered arrays? Yes, a little bit.

Let's look at searching first because it affects insertion and deletion. If finding only a single matching item is sufficient, then there's no difference whether the array has duplicates or not. When you find the first one, you're done. If searches must return *all* matching items, the binary search algorithm finds only the first of them. After that, you would need to find any duplicates to the left or right of the one discovered by binary search. That could be done with linear searching, and it would need to search only the last `lo` to `hi` range explored by the binary search. That would add extra steps, possibly up to visiting all N items because the entire array could be duplicates of the same value. On average, however, it would be much less.

Note that a successful binary search does not guarantee finding the item with the lowest or highest index among those with duplicate keys. It guarantees finding one of them, but finding the relative position of that item to the others requires the extra linear searching.

Insertion remains the same for ordered arrays when duplicates are permitted. If a duplicate key exists, the binary search will find one of the duplicates and insert the new item beside it. If the new item has a unique key, it will be placed in order as before. You still need to shift values over to correctly insert any new value. The presence of duplicates might mean shifting fewer values, if the item

to insert matches the value of one or more existing items. The presence of a few duplicates, however, does not significantly change the average number of comparisons and shifts needed, which remain about N/2.

For deletions, the effect of duplicates is also a bit complicated. If deleting only one of the matching items is sufficient, you can use binary search to find the item and shift items to the right of it to fill the hole caused by its removal. If there are many duplicates, you could save some shifts by shifting only the rightmost duplicate to fill the hole, but finding the rightmost takes almost as much time as shifting all the duplicates in between.

If deletion requires deleting *all* matching items, the complexity that we discussed for the search operation applies to the deletion process too. There would be fewer shifts needed after finding all the duplicates because you can shift values over D cells as fast as shifting over 1 cell, where D is the number of matching duplicate items. Overall, however, there is not much of a difference compared to the no-duplicates case because the number of operations is still proportional to N.

# Python Code for an Ordered Array Class

Let's examine some Python code that implements an ordered array. This example uses the `OrderedArray` class to encapsulate the underlying `list` and its algorithms. The heart of this class is the `find()` method, which uses a binary search to locate a specified data item. We examine this method in detail before showing the complete program.

# Binary Search with the find() Method

The `find()` method searches for the index to a specified item by repeatedly dividing in half the range of list items to be considered. The method looks like this:

```
    def find(self, item):      # Find index at or just below
        lo = 0    # item in ordered list
        hi = self.__nItems-1    # Look between lo and hi

        while lo <= hi:
    mid = (lo + hi) // 2        # Select the midpoint
    if self.__a[mid] == item:   # Did we find it at midpoint?
        return mid        # Return location of item
```

```
        elif self.__a[mid] < item: # Is item in upper half?
            lo = mid + 1      # Yes, raise the lo boundary
        else:
            hi = mid - 1      # No, but could be in lower half

        return lo   # Item not found, return insertion point instead
```

The method begins by setting the `lo` and `hi` variables to the first and last indices in the array. Setting these variables specifies the range for where the `item` may be found. Then, within the `while` loop, the index, `mid`, is set to the middle of this range.

If you're lucky, `mid` may already be pointing to the desired item, so you first check if `self.__a[mid] == item` is true. If it is, you've found the item, and you return with its index, `mid`.

If `mid` does not point to the `item` being sought, then you need to figure out which half of the range it falls in. You check whether the `item` is bigger than the one at the midpoint by testing `self.__a[mid] < item`. If the `item` is bigger, then you can shrink the search range by setting the `lo` boundary to be 1 above the midpoint. Note that setting `lo` to be the same as `mid` would mean including that midpoint item in the remaining search. You don't want to include it because the comparison with the item at `mid` already showed that its value is too low. Finally, if the midpoint item is neither equal to nor less than the `item` being sought, it must be bigger. In this case, you can shrink the search range by setting `hi` to be 1 below the midpoint. Figure 2-7 shows how the range is altered in these two situations.

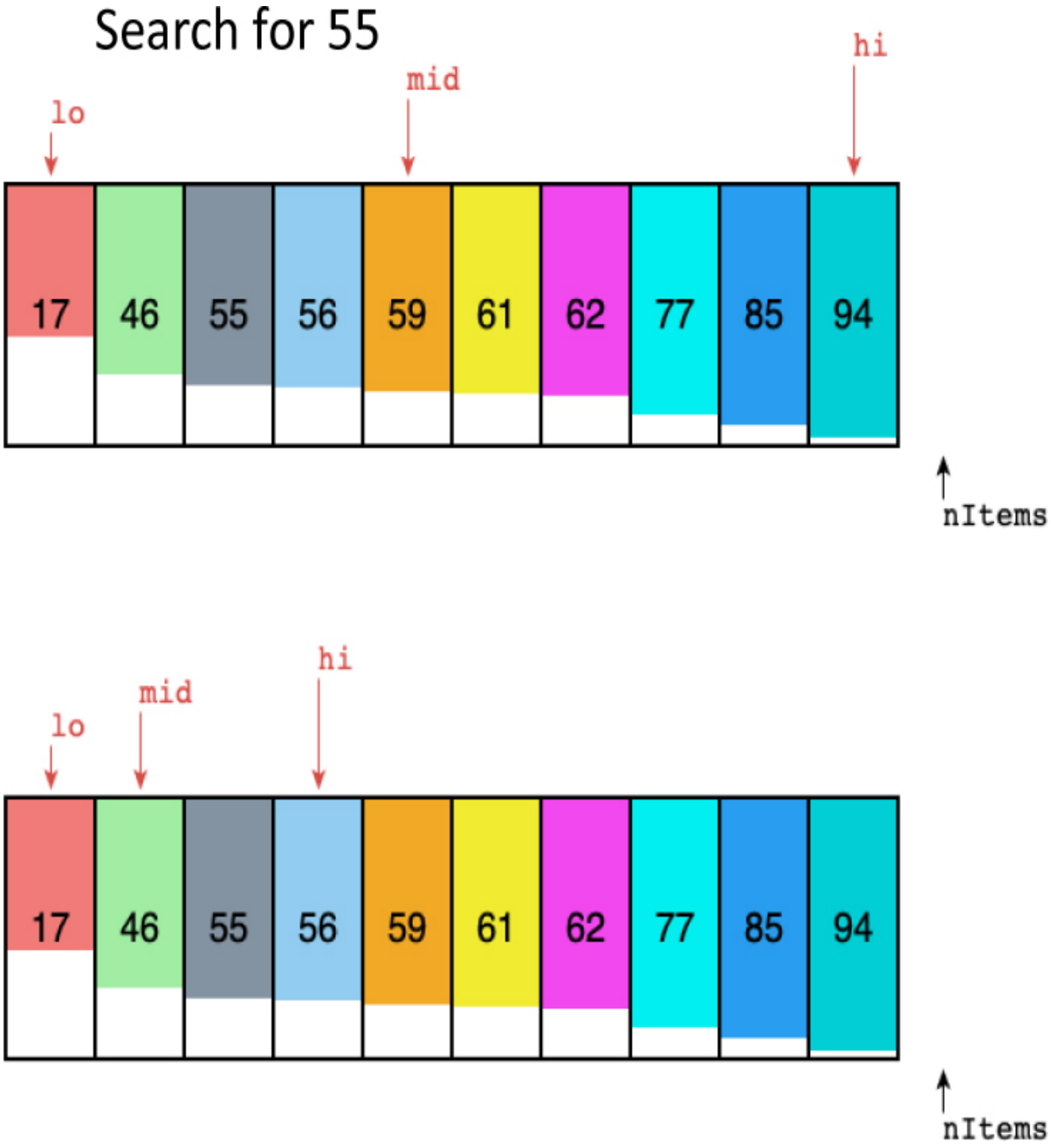



**Figure 2-7** *Dividing the range in a binary search*

Each time through the loop you divide the range in half. Eventually, the range becomes so small that it can't be divided any more. You check for this in the loop condition: if `lo` is greater than `hi`, the range has ceased to exist. (When `lo` equals `hi`, the range is one and you need one more pass through the loop.) You can't continue the search without a valid range, but you haven't found the desired item, so you return `lo`, the last lower bound of the search range. This might seem odd because you're returning an index that doesn't point to the

item being sought. It still could be useful, however, because it specifies where an item with that value would be placed in the ordered array.

## The OrderedArray Class

In general, the `OrderedArray.py` program is similar to `Array.py` (refer to Listing 2-3). The main difference is that the `find()` method is changed to do a binary search, as we've discussed. Here, we show the class in two parts. Listing 2-5 shows the basic class infrastructure including the constructor, utility methods, and the traverse operation. Listing 2-6 shows the other three core operations, including the `find()` method.

**Listing 2-5** *The Basic* `OrderedArray` *Class Definition*

```
# Implement an Ordered Array data structure

class OrderedArray(object):
   def __init__(self, initialSize):    # Constructor
      self.__a = [None] * initialSize  # The array stored as a list
      self.__nItems = 0  # No items in array initially

   def __len__(self):    # Special def for len() func
      return self.__nItems      # Return number of items

   def get(self, n):     # Return the value at index n
      if 0 <= n and n < self.__nItems: # Check if n is in bounds, and
  return self.__a[n]     # only return item if in bounds
      raise IndexError("Index " + str(n) + " is out of range")

   def traverse(self, function=print): # Traverse all items
      for j in range(self.__nItems):   # and apply a function
  function(self.__a[j])

   def __str__(self):    # Special def for str() func
      ans = "["   # Surround with square brackets
      for i in range(self.__nItems):   # Loop through items
  if len(ans) > 1:       # Except next to left bracket,
     ans += ", "  # separate items with comma
  ans += str(self.__a[i])       # Add string form of item
      ans += "]"         # Close with right bracket
      return ans
```

The `OrderedArray` constructor is identical to that of the `Array` class; it allocates a list of the specified `initialSize` and sets the item count to 0. The `__len__()` and `traverse()` methods are identical too. In the `get()` method, one change has been added: it raises an exception if called on an index outside the range of active cells. The `IndexError` is a standard Python exception type used for this condition. A customized string message explains the problem.

The `OrderedArray` class includes a new `__str__()` method in Listing 2-5, which builds a string representation of the data items currently in the array. This is more than just a convenient utility for these tests; a Python object's `__str__()` method is invoked by the built-in `str()` function to create a string in contexts where one is needed, such as when the object is passed to the `print()` function. It uses the same syntax that Python uses for a string version of lists: a list of comma-separated values enclosed in square brackets.

Note that we intentionally leave out the `set()` method because that would allow callers to change values in ways that might not keep the items in order.

**Listing 2-6** *The Core Operations of the* `OrderedArray` *Class*

```
class OrderedArray(object):
…
   def find(self, item):      # Find index at or just below
      lo = 0    # item in ordered list
      hi = self.__nItems-1    # Look between lo and hi

      while lo <= hi:
  mid = (lo + hi) // 2        # Select the midpoint
  if self.__a[mid] == item:   # Did we find it at midpoint?
     return mid               # Return location of item
  elif self.__a[mid] < item: # Is item in upper half?
     lo = mid + 1             # Yes, raise the lo boundary
  else:
     hi = mid - 1             # No, but could be in lower half

      return lo    # Item not found, return insertion point instead

   def search(self, item):
      index = self.find(item)         # Search for item
      if index < self.__nItems and self.__a[index] == item:
  return self.__a[index]      # and return item if found

   def insert(self, item): # Insert item into correct position
```

```
    if self.__nItems >= len(self.__a): # If array is full,
raise Exception("Array overflow") # raise exception

    index = self.find(item)      # Find index where item should go
    for j in range(self.__nItems, index, -1): # Move bigger items
self.__a[j] = self.__a[j-1]      # to the right

    self.__a[index] = item       # Insert the item
    self.__nItems += 1    # Increment the number of items

 def delete(self, item):       # Delete any occurrence
    j = self.find(item)        # Try to find the item
    if j < self.__nItems and self.__a[j] == item:  # If found,
self.__nItems -= 1      # One fewer at end
for k in range(j, self.__nItems): # Move bigger items left
   self.__a[k] = self.__a[k+1]
return True      # Return success flag

    return False      # Made it here; item not found
```

Listing 2-6 starts with the `find()` method that implements the binary search algorithm. The `search()` method changes a little from that of the `Array`. It first calls `find()` and verifies that the index returned is in bounds. If not, or if the indexed item doesn't match the sought item, it returns `None`. That means searching for an item not in the array will return `None` without raising an exception.

A check at the beginning of the `insert()` method determines whether the array is full. This is done by comparing the length of the Python `list`, `__a`, to the number of items currently in the array, `__nItems`. If `__nItems` is equal to (or somehow, larger than) the size of the `list`, inserting another item will overflow it, so the method raises an exception.

Otherwise, the `insert()` method calls `find()` to locate where the new item goes. Then it uses a loop over the indices to the right of the insertion `index` to move those items one cell to the right. The loop uses `range(self.__nItems, index, -1)` to go backward through the indices from `__nItems` to `index + 1`. The number of items to be moved could be all N of them if the new item is the smallest. On average, it will move half the current items.

The `delete()` method calls `find()` to figure out the location of the item to be deleted and whether it is in the array. If it does find the item, it also must move

half the current items to the left on average. If not, it can return `False` without moving anything.

Like before, we use a separate client program to test the operations of the class and the utility methods. The `OrderedArrayClient.py` program appears in Listing 2-7.

**Listing 2-7** *The `OrderedArrayClient.py` Program*

```
from OrderedArray import *

maxSize = 1000  # Max size of the array
arr = OrderedArray(maxSize)    # Create the array object

arr.insert(77)  # Insert 11 items
arr.insert(99)
arr.insert(44)  # Inserts not in order
arr.insert(55)
arr.insert(0)
arr.insert(12)
arr.insert(44)
arr.insert(99)
arr.insert(77)
arr.insert(0)
arr.insert(3)

print("Array containing", len(arr), "items:", arr)

arr.delete(0)   # Delete a few items
arr.delete(99)
arr.delete(0)   # Duplicate deletes
arr.delete(0)
arr.delete(3)

print("Array after deletions has", len(arr), "items:", arr)

print("find(44) returns", arr.find(44))
print("find(46) returns", arr.find(46))
print("find(77) returns", arr.find(77))
```

Note that you can pass the `arr` variable directly to `print` and expect a reasonable output because of the `__str__()` method. We also use a different form of the `import` statement in `OrderedArrayClient.py`. By importing the

module using the "`from` *module* `import *`" syntax, the definitions it contains are added in the same namespace as the client program, not in a new namespace for the module. That means you can create the object using the expression `OrderedArray(maxSize)` instead of `OrderedArray.OrderedArray(maxSize)`. The output of the program looks like this:

```
$ python3 OrderedArrayClient.py
Array containing 11 items: [0, 0, 3, 12, 44, 44, 55, 77, 77, 99, 99]
Array after deletions has 7 items: [12, 44, 44, 55, 77, 77, 99]
find(44) returns 1
find(46) returns 3
find(77) returns 5
```

The last three print statements illustrate some particular cases of the binary search with duplicate entries. The result of `find(46)` shows that even though 46 is not in `arr`, it should be inserted after the first three items to preserve ordering. The `find(44)` finds the first occurrence of 44 at position 1. If `delete(44)` were called at this point, it would delete the first of the 44s currently in the array. By contrast, `find(77)` points at the second of the two 77s in the array. The binary search stops after it finds the first matching item, which could be any instance of an item that appears multiple times.

# Advantages of Ordered Arrays

What have we gained by using an ordered array? The major advantage is that search times are much faster than in an unordered array. The disadvantage is that insertion takes longer because all the data items with a higher key value must be moved up to make room. Even though it uses binary search to find where the key belongs, it still must move most of the items in the array.

Deletions are slow in both ordered and unordered arrays because items must be moved down to fill the hole left by the deleted item. There is a bit of speed-up for requests to delete items not in the array. By using `find()`, you can quickly discover whether any items need to be moved in the array. That benefit, however, is reduced when the requested item is in the array. Finding the item with a binary search replaces a linear search over the left side of the array. Both linear and binary searching require shifting items to right of the deleted item.

Going back to insertion for a moment, would it be simpler to skip the binary search of `find()` and just move items to the right until you reach an item

smaller than the item being inserted? That saves a function call to `find()` but requires more comparisons. The binary search algorithm uses way fewer than N comparisons to find the insertion point, and if it doesn't call `find()`, the insert code must compare values for all N items being shifted.

Ordered arrays are therefore useful in situations in which searches are frequent, but insertions and deletions are not. An ordered array might be appropriate for a database of a transport company that tracks the location of its vehicles, for example. The need to add and delete the names of a fleet of vehicles happens much less frequently than the events that require updates to their position, so having fast search find the right vehicle for each position update would be important and worth the increased time needed to add each new vehicle. On the other hand, if the transport company keeps a log of the tasks assigned to each vehicle or their actions in picking up and delivering cargo, there would be frequent additions to the log, but perhaps little need to find a particular log entry. The task log could benefit from the data structure with fast insertion time at the expense of a longer search.

# Logarithms

In this section we explain how you can use logarithms to calculate the number of steps necessary in a binary search. If you're a math fan, you can probably skip this section. If thinking about math makes you nervous, give it a try, and make sure to take a long, hard look at Table 2-3.

A binary search provides a significant speed increase over a linear search. In the number-guessing game, with a range from 1 to 100, a maximum of seven guesses is needed to identify any number using a binary search; just as in an array of 100 records, a maximum of seven comparisons is needed to find a record with a specified key value. How about other ranges? Table 2-3 shows some representative ranges and the number of comparisons needed for a binary search.

**Table 2-3** *Comparisons Needed in a Binary Search*

| Range | Comparisons Needed |
|---|---|
| 10 | 4 |
| 100 | 7 |
| 1,000 | 10 |
| 10,000 | 14 |
| 100,000 | 17 |
| 1,000,000 | 20 |
| 10,000,000 | 24 |
| 100,000,000 | 27 |
| 1,000,000,000 | 30 |

Notice the differences between binary search times and linear search times. For very small numbers of items, the difference isn't dramatic. Searching 10 items would take an average of 5 comparisons with a linear search (N/2) and a maximum of 4 comparisons with a binary search. But the more items there are, the bigger the difference. With 100 items, there are 50 comparisons in a linear search, but only 7 in a binary search. For 1,000 items, the numbers are 500 versus 10, and for 1,000,000 items, they're 500,000 versus 20. You can conclude that for all but very small arrays, the binary search is greatly superior.

## The Equation

You can verify the results of Table 2-3 by repeatedly dividing a range (from the first column) in half until it's too small to divide further. The number of divisions this process requires is the number of comparisons shown in the second column.

Repeatedly dividing the range by two is an algorithmic approach to finding the number of comparisons. You might wonder if you could also find the number using a simple equation. Of course, there is such an equation, and it's worth exploring here because it pops up from time to time in the study of data structures. This formula involves logarithms. (Don't panic yet.)

You have probably already experienced logarithms, without having recognized them. Have you ever heard someone say “a six figure salary” or read about “a deal worth eight figures”? Those simplified expressions tell you the approximate amount of the salary or deal by telling you how many digits are needed to write the number. The number of digits could be found by repeatedly dividing the number by 10. When it’s less than 1, the number of divisions is the number of digits.

The numbers in Table 2-3 leave out some interesting data. They don’t answer such questions as “What is the exact size of the maximum range that can be searched in five steps?” To solve this problem, you can create a similar table, but one that starts at the beginning, with a range of one, and works up from there by multiplying the range by two each time. Table 2-4 shows how this looks for the first seven steps.

**Table 2-4** *Powers of Two*

| Step s, same as $\log_2(r)$ | Range r | Range Expressed as Power of 2 ($2^s$) |
|---|---|---|
| 0 | 1 | $2^0$ |
| 1 | 2 | $2^1$ |
| 2 | 4 | $2^2$ |
| 3 | 8 | $2^3$ |
| 4 | 16 | $2^4$ |
| 5 | 32 | $2^5$ |
| 6 | 64 | $2^6$ |
| 7 | 128 | $2^7$ |
| 8 | 256 | $2^8$ |
| 9 | 512 | $2^9$ |
| 10 | 1,024 | $2^{10}$ |

For the original problem with a range of 100, you can see that 6 steps don’t produce a range quite big enough (64), whereas 7 steps cover it handily (128).

Thus, the 7 steps that are shown for 100 items in Table 2-3 are correct, as are the 10 steps for a range of 1,000.

Doubling the range each time creates a series that's the same as raising 2 to a power, as shown in the third column of Table 2-4. We can express this power as a formula. If *s* represents steps (the number of times you multiply by 2—that is, the power to which 2 is raised) and *r* represents the range, then the equation is

$$r = 2^s$$

If you know *s*, the number of steps, this tells you *r*, the range. For example, if *s* is 6, the range is $2^6$, or 64.

## The Opposite of Raising 2 to a Power

The original question was the opposite of the one just described: Given the range, you want to know how many comparisons are required to complete a search. That is, given *r*, you want an equation that gives you *s*.

The inverse of raising something to a power is called a **logarithm**. Here's the formula you want, expressed with a logarithm:

$$s = \log_2(r)$$

This equation says that the number of steps (comparisons) is equal to the logarithm to the base 2 of the range. What's a logarithm? The base 2 logarithm of a number *r* is the number of times you must multiply 2 by itself to get *r*. In Table 2-4, the step numbers in the first column, *s*, are equal to $\log_2(r)$.

How do you find the logarithm of a number without doing a lot of dividing? Most calculators and computer languages have a log function. For those that don't, sometimes it can be added as option, such as with Python's `math` module. It might only provide a function for log to the base 10, but you can convert easily to base 2 by multiplying by 3.322. For example, $\log_{10}(100) = 2$, so $\log_2(100) = 2$ times 3.322, or 6.644. Rounded up to the whole number 7, this is what appears in the column to the right of 100 in Table 2-3.

In any case, the point here isn't to calculate logarithms. It's more important to understand the relationship between a number and its logarithm. Look again at Table 2-3, which compares the number of items and the number of steps needed to find a particular item. Every time you multiply the number of items

(the range) by a factor of 10, you add only three or four steps (actually 3.322, before rounding off to whole numbers) to the number needed to find a particular item. This is true because, as a number grows larger, its logarithm doesn't grow nearly as fast. We compare this logarithmic growth rate with that of other mathematical functions when we talk about Big O notation later in this chapter.

