Guy Lebanon · Mohamed El-Geish

Computing with Data

An Introduction to the Data Industry

Springer

Computing with Data

Guy Lebanon • Mohamed El-Geish

Computing with Data

An Introduction to the Data Industry

www.computingwithdata.com

Springer

Guy Lebanon
Amazon
Menlo Park
CA, USA

Mohamed El-Geish
Voicera
Santa Clara
CA, USA

ISBN 978-3-319-98148-2 ISBN 978-3-319-98149-9 (eBook)
https://doi.org/10.1007/978-3-319-98149-9

Library of Congress Control Number: 2018954275

© Springer Nature Switzerland AG 2018
This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

To Anat Lebanon

Guy

To my family and friends who put up with me while writing this (and elsewhen).

Mohamed

Contents

Chapter 1
Introduction: How to Use This Book?

Machine learning, data analysis, and artificial intelligence are becoming increasingly ubiquitous in our lives, and more central to the high-tech industry. These fields play a central role in many of the recent and upcoming revolutions in computing; for example, social networks, streaming video on demand, personal assistants (e.g., Alexa, Siri, and Google Assistant), and self-driving cars. Alphabet's Executive Chairman, Eric Schmidt, went a step further at the 2016 Google Cloud Computing Conference in San Francisco when he said, "Machine learning and crowdsourcing data will be the basis and fundamentals of every successful huge IPO win in five years."

On the other hand, there is a massive talent gap in the Big Data job market. The McKinsey Global Institute predicted that 1.5 million people, who know how to wield Big Data, are going to be in high demand by the year 2018; and that's in the US job market alone.[1]

In pursuit of innovation, organizations seek after people who possess a set of skills that combines data analysis, software engineering, applied statistics, machine learning, system design, databases, programming languages, and software tools. This set of skills is extremely broad and goes much beyond the traditional computer science undergraduate curriculum. It splits into two broad categories: *computing skills* and *algorithmic and mathematical skills*.

Computing Skills for Data Analysis

- Operating system concepts and use of the command shell
- Basic hardware concepts such as the memory hierarchy, caching, and binary floating-point representations leading to overflow and underflow
- Programming languages for low-latency production systems such as C++

[1] https://computingwithdata.com/redirect/mckinsey.

© Springer Nature Switzerland AG 2018

G. Lebanon, M. El-Geish, *Computing with Data*,
https://doi.org/10.1007/978-3-319-98149-9_1

- Scripting and programming languages for high-level data analysis such as Python and R
- Big data frameworks such as Apache Kafka
- SQL and NoSQL databases
- Processing and streaming data
- Graphing and visualizing data
- Software testing
- Programming tools such as version control, build tools, and documentation tools
- etc.

Algorithmic and Mathematical Skills for Data Analysis

- Mathematical prerequisites such as probability, multivariate calculus, and linear algebra
- Maximum-likelihood estimation
- Bayesian statistics
- Linear classification such as logistic regression and support vector machines
- Nonlinear classification methods such as gradient boosted decision trees and random forests
- Optimization algorithms such as variations of stochastic gradient descent
- Estimation in high dimensions including regularization and variable selection
- Clustering and topic modeling
- Density estimation
- Dimensionality reduction
- Statistical testing theory for conducting A/B tests
- Natural language processing for handling text or speech data
- Recommendation systems
- Deep learning
- etc.

These two sets of skills are extremely broad and learning them requires reading dozens of different textbooks. The traditional learning method of reading textbooks sequentially and learning the above skills in depth, one after another, is very challenging.

This book introduces the first set of skills above—computing skills—in a way that does not rely on external sources and that's accessible for people without strong computer science background. The introduction is self-contained and progresses from basic hardware concepts, to operating systems, programming languages, graphing and processing data, testing and programming tools, big data frameworks, and cloud computing. While this book provides an in-depth introduction, readers who require deeper expertise can consult additional sources afterwards.

We made a conscious decision to avoid the second set of skills that are more algorithmic and mathematical in nature. There are many textbooks that specialize each in a subset of these areas, for example, linear algebra (Strang, 2009), calculus and real analysis (Rudin, 1976; Trench, 2003; Thomas et al., 2009), probability

(Feller, 1968; Ross, 2009; DasGupta, 2010), statistics (Casella and Berger, 2001), regression (Seber and Lee, 2003; Kutner et al., 2004), kernel methods (Schölkopf and Smola, 2002), and natural language processing (Manning and Schutze, 1999). Two popular textbooks that provide an overview of machine learning are (Bishop, 2006; Murphy, 2012).