## Storing Objects

In the examples we've shown so far, we've stored single values in array data structures such as integers, floating-point numbers, and strings. Storing such simple values simplifies the program examples, but it's not representative of how you use data storage structures in the real world. Usually, the data you want to store comprises many values or fields, usually called a **record**. For a personnel record, you might store the family name, given name, birth date, first working date, identification number, and so forth. For a fleet of vehicles, you might store the type of vehicle, the name, the date it entered service, a license tag, and so forth. In object-oriented programs, you want to store the objects themselves in data structures. The objects can represent records.

When storing objects or records in ordered data structures, like the `OrderedArray` class, you need to define the way the records are ordered by specifying a **key** that can be used on all of them. Let's look at how that changes the implementation.

## The OrderedRecordArray Class

As shown in the previous examples, you can insert any data type into Python arrays. It's very convenient to allow complex data types to be inserted, deleted, and managed in the arrays, along with the benefits of storing them in order, which makes search faster. To distinguish them (and order them), you need a key for each record. The best way to do that is to define a function that extracts the key from a record and then use that function when comparing keys. By using a function, the array data structure doesn't need to know anything about format or organization of the record. All it must do is pass one of the records, let's say record R, as the argument to the function, F, to get that record's key, F(R).

The function for the key could be provided to the array data structure in several ways. The program needing the array could define a function with a known name like `array_key` that fetches the key. That approach wouldn't be very portable, and it would make it impossible to have different arrays using different key functions. The key function could also be passed as an argument to the array's methods like `find` and `insert`. That would allow different arrays to use different key functions, but it has a potential problem. If the program using the arrays accidentally passes the wrong key function to an array that ordered its records by a different key, then the records could be out of order using the new key. Instead, it's better to define the key function *when the array is created* and not allow it to change. That's how we implement the `OrderedRecordArray` class, as shown in Listing 2-8.

**Listing 2-8** *The Basic* `OrderedRecordArray` *Class*

```
# Implement an Ordered Array of Records structure

def identity(x):      # The identity function
   return x

class OrderedRecordArray(object):
   def __init__(self, initialSize, key=identity):    # Constructor
      self.__a = [None] * initialSize  # The array stored as a list
      self.__nItems = 0  # No items in array initially
      self.__key = key   # Key function gets record key

   def __len__(self):    # Special def for len() func
      return self.__nItems      # Return number of items

   def get(self, n):     # Return the value at index n
      if n >= 0 and n < self.__nItems: # Check if n is in bounds, and
  return self.__a[n]     # only return item if in bounds
      raise IndexError("Index " + str(n) + " is out of range")

   def traverse(self, function=print): # Traverse all items
      for j in range(self.__nItems):   # and apply a function
  function(self.__a[j])

   def __str__(self):    # Special def for str() func
      ans = "["   # Surround with square brackets
      for i in range(self.__nItems):   # Loop through items
  if len(ans) > 1:       # Except next to left bracket,
```

```
            ans += ", "  # separate items with comma
        ans += str(self.__a[i])       # Add string form of item
            ans += "]"          # Close with right bracket
            return ans
```

The constructor for `OrderedRecordArray` takes a new argument, `key`, which is the key function. That function defaults to being the `identity` function, which is defined in the module and simply returns the first argument as the result. This makes the default behavior the same as the `OrderedArray` class shown in Listings 2-5 and 2-6. The key function is stored in the private instance variable `__key` so it should not be modified by the clients using `OrderedRecordArray`s.

Listing 2-9 shows that the `find()` and `search()` methods change to take a `key` as an argument, instead of the item or record used in the `OrderedArray` class. This key is a value, not a function, and is used to compare with the keys extracted from the records in the array. The `find` and `search` methods use the internal `__key` function on the records to get the right value to compare with the `key` being sought. The `insert` and `delete` method signatures don't change —they still operate on `item` records—but internally they change the way they pass the appropriate key to `find()`.

**Listing 2-9** *The Item Operations of the* `OrderedRecordArray` *Class*

```
class OrderedRecordArray(object):
…
    def find(self, key):      # Find index at or just below key
        lo = 0    # in ordered list
        hi = self.__nItems-1    # Look between lo and hi

        while lo <= hi:
    mid = (lo + hi) // 2        # Select the midpoint

    if self.__key(self.__a[mid]) == key:  # Did we find it?
        return mid        # Return location of item

    elif self.__key(self.__a[mid]) < key: # Is key in upper half?
        lo = mid + 1      # Yes, raise the lo boundary

    else:
        hi = mid - 1      # No, but could be in lower half

        return lo    # Item not found, return insertion point instead
```

```
    def search(self, key):
        idx = self.find(key)    # Search for a record by its key
        if idx < self.__nItems and self.__key(self.__a[idx]) == key:
    return self.__a[idx]        # and return item if found

    def insert(self, item):     # Insert item into the correct position
        if self.__nItems >= len(self.__a): # If array is full,
    raise Exception("Array overflow") # raise exception

        j = self.find(self.__key(item))      # Find where item should go

        for k in range(self.__nItems, j, -1): # Move bigger items right
    self.__a[k] = self.__a[k-1]

        self.__a[j] = item       # Insert the item
        self.__nItems += 1       # Increment the number of items

    def delete(self, item):        # Delete any occurrence
        j = self.find(self.__key(item))    # Try to find the item
        if j < self.__nItems and self.__a[j] == item:  # If found,
    self.__nItems -= 1       # One fewer at end
    for k in range(j, self.__nItems): # Move bigger items left
       self.__a[k] = self.__a[k+1]
    return True      # Return success flag

        return False      # Made it here; item not found
```

The test program for this new class, `OrderedRecordArrayClient.py` shown in Listing 2-10, uses records with two elements or fields. The key for the records is set to be the second element of each record. Loops in this version perform the insertions, deletions, and searches.

**Listing 2-10** *The `OrderedRecordArrayClient.py` Program*

```
from OrderedRecordArray import *

def second(x):  # Key on second element of record
    return x[1]

maxSize = 1000        # Max size of the array
arr = OrderedRecordArray(maxSize, second)  # Create the array object

# Insert 10 items
```

```
for rec in [('a', 3.1), ('b', 7.5), ('c', 6.0), ('d', 3.1),
     ('e', 1.4), ('f', -1.2), ('g', 0.0), ('h', 7.5),
     ('i', 7.5), ('j', 6.0)]:
    arr.insert(rec)

print("Array containing", len(arr), "items:\n", arr)

# Delete a few items, including some duplicates
for rec in [('c', 6.0), ('g', 0.0), ('g', 0.0),
     ('b', 7.5), ('i', 7.5)]:
    print("Deleting", rec, "returns", arr.delete(rec))

print("Array after deletions has", len(arr), "items:\n", arr)

for key in [4.4, 6.0, 7.5]:
    print("find(", key, ") returns", arr.find(key),
   "and get(", arr.find(key), ") returns",
   arr.get(arr.find(key)))
```

---

After putting 10 records in the array including some with duplicate keys, the test program deletes a few records, showing the result of the deletion. It then tries to find a few keys in the reduced array. The result of running the program is

```
$ python3 OrderedRecordArrayClient.py
Array containing 10 items:
 [('f', -1.2), ('g', 0.0), ('e', 1.4), ('d', 3.1), ('a', 3.1), ('j',
6.0), ('c', 6.0), ('i', 7.5), ('h', 7.5), ('b', 7.5)]
Deleting ('c', 6.0) returns False
Deleting ('g', 0.0) returns True
Deleting ('g', 0.0) returns False
Deleting ('b', 7.5) returns False
Deleting ('i', 7.5) returns True
Array after deletions has 8 items:
 [('f', -1.2), ('e', 1.4), ('d', 3.1), ('a', 3.1), ('j', 6.0), ('c',
6.0), ('h', 7.5), ('b', 7.5)]
find( 4.4 ) returns 4 and get( 4 ) returns ('j', 6.0)
find( 6.0 ) returns 5 and get( 5 ) returns ('c', 6.0)
find( 7.5 ) returns 6 and get( 6 ) returns ('h', 7.5)
```

The program output shows that deleting the record `('c', 6.0)` fails. Why? The next two deletions show that deleting `('g', 0.0)` succeeds the first time but fails the second time because only one record has that key, 0.0. That's what is expected, but the next deletions are unexpected. The deletion of the record `('b', 7.5)` fails, but the deletion of `('i', 7.5)` succeeds. What is going on?

The issue comes up because of the *duplicate keys*. The program inserts three records that have the key 7.5. When the `find()` method runs, it uses binary search to get the index to one of those records. The exact one it finds depends on the sequence of values for the `mid` variable. You can see which one it finds in the output of the `find` tests. Note that `find(4.4)` returns a valid index, 4, and that points to the location where a record with that key should go. The record at index 4 has the next higher key value, 6.0. When you call `find(7.5)` on the final `Array`, it returns 6, which points to the `('h', 7.5)` record. That isn't equal to the `('b', 7.5)` record using Python's == test. The `delete()` method removes only items that pass the == test. You can also deduce that `find(7.5)` did find the `('i', 7.5)` record on the earlier delete operation. This example illustrates an important issue when duplicate keys are allowed in a sorted data structure like `OrderedRecordArray`. One of the end-of-chapter programming projects asks you to change the behavior of this class to correctly delete records with duplicate keys.

# Big O Notation

Which algorithms are faster than others? Everyone wants their results as soon as possible, so you need to be able compare the different approaches. You can certainly run experiments with each program on a particular computer and with a particular set of data to see which is fastest. That capability is useful, but when the computer changes or the data changes, you could get different results. Computers generally get faster as better technologies are invented, and that makes all algorithms run faster. The changes with the data, however, are harder to predict. You've already seen that a binary search takes far fewer steps than a linear search because the number of items to search increases. We'd like to be able to extend that reasoning to help predict what will happen with other algorithms.

People like to categorize things, especially by what they are capable of doing. If you think about cutting grass, there are push lawn mowers, powered lawn mowers, riding lawn mowers, and towed grass cutters. Each one of them is good for different size jobs of grass cutting. Similarly for refrigeration, there are personal refrigerators, household refrigerators, restaurant kitchen refrigerators, walk-in refrigerators, and refrigerated warehouses for different quantities of perishable items. In each case, choosing something too big or too small for the job would be costly in time or money.

In computer science, a rough measure of performance called **Big O** notation is used to describe algorithms. It's primarily used to describe the speed of algorithms but is also used to describe how much storage they need. Algorithms with the same Big O speed are in the same category. The category gives a rough idea of what amount of data they can process (or storage they need). For example, the linear search `Array` class in Listing 2-3 should be plenty fast for small jobs like keeping a list of contacts in a personal computer or phone or watch. It would probably not be acceptable for the list of contacts of a large corporation with tens of thousands of employees, and it certainly would be too slow to manage all the contact information for a nation of tens of millions of people. By using the `OrderedRecordArray` class in Listing 2-8, you can get the benefit of binary search and drastically reduce the search time by making it proportional to the logarithm of the number of items. Are there even better algorithms? Big O notation helps answer that question.

# Insertion in an Unordered Array: Constant

Insertion into an unordered array is the only algorithm we've discussed that doesn't depend on how many items are in the array. The new item is always placed in the next available position, at `__a[self.__nItems]`, and `self.__nItems` is then incremented. Instead of using the variable names in a particular program, Big O notation uses N to stand for the number of items being managed. Insertion into an unordered array requires the same amount of time no matter how big N is. You can say that the time, T, to insert an item into an unsorted array is a constant, K:

$$T = K$$

In a real situation, the actual time required by the insertion is related to the speed of the processor, how efficiently the compiler has generated the program code, how much data is in the item being copied into the array, and other factors. The constant K in the preceding equation is used to account for all such factors. To find out what K is in a real situation, you need to measure how long an insertion took. (Software exists for this very purpose.) K would then be equal to that time.

# Linear Search: Proportional to N

You've seen that, in a linear search of items in an array, the number of comparisons that must be made to find a specified item is, on the average, half of the total number of items. Thus, if N is the total number of items, the search time T is proportional to half of N:

$$T = K \times N / 2$$

As with insertions, discovering the value of K in this equation would require timing searches for some (probably large) values of N and then using the resulting values of T to calculate K. There is probably some time spent launching the program and cleaning up when it's done that would add a small constant factor to the total time. By measuring the time taken for several searches, you can account for the variations for where the item falls in the array and for the extra constant factors added by the measurement. When you know K and any additional constants, you can calculate T for any other value of N.

For a handier formula, you could lump the 2 into the K. The new K is equal to the old K divided by 2. Now you have

$$T = K \times N$$

This equation says that average linear search times are proportional to the size of the array. If an array is twice as big, searching it will take twice as long. In this case, we are less concerned with getting a precise estimate of the time it will take as we are knowing how fast it will grow as N gets bigger.

## Binary Search: Proportional to log(N)

Similarly, you can concoct a formula relating T and N for a binary search:

$$T = K \times \log_2(N)$$

As you saw earlier, the search time is proportional to the base 2 logarithm of N. Actually, because any logarithm is related to any other logarithm by a constant (for example, multiplying by 3.322 to go from base 2 to base 10), you can lump this constant into K as well. Then you don't need to specify the base:

$$T = K \times \log(N)$$

## Don't Need the Constant

Big O notation looks like the formulas just described, but it dispenses with the constant K. When comparing algorithms, you don't really care about the particular processor or compiler; all you want to compare is how T changes for different values of N, not what the actual numbers are. Although the K might be important for getting a precise estimate for small values of N, when N is really large, it "dominates" the time calculation. Therefore, we drop the constant in Big O notation.

Big O notation uses the uppercase letter *O*, which you can think of as meaning "order of." In Big O notation, you would say that a linear search takes O(N) or "Order of N" or simply "Order N" time, and a binary search takes O(log(N)) time. A further simplification of the notation gets rid of the parentheses for the log function, and you simply write O(log N). Insertion into an unordered array takes O(1), or constant time. (That's the numeral 1 in the parentheses.)

Table 2-5 summarizes the running times of the algorithms we've discussed so far.

**Table 2-5** *Running Times in Big O Notation*

| Algorithm | Unordered Array | Ordered Array |
|---|---|---|
| Linear search | O(N) | |
| Binary search | Not possible | O(log N) |
| Insertion | O(1) | O(N) |
| Deletion | O(N) | O(N) |

You might ask why deletion in ordered arrays isn't shown as O(log N) + O(N) or maybe O(log N + N) because it uses binary search to find the location of the item to delete. The reason is that the O(N) part needed for shifting the items of the array is so much larger than the O(log N) part that it really doesn't matter when N gets big. Big O notation is intended to describe how the algorithm behaves for very large numbers of items.

Figure 2-8 graphs some Big O relationships between time (in number of steps) and number of items, N. Based on this graph, you might rate the various Big O values (very subjectively) like this:

- O(1) is excellent,

- O(log N) is good,
- O(N) is fair,
- O(N × log N) is poor, and
- O($N^2$) is bad.

O(N × log N) occurs in many kinds of sorting. O($N^2$) occurs in simple sorting and in certain graph algorithms, all of which we look at later in this book.


**Figure 2-8** *Graph of Big O times*

The idea in Big O notation isn't to give actual figures for running times but to convey how the running time grows as the number of items increases. This is the most meaningful way to compare algorithms, without actually measuring running times in a real installation. This is often called the **computational complexity** of algorithms, or the **order of the function** that characterizes the running time (that's where the "O" comes from). Algorithms that are more complex for the computer to process have a higher order and are usually avoided.

# Why Not Use Arrays for Everything?

Arrays seem to get the job done, so why not use them for all data storage? You've already seen some of their disadvantages. In an unordered array, you can insert items quickly, in O(1) time, but searching takes slow O(N) time. In an ordered array, you can search quickly, in O(log N) time, but insertion takes

O(N) time. For both kinds of arrays, deletion takes O(N) time because half the items (on the average) must be moved to fill in the hole.

It would be nice if there were data structures that could do everything—insertion, deletion, and searching—quickly, ideally in O(1) time, but if not that, then in O(log N) time. Traversal, by definition, needs O(N) time, but in more complex data structures, it could be larger. In the chapters ahead, we examine how closely these ideals can be approached and the price that must be paid in complexity.

Another problem with arrays is that their size is fixed when they are first created. The reason for that is the compiler needs to know how much space to set aside for the whole array and keep it separate from all the other data. Usually, when the program first starts, you don't know exactly how many items will be placed in the array later, so you guess how big it should be. If the guess is too large, you'll waste memory by having cells in the array that are never filled. If the guess is too small, you'll overflow the array, causing at best a message to the program's user, and at worst a program crash.

Other data structures are more flexible and can expand to hold the number of items inserted in them. The linked list, discussed in Chapter 5, is such a structure. One of the programming projects in this chapter asks you to make an expanding array data structure.

# Summary

- Arrays are sequential groupings of data elements. Each element can store a value called an item.
- Each element of the array can be accessed by knowing the start of the array and an integer index to the element.
- Object-oriented programs are used to implement data structures to encapsulate the algorithms that manipulate the data.
- Data structures use private instance variables to restrict access to important values of the structure that could cause errors if changed by the calling program.
- Unordered arrays offer fast insertion but slow searching and deletion.

- A binary search can be applied to an ordered array.
- The logarithm to the base B of a number A is (roughly) the number of times you can divide A by B before the result is less than 1.
- Linear searches require time proportional to the number of items in an array.
- Binary searches require time proportional to the logarithm of the number of items.
- Data structures usually store complex data types like records.
- A key must be defined to order complex data types.
- If duplicate items or keys are allowed in a data structure, the algorithms should have a predictable behavior for how they are managed.
- Big O notation provides a convenient way to compare the speed of algorithms.
- An algorithm that runs in O(1) time is the best, O(log N) is good, O(N) is fair, and O($N^2$) is bad.

# Questions

These questions are intended as a self-test for readers. Answers may be found in Appendix C.

1. Aside from the insert, delete, search, and traverse methods common to all "database" data structures, array data structures should have _____ method(s).
2. When constructing a new instance of an `Array` (Listing 2-3):

   a. the initial value for at least one of the array cells must be set.

   b. the data type of all the array cells must be set.

   c. the key for each data item must be set.

   d. the maximum number of cells the array can hold must be set.

   e. none of the above.

3. Why is it important to use private instance attributes like `__nItems` in the definition of the array data structure?

4. Inserting an item into an unordered array

   a. takes time proportional to the size of the array.

   b. requires multiple comparisons.

   c. requires shifting other items to make room.

   d. takes the same time no matter how many items there are.

5. True or False: When you delete an item from an unordered array, in most cases you shift other items to fill in the gap.

6. In an unordered array, allowing duplicates

   a. increases times for all operations.

   b. increases search times in some situations.

   c. always increases insertion times.

   d. sometimes decreases insertion times.

7. True or False: In an unordered array, it's generally faster to find out an item is not in the array than to find out it is.

8. Ordered arrays, compared with unordered arrays, are

   a. much quicker at deletion.

   b. quicker at insertion.

   c. quicker to create.

   d. quicker at searching.

9. Keys are used with arrays

   a. to provide a single value for each array item that can be used to order the items.

   b. to decrypt the values stored in the array cell.

   c. to decrease the insertion time in unordered arrays.

   d. to allow complex data types to be stored as a single key value in the array.

10. The `OrderedArray.py` (Listing 2-6) and `OrderedRecordArray.py` (Listing 2-9) modules have both a `find()` and a `search()` method. How are the two methods the same and how do they differ?

11. A logarithm is the inverse of _____.

12. The base 10 logarithm of 1,000 is _____.

13. The maximum number of items that must be examined to complete a binary search in an array of 200 items is

    a. 200.

    b. 8.

    c. 1.

    d. 13.

14. The base 2 logarithm of 64 is _____.

15. True or False: The base 2 logarithm of 100 is 2.

16. Big O notation tells

    a. how the speed of an algorithm relates to the number of items.

    b. the running time of an algorithm for a given size data structure.

    c. the running time of an algorithm for a given number of items.

    d. how the size of a data structure relates to the speed of one of its algorithms.

17. O(1) means a process operates in _____ time.

18. Advantages of using arrays include

    a. the variable size of array cells.

    b. the variable length of the array over the lifetime of the data structure.

    c. the O(1) access time to read or write an array cell.

    d. the O(1) time to traverse all the items in the array.

    e. all of the above.

19. A colleague asks for your comments on a data structure using an unordered array without duplicates and binary search. Which of the following comments makes sense?

a. Because the array can store any data type, a binary search won't be efficient.

b. Because the array is unordered, a binary search cannot guarantee finding the item being sought.

c. Because binary search takes O(N) time, it would be better to use an ordered array.

d. Because the array doesn't have duplicates, binary search doesn't really have an advantage over the simpler linear search.

20. You've been asked to adapt some code that maintains a record about each planet and their moons in a solar system like ours into a system that will store a record about every planet and moon in every known galaxy. The record structure will be a little larger for each planet to hold some new attributes. It's likely that the records will be added and updated "randomly" as telescopes and other sensors point at different parts of the universe over time, filling in some initial attributes of the records, and then updating others during frequent observations. The current code uses an unordered array for the records. Would you recommend any changes? If so, why?

# Experiments

Carrying out these experiments will help to provide insights into the topics covered in the chapter. No programming is involved.

2-A Use the Array Visualization tool to insert, search for, and delete items. Make sure you can predict what it's going to do. Do this both with duplicate values present and without.

2-B Make sure you can predict in advance what indices the Ordered Array Visualization tool will select at each step for `lo`, `mid`, and `hi` when you search for the lowest, second lowest, one above middle, and highest values in the array.

2-C In the Ordered Array Visualization tool, create an array of 12 cells and then use the Random Fill button to fill them with values. Use the Delete Rightmost button to remove the five highest values. Then insert five of the same value, somewhere in the middle of the array. Note the colors that are assigned to inserted values and the order they were inserted.

Can you predict the order they will be deleted? Try deleting the value you chose several times to see if your prediction is right.

## Programming Projects

Writing programs to solve the Programming Projects helps to solidify your understanding of the material and demonstrates how the chapter's concepts are applied. (As noted in the Introduction, qualified instructors may obtain completed solutions to the Programming Projects on the publisher's website.)

2.1 To the `Array` class in the `Array.py` program (Listing 2-3), add a method called `getMaxNum()` that returns the value of the highest number in the array, or `None` if the array has no numbers. You can use the expression `isinstance(`*x*`, (int, float))` to test for numbers. Add some code to `ArrayClient.py` (Listing 2-4) to exercise this method. You should try it on arrays containing a variety of data types and some that contain zeros and some that contain no numbers.

2.2 Modify the method in Programming Project 2.1 so that the item with the highest numeric value is not only returned by the method but also removed from the array. Call the method `deleteMaxNum()`.

2.3 The `deleteMaxNum()` method in Programming Project 2.2 suggests a way to create an array of numbers sorted by numeric value. Implement a sorting scheme that does not require modifying the `Array` class from Project 2.2, but only the code in `ArrayClient.py` (Listing 2-4).

2.4 Write a `removeDupes()` method for the `Array.py` program (Listing 2-3) that removes any duplicate entries in the array. That is, if three items with the value `'bar'` appear in the array, `removeDupes()` should remove two of them. Don't worry about maintaining the order of the items. One approach is to make a new, empty `list`, move items one at a time into it after first checking that they are not already in the new list, and then set the array to be the new list. Of course, the array size will be reduced if any duplicate entries exist. Write some tests to show it works on arrays with and without duplicate values.

2.5 Add a `merge()` method to the `OrderedRecordArray` class (Listing 2-8 and Listing 2-9) so that you can merge one ordered source array into that object's existing ordered array. The merge should occur only if both objects' key functions are identical. Your solution should create a new

list big enough to hold the contents of the current (`self`) list and the merging array list. Write tests for your class implementation that creates two arrays, inserts some random numbers into them, invokes `merge()` to add the contents of one to the other, and displays the contents of the resulting array. The source arrays may hold different numbers of data items. Your algorithm needs to compare the keys of the source arrays, picking the smallest one to copy to the destination. You also need to handle the situation when one source array exhausts its contents before the other. Note that, in Python, you can access a parameter's private attributes in a manner similar to using `self`. If the parameter `arr` is an `OrderedRecordArray` object, you can access its number of items as `arr.__nItems`.

2.6 Modify the `OrderedRecordArray` class (Listing 2-8 and Listing 2-9) so that requests to delete records that have duplicate keys correctly find the target records and delete them if present. Make sure you test the program thoroughly so that regardless of the number of records with duplicate keys or their order within the internal `list`, your modified version finds the matching record if it exists and leaves the `list` unchanged if it is not present.

2.7 Modify the `OrderedRecordArray` class (Listing 2-8 and Listing 2-9) so that it stores the maximum size of the array. When an insertion would go beyond the current maximum size, create a new list capable of holding more data and copy the existing list contents into it. The new size can be a fixed increment or a multiple of the current size. Test your new class by inserting data in a way that forces the object to expand the list several times and determine which is the best strategy—growing the list by adding a fixed amount of storage each time it fills up, or multiplying the list's storage by a fixed multiple each time it fills up.

# 3. Simple Sorting

## In This Chapter



As soon as you create a significant database, you'll probably think of reasons to sort the records in various ways. You need to arrange names in alphabetical order, students by grade, customers by postal code, home sales by price, cities in order of population, countries by land mass, stars by magnitude, and so on.

Sorting data may also be a preliminary step to searching it. As you saw in Chapter 2, "Arrays," a binary search, which can be applied only to sorted data, is much faster than a linear search.

Because sorting is so important and potentially so time-consuming, it has been the subject of extensive research in computer science, and some very sophisticated methods have been developed. In this chapter we look at three of the simpler algorithms: the bubble sort, the selection sort, and the insertion sort. Each is demonstrated in a Visualization tool. In Chapter 6, "Recursion," and Chapter 7, "Advanced Sorting," we return to look at more sophisticated approaches including Shellsort and quicksort.

The techniques described in this chapter, while unsophisticated and comparatively slow, are nevertheless worth examining. Besides being easier to understand, they are actually better in some circumstances than the more sophisticated algorithms. The insertion sort, for example, is preferable to

quicksort for small arrays and for almost-sorted arrays. In fact, an insertion sort is commonly used in the last stage of a quicksort implementation.

The sample programs in this chapter build on the array classes developed in the preceding chapter. The sorting algorithms are implemented as methods of the `Array` class.

Be sure to try out the algorithm visualizations provided with this chapter. They are very effective in explaining how the sorting algorithms work in combination with the descriptions and static pictures from the text.

# How Would You Do It?

Imagine that a kids-league football team is lined up on the field, as shown in Figure 3-1. You want to arrange the players in order of increasing height (with the shortest player on the left) for the team picture. How would you go about this sorting process?

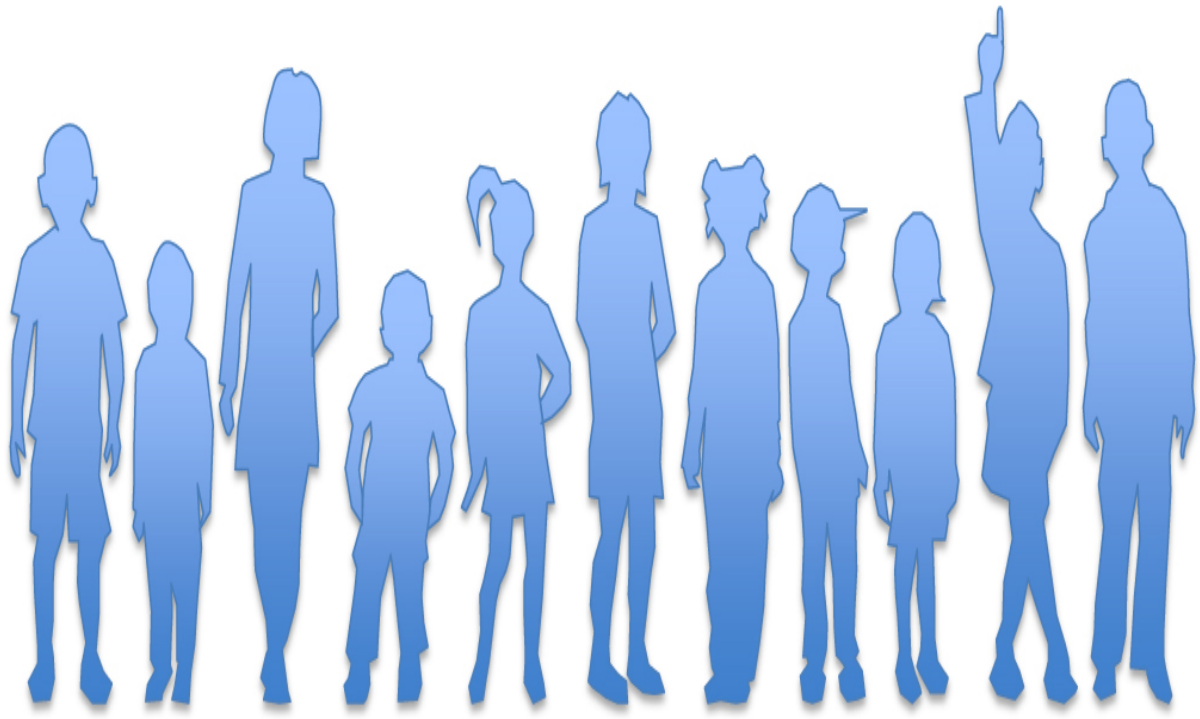


**Figure 3-1** *The unordered football team*

As a human being, you have advantages over a computer program. You can see all the players at once, and you can pick out the tallest player almost instantly. You don't need to laboriously measure and compare everyone. The players are

intelligent too and can self-sort pretty well. Also, the players don't need to occupy particular places. They can jostle each other, push each other a little to make room, and stand behind or in front of each other. After some ad hoc rearranging, you would have no trouble in lining up all the players, as shown in Figure 3-2.

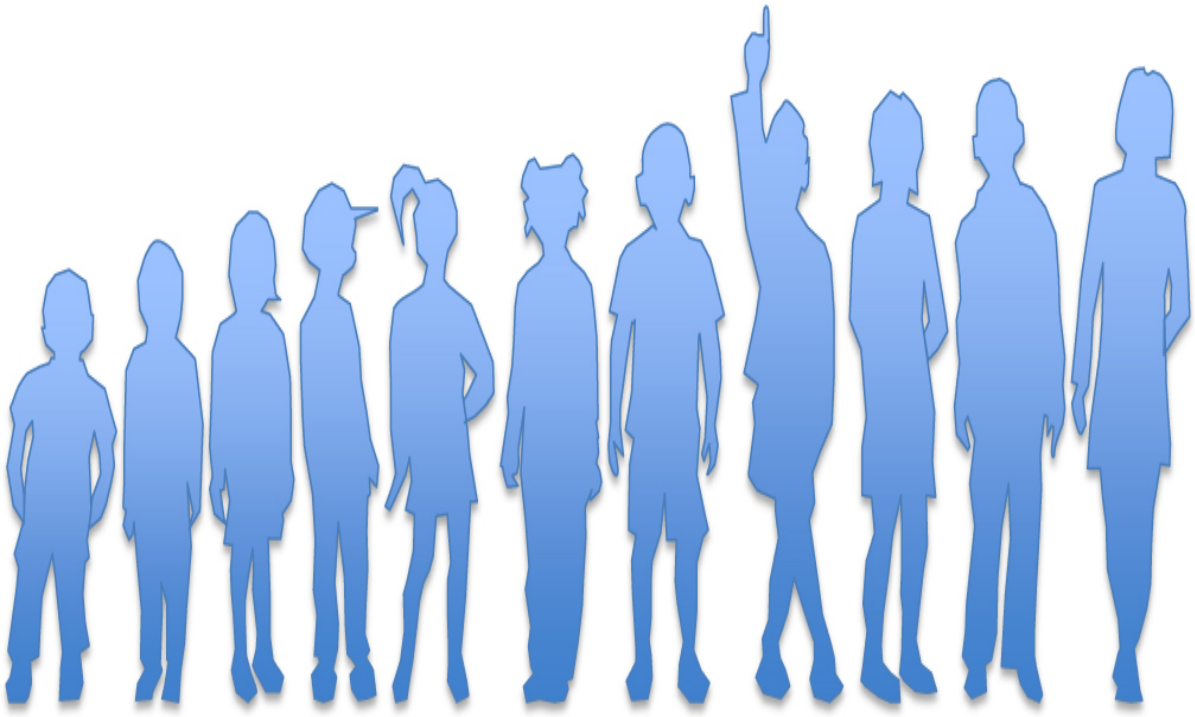


**Figure 3-2** *The football team ordered by height*

A computer program isn't able to glance over the data in this way. It can compare only two players at one time because that's how the comparison operators work. These operators expect players to be in exact positions, moving only one or two players at a time. This tunnel vision on the part of algorithms is a recurring theme. Things may seem simple to us humans, but the computer can't see the big picture and must, therefore, concentrate on the details and follow strict rules.

The three algorithms in this chapter all involve two operations, executed over and over until the data is sorted:

A. Compare two items.

B. Swap two items, or copy one item.

Each algorithm, of course, handles the details in a different way.

# Bubble Sort

The bubble sort is notoriously slow, but it's conceptually the simplest of the sorting algorithms and for that reason is a good beginning for our exploration of sorting techniques.

# Bubble Sort on the Football Players

Imagine that you are wearing blinders or are near-sighted so that you can see only two of the football players at the same time, if they're next to each other and if you stand very close to them. Given these constraints (like the computer algorithm faces), how would you sort them? Let's assume there are N players, and the positions they're standing in are numbered from 0 on the left to N−1 on the right.

The bubble sort routine works like this: you start at the left end of the line and compare the two players in positions 0 and 1. If the one on the left (in 0) is taller, you swap them. If the one on the right is taller, you don't do anything. Then you move over one position and compare the players in positions 1 and 2. Again, if the one on the left is taller, you swap them. This sorting process is shown in Figure 3-3.

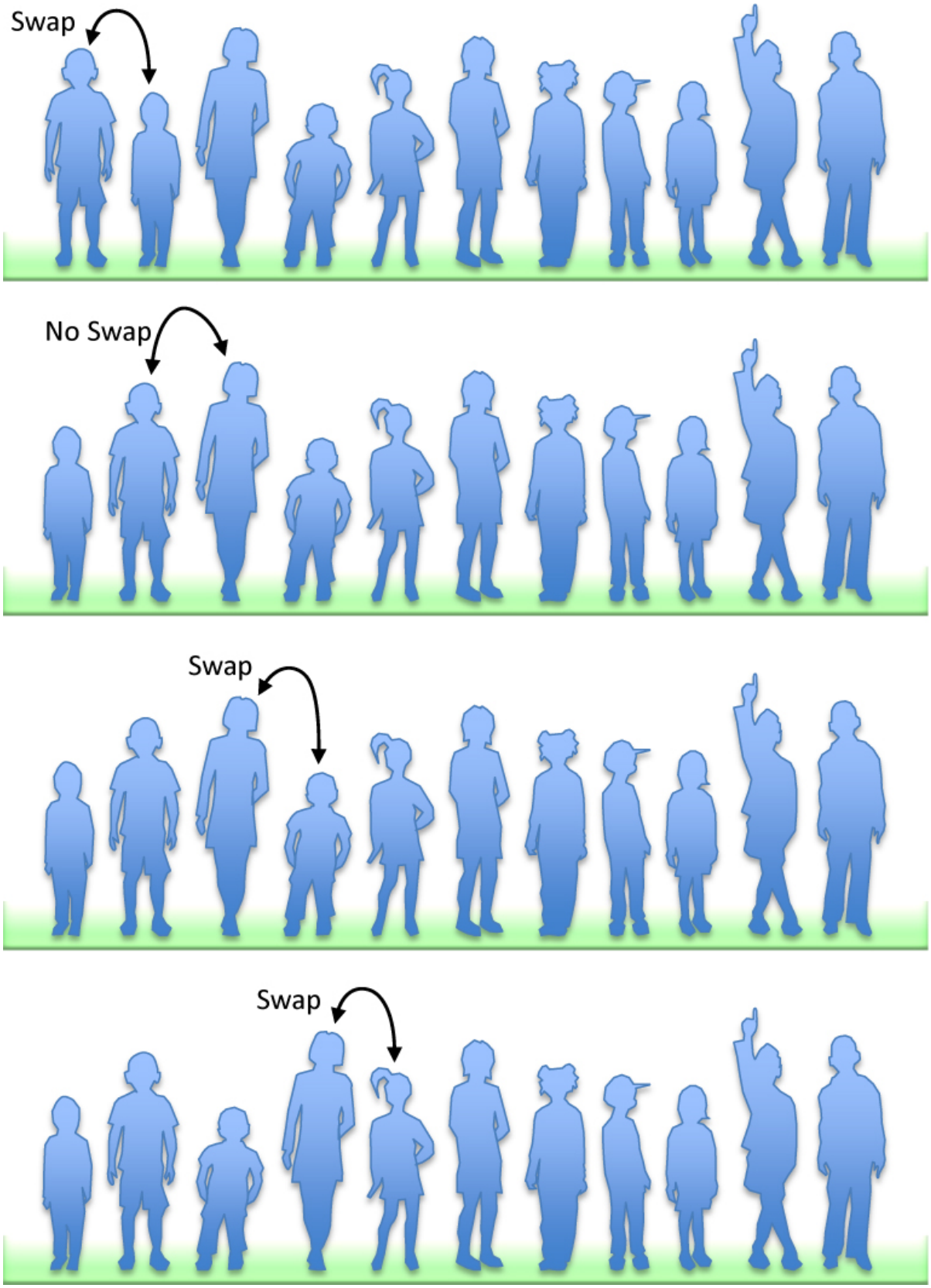

Swap
No Swap
Swap
Swap

**Figure 3-3** *Bubble sort: swaps made during the first pass*

Here are the rules you're following:

1. Compare two players.
2. If the one on the left is taller, swap them.
3. Move one position right.

You continue down the line this way until you reach the right end. You have by no means finished sorting the players, but you do know that the tallest player now is on the right. This must be true because, as soon as you encounter the tallest player, you end up swapping them every time you compare two players, until eventually the tallest reaches the right end of the line. This is why it's called the bubble sort: as the algorithm progresses, the biggest items "bubble up" to the top end of the array. Figure 3-4 shows the players at the end of the first pass.

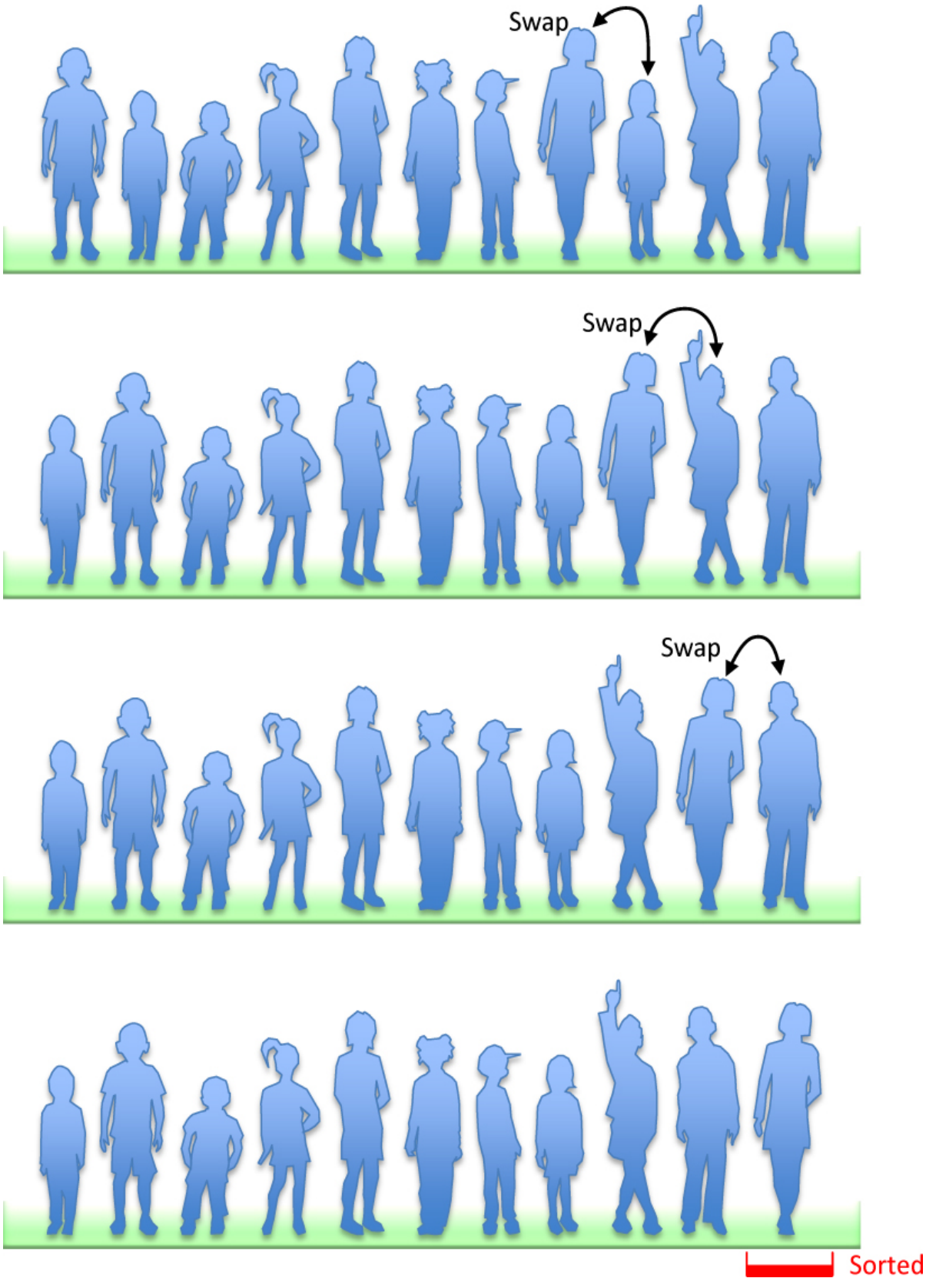

Swap
Swap
Swap
Sorted

**Figure 3-4** *Bubble sort: the end of the first pass*

After this first pass through all the players, you've made N−1 comparisons and somewhere between 0 and N−1 swaps, depending on the initial arrangement of the players. The player at the end of the line is sorted and won't be moved again.

Now you go back and start another pass from the left end of the line. Again, you go toward the right, comparing and swapping when appropriate. This time, however, you can stop one player short of the end of the line, at position N−2, because you know the last position, at N−1, already contains the tallest player. This rule could be stated as:

> When you reach the first sorted player, start over at the left end of the line.

You continue this process until all the players are in order. Describing this process is much harder than demonstrating it, so let's watch its work in the Simple Sorting Visualization tool.

# The Simple Sorting Visualization Tool

Start the Simple Sorting Visualization tool (run `python3 SimpleSorting.py` as described in Appendix A). This program shows an array of values and provides multiple ways of sorting and manipulating it. Figure 3-5 shows the initial display.


**Figure 3-5** *The Simple Sorting Visualization tool*

The Insert, Search, Delete, New, Random Fill, and Delete Rightmost buttons function like those we saw in the Array Visualization tool used in Chapter 2. There are new buttons on the right relevant to sorting. Select the Bubble Sort button to start sorting the items. As with the other visualization tools, you can slow down or speed up the animation using the slider on the bottom. You can pause and resume playing the animation with the play, pause, and skip buttons (▶, ❙❙, ▶|).

When the bubble sort begins, it adds two arrows labeled "inner" and "last" next to the array. These point to where the algorithm is doing its work. The inner arrow walks from the leftmost cell to the right, swapping item pairs when it finds a taller one to the left of a shorter one, just like the players on the team. It stops when it reaches the last arrow, which marks the first item that is in the final sorted order.

As each maximum item "bubbles" up to the far right, the last arrow moves to the left by one cell, to keep track of which items are sorted. When the last arrow moves all the way to the first cell on the left, the array is fully sorted. To run a new sort operation, select Shuffle, and then Bubble Sort again.