In writing this book, we took an approach to cover a breadth of computational topics requisite for data scientists, analysts, and engineers (and those who aspire to be) to have a productive start in the industry. You can think of it as your mentor that gets you up to speed during the first few months of starting a new job at one of the fields mentioned above. We selected the topics to cover in this book by examining what it takes to be successful in a role that entails computing with data. We looked deeply into what skills are required; and we drew from our combined experience working on big data products at companies like Microsoft, Amazon, Google, LinkedIn, and Netflix; and building big data systems for an AI startup—Voicera—from the ground up. By covering a breadth of topics—that range from the basics of how a computer works to advanced data manipulation techniques—this book opens more doors for you to explore and enhance your knowledge.

This book was written with several audiences in mind. Readers with a strong traditional educational background in CS but without significant industry background will find the following chapters particularly useful: 7 (Learning R), 11 (Essential Knowledge: Testing), 12 (A Few More Things About Programming), 8 (Visualizing Data in R and Python), 9 (Processing Data in R and Python), 14 (Thoughts on System Design for Big Data—including big data frameworks and the cloud), 13 (Essential Knowledge: Data Stores), and 15 (Thoughts on Software Craftsmanship). Readers who do not have a strong traditional educational background in CS (or readers who need a refresher) may find—additionally—the following chapters particularly useful: 2 (Essential Knowledge: Hardware), 3 (Essential Knowledge: Operating Systems), and 4–7 (Programming Languages).

A key advantage of this book is the plethora of examples we use to explain a multitude of interconnected concepts that may otherwise feel dry. This book is intended to help you understand and apply said concepts so that you can recall them in the same context when needed. The principles we cover in this book can be used in many applications ranging from software simulations to real-world web applications that serve billions of users; in fact, that's the scale we had in mind when selecting topics for this book. We've worked on web applications that serve the vast majority of internet users worldwide; we want you to have the skills that enable you to do that—and much more. We don't claim that the material in this book is all that you need to do so; this book—like a good coach—introduces you to the beginnings of many paths and you need to do more work to explore them farther.

We recognize the fact that many readers may want to go the extra mile, with the help of more advanced material and specialized texts, so we added references to such material as recommended readings. The companion booksite, https://www.computingwithdata.com, is a great resource where you can find references, bibliography, and useful links. Also, the myriad of code and script examples

you'll find in this book are available online on our booksite along with other examples that we didn't include here; for instance, data sets for practice and listings of more involved code and script examples that are too long to print. We highly recommend that you run the code and script examples as you encounter them while reading the relevant material. All content on the booksite is gratis, so feel free to share it as well; this way, for each concept you learn there are examples to help cement your understanding of it: You see one, do one, and teach one.

This is not your typical introduction-to-data-science book; it's a handbook that guides you through a journey to explore various topics and takes you through many roads to the goal you set each time you pick up the book. One goal we envisioned for strata of our readers is bridging the gap between a background in statistics and a career in the data science industry that requires honed programming skills; for that purpose, we introduce the reader to prevalent programming languages and data processing systems that are commonly used in the industry to accomplish great feats of engineering. Another main goal is introducing programmers to data science concepts and practices through new apparatuses like R programming and data processing techniques; moreover, said readers can explore new tools and libraries—to use in big data projects—that work with programming languages they may already know (like pandas with Python); the programming examples in this book are geared towards practical data science applications. This book is here to help you hone those skills, introduce you to new ones to add to your arsenal, and help you be a more productive data scientist, analyst, and engineer. It's also a helpful guide for self-study to survey data science and data engineering for those who aspire to start a career in said fields or expand on their current roles in areas like applied statistics, big data, machine learning, data mining, informatics, etc.

Since this book covers various topics, you don't have to read it cover-to-cover; however, we recommend that you read related topics together to establish a common context that connects them together. For example, it makes sense to read about programming in Python before starting to read about the NumPy and SciPy packages for scientific computing. One approach you can take is depth-first: Pick a topic, read the relevant material and practice the respective examples, apply the concepts in the real world, and optionally explore more advanced material for a deep dive into said topic. Another approach is breadth-first: Read the entirety of this book first, pick the topics that are most relevant to you to pursue, and find the specialized material to help you explore further. Regardless of which approach you choose, it's key to practice and work through the examples included in this book; computing with data is part science and part art, both of which require rigorous practice. So let's get started!