Try going through the bubble sort process using the step (▶|) or play/pause (▶/❙❙) buttons. At each step, think about what should happen next. Will a swap occur? How will the two arrows change?

Try making a new array with many cells (30+) and filling it with random items. The cells will become too narrow to show the numbers of each item. The heights of the colored rectangles indicate their number, but it's harder to see which one is taller when they are nearly equal in height. Can you still predict what will happen at each step?

The visualization tool also has buttons to fill empty cells with keys in increasing order and decreasing order. It's interesting to see what happens when sorting algorithms encounter data that is already sorted or reverse sorted. You can also scramble the data with the Shuffle button to try running a new sort.

### Note

The Stop (■) button stops an animation and allows you to start other operations. The array contents, however, may be different than what they were at the start of the animation. There may be missing items or extra copies of items depending on when the operation was interrupted. In this way, the visualization mimics what happens in computer memory if something stops the execution.

# Python Code for a Bubble Sort

The bubble sort algorithm is pretty straightforward to explain, but we need to look at how to write the program. Listing 3-1 shows the `bubbleSort()` method of an `Array` class. It is almost the same as the `Array` class introduced in Chapter 2, and the full `SortArray` module is shown later in this chapter in Listing 3-4.

**Listing 3-1** *The `bubbleSort()` method of the `Array` class*

```
def bubbleSort(self):                  # Sort comparing adjacent vals
   for last in range(self.__nItems-1, 0, -1):  # and bubble up
      for inner in range(last):        # inner loop goes up to last
         if self.__a[inner] > self.__a[inner+1]:  # If item less
            self.swap(inner, inner+1) # than adjacent item, swap
```

In the `Array` class, each element of the array is assumed to be a simple value that can be compared with any of the other values for the purposes of ordering them. We reintroduce a key function to handle ordering records later.

The `bubbleSort()` method has two loops. The inner loop handles stepping through the array and swapping elements that are out of order. The outer loop handles the decreasing length of the unsorted part of the array. The outer loop sets the `last` variable to point initially at the last element of the array. That's done by using Python's `range` function to start at `__nItems-1`, and step down to 1 in increments of −1. The inner loop starts each time with the `inner` variable set to 0 and increments by 1 to reach `last-1`. The elements at indices greater than `last` are always completely sorted. After each pass of the inner loop, the `last` variable can be reduced by 1 because the maximum element between 0 and `last` bubbled up to the `last` position.

The inner loop body performs the test to compare the elements at the `inner` and `inner+1` indices. If the value at `inner` is larger than the one to its right, you have to swap the two array elements (we'll see how `swap()` works when we look at all the `SortArray` code in a later section).

# Invariants

In many algorithms there are conditions that remain unchanged as the algorithm proceeds. These conditions are called **invariants**. Recognizing invariants can be useful in understanding the algorithm. In certain situations they may help in debugging; you can repeatedly check that the invariant is true, and signal an error if it isn't.

In the `bubbleSort()` method, the invariant is that the array elements to the right of `last` are sorted. This remains true throughout the running of the algorithm. On the first pass, nothing has been sorted yet, and there are no items to the right of `last` because it starts on the rightmost element.

# Efficiency of the Bubble Sort

If you use bubble sort on an array with 11 cells, the `inner` arrow makes 10 comparisons on the first pass, nine on the second, and so on, down to one comparison on the last pass. For 11 items, this is

$$10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 = 55$$

In general, where N is the number of items in the array, there are N−1 comparisons on the first pass, N−2 on the second, and so on. The formula for the sum of such a series is

$$(N-1) + (N-2) + (N-3) + \ldots + 1 = N \times (N-1)/2$$

$N \times (N-1)/2$ is 55 ($11 \times 10/2$) when N is 11.

Thus, the algorithm makes about $N^2/2$ comparisons (ignoring the −1, which doesn't make much difference, especially if N is large).

There are fewer swaps than there are comparisons because two bars are swapped only if they need to be. If the data is random, a swap is necessary about half the time, so there would be about $N^2/4$ swaps. (In the worst case, with the initial data inversely sorted, a swap is necessary with every comparison.)

Both swaps and comparisons are proportional to $N^2$. Because constants don't count in Big O notation, you can ignore the 2 and the 4 in the divisor and say that the bubble sort runs in $O(N^2)$ time. This is slow, as you can verify by running the Bubble Sort in the Simple Sorting Visualization tool with arrays of 20+ cells.

Whenever you see one loop nested within another, such as those in the bubble sort and the other sorting algorithms in this chapter, you can suspect that an algorithm runs in $O(N^2)$ time. The outer loop executes N times, and the inner loop executes N (or perhaps N divided by some constant) times for each cycle of the outer loop. This means you're doing something approximately N×N or $N^2$ times.

## Selection Sort

How can you improve the efficiency of the sorting operation? You know that $O(N^2)$ is pretty bad. Can you get to O(N) or maybe even O(log N)? We look next at a method called selection sort, which reduces the number of swaps. That could be significant when sorting large records by a key. Copying entire records rather than just integers could take much more time than comparing two keys. In Python as in other languages, the computer probably just copies pointers or references to record objects rather than the entire record, but it's important to understand which operations are happening the most times.

## Selection Sort on the Football Players

Let's consider the football players again. In the selection sort, you don't compare players standing next to each other, but you still can look at only two players and compare their heights. In this algorithm, you need to remember a certain player's height—perhaps by using a measuring stick and writing the number in a notebook. A ball also comes in handy as a marker.

## A Brief Description

What's involved in the selection sort is making a pass through all the players and picking (or *selecting*, hence the name of the sort) the shortest one. This shortest player is then swapped with the player on the left end of the line, at position 0. Now the leftmost player is sorted and doesn't need to be moved again. Notice that in this algorithm the sorted players accumulate on the left (lower indices), whereas in the bubble sort they accumulated on the right.

The next time you pass down the row of players, you start at position 1, and, finding the minimum, swap with position 1. This process continues until all the players are sorted.

## A More Detailed Description

Let's start at the left end of the line of players. Record the leftmost player's height in your notebook and put a ball on the ground in front of this person. Then compare the height of the next player to the right with the height in your notebook. If this player is shorter, cross out the height of the first player and record the second player's height instead. Also move the ball, placing it in front of this new "shortest" (for the time being) player. Continue down the row, comparing each player with the minimum. Change the minimum value in your notebook and move the ball whenever you find a shorter player. When you reach the end of the line, the ball will be in front of the shortest player. For example, the top of Figure 3-6 shows the ball being placed in front of the fourth player from the left.

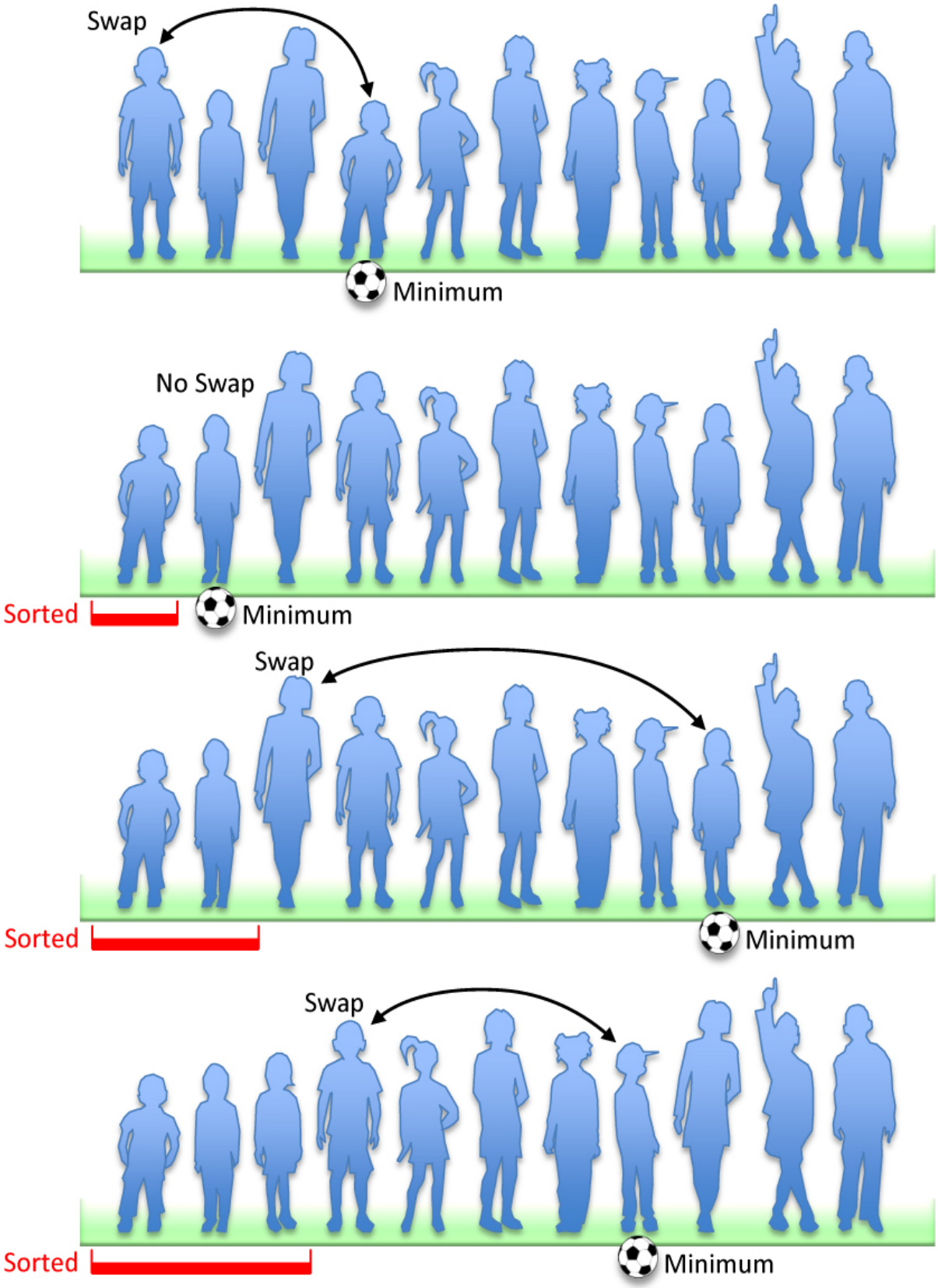

Swap
Minimum
No Swap
Sorted
Minimum
Swap
Sorted
Minimum
Swap
Sorted
Minimum

**Figure 3-6** *Selection sort on football players*

Swap this shortest player with the player on the left end of the line. You've now sorted one player. You've made N−1 comparisons, but only one swap.

On the next pass, you do exactly the same thing, except that you can skip the player on the left because this player has already been sorted. Thus, the algorithm starts the second pass at position 1, instead of 0. In the case of Figure 3-6, this second pass finds the shortest player at position 1, so no swap is needed. With each succeeding pass, one more player is sorted and placed on the left, and one fewer player needs to be considered when finding the new minimum. The bottom of Figure 3-6 shows how this sort looks after the first three passes.

## The Selection Sort in the Simple Sorting Visualization Tool

To see how the selection sort looks in action, try out the Selection Sort button in the Simple Sorting Visualization tool. Use the New and Random Fill buttons to create a new array of 11 randomly arranged items. When you select the Selection Sort button, it adds arrows to the display that look something like those shown in Figure 3-7. The arrow labeled "outer" starts on the left along with a second arrow labeled "min." The "inner" arrow starts one cell to the right and marks where the first comparison happens.


**Figure 3-7** *The start of a selection sort in the Simple Sorting Visualization tool*

As inner moves right, each cell it points at is compared with the one at min. If the value at min is lower, then the min arrow is moved to where inner is, just as you moved the ball to lie in front of the shortest player.

When inner reaches the right end, the items at the outer and min arrows are swapped. That means that one more cell has been sorted, so outer can be moved one cell to the right. The next pass starts with min pointing to the new position of outer and inner one cell to their right.

The outer arrow marks the position of the first unsorted item. All items from outer to the right end are unsorted. Cells to the left of outer are fully sorted. The sorting continues until outer reaches the last cell on the right. At that point, the next possible comparison would have to be past the last cell, so the sorting must stop.

# Python Code for Selection Sort

As with the bubble sort, implementing a selection sort requires two nested loops. The Python method is shown in Listing 3-2.

**Listing 3-2** *The* `selectionSort()` *Method of the* `Array` *Class*

```
def selectionSort(self):            # Sort by selecting min and
   for outer in range(self.__nItems-1): # swapping min to leftmost
      min = outer                   # Assume min is leftmost
      for inner in range(outer+1, self.__nItems): # Hunt to right
         if self.__a[inner] < self.__a[min]: # If we find new min,
            min = inner             # update the min index

      # __a[min] is smallest among __a[outer]...__a[__nItems-1]
      self.swap(outer, min)       # Swap leftmost and min
```

The outer loop starts by setting the `outer` variable to point at the beginning of the array (index 0) and proceeds right toward higher indices. The minimum valued item is assumed to be the first one by setting `min` to be the same as `outer`. The inner loop, with loop variable `inner`, begins at `outer+1` and likewise proceeds to the right.

At each new position of `inner`, the elements `__a[inner]` and `__a[min]` are compared. If `__a[inner]` is smaller, then `min` is given the value of `inner`. At

the end of the inner loop, `min` points to the minimum value, and the array elements pointed to by `outer` and `min` are swapped.

# Invariant

In the `selectionSort()` method, the array elements with indices less than `outer` are always sorted. This condition holds when `outer` reaches `__nItems-1`, which means all but one item are sorted. At that point, you could try to run the inner loop one more time, but it couldn't change the `min` index from `outer` because there is only one item left in the unsorted range. After doing nothing in the inner loop, it would then swap the item at `outer` with itself—another do nothing operation—and then `outer` would be incremented to `__nItems`. Because that entire last pass does nothing, you can end when `outer` reaches `__nItems-1`.

# Efficiency of the Selection Sort

The comparisons performed by the selection sort are $N \times (N-1)/2$, the same as in the bubble sort. For 11 data items, this is 55 comparisons. That process requires exactly 10 swaps. The number of swaps is always $N-1$ because the outer loop does it for each iteration. That compares favorably with the approximately $N^2/4$ swaps needed in bubble sort, which works out to about 30 for 11 items. With 100 items, the selection sort makes 4,950 comparisons and 99 swaps. For large values of N, the comparison times dominate, so you would have to say that the selection sort runs in $O(N^2)$ time, just as the bubble sort did. It is, however, unquestionably faster because there are O(N) instead of $O(N^2)$ swaps. Note that the number of swaps in bubble sort is dependent on the initial ordering of the array elements, so in some special cases, they could number fewer than those made by the selection sort.

# Insertion Sort

In most cases the insertion sort is the best of the elementary sorts described in this chapter. It still executes in $O(N^2)$ time, but it's about twice as fast as the bubble sort and somewhat faster than the selection sort in normal situations. It's also not too complex, although it's slightly more involved than the bubble

and selection sorts. It's often used as the final stage of more sophisticated sorts, such as quicksort.

## Insertion Sort on the Football Players

To begin the insertion sort, start with the football players lined up in random order. (They wanted to play a game, but clearly they've got to wait until the picture can be taken.) It's easier to think about the insertion sort if you begin in the middle of the process, when part of the team is sorted.

## Partial Sorting

You can use your handy ball to mark a place in the middle of the line. The players to the left of this marker are **partially sorted**. This means that they are sorted among themselves; each one is taller than the person to their left. The players, however, aren't necessarily in their final positions because they may still need to be moved when previously unsorted players are inserted between them.

Note that partial sorting did not take place in the bubble sort and selection sort. In these algorithms, a group of data items was completely sorted at any given time; in the insertion sort, one group of items is only partially sorted.

## The Marked Player

The player where the marker ball is, whom we call the "marked" player, and all the players to the right are as yet unsorted. Figure 3-8 shows the process. At the top, the players are unsorted. The next row in the figure shows the situation three steps later. The marker ball is put in front of a player who has stepped forward.

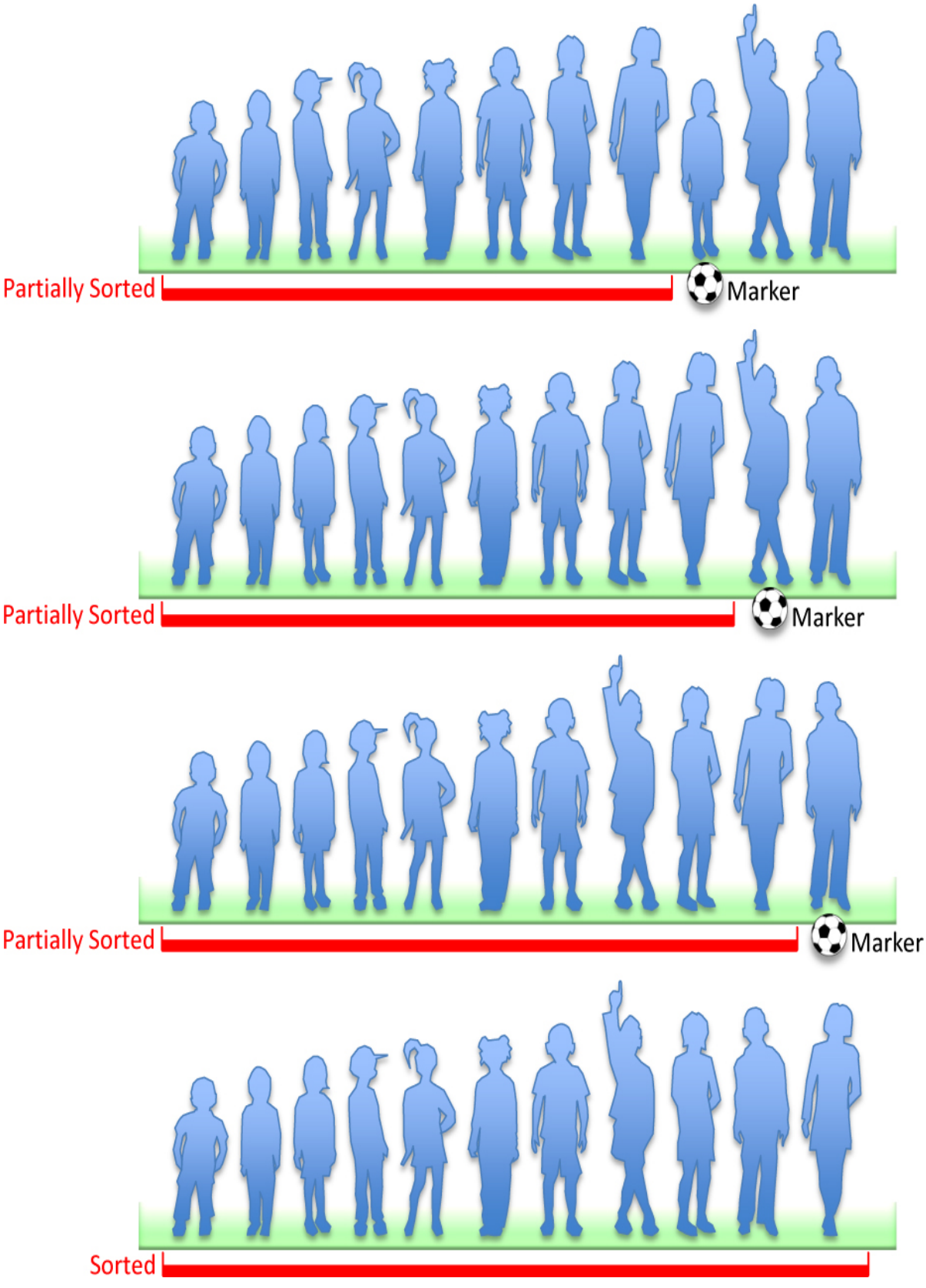

Partially Sorted
Marker
Partially Sorted
Marker
Partially Sorted
Marker
Sorted

**Figure 3-8** *The insertion sort on football players*

What you're going to do is insert the marked player in the appropriate place in the (partially) sorted group. To do this, you need to shift some of the sorted players to the right to make room. To provide a space for this shift, the marked player is taken out of line. (In the program this data item is stored in a temporary variable.) This step is shown in the second row of Figure 3-8.

Now you shift the sorted players to make room. The tallest sorted player moves into the marked player's spot, the next-tallest player into the tallest player's spot, and so on as shown by the arrows in the third row of the figure.

When does this shifting process stop? Imagine that you and the marked player are walking down the line to the left. At each position you shift another player to the right, but you also compare the marked player with the player about to be shifted. The shifting process stops when you've shifted the last player that's taller than the marked player. The last shift opens up the space where the marked player, when inserted, will be in sorted order. This step is shown in the bottom row of Figure 3-8.

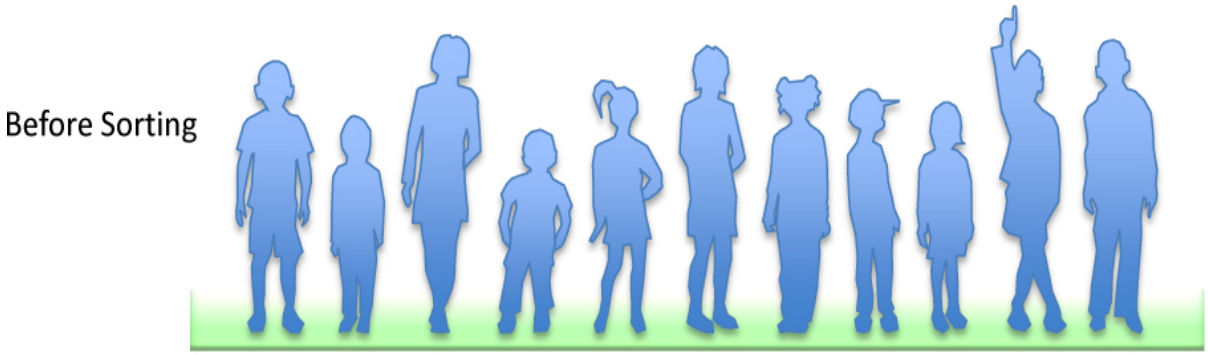
Now the partially sorted group is one player bigger, and the unsorted group is one player smaller. The marker ball is moved one space to the right, so it's again in front of the leftmost unsorted player. This process is repeated until all the unsorted players have been inserted (hence the name *insertion* sort) into the appropriate place in the partially sorted group.

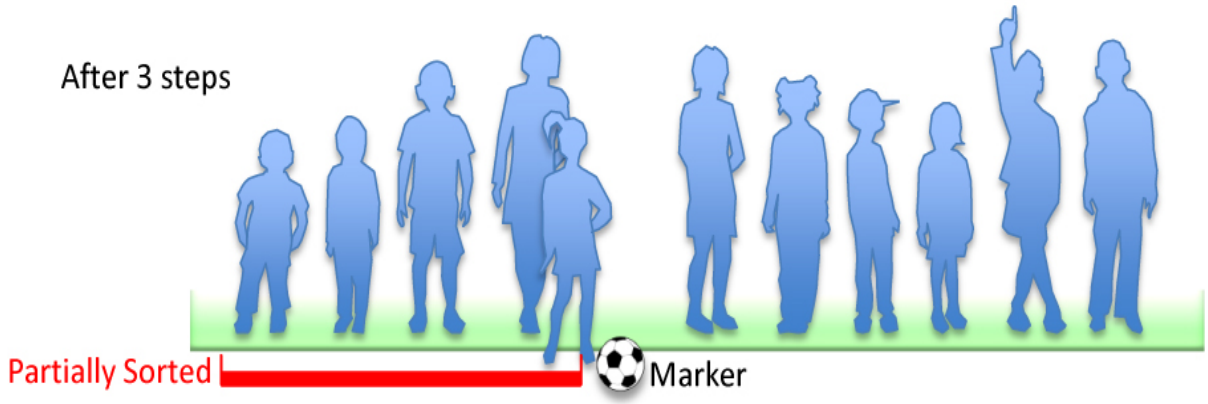
# The Insertion Sort in the Simple Sorting Visualization Tool

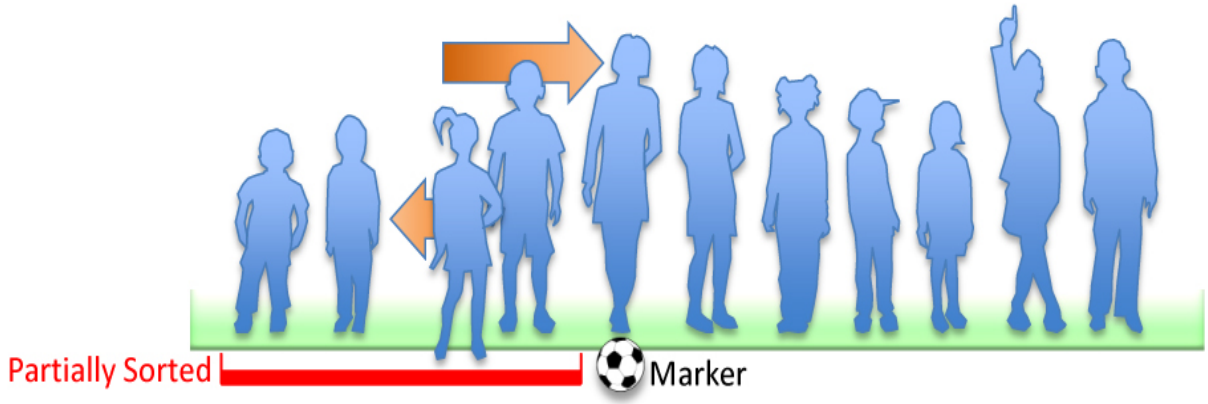
Return to the Simple Sorting Visualization tool and create another 11-cell array of random values. Then select the Insertion Sort button to start the sorting process. As in the other sort operations, the algorithm puts up two arrows: one for the outer and one for the inner loops. The arrow labeled "outer" is the equivalent of the marker ball in sorting the players.

In the example shown in Figure 3-9, the outer arrow is pointing at the fifth cell in the array. It already copied item 61 to the `temp` variable below the array. It has also copied item 94 into the fifth cell and is starting to copy item 85 into the fourth. The four cells to the left of the outer arrow are partially sorted; the cells to its right are unsorted. Even while it decides where the marked player

should go and there are extra copies of one of them, the left-hand cells remain partially sorted. This matches what happens in the computer memory when you copy the array item to a temporary variable.


Before Sorting


After 3 steps
Partially Sorted
Marker


Partially Sorted
Marker

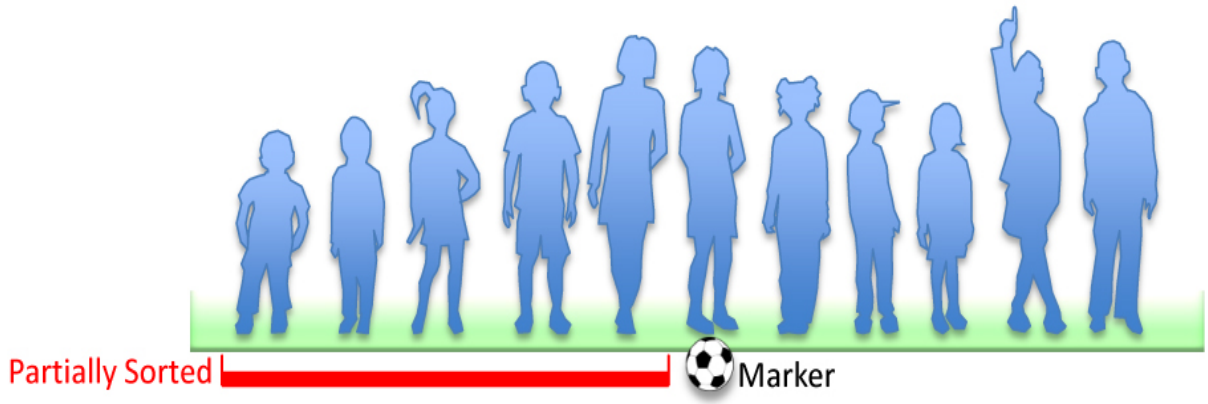

Partially Sorted
Marker

**Figure 3-9** *In the middle of an insertion sort in the Simple Sorting Visualization tool*

The inner arrow starts at same cell as outer and moves to the left, shifting items taller than the marked `temp` item to the right. In Figure 3-9, item 85 is being copied to where inner points because it is larger than item 61. After that copy is completed, inner moves one cell to the left and finds that item 77 needs to be moved too. On the next step for inner, it finds item 59 to its left. That item is smaller than 61, and the marked item is copied from temp back to the inner cell, where item 77 used to be.

Try creating a larger array filled with random values. Run the insertion sort and watch the updates to the arrows and items. Everything to the left of the outer arrow remains sorted by height. The inner arrow moves to the left, shifting items until it finds the right cell to place the item that was marked (copied from the outer cell).

Eventually, the outer arrow arrives at the right end of the array. The inner arrow moves to the left from that right edge and stops wherever its appropriate to insert the last item. It might finish anywhere in the array. That's a little different than the bubble and selection sorts where the outer and inner loop variables finish together.

Notice that, occasionally, the item copied from outer to temp is copied right back to where it started. That occurs when outer happens to point at an item larger than every item to its left. A similar thing happens in the selection sort when the outer arrow points at an item smaller than every item to its right; it ends up doing a swap with itself. Doing these extra data moves might seem inefficient, but adding tests for these conditions could add its own inefficiencies. An experiment at the end of the chapter asks you to explore when it might make sense to do so.

# Python Code for Insertion Sort

Let's now look at the `insertionSort()` method for the `Array` class as shown in Listing 3-3. You still need two loops, but the inner loop hunts for the insertion point using a `while` loop this time.

**Listing 3-3** *The `insertionSort()` method of the `Array` class*

```
def insertionSort(self):            # Sort by repeated inserts
   for outer in range(1, self.__nItems): # Mark one element
      temp = self.__a[outer]        # Store marked elem in temp
      inner = outer                 # Inner loop starts at mark
      while inner > 0 and temp < self.__a[inner-1]: # If marked
         self.__a[inner] = self.__a[inner-1] # elem smaller, then
         inner -= 1                 # shift elem to right
      self.__a[inner] = temp        # Move marked elem to 'hole'
```

In the outer `for` loop, `outer` starts at 1 and moves right. It marks the leftmost unsorted array element. In the inner `while` loop, `inner` starts at `outer` and moves left to `inner-1`, until either `temp` is smaller than the array element there, or it can't go left any further. Each pass through the `while` loop shifts another sorted item one space right.

## Invariants in the Insertion Sort

The data items with smaller indices than `outer` are partially sorted. This is true even at the beginning when `outer` is 1 because the single item in cell 0 forms a one-element sequence that is always sorted. As items shift, there are multiple copies of one of them, but they all remain in partially sorted order.

## Efficiency of the Insertion Sort

How many comparisons and copies does this algorithm require? On the first pass, it compares a maximum of one item. On the second pass, it's a maximum of two items, and so on, up to a maximum of N−1 comparisons on the last pass. (We ignore the check for `inner > 0` and count only the intra-element comparisons). This adds up to

$$1 + 2 + 3 + \ldots + N-1 = N\times(N-1)/2$$

On each pass, however, the number of items actually compared before the insertion point is found is, on average, half of the partially sorted items, so you can divide by 2, which gives

$$N\times(N-1)/4$$

The number of copies is approximately the same as the number of comparisons because it makes one copy for each comparison but the last. Copying one item is less time-consuming than a swap of two items, so for random data this algorithm runs twice as fast as the bubble sort and faster than the selection sort, on average.

In any case, like the other sort routines in this chapter, the insertion sort runs in $O(N^2)$ time for random data.

For data that is already sorted or almost sorted, the insertion sort does much better. When data is in order, the condition in the `while` loop is never true, so it becomes a simple statement in the outer loop, which executes N−1 times. In this case the algorithm runs in O(N) time. If the data is almost sorted, the insertion sort runs in almost O(N) time, which makes it a simple and efficient way to order an array that is only slightly out of order.

For data arranged in inverse sorted order, however, every possible comparison and shift is carried out, so the insertion sort runs no faster than the bubble sort. Try some experiments with the Simple Sorting Visualization tool. Sort an 8+ element array with the selection sort and then try sorting the result with the selection sort again. Even though the second attempt doesn't make any "real" swaps, it takes the same number of steps to go through the process. Then shuffle the array and sort it twice using the insertion sort. The second attempt at the insertion sort is significantly shorter because the inner loop ends after one comparison.

# Python Code for Sorting Arrays

Now let's combine the three sorting algorithms into a single object class to compare them. Starting from the `Array` class introduced in Chapter 2, we add the new methods for swapping and sorting plus the `__str__()` method for displaying the contents of arrays as a string and put them in a module called `SortArray.py`, as shown in Listing 3-4.

**Listing 3-4** *The `SortArray.py` module*

```
# Implement a sortable Array data structure

class Array(object):
```

```
def __init__(self, initialSize):    # Constructor
   self.__a = [None] * initialSize  # The array stored as a list
   self.__nItems = 0                # No items in array initially

def __len__(self):                  # Special def for len() func
   return self.__nItems             # Return number of items

def get(self, n):                   # Return the value at index n
   if 0 <= n and n < self.__nItems: # Check if n is in bounds, and
      return self.__a[n]            # only return item if in bounds

def set(self, n, value):            # Set the value at index n
   if 0 <= n and n < self.__nItems: # Check if n is in bounds, and
      self.__a[n] = value           # only set item if in bounds

def swap(self, j, k):               # Swap the values at 2 indices
   if (0 <= j and j < self.__nItems and # Check if indices are in
       0 <= k and k < self.__nItems): # bounds, before processing
      self.__a[j], self.__a[k] = self.__a[k], self.__a[j]

def insert(self, item):             # Insert item at end
   self.__a[self.__nItems] = item   # Item goes at current end
   self.__nItems += 1               # Increment number of items

def find(self, item):               # Find index for item
   for j in range(self.__nItems):   # Among current items
      if self.__a[j] == item:       # If found,
         return j                   # then return index to element
   return -1                        # Not found -> return -1

def search(self, item):             # Search for item
   return self.get(self.find(item)) # and return item if found

def delete(self, item):             # Delete first occurrence
   for j in range(self.__nItems):   # of an item
      if self.__a[j] == item:       # Found item
         self.__nItems -= 1         # One fewer at end
         for k in range(j, self.__nItems):  # Move items from
            self.__a[k] = self.__a[k+1]     # right over 1
         return True                # Return success flag

   return False     # Made it here, so couldn't find the item

def traverse(self, function=print): # Traverse all items
   for j in range(self.__nItems):   # and apply a function
      function(self.__a[j])
```

```
def __str__(self):                    # Special def for str() func
   ans = "["                          # Surround with square brackets
   for i in range(self.__nItems):     # Loop through items
      if len(ans) > 1:                # Except next to left bracket,
         ans += ", "                  # separate items with comma
      ans += str(self.__a[i])         # Add string form of item
   ans += "]"                         # Close with right bracket
   return ans

def bubbleSort(self):                 # Sort comparing adjacent vals
   for last in range(self.__nItems-1, 0, -1):  # and bubble up
      for inner in range(last):       # inner loop goes up to last
         if self.__a[inner] > self.__a[inner+1]:  # If elem less
            self.swap(inner, inner+1) # than adjacent value, swap

def selectionSort(self):              # Sort by selecting min and
   for outer in range(self.__nItems-1): # swapping min to leftmost
      min = outer                     # Assume min is leftmost
      for inner in range(outer+1, self.__nItems): # Hunt to right
         if self.__a[inner] < self.__a[min]: # If we find new min,
            min = inner               # update the min index

      # __a[min] is smallest among __a[outer]...__a[__nItems-1]
      self.swap(outer, min)          # Swap leftmost and min

def insertionSort(self):              # Sort by repeated inserts
   for outer in range(1, self.__nItems): # Mark one element
      temp = self.__a[outer]          # Store marked elem in temp
      inner = outer                   # Inner loop starts at mark
      while inner > 0 and temp < self.__a[inner-1]: # If marked
         self.__a[inner] = self.__a[inner-1] # elem smaller, then
         inner -= 1                   # shift elem to right
      self.__a[inner] = temp          # Move marked elem to 'hole'
```

The `swap()` method swaps the values in two cells of the array. It ensures that swaps happen only with items that have been inserted in the array and not with allocated but uninitialized cells. Those tests may not be necessary with the sorting methods in this module that already ensure proper indices, but they are a good idea for a general-purpose routine.

We use a separate client program, `SortArrayClient.py`, to test this new module and compare the performance of the different sorting methods. This program uses some other Python modules and features to help in this process and is shown in Listing 3-5.

**Listing 3-5** *The* `SortArrayClient.py` *program*

```
from SortArray import *
import random
import timeit

def initArray(size=100, maxValue=100, seed=3.14159):
    """Create an Array of the specified size with a fixed sequence of
       'random' elements"""
    arr = Array(size)                    # Create the Array object
    random.seed(seed)                    # Set random number generator
    for i in range(size):                # to known state, then loop
        arr.insert(random.randrange(maxValue)) # Insert random numbers
    return arr                           # Return the filled Array

arr = initArray()
print("Array containing", len(arr), "items:\n", arr)

for test in ['initArray().bubbleSort()',
             'initArray().selectionSort()',
             'initArray().insertionSort()']:
    elapsed = timeit.timeit(test, number=100, globals=globals())
    print(test, "took", elapsed, "seconds", flush=True)

arr.insertionSort()
print('Sorted array contains:\n', arr)
```

The `SortArrayClient.py` program starts by importing the `SortArray` module to test the sorting methods, and the `random` and `timeit` modules to assist. The `random` module provides pseudo-random number generators that can be used to make test data. The `timeit` module provides a convenient way to measure the execution times of Python code.

The test program first defines a function, `initArray()`, that generates an unsorted array for testing. Right after the parameter list, it has a documentation string that helps explain its purpose, creating arrays of a fixed size filled with a sequence of random numbers. Because we'd like to test the different sort operations on the exact same sequence of elements, we want `initArray()` to return a fresh copy of the same array every time it runs, but conforming to the parameters it is given. The function first creates an `Array` of the desired `size`. Next, it initializes the `random` module to a known state by setting the `seed`

value. This means that pseudo-random functions produce the same sequence of numbers when called in the same order.

Inside the following `for` loop, the `random.randrange(maxValue)` function generates integers in the range [0, `maxValue`) that are inserted into the array (the math notation *[a, b)* represents a range of numbers that includes *a* but excludes *b*). Then the filled array is returned.

The test program uses the `initArray()` function to generate an array that is stored in the `arr` variable. After printing the initial contents of `arr`, a `for` loop is used to step through the three sort operations to be timed. The loop variable, `test`, is bound to a string on each pass of the loop. The string has the Python expression whose execution time is to be measured. Inside the loop body, the `timeit.timeit()` function call measures the elapsed time of running the `test` expression. It runs the expression a designated `number` of times and returns the total elapsed time in seconds. Running it many times helps deal with the variations in running times due to other operations happening on the computer executing the Python interpreter. The last argument to `timeit.timeit()` is `globals`, which is needed so that the interpreter has all the definitions that have been loaded so far, including `SortArray` and `initArray`. The interpreter uses those definitions when evaluating the Python expression in the `test` variable.

The result of `timeit.timeit()` is stored in the `elapsed` variable and then printed out with `flush=True` so that it doesn't wait until the output buffer is full or input needs to be read before printing. After all three test times are printed, it performs one more sort on the original `arr` array and prints its contents. The results look like this:

```
$ python3 SortArrayClient.py
Array containing 100 items:
 [77, 94, 59, 85, 61, 46, 62, 17, 56, 37, 18, 45, 76, 21, 91, 7, 96,
50,
31, 69, 80, 69, 56, 60, 26, 25, 1, 2, 67, 46, 99, 57, 32, 26, 98, 51,
77,
34, 20, 81, 22, 40, 28, 23, 69, 39, 23, 6, 46, 1, 96, 51, 71, 61, 2,
34,
1, 55, 78, 91, 69, 23, 2, 8, 3, 78, 31, 25, 26, 73, 28, 88, 88, 38,
22,
97, 9, 18, 18, 66, 47, 16, 82, 9, 56, 45, 15, 76, 85, 52, 86, 5, 28,
67,
34, 20, 6, 33, 83, 68]
initArray().bubbleSort() took 0.25465869531035423 seconds
initArray().selectionSort() took 0.11350362841039896 seconds
```

```
initArray().insertionSort() took 0.12348053976893425 seconds
Sorted array contains:
 [1, 1, 1, 2, 2, 2, 3, 5, 6, 6, 7, 8, 9, 9, 15, 16, 17, 18, 18, 18,
20,
20, 21, 22, 22, 23, 23, 23, 25, 25, 26, 26, 26, 28, 28, 28, 31, 31,
32,
33, 34, 34, 34, 37, 38, 39, 40, 45, 45, 46, 46, 46, 47, 50, 51, 51,
52,
55, 56, 56, 56, 57, 59, 60, 61, 61, 62, 66, 67, 67, 68, 69, 69, 69,
69,
71, 73, 76, 76, 77, 77, 78, 78, 80, 81, 82, 83, 85, 85, 86, 88, 88,
91,
91, 94, 96, 96, 97, 98, 99]
```

Looking at the output, you can see that the `randrange()` function provided a broad variety of values for the array in a random order. The results of the timing of the sort tests show the bubble sort taking at least twice as much time as the selection and insertion sorts do. The final sorted version of the array confirms that sorting works and shows that there are many duplicate elements in the array.

# Stability

Sometimes it matters what happens to data items that have equal keys when sorting. The `SortArrayClient.py` test stored only integers in the array, and equal integers are pretty much indistinguishable. If the array contained complex records, however, it could be very important how records with the same key are sorted. For example, you may have employee records arranged alphabetically by family names. (That is, the family names were used as key values in the sort.) Let's say you want to sort the data by postal code too, but you want all the items with the same postal code to continue to be sorted by family names. This is called a **secondary sort key**. You want the sorting algorithm to shift only what needs to be sorted by the current key and leave everything else in its order after previous sorts using other keys. Some sorting algorithms retain this secondary ordering; they're said to be **stable**. Stable sorting methods also minimize the number of swaps or copy operations.

Some of the algorithms in this chapter are stable. We have included an exercise at the end of the chapter for you to decide which ones are. The answer is not obvious from the output of their test programs, especially on simple integers. You need to review the algorithms to see that swaps and copies only happen on

items that need to be moved to be in the right position for the final order. They should also move items with equal keys the minimum number of array cells necessary so that they remain in the same relative order in the final arrangement.

## Comparing the Simple Sorts

There's probably no point in using the bubble sort, unless you don't have your algorithm book handy. The bubble sort is so simple that you can write it from memory. Even so, it's practical only if the amount of data is small. (For a discussion of what "small" means and what trade-offs such decisions entail, see Chapter 16, "What to Use and Why.")

Table 3-1 summarizes the time efficiency of each of the sorting methods. We looked at what the algorithms would do when presented with elements whose keys are in random order to determine what they would do in the average case. We also considered what would happen in the worst case when the keys were in some order that would maximize the number of operations. The table includes both the Big O notation followed by the detailed value in square brackets [ ]. The Big O notation is what matters most; it gives the broad classification of the algorithms and tells what will happen for very large N. Typically, we are most interested in the average case behavior because we cannot anticipate what data they will encounter, but sometimes the worst case is the driving factor. For example, if the sort must be able to complete within a specific amount of time for a particular size of data, we can use the best of the worst-case values to choose the algorithm that will guarantee that completion time.