References

G. Strang. *Introduction to Linear Algebra*. Wellesley Cambridge Press, fourth edition, 2009.
W. Rudin. *Principles of Mathematical Analysis*. McGraw-Hill, third edition, 1976.
W. F. Trench. *Introduction to Real Analysis*. Pearson, 2003.
G. Thomas, M. D. Weir, and J. Hass. *Thomas' Calculus*. Addison Wesley, twelfth edition, 2009.
W. Feller. *An Introduction to Probability Theory and its Application*, volume 1. John Wiley and Sons, third edition, 1968.
Sheldon M. Ross. *Introduction to Probability Models*. Academic Press, tenth edition, 2009.
A. DasGupta. *Fundamentals of Probability: A First Course*. Springer, 2010.
G. Casella and R. L. Berger. *Statistical Inference*. Duxbury, second edition, 2001.
G. A. Seber and A. J. Lee. *Linear Regression Analysis*. Wiley Interscience, 2003.
M. Kutner, C. Nachtsheim, J. Neter, and W. Li. *Applied Linear Statistical Models*. McGraw-Hill, fifth edition, 2004.
B. Schölkopf and A. Smola. *Learning with Kernels*. MIT Press, 2002.
C. D. Manning and H. Schutze. *Foundations of Statistical Natural Language Processing*. MIT Press, 1999.
C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.
K. P. Murphy. *Machine Learning: A Probabilistic Perspective*. MIT Press, 2012.

Chapter 2
Essential Knowledge: Hardware

In order to implement efficient computer programs, it's essential to understand the basic hardware structure of computers. In this chapter we examine the hardware components of a typical computer (CPU, memory, storage, GPU, etc.) focusing on issues that are relevant for software development and algorithm design. We also explore concepts like binary representations of numbers and strings, assembly language, multiprocessors, and the memory hierarchy.

A typical computer is composed of several important components that are connected to the computer's motherboard, including the central processing unit (CPU), the graphics processing unit (GPU), the memory, and the disk drive. The motherboard facilitates communication between these components and supplies electricity to them. The motherboard also connects the components listed above with external devices such as keyboard, mouse, display, printer, and network card. Figure 2.1 shows a schematic illustration of a motherboard.

2.1 RAM and ROM

The random access memory (RAM) is a collection of chips that store information in the form of a sequence of digital bits, where each bit is set to either 0 or 1. For example, the RAM may contain the following sequence.

```
00111010010110101110111100 1...0101110000110100011101
```

Definition 2.1.1 One byte is a sequence of eight bits. A kilobyte (KB) is 2^{10} bytes, a megabyte (MB) is 2^{20} bytes, and a gigabyte (GB) is 2^{30} bytes.

Definition 2.1.2 The memory size is the length of the memory sequence in bits divided by 8 (or alternatively the number of bytes).

A typical laptop computer manufactured during the year 2018 has between four and sixteen GB of RAM.

© Springer Nature Switzerland AG 2018

G. Lebanon, M. El-Geish, *Computing with Data*,
https://doi.org/10.1007/978-3-319-98149-9_2

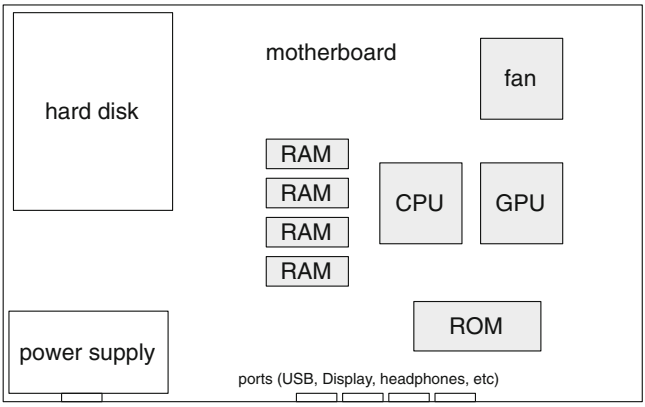


Fig. 2.1 A schematic illustration of a motherboard. The motherboard hosts devices such as the central processing unit (CPU), the graphics processing unit (GPU), the random access memory (RAM), and the read-only memory (ROM). The ports enable communication between the motherboard components and external devices such as display, mouse, and keyboard

Definition 2.1.3 The address of a part of the memory is its position in the memory sequence.

The contents of the RAM may be read or written by the central processing unit (see Sect. 2.3 for an overview of the CPU). In most cases, the contents of the RAM are lost when power is turned off. The read-only memory (ROM), is a different kind of memory that is impossible or harder to modify, and whose contents persist after power is turned off.

The qualifier "random access" in RAM implies that it takes constant time for the CPU to read from a portion of RAM or write to it, regardless of its position. Specifically, accessing RAM bits that are close to previously accessed RAM bits takes as much time as accessing RAM bits that are far away from previously accessed RAM bits. This important property is not shared by mechanical disk drives.

2.2 The Disk

The hard disk drive (HDD) stores a sequence of bits much like the memory. We distinguish between two types of disks: solid state disks and mechanical disks. Solid state disks store the bit sequence in chips. Mechanical disks store the bit sequence on disks that are coated with magnetic material. Reading or writing the contents of a mechanical disk is done by mechanically rotating the disk so the disk head is aligned with the appropriate disk location. In contrast to RAM, the content of either a solid state disk or a mechanical disk persists after power is turned off.