**Table 3-1** *Time Order of the Sort Methods*

| Algorithm | Comparisons | | Swaps or Copies | |
|---|---|---|---|---|
| | Average case | Worst case | Average case | Worst case |
| *Bubble sort* | $O(N^2)$ $[N^2/2]$ | $O(N^2)$ $[N^2/2]$ | $O(N^2)$ $[N^2/4]$ | $O(N^2)$ $[N^2/2]$ |
| *Selection sort* | $O(N^2)$ $[N^2/2]$ | $O(N^2)$ $[N^2/2]$ | $O(N)$ $[N-1]$ | $O(N)$ $[N-1]$ |
| *Insertion sort* | $O(N^2)$ $[N^2/4]$ | $O(N^2)$ $[N^2/2]$ | $O(N^2)$ $[N^2/4]$ | $O(N^2)$ $[N^2/2]$ |

The detailed values shown in the table can be somewhat useful in narrow circumstances. For example, the relative number of comparisons in the average case between the bubble sort and the insertion sort corresponds well with the relative time measurements in the test program. That would also predict that the selection sort average case should be closer to the bubble sort, but the one measurement above doesn't agree with that. Perhaps that means that the swaps or copies were a bigger factor on running time with N around 100 on the test platform.

Overall, the selection sort minimizes the number of swaps, but the number of comparisons is still high. This sort might be useful when the amount of data is small and swapping data items is very time-consuming relative to comparing them. In real-world scenarios, there can be situations where keeping the swaps to a minimum is more important than the number of comparisons. For example, in moving patients between hospital rooms (in response to some other disaster), it could be critical to displace as few patients as possible.

The insertion sort is the most versatile of the three sorting methods and is the best bet in most situations, assuming the amount of data is small, or the data is almost sorted. For larger amounts of data, quicksort and Timsort are generally considered the fastest approach; we examine them in Chapter 7.

We've compared the sorting algorithms in terms of speed. Another consideration for any algorithm is how much memory space it needs. All three of the algorithms in this chapter carry out their sort **in place**, meaning that they use the input array as the output array and very little extra memory is required. All the sorts require an extra variable to store an item temporarily while it's being swapped or marked. Because that item can be quite large, you need to consider it, but the local variables for indices into the array are typically ignored.

# Summary

- The sorting algorithms in this chapter all assume an array as a data storage structure.
- Sorting involves comparing the keys of data items in the array and moving the items (usually references to the items) around until they're in sorted order.

- All the algorithms in this chapter execute in $O(N^2)$ time. Nevertheless, some can be substantially faster than others.
- An invariant is a condition that remains unchanged while an algorithm runs.
- The bubble sort is the least efficient but the simplest sort.
- The insertion sort is the most commonly used of the $O(N^2)$ sorts described in this chapter.
- The selection sort performs $O(N^2)$ comparisons and only O(N) swaps, which can be important when swap time is much more significant than comparison time.
- A sort is stable if the order of elements with the same key is retained.
- None of the sorts in this chapter require more than a single temporary record variable, in addition to the original array.

# Questions

These questions are intended as a self-test for readers. Answers may be found in Appendix C.

1. Computer sorting algorithms are more limited than humans in that
   a. the amount of data that computers can sort is much less than what humans can.
   b. humans can invent new sorting algorithms, whereas computers cannot.
   c. humans know what to sort, whereas computers need to be told.
   d. computers can compare only two things at a time, whereas humans can compare small groups.
2. The two basic operations in simple sorting are _____ items and _____ them (or sometimes _____ them).
3. True or False: The bubble sort always ends up comparing every possible pair of items in the initial array.

4. The bubble sort algorithm alternates between

   a. comparing and swapping.

   b. moving and copying.

   c. moving and comparing.

   d. copying and comparing.

5. True or False: If there are N items, the bubble sort makes exactly N×N comparisons.

6. In the selection sort,

   a. the largest keys accumulate on the left (low indices).

   b. a minimum key is repeatedly discovered.

   c. a number of items must be shifted to insert each item in its correctly sorted position.

   d. the sorted items accumulate on the right.

7. True or False: If, on a particular computing platform, swaps take much longer than comparisons, the selection sort is about twice as fast as the bubble sort for all values of N.

8. Ignoring the details of where the computer stores each piece of data, what is a reasonable assumption about the ratio of the amounts of time taken for a copy operation versus a swap operation?

9. What is the invariant in the selection sort?

10. In the insertion sort, the “marked player” described in the text corresponds to which variable in the `insertionSort()` method?

    a. `inner`

    b. `outer`

    c. `temp`

    d. `__a[outer]`

11. In the insertion sort, the “partially sorted” group members are

    a. the items that are already sorted but still need to be moved as a block.

    b. the items that are in their final block position but may still need to be sorted.

c. only partially sorted in order by their keys.

d. the items that are sorted among themselves, but items outside the group may need to be inserted in the group.

12. Shifting a group of items left or right requires repeated _____.

13. In the insertion sort, after an item is inserted in the partially sorted group, it

    a. is never moved again.

    b. is never shifted to the left.

    c. is often moved out of this group.

    d. finds that its group is steadily shrinking.

14. The invariant in the insertion sort is that _____.

15. Stability might refer to

    a. items with secondary keys being excluded from a sort.

    b. keeping cities sorted by increasing population within each state, in a sort by state.

    c. keeping the same given names matched with the same family names.

    d. items keeping the same order of secondary keys without regard to primary keys.

# Experiments

Carrying out these experiments will help to provide insights into the topics covered in the chapter. No programming is involved.

3-A The "Stability" section explains that stable sorting algorithms don't change the *relative* position within the array of items having equal valued keys. For example, when you're sorting an array containing tuples by their first element such as

(elm, 1), (asp, 1), (elm, 2), (oak, 1)

the algorithm always produces

(asp, 1), (elm, 1), (elm, 2), (oak, 1)

and never

(asp, 1), (elm, 2), (elm, 1), (oak, 1)

Review each of the simple sorting algorithms in this chapter and determine if they are always stable.

Determining whether a sorting algorithm is stable or not can be difficult. Try walking through some sample inputs that include some equal-valued keys to see when those items are moved. If you can think of even one example where two items with equal keys are reordered from their original relative order, then you have proven that the algorithm is unstable. If you can't think of an example where equal valued keys are moved out of order, then the algorithm *might* be stable. To be sure, you need to provide more proof. That's usually done by finding invariant conditions about a partial sequence of items as the algorithm progresses. For example, if you can show that the output array after index `i` contains only items in stable order, that condition stays valid at iteration $j$ if it held in iteration $j - 1$, and that `i` always ends at index 0, then the algorithm must be stable.

Note that if you get stuck, we provide an answer for this exercise in Appendix C.

3-B Sometimes the items in an array might have just a few distinct keys with many copies. That creates a situation where there are many duplicates of the same key. Use the Simple Sorting Visualization tool to create an array with 15 cells and filled with only two distinct values, say 10 and 90. Try shuffling and sorting that kind of array with the three sorting algorithms. Do any of them show advantages or disadvantages over the others for this kind of data?

3-C In the `selectionSort()` method shown in Listing 3-2, the inner loop makes a swap on every pass. It doesn't need to swap them if the value of the `min` and `outer` indices are the same. Would it make sense to add an additional comparison of those indices and perform the swap only if they are different? If so, under what conditions would that improve the performance? If not, why not?

# Programming Projects

Writing programs to solve the Programming Projects helps to solidify your understanding of the material and demonstrates how the chapter's concepts are applied. (As noted in the Introduction, qualified instructors may obtain completed solutions to the Programming Projects on the publisher's website.)

3.1 In the `bubbleSort()` method (Listing 3-1) and the Visualization tool, the `inner` index always goes from left to right, finding the largest item and carrying it out on the right. Modify the `bubbleSort()` method so that it's bidirectional. This means the `in` index will first carry the largest item from left to right as before, but when it reaches `last`, it will reverse and carry the smallest item from right to left. You need two outer indexes, one on the right (the old `last`) and another on the left.

3.2 Add a method called `median()` to the `Array` class in the `SortArray.py` module (Listing 3-4). This method should return the median value in the array. (Recall that in a group of numbers, half are larger than the median and half are smaller.) Do it the easy way.

3.3 Add a method called `deduplicate()` to the `Array` class in the `SortArray.py` module (Listing 3-4) that removes duplicates from a previously sorted array without disrupting the order. You can use any of the sort methods in your test program to sort the data. You can imagine schemes in which all the items from the place where a duplicate was discovered to the end of the array would be shifted down one space every time a duplicate was discovered, but this would lead to slow $O(N^2)$ time, at least when there were a lot of duplicates. In your algorithm, make sure no item is moved more than once, no matter how many duplicates there are. This will give you an algorithm with O(N) time.

3.4 Another simple sort is the odd-even sort. The idea is to repeatedly make two passes through the array. On the first pass, you look at all the pairs of items, a[j] and a[j+1], where j is odd (j = 1, 3, 5, …). If their key values are out of order, you swap them. On the second pass, you do the same for all the even values (j = 0, 2, 4, …). You do these two passes repeatedly until the array is sorted. Add an `oddEvenSort()` method to the `Array` class in the `SortArray.py` module (Listing 3-4). Perform the outer loop until no swaps occur to see how many passes are needed; one pass includes both odd-pair and even-pair swapping. Make sure it works for varying amounts of data and on good and bad initial orderings. After testing how many passes are needed before no more swaps occur,

determine the maximum number of passes of the outer loop based on the length of the input array.

The odd-even sort is actually useful in a multiprocessing environment, where a separate processor can operate on each odd pair simultaneously and then on each even pair. Because the odd pairs are independent of each other, each pair can be checked—and swapped, if necessary—by a different processor. This makes for a very fast sort.

3.5 Modify the `insertionSort()` method in `SortArray.py` (Listing 3-4) so that it counts the number of copies and the number of item comparisons it makes during a sort and displays the totals. You need to look at the loop condition in the inner `while` loop and carefully count item comparisons. Use this program to measure the number of copies and comparisons for different amounts of inversely sorted data. Do the results verify $O(N^2)$ efficiency? Do the same for almost-sorted data (only a few items out of place). What can you deduce about the efficiency of this algorithm for almost-sorted data?

3.6 Here's an interesting way to remove items with duplicate keys from an array. The insertion sort uses a loop-within-a-loop algorithm that compares every item in the array with the partially sorted items so far. One way to remove items with duplicate keys would be to modify the algorithm in the `Array` class in the `SortArray.py` module (Listing 3-4) so that it removes the duplicates as it sorts. Here's one approach: When a duplicate key is found, instead of copying the duplicate back into the array cell at its sorted position, change the key for the marked item to be a special value that is treated as lower than any other possible key. With that low key value, it is automatically moved into place at the beginning of the array. By keeping track of how many duplicates are found, you know the end of duplicates and beginning of the remaining elements in the array. When the outer loop ends, the algorithm would have to make one more pass to shift the unique keys into the cells occupied by the duplicates. Write an `insertionSortAndDedupe()` method that performs this operation. Make sure to test that it works with all different kinds of input array data.

# 4. Stacks and Queues

### In This Chapter



This chapter examines three data storage structures widely used in all kinds of applications: the stack, the queue, and the priority queue. We describe how these structures differ from arrays, examining each one in turn. In the last section, we look at an operation in which the stack plays a significant role: parsing arithmetic expressions.

## Different Structures for Different Use Cases

You choose data structures to use in programs based on their suitability for particular tasks. The structures in this chapter differ from what you've already seen in previous chapters in several ways. Those differences guide the decision on when to apply them to a new task.

## Storage and Retrieval Pattern

Arrays—the data storage structure examined thus far—as well as many other structures you encounter later in this book (linked lists, trees, and so on) are appropriate for all kinds of data storage. They are sometimes referred to as low-level data structures because they are used in implementing many other data structures. The organization of the data dictates what it takes to retrieve a

particular record or object that is stored inside them. Arrays are especially good for applications where the items can be indexed by an integer because the retrieval is fast and independent of the number of items stored. If the index isn't known, the array still can be searched to find the object. As you've seen, sorting the objects made the search faster at the expense of other operations. Insertion and deletion operations can either be constant time or have the same Big O behavior as the search operation.

The structures and algorithms we examine in this chapter, on the other hand, are used in situations where you are unlikely to need to search for a particular object or item that is stored but rather where you want to process those objects in a particular order.

## Restricted Access

In an array, any item can be accessed, either immediately—if its index number is known—or by searching through a sequence of cells until it's found. In the data structures in this chapter, however, access is restricted: only one item can be read or removed at a time.

The interface of these structures is designed to enforce this restricted access. Access to other items is (in theory) not allowed.

## More Abstract

Stacks, queues, and priority queues are more abstract structures than arrays and many other data storage structures. They're defined primarily by their interface—the permissible operations that can be carried out on them. The underlying mechanism used to implement them is typically not visible to their user.

The underlying mechanism for a stack, for example, can be an array, as shown in this chapter, or it can be a linked list. The underlying mechanism for a priority queue can be an array or a special kind of tree called a **heap**. When one data structure is used to implement a more abstract one, we often say that it is a *lower-level data structure* by picturing the more abstract structure being layered on top of it. We return to the topic of one data structure being implemented by another when we discuss abstract data types (ADTs) in Chapter 5, "Linked Lists."

# Stacks

A **stack** data structure allows access to only one data item in the collection: the last item inserted. If you remove this item, you can access the next-to-last item inserted, and so on. Although that constraint seems to be quite limiting, this behavior occurs in many programming situations. In this section we show how a stack can be used to check whether parentheses, braces, and brackets are balanced in a computer program source file. At the end of this chapter, a stack plays a vital role in parsing (analyzing) arithmetic expressions such as 3×(4+5).

A stack is also a handy aid for algorithms applied to certain complex data structures. In Chapter 8, "Binary Trees," you will see it used to help traverse the nodes of a tree. In Chapter 14, "Graphs," you will apply it to searching the vertices of a graph (a technique that can be used to find your way out of a maze).

Nearly all computers use a stack-based architecture. When a function or method is called, the return address for where to resume execution and the function arguments are **pushed** (inserted) onto a stack in a particular order. When the function returns, they're **popped** off. The stack operations are often accelerated by the computer hardware.

The idea of having all the function arguments pushed on a stack makes very clear what the operands are for a particular function or operator. This scheme is used in some older pocket calculators and formal languages such as PostScript and PDF content streams. Instead of entering arithmetic expressions using parentheses to group operands, you insert (or push) those values onto a stack. The operator comes after the operands and pops them off, leaving the (intermediate) result on the stack. You will learn more about this approach when we discuss parsing arithmetic expressions in the last section in this chapter.

# The Postal Analogy

To understand the idea of a stack, consider an analogy provided by postal, and to a lesser extent, some email and messaging systems, especially ones where you can see only one message at a time. Many people, when they get their mail, toss it onto a stack on a table. If they've been away awhile, the stack might be quite large. If they don't open the mail, the following day, they may add more messages on top of the stack. In messaging systems, a large number

of messages can accumulate since the last time people viewed them. Then, when they have a spare moment, they start to process the accumulated mail or messages from the top down, one at a time. First, they open the letter on the top of the stack and take appropriate action—paying the bill, reading the holiday newsletter, considering an advertisement, throwing it away, or whatever. After disposing of the first message, they examine the next one down, which is now the top of the stack, and deal with that. The idea is to start with the most recently delivered letter or message first. Eventually, they work their way down to the message on the bottom of the stack (which is now the topmost one). Figure 4-1 shows a stack of mail.

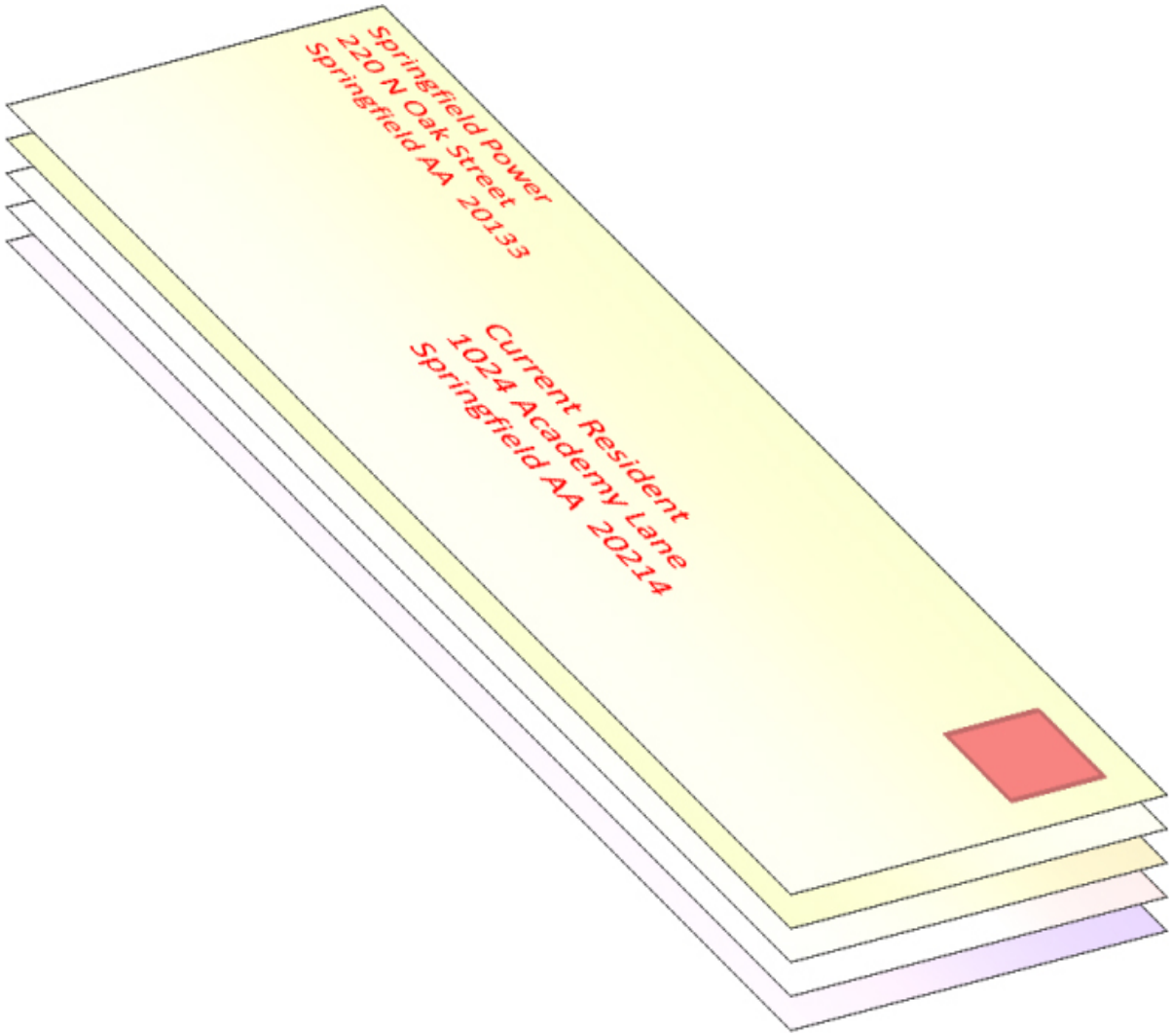



**Figure 4-1** *A stack of letters*

This “do the top one first” approach works all right as long as people can easily process all the items in a reasonable time. If they can’t, there’s the danger that messages on the bottom of the stack won’t be examined for months, and the bills or other important requests for action are not handled soon enough. By reviewing the most recent one first, however, they’re at least guaranteed to find any updates to bills or messages that occurred after the earlier deliveries.

Of course, many people don’t rigorously follow this top-to-bottom approach. They may, for example, take the mail off the bottom of the stack, to process the oldest letter first. That way, they see the messages in chronological order of arrival. Or they might shuffle through the messages before they begin processing them and put higher-priority messages on top. In these cases, their processing system is no longer a stack in the computer-science sense of the word. If they take mail off the bottom, it’s a **queue**; and if they prioritize it, it’s a **priority queue**. We’ll look at these possibilities later.

The tasks you perform every day can be seen as a stack. You typically have a set of goals, often with different priorities. They might include things like these: be happy, be with my family, or live a long life. Each of those goals can have sub-goals. For instance, to be happy, you may have a sub-goal: to be financially stable. A sub-goal of that could be: to have a job that pays enough. The tasks that you perform each day are organized around these goals. So, if “be with my family” is a main goal, you go to work to satisfy your other goal of being financially stable. When work is done, you return to your main goal by returning home. While at work, you might start the day by working on a long-term project. When the manager asks you to handle some new task that just popped up, you set aside the project and work the pop-up task. If a more critical problem comes up, you set aside the pop-up task and solve the problem. When each higher-priority activity is finished, you return to the next lower-priority activity on your *stack* of tasks.

In data structures, placing a data item on the top of the stack is called *pushing* it. Removing it from the top of the stack is called *popping* it. Looking at the top item without popping it is called **peeking**. These are the primary stack operations. A stack is said to be a last-in, first-out (LIFO) storage mechanism because the last item inserted is the first one to be removed.

# The Stack Visualization Tool

Let's use the Stack Visualization tool to get an idea how stacks work. When you launch this tool by following the guide in Appendix A, "Running the Visualizations," you see an empty stack with operations for Push, New, Pop, Peek, and the animation controls shown in Figure 4-2.


**Figure 4-2** *The Stack Visualization tool*

The Stack Visualization tool is based on an array, so you see an array of data items, this time arranged in a column. Although it's based on an array, a stack restricts access, so you can't access elements using an index (even though they appear next to the cells in the visualization). In fact, the concept of a stack and the underlying data structure used to implement it are quite separate. As we noted earlier, stacks can also be implemented by other kinds of storage structures, such as linked lists.

## The Push Button

To insert a data item on the stack, use the button labeled Push. The item can be any text string, and you simply type it in the text entry box next to Push. The tool limits the number of characters in the string, so they fit in the array cells.

To the left of the stack is a reddish-brown arrow labeled "`__top`." This is part of the data structure, an attribute that tracks the last item inserted. Initially, nothing has been inserted, so it points below the first array cell, the one labeled with a small 0. When you push an item, it moves up to cell 0, and the item is inserted in that cell. Notice that the `top` arrow is incremented before the item is inserted. That's important because copying the data before incrementing could either cause an error by writing outside of the array bounds or overwriting the last element in the array when there are multiple items.

Try pushing several items on the stack. The tool notes the last operation you chose when typing an argument, so if you press Return or Enter while typing the next one, it will run the push operation again. If the stack is full and you try to push another item, you'll get the `Error! Stack is already full` message. (Theoretically, an ADT stack doesn't become full, but any practical implementation does.)

## The New Button

As with all data structures, you create (and remove) stacks as needed for an algorithm. Because this implementation is based on an array, it needs to know how big the array should be. The Stack Visualization tool starts off with eight cells allocated and none of them filled. To make a new, empty stack, type the number of cells it should have in the text entry box and then select New. The tool adjusts the height of the cells to fit them within the display window. If you ask for a size that won't fit, you'll get an error message.

## The Pop Button

To remove a data item from the top of the stack, use the Pop button. The value popped is moved to the right as it is being copied to a variable named `top`. It is stored there temporarily as the `pop()` method finishes its work.

Again, notice the steps involved. First, the item is copied from the cell pointed to by `__top`. Then the cell is cleared, and then `__top` is decremented to point to the highest occupied cell. The order of copying and changing the index pointer is the reverse of the sequence used in the push operation.

The tool's Pop operation shows the item actually being removed from the array and the cell becoming empty. This is not strictly necessary because the value could be left in the array and simply ignored. In programming languages like Python, however, references to items left in an array like this can consume memory. It's important to remove those references so that the memory can be reclaimed and reused. We show this in detail when looking at the code. Note that you still have a reference to the item in the `top` variable so that it won't be lost.

After the topmost item is removed, the `__top` index decrements to the next lower cell. After you pop the last item off the stack, it points to −1, below the lowest cell. This position indicates that the stack is empty. If the stack is empty and you try to pop an item, you'll get the `Error! Stack is empty` message.

### The Peek Button

Push and pop are the two primary stack operations. It's sometimes useful, however, to be able to read the value from the top of the stack without removing it. The peek operation does this. By selecting the Peek button a few times, you see the value of the item at the `__top` index copied to the output box on the right, but the item is not removed from the stack, which remains unchanged.

Notice that you can peek only at the top item. By design, all the other items are invisible to the stack user. If the stack is empty in the visualization tool when you select Peek, the output box is not created. We show how to implement the empty test later in the code.

### Stack Size

Stacks come in all sizes and are typically capped in size so that they can be allocated in a single block of memory. The application allocates some initial size based on the maximum number of items expected to be stacked. If the application tries to push items beyond what the stack can hold, then either the stack needs to increase in size, or an exception must occur. The visualization tool limits the size to what fits conveniently on the screen, but stacks in many applications can have thousands or millions of cells.

## Python Code for a Stack

The Python `list` type is a natural choice for implementing stacks. As you saw in Chapter 2, "Arrays," the `list` type can act like an array (as opposed to a linked list, as shown in Chapter 5, "Linked Lists"). Instead of using Python slicing to get the last element in the list, we continue to use the list as a basic array and keep a separate `__top` pointer for the topmost item, as shown in Listing 4-1.

**Listing 4-1** *The `SimpleStack.py` Module*

```
# Implement a Stack data structure using a Python list

class Stack(object):
   def __init__(self, max):            # Constructor
```

```
        self.__stackList = [None] * max  # The stack stored as a list
        self.__top = -1                  # No items initially

    def push(self, item):                # Insert item at top of stack
        self.__top += 1                  # Advance the pointer
        self.__stackList[self.__top] = item  # Store item

    def pop(self):                       # Remove top item from stack
        top = self.__stackList[self.__top]   # Top item
        self.__stackList[self.__top] = None  # Remove item reference
        self.__top -= 1                  # Decrease the pointer
        return top                       # Return top item

    def peek(self):                      # Return top item
        if not self.isEmpty():           # If stack is not empty
            return self.__stackList[self.__top] # Return the top item

    def isEmpty(self):                   # Check if stack is empty
        return self.__top < 0

    def isFull(self):                    # Check if stack is full
        return self.__top >= len(self.__stackList) - 1

    def __len__(self):                   # Return # of items on stack
        return self.__top + 1

    def __str__(self):                    # Convert stack to string
        ans = "["                         # Start with left bracket
        for i in range(self.__top + 1):   # Loop through current items
            if len(ans) > 1:              # Except next to left bracket,
                ans += ", "               # separate items with comma
            ans += str(self.__stackList[i]) # Add string form of item
        ans += "]"                        # Close with right bracket
        return ans
```

---

The `SimpleStack.py` implementation has just the basic features needed for a stack. Like the `Array` class you saw in Chapter 2, the constructor allocates an array of known size, called `__stackList`, to hold the items with the `__top` pointer as an index to the topmost item in the stack. Unlike the `Array` class, `__top` points to the topmost item and not the next array cell to be filled. Instead of `insert()`, it has a `push()` method that puts a new item on top of the stack. The `pop()` method returns the top item on the stack, clears the array cell that held it, and decreases the stack size. The `peek()` method returns the top item without decreasing the stack size.

In this simple implementation, there is very little error checking. It does include `isEmpty()` and `isFull()` methods that return Boolean values indicating whether the stack has no items or is at capacity. The `peek()` method checks for an empty stack and returns the top value only if there is one. To avoid errors, a client program would need to use `isEmpty()` before calling `pop()`. The class also includes methods to measure the stack depth and a string conversion method for convenience in displaying stack contents.

To exercise this class, you can use the `SimpleStackClient.py` program shown in Listing 4-2.

**Listing 4-2** *The `SimpleStackClient.py` Program*

```
from SimpleStack import *

stack = Stack(10)

for word in ['May', 'the', 'force', 'be', 'with', 'you']:
   stack.push(word)

print('After pushing', len(stack),
      'words on the stack, it contains:\n', stack)
print('Is stack full?', stack.isFull())

print('Popping items off the stack:')
while not stack.isEmpty():
   print(stack.pop(), end=' ')
print()
```

The client creates a small stack, pushes some strings onto the stack, displays the contents, then pops them off, printing them left to right separated by spaces. The transcript of running the program shows the following:

```
$ python3 SimpleStackClient.py
After pushing 6 words on the stack, it contains:
 [May, the, force, be, with, you]
Is stack full? False
Popping items off the stack:
you with be force the May
```

Notice how the program reverses the order of the data items. Because the last item pushed is the first one popped, the `you` appears first in the output. Figure

4-3 shows how the data gets placed in the array cells of the stack and then returned for push and pop operations. The array cells shown as empty in the illustration still have the value `None` in them, but because only the values from the bottom up through the `__top` pointer are occupied, the ones beyond `__top` can be considered unfilled. The visualization tool uses the same `pop()` method as in Listing 4-1, setting the popped cell to `None`.

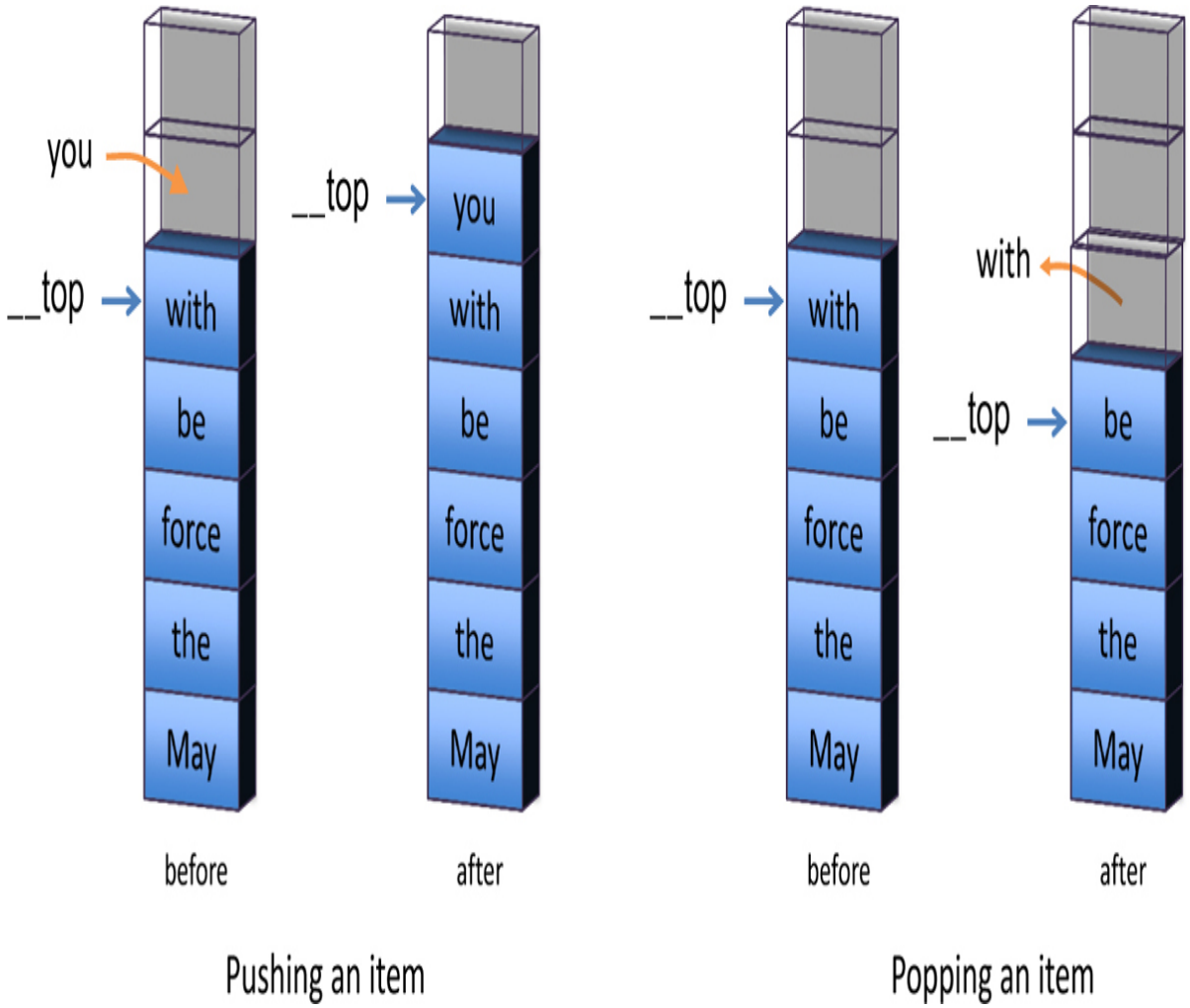



**Figure 4-3** *Operation of the* `Stack` *class* `push()` *and* `pop()` *methods*

## Error Handling

There are different philosophies about how to handle stack errors. What should happen if you try to push an item onto a stack that's already full or pop an item from a stack that's empty?

In this implementation, the responsibility for avoiding or handling such errors has been left up to the program using the class. That program should always check to be sure the stack is not full before inserting an item, as in

```
if not stack.isFull():
   stack.push(item)
else:
   print("Can't insert, stack is full")
```

The client program in Listing 4-2 checks for an empty stack before it calls `pop()`. It does not, however, check for a full stack before calling `push()`. Alternatively, many stack classes check for these conditions internally, in the `push()` and `pop()` methods. Typically, a stack class that discovers such conditions either throws an exception, which can then be caught and processed by the program using the class, or takes some predefined action, such as returning `None`. By providing both the ability for users to query the internal state and the possibility of raising exceptions when constraints are violated, the data structure enables both **proactive** and **reactive** approaches.

# Stack Example 1: Reversing a Word

For this first example of using a stack, we examine a simple task: reversing a word. When you run the program, you type in a word or phrase, and the program displays the reversed version.

A stack is used to reverse the letters. First, the characters are extracted one by one from the input string and pushed onto the stack. Then they're popped off the stack and displayed. Because of its last-in, first-out characteristic, the stack reverses the order of the characters. Listing 4-3 shows the code for the `ReverseWord.py` program.

**Listing 4-3** *The `ReverseWord.py` Program*

```
# A program to reverse the letters of a word

from SimpleStack import *

stack = Stack(100)          # Create a stack to hold letters

word = input("Word to reverse: ")
```

```
for letter in word:          # Loop over letters in word
   if not stack.isFull():    # Push letters on stack if not full
      stack.push(letter)

reverse = ''                  # Build the reversed version
while not stack.isEmpty():    # by popping the stack until empty
   reverse += stack.pop()

print('The reverse of', word, 'is', reverse)
```

The program makes use of Python's `input()` function, which prints a prompt string and then waits for a user to type a response string. The program constructs a stack instance, gets the word to be reversed from the user, loops over the letters in the word, and pushes them on the stack. Here the program avoids pushing letters if the stack is already full. After all the letters are pushed on the stack, the program creates the reversed version, starting with an empty string and appending each character popped from the stack. The results look like this:

```
$ python3 ReverseWord.py
Word to reverse: draw
The reverse of draw is ward
$ python3 ReverseWord.py
Word to reverse: racecar
The reverse of racecar is racecar
$ python3 ReverseWord.py
Word to reverse: bolton
The reverse of bolton is notlob
$ python3 ReverseWord.py
Word to reverse: A man a plan a canal Panama
The reverse of A man a plan a canal Panama is amanaP lanac a nalp a
nam A
```

The results show that single-word palindromes come out the same as the input. If the input "word" has spaces, as in the last example, the spaces are treated just as any other letter in the string.

## Stack Example 2: Delimiter Matching

One common use for stacks is to parse certain kinds of text strings. Typically, the strings are lines of code in a computer language, and the programs parsing them are compilers or interpreters.

To give the flavor of what's involved, let's look at a program that checks the delimiters in an expression. The text expression doesn't need to be a line of real code (although it could be), but it should use delimiters the same way most programming languages do. The delimiters are the curly braces { and }, square brackets [ and ], and parentheses ( and ). Each opening, or left, delimiter should be matched by a closing, or right, delimiter; that is, every { should be followed by a matching } and so on. Also, opening delimiters that occur later in the string should be closed before those occurring earlier. Here are some examples:

```
c[d]        # correct
a{b[c]d}e   # correct
a{b(c]d}e   # not correct; ] doesn't match (
a[b{c}d]e}  # not correct; nothing matches final }
a{b(c)      # not correct; nothing matches opening {
```

## Opening Delimiters on the Stack

This delimiter-matching program works by reading characters from the string one at a time and placing opening delimiters when it finds them, on a stack. When it reads a closing delimiter from the input, it pops the opening delimiter from the top of the stack and attempts to match it with the closing delimiter. If they're not the same type (there's an opening brace but a closing parenthesis, for example), an error occurs. Also, if there is no opening delimiter on the stack to match a closing one, or if a delimiter has not been matched, an error occurs. It discovers unmatched delimiters because they remain on the stack after all the characters in the string have been read.

Let's see what happens on the stack for a typical correct string:

```
a{b(c[d]e)f}
```

Table 4-1 shows how the stack looks as each character is read from this string. The entries in the second column show the stack contents, reading from the bottom of the stack on the left to the top on the right.

**Table 4-1** *Stack Contents in Delimiter Matching*

| Character Read | Stack Contents |
|---|---|
| a | |
| { | { |
| b | { |
| ( | { ( |
| c | { ( |
| [ | { ( [ |
| d | { ( [ |
| ] | { ( |
| e | { ( |
| ) | { |
| f | { |
| } | |

As the string is read, each opening delimiter is placed on the stack. Each closing delimiter read from the input is matched with the opening delimiter popped from the top of the stack. If they form a pair, all is well. Nondelimiter characters are not inserted on the stack; they're ignored.

This approach works because pairs of delimiters that are opened last should be closed first. This matches the last-in, first-out property of the stack.

## Python Code for DelimiterChecker.py

Listing 4-4 shows the code for the parsing program, `DelimiterChecker.py`. Like `ReverseWord.py`, the program accepts an expression after prompting the user using `input()` and prints out any errors found or a message saying the delimiters are balanced.

**Listing 4-4** *The* `DelimiterChecker.py` *Program*

```
# A program to check that delimiters are balanced in an expression

from SimpleStack import *

stack = Stack(100)          # Create a stack to hold delimiter tokens

expr = input("Expression to check: ")

errors = 0                  # Assume no errors in expression

for pos, letter in enumerate(expr): # Loop over letters in expression
   if letter in "{[(":      # Look for starting delimiter
      if stack.isFull():    # A full stack means we can't continue
         raise Exception('Stack overflow on expression')
      else:
         stack.push(letter) # Put left delimiter on stack

   elif letter in "}])":    # Look for closing delimiter
      if stack.isEmpty():   # If stack is empty, no left delimiter
         print("Error:", letter, "at position", pos,
               "has no matching left delimiter")
         errors += 1
      else:
         left = stack.pop() # Get left delimiter from stack
         if not (left == "{" and letter == "}" or  # Do delimiters
                 left == "[" and letter == "]" or  # match?
                 left == "(" and letter == ")"):
            print("Error:", letter, "at position", pos,
                  "does not match left delimiter", left)
            errors += 1

# After going through entire expression, check if stack empty
if stack.isEmpty() and errors == 0:
   print("Delimiters balance in expression", expr)

elif not stack.isEmpty(): # Any delimiters on stack weren't matched
   print("Expression missing right delimiters for", stack)
```

The program doesn't define any functions; it processes one expression from the user and exits. It creates a `Stack` object to hold the delimiters as they are found. The `errors` variable is used to track the number of errors found in parsing the expression. It loops over the characters in the expression using Python's `enumerate()` sequencer, which gets both the index and the value of the string at that index. As it finds starting (or left) delimiters, it pushes them on the

stack, avoiding overflowing the stack. When it finds ending (or closing or right) delimiters, it checks whether there is a matching delimiter on the top of the stack. If not, it prints the error and continues. Some sample outputs are

```
$ python3 DelimiterChecker.py
Expression to check: a()
Delimiters balance in expression a()
$ python3 DelimiterChecker.py
Expression to check: a( b[4]d )
Delimiters balance in expression a( b[4]d )
$ python3 DelimiterChecker.py
Expression to check: a( b]4[d )
Error: ] at position 4 does not match left delimiter (
Error: ) at position 9 does not match left delimiter [
$ python3 DelimiterChecker.py
Expression to check: {{a( b]4[d )
Error: ] at position 6 does not match left delimiter (
Error: ) at position 11 does not match left delimiter [
Expression missing right delimiters for [{, {]
```

The messages printed by `DelimiterChecker.py` show the position and value of significant characters (delimiters) in the expression. After processing all the characters, any remaining delimiters on the stack are unmatched. The program prints an error for that using the Stack's string conversion method, which surrounds the list in square brackets, possibly leading to some confusion with the input string delimiters.

### The Stack as a Conceptual Aid

Notice how convenient the stack is in the `DelimiterChecker.py` program. You could have set up an array to do what the stack does, but you would have had to worry about keeping track of an index to the most recently added character, as well as other bookkeeping tasks. The stack is conceptually easier to use. By providing limited access to its contents, using the `push()` and `pop()` methods, the stack has made the delimiter-checking program easier to understand and less error prone. (As carpenters will tell you, it's safer and faster to use the right tool for the job.)

## Efficiency of Stacks

Items can be both pushed and popped from the stack implemented in the `Stack` class in constant O(1) time. That is, the time is not dependent on how many

items are in the stack and is therefore very quick. No comparisons or moves within the stack are necessary.

# Queues

In computer science, a **queue** is a data structure that is somewhat like a stack, except that in a queue the first item inserted is the first to be removed (first-in, first-out, or FIFO). In stacks, as you've seen, the last item inserted is the first to be removed (LIFO). A queue models the way people wait for something, such as tickets being sold at a window or a chance to greet the bride and groom at a big wedding. The first person to arrive goes first, the next goes second, and so forth. Americans call it a waiting line, whereas the British call it a queue. The key aspect is that the first items to arrive are the first to be processed.

Queues are used as a programmer's tool just like stacks are. They are found everywhere in computer systems: the jobs waiting to run, the messages to be passed over a network, the sequence of characters waiting to be printed on a terminal. They're used to model real-world situations such as people waiting in line for tickets, airplanes waiting to take off, or students waiting to see whether they get into a particular course. This ordering is sometimes called **arrival ordering** because the time of arrival in the queue determines the order.

Various queues are quietly doing their job in your computer's (or the network's) operating system. There's a printer queue where print jobs wait for the printer to be available. Queues also store user input events like keystrokes, mouse clicks, touchscreen touches, and microphone inputs. They are really important in multiprocessing systems so that each event can be processed in the correct order even when the processor is busy doing something else when the event occurs.

The two basic operations on the queue are called **insert** and **remove**. Insert corresponds to a person inserting themself at the rear of a ticket line. When that person makes their purchase, they remove themself from the front of the line.

The terms for insertion and removal in a stack are fairly standard; everyone says *push* and *pop*. The terminology for queues is not quite as standardized. *Insert* is also called *put* or *add* or *enqueue*, whereas *remove* may be called *delete* or *get* or *dequeue*. The rear of the queue, where items are inserted, is also called the *back* or *tail* or *end*. The front, where items are removed, may

also be called the *head*. In this book, we use the terms *insert*, *remove*, *front*, and *rear*.

## A Shifty Problem

In thinking about how to implement a queue using arrays, the first option would be to handle inserts like a push on a stack. The first item goes at the last position (first empty cell) of the array. Then when it's time to remove an item from the queue, you would take the first filled cell of the array. To avoid hunting for the position of those cells, you could keep two indices, `front` and `rear`, to track where the filled cells begin and end, as shown in Figure 4-4. When Ken arrives, he's placed in the cell indexed by `rear`. When you remove the first item in the queue, you get Raj from the cell indexed by `front`.

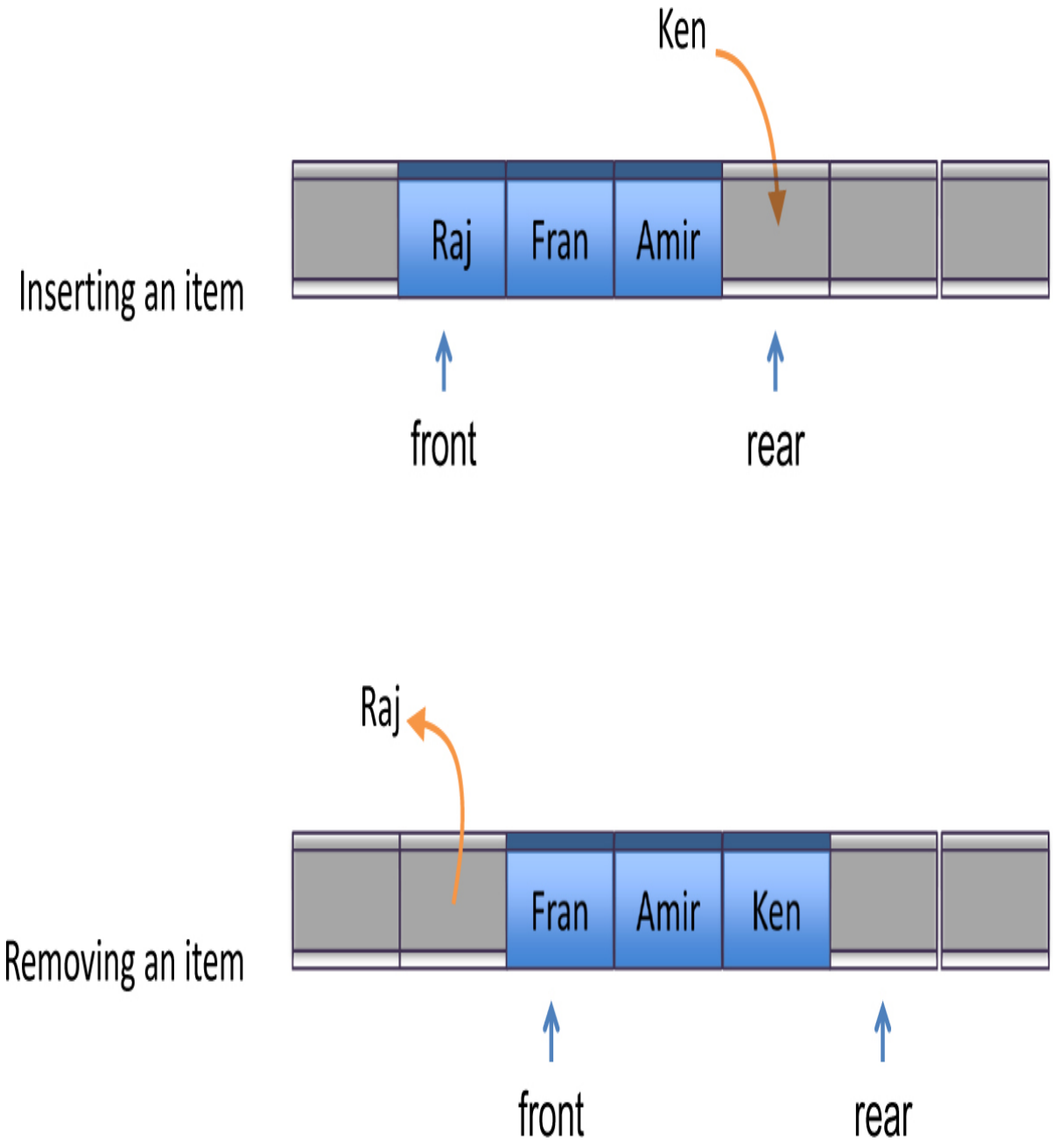



**Figure 4-4** *Queue operations in a linear array*

This operation works nicely because both insert and remove simply copy an item and update an index pointer. It's just as fast as the push and pop operations of the stack and takes only one more variable to manage.

What happens when you get to the end of the array? If the `rear` index reaches the end of the array, there's no space to insert new items. That might be acceptable because it's no worse than when a stack runs out of room. If the

`front` index has moved past the beginning of the array, however, free cells could be used for item storage. It seems wasteful not to take advantage of them.

One way to reclaim that unused storage space would be to shift all the items in the array when an insertion would go past the end. Shifting is similar to what happens with people standing in a line/queue; they all step forward as people leave the front of the queue. In the array, you would move the item indexed by `front` to cell 0 and move all the items up to `rear` the same number of cells; then you would set `front` to 0 and `rear` to `rear` − `front`. Shifting items, however, takes time, and doing so would make some insert operations take O(N) time instead of O(1). Is there a way to avoid the shifts?

# A Circular Queue

To avoid the problem of not being able to insert more items into a queue when it's not full, you let the `front` and `rear` pointers **wrap around** to the beginning of the array. The result is a **circular queue** (sometimes called a **ring buffer**). This is easy to visualize if you take a row of cells and bend them around in the form of a circle so that the last cell and first cell are adjacent, as shown in Figure 4-5. The array has N cells in it, and they are numbered 0, 1, 2, …, N−2, N−1. When one of the pointers is at N−1 and needs to be incremented, you simply set it to 0. You still need to be careful not to let the wraparound go too far and start writing over cells that have valid items in them. To see how, let's look first at the Queue Visualization tool and then the code that implements it.

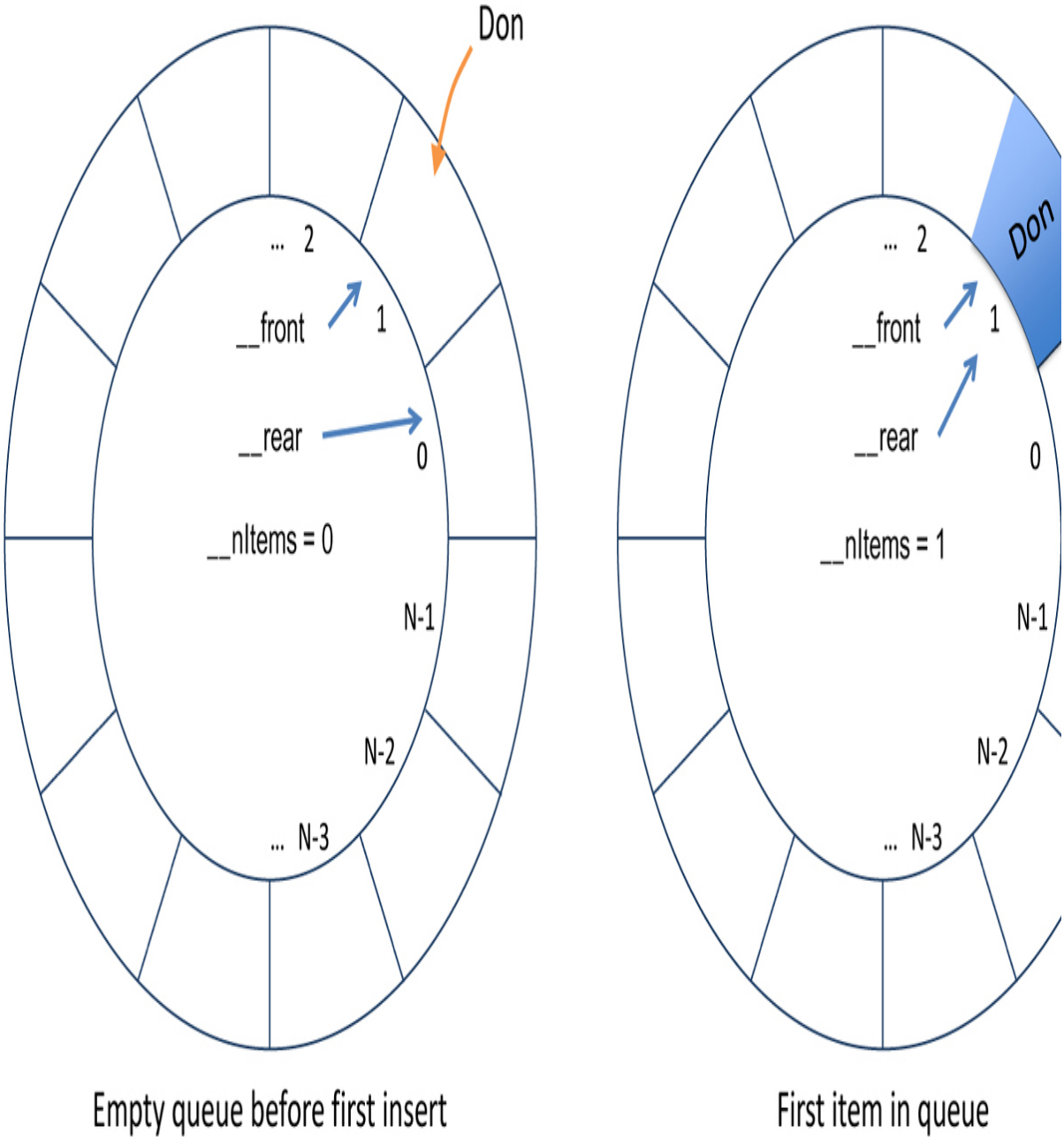



**Figure 4-5** *Operation of the* `Queue.insert()` *method on an empty queue*

# The Queue Visualization Tool

Let's use the Queue Visualization tool to get an idea how queues and circular arrays work. When you start the Queue tool, you see an empty queue that can hold 10 items, as shown in Figure 4-6. The array cells are numbered from 0 to 9 starting from the right edge. (The indices increase in the counterclockwise

direction to match the convention in trigonometry where angles increase from the horizontal, X-axis.)


**Figure 4-6** *The Queue Visualization tool*

The `_front` and `_rear` indices are shown in the center (with underscore prefixes to be somewhat like the attributes named in the code). There's also an `_nItems` counter at the top left. It might seem a little odd to have `front` point to 1 and `rear` point to 0, but the reason will become clearer shortly.

## The Insert Button

After typing some text in the text entry box, select the Insert button. The tool responds by incrementing the `_rear` index and copying the value to cell 1. As with the stack, you can insert string values of limited length. Typing another string followed by pressing Return advances `_rear` to cell 2 and copies the value into it.

## The Remove Button

When the queue has one or more items in it, selecting the Remove button copies the item at the `_front` cell to a variable named `front`. The `_front` cell is cleared and the index advances by one. The `front` variable holds the value returned by the operation.

Note that the `_front` and `_rear` indices can appear in any order. In the initial, empty queue, `_rear` was one less than `_front`. When the first item was inserted, both `_rear` and `_front` pointed to the same cell. Additional inserts

advance `_rear` past `_front`. The remove operations advance `_front` past `_rear`.

Keep inserting values until all the cells are filled. Note how the `_rear` index wraps around from 9 to 0. When all the cells are filled, `_rear` is one less than `_front`. That's the same relationship as when the queue was empty, but now the `_nItems` counter is 10, not 0.

### The Peek Button

The peek operation returns the value of the item at the front of the queue without removing the item. (Like insert and remove, peek, when applied to a queue, is also called by a variety of other names.) If you select the Peek button, you see the value at `_front` copied to an output box. The queue remains unchanged.

Some queue implementations have a `rearPeek()` and a `frontPeek()` method, but usually you want to know what you're about to remove, not what you just inserted.

### The New Button

If you want to start with an empty queue, you can use the New button to create one. Because it's based on an array, the size of the array is the argument that's required. The Queue Visualization tool lets you choose a range of queue sizes up to a limit that allows for values to be easily displayed. The animation shows the steps taken in the call to the object constructor.

### Empty and Full

If you try to remove an item when there are no items in the queue, you'll get the `Queue is empty!` message. You'll also see that the code is highlighted in a different color because this operation has raised an exception. Similarly, if you try to insert an item when all the cells are already occupied, you'll get the `Queue is full!` message from a Queue overflow exception. These operations are shown in detail in the next section.

## Python Code for a Queue

Let's look at how to implement a queue in Python using a circular array. Listing 4-5 shows the code for the `Queue` class. The constructor is a bit more complex than that of the stack because it must manage two index pointers for the front and rear of the queue. We also choose to maintain an explicit count of the number of items in this implementation, as explained later.

**Listing 4-5** *The `Queue.py` Module*

```
# Implement a Queue data structure using a Python list

class Queue(object):
   def __init__(self, size):           # Constructor
      self.__maxSize = size            # Size of [circular] array
      self.__que = [None] * size       # Queue stored as a list
      self.__front = 1                 # Empty Queue has front 1
      self.__rear = 0                  # after rear and
      self.__nItems = 0                # No items in queue

   def insert(self, item):             # Insert item at rear of queue
      if self.isFull():                # if not full
         raise Exception("Queue overflow")
      self.__rear += 1                 # Rear moves one to the right
      if self.__rear == self.__maxSize: # Wrap around circular array
         self.__rear = 0
      self.__que[self.__rear] = item   # Store item at rear
      self.__nItems += 1
      return True

   def remove(self):                   # Remove front item of queue
      if self.isEmpty():               # and return it, if not empty
         raise Exception("Queue underflow")
      front = self.__que[self.__front] # get the value at front
      self.__que[self.__front] = None # Remove item reference
      self.__front += 1                # front moves one to the right
      if self.__front == self.__maxSize: # Wrap around circular arr.
         self.__front = 0
      self.__nItems -= 1
      return front

   def peek(self):                    # Return frontmost item
      return None if self.isEmpty() else self.__que[self.__front]

   def isEmpty(self): return self.__nItems == 0
```

```
    def isFull(self): return self.__nItems == self.__maxSize

    def __len__(self): return self.__nItems

    def __str__(self):                  # Convert queue to string
        ans = "["                       # Start with left bracket
        for i in range(self.__nItems):  # Loop through current items
            if len(ans) > 1:            # Except next to left bracket,
                ans += ", "             # separate items with comma
            j = i + self.__front        # Offset from front
            if j >= self.__maxSize:     # Wrap around circular array
                j -= self.__maxSize
            ans += str(self.__que[j])   # Add string form of item
        ans += "]"                      # Close with right bracket
        return ans
```

The `__front` and `__rear` pointers point at the first and last items in the queue, respectively. These and other attributes are named with double underscore prefixes to indicate they are private. They should not be changed directly by the object user.

When the queue is empty, where should `__front` and `__rear` point? We typically set one of them to 0, and we choose to do that for `__rear`. If we also set `__front` to be 0, we will have a problem inserting the first element. We set `__front` to 1 initially, as shown in the empty queue of Figure 4-5, so that when the first element is inserted and `__rear` is incremented, they both are 1. That's desirable because the first and last items in the queue are one and the same. So `__rear` and `__front` are 1 for the first item, and `__rear` is increased for the insertions that follow. That means the frontmost items in the queue are at lower indices, and the rearmost are at higher indices, in general.

The `insert()` method adds a new item to the rear of the queue. It first checks whether the queue is full. If it is, `insert()` raises an exception. This is the preferred way to implement data structures: provide tests so that callers can check the status in advance, but if they don't, raise an exception for invalid operations. Python's most general-purpose `Exception` class is used here with a custom reason string, `"Queue overflow"`. Many data structures define their own exception classes so that they can be easily distinguished from exception conditions like `ValueError` and `IndexError`.

You can avoid shifting items in the array during inserts by verifying that space is available before incrementing the `__rear` pointer and placing the new item at that empty cell of the array. The increment takes an extra step to handle the

circular array logic when the pointer would go beyond the maximum size of the array by setting `__rear` back to zero. Finally, `insert()` increases the item count to reflect the inserted item at the rear.

The `remove()` method is similar in operation but acts on the `__front` of the queue. First, it checks whether the queue is empty and raises an "underflow" exception if it is. Then it makes a copy of the first item, clears the array cell, and increments the `__front` pointer to point at the next cell. The `__front` pointer must wrap around, just like `__rear` did, returning to 0 when it gets to the end of the array. The item count decreases because the front item was removed from the array, and the copy of the item is returned.

The `peek()` method looks at the frontmost item of the queue. You could create `peekfront()` and `peekrear()` methods to look at either end of the queue, but it's rare to need both in practice. The `peek()` method returns `None` when the queue is empty, although it might be more consistent to use the same underflow exception produced by `remove()`.

The `isEmpty()` and `isFull()` methods are simple tests on the number of items in the queue. Note that a slightly different Python syntax is used here. The whole body of the method is a single return statement. In that case, Python allows the statement to be placed after the colon ending the method signature. The `__len__()` method also uses the shortened syntax. Note also that these tests look at the `__nItems` value of the attribute rather than the `__front` and `__rear` indices. That's needed to distinguish the empty and full queues. We look at how wrapping the indices around makes that choice harder in Figure 4-9.

The last method for `Queue` is `__str__()`, which creates a string showing the contents of the queue enclosed in brackets and separated by commas for display. This method illustrates how circular array indices work. The beginning of the string has the front of the queue, and the end of the string is the rearmost item. The `for` loop uses the variable `i` to index all current items in the queue. A separate variable, `j`, starts at the `__front` and increments toward the `__rear` wrapping around if it passes the maximum size of the array.

Some simple tests of the `Queue` class are shown in Listing 4-6 and demonstrate the basic operations. The program creates a queue and inserts some names in it. The initial queue is empty, and Figure 4-5 shows how the first name, `'Don'`, is inserted. After that first insertion, both `__front` and `__rear` point at array cell 1 and the number of items is 1.

**Listing 4-6** *The* `QueueClient.py` *Program*

```
from Queue import *

queue = Queue(10)

for person in ['Don', 'Ken', 'Ivan', 'Raj', 'Amir', 'Adi']:
   queue.insert(person)

print('After inserting', len(queue),
      'persons on the queue, it contains:\n', queue)
print('Is queue full?', queue.isFull())

print('Removing items from the queue:')
while not queue.isEmpty():
   print(queue.remove(), end=' ')
print()
```

The person names inserted into the queue keep advancing the `__rear` pointer and increasing the `__nItems` count, as shown in Figure 4-7. After inserting all the names, the `QueueClient.py` program uses the `__str__()` method (implicitly) and prints the contents of the queue along with the status of whether the queue is full or not. After that's complete, the program removes items one at a time from the queue until the queue is empty. The items are printed separated by spaces (which is different from the way the `__str__()` method displays them). The result looks like this:

```
$ python3 QueueClient.py
After inserting 6 persons on the queue, it contains:
 [Don, Ken, Ivan, Raj, Amir, Adi]
Is queue full? False
Removing items from the queue:
Don Ken Ivan Raj Amir Adi
```


**Figure 4-7** *Inserting an item into a partially full queue*

The printed result shows that items are deleted from the queue in the same order they were inserted in the queue. The first step of the deletion process is shown in Figure 4-8. When the first item, `'Don'`, is deleted from the front of the queue, the `__front` pointer is advanced to 2, and the number of items decreases to 5. The `__rear` pointer stays the same.

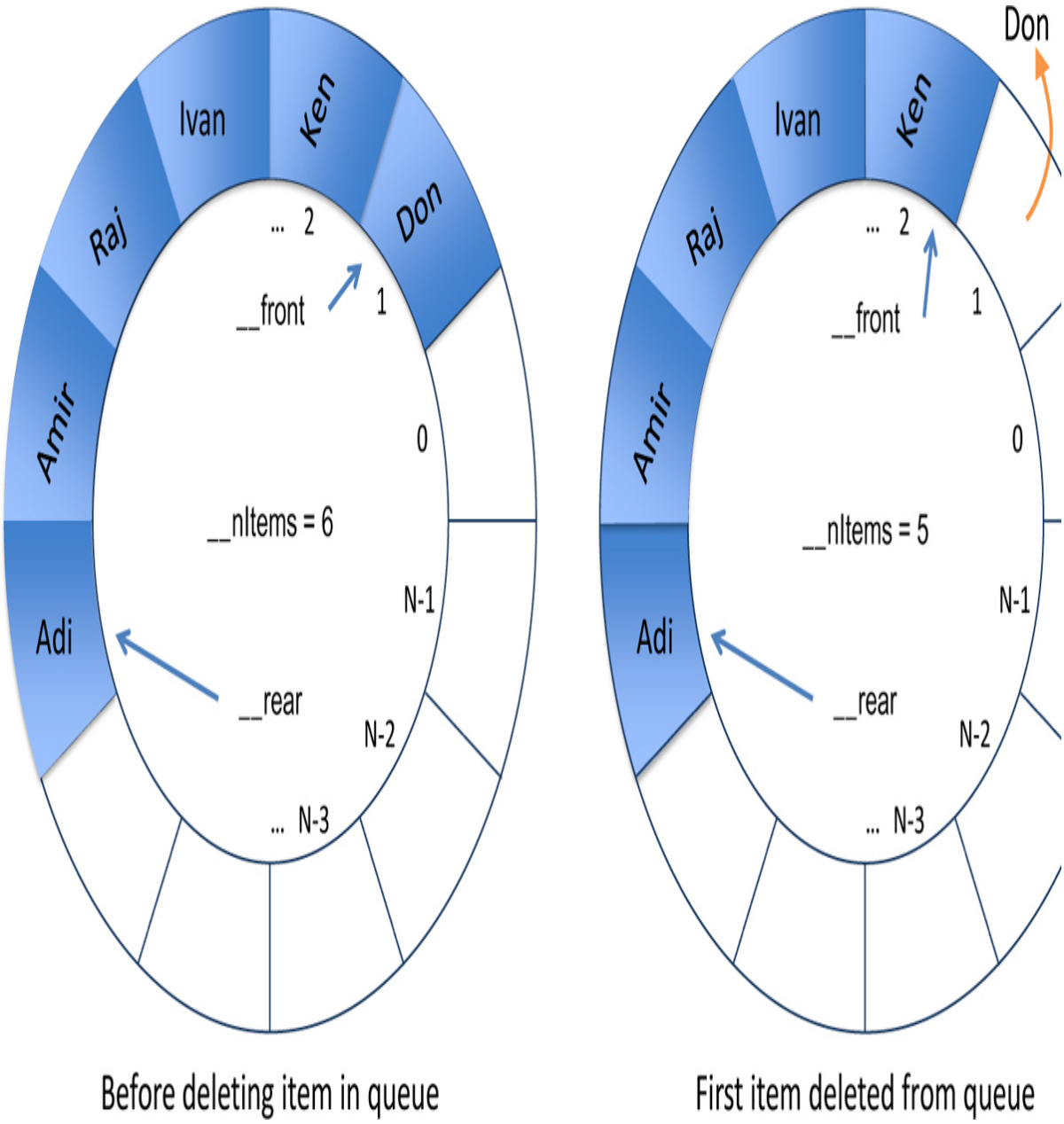



**Figure 4-8** *Deleting an item from the queue*

Let's look at what happens when the queue wraps around the circular array. If you delete only one name and then insert more names into the queue, you'll eventually get to the situation shown in Figure 4-9. The `__rear` pointer keeps increasing with each insertion. After it gets to `__maxSize - 1`, the next insertion forces `__rear` to point at cell 0.