Due to the need for mechanical rotation, mechanical disks are not random access in the sense that accessing bits near previously accessed bits takes less time than accessing bits far away from previously accessed bits. This has significant impact on the design of algorithms that access the disk in that a sequential passage over a contiguous sequence of bits is preferred over accessing multiple noncontiguous bits.

A mechanical disk can store more bits than a solid state disk (per dollar cost), which in turn can store significantly more bits than RAM (per dollar cost). Technological advances are reducing the cost of solid state disks and are making it a viable alternative to mechanical disks in laptop and desktop computers that do not require large disk storage.

2.3 The Central Processing Unit

The central processing unit (CPU) reads a sequence of instructions that are stored in binary form at a memory location, and executes them one by one. Each CPU has a specific set of possible instructions, called the instruction set, and a scheme to encode these instructions as bit sequences that are stored in memory. The CPU instructions are simple and in most cases fall into one of the following categories:

- read the content of a memory location,
- write content to a memory location,
- transfer execution to instructions in a different memory location, or
- compute an arithmetic operation.

Despite the simplicity of the individual instructions, they are the building blocks of all computer programs and conversely all computer programs are essentially long sequences of such instructions.

CPUs can also read and write information to registers, which are components of the CPU that can hold a small amount of memory and feature faster access time than the RAM.

Definition 2.3.1 The program counter (PC) is a part of the CPU that contains the address (see Definition 2.1.3) where the instruction that should be executed next is stored.

The CPU executes a program by repeating the following three steps.

1. The CPU finds the address in memory where the instruction that should be executed next is stored and reads that instruction.
2. The CPU executes the instruction.
3. The CPU updates the program counter, typically by incrementing it by the number of bytes that are used to store an instruction.

The portion of memory that holds the instructions that are being executed by the CPU must be protected from being overwritten by the program itself. For that reason, the portion of memory that is used for storing the instructions and the portion of the memory that is used by the program do not overlap.

2.4 The Clock

The CPU clock is a repetitive, metronome-like signal that synchronizes the different CPU and motherboard components.

Definition 2.4.1 The clock period is the time between successive CPU clock signals (ticks), measured in fractions of a second. The clock rate is the inverse of the clock period, representing the number of clock periods in a second. The clock rate is usually measured in multiples of Giga-Hertz (GHz) that equals billion clock periods per second.

A typical laptop computer manufactured during the year 2018 has a clock frequency of 2 GHz and a clock period of $1/(2 \cdot 10^9) = 0.5 \cdot 10^{-9}$ (one half of a billionth of a second).

Executing instructions may take several clock cycles, with more complex instructions typically taking longer than simple instructions.

Definition 2.4.2 The CPU time of a particular program is the amount of time the CPU spends executing that program.

Definition 2.4.3 Clock cycles per instruction (CPI) is the average number of clock cycles per instruction. The averaging is weighted by the frequency with which the different instructions appear.

Denoting the number of instructions in a program as the instruction count, we can express the CPU time as

$$\begin{aligned}\text{CPU time} &= \text{number of clock cycles} \cdot \text{clock period} \\ &= \frac{\text{number of clock cycles}}{\text{clock frequency}} \\ &= \text{instruction count} \cdot \text{CPI} \cdot \text{clock period} \\ &= \frac{\text{instruction count} \cdot \text{CPI}}{\text{clock frequency}}.\end{aligned}$$

The third equality above depends on the accuracy of the instruction weighting that is used in the calculation of the CPI.

Definition 2.4.4 Million instructions per second (MIPS) is the number of instructions that can be executed during a second, divided by a million:

$$\text{MIPS} = \frac{\text{instruction count}}{\text{execution time} \cdot 10^6}. \tag{2.1}$$

Since the MIPS formula (2.1) depends on the type of instructions that are being executed, it is common to use a representative mix of instruction types.

Definition 2.4.5 Floating-point operations per second (FLOPS) is the number of floating-point operations (addition, subtraction, and multiplication of non-integers) that can be executed during a second.

As in the case of MIPS, the FLOPS measure depends on the type of floating-point operations (addition takes less time than multiplication), and is thus based on a representative mix of instruction types. Below are several standard abbreviations.

Kilo	FLOPS	=	10^3 FLOPS
Mega	FLOPS	=	10^6 FLOPS
Giga	FLOPS	=	10^9 FLOPS
Tera	FLOPS	=	10^{12} FLOPS
Peta	FLOPS	=	10^{15} FLOPS
Exa	FLOPS	=	10^{18} FLOPS

Supercomputers during 2017 exceeded 100 peta-FLOPS; future plans predict an exa-FLOPS supercomputer before 2020.

The three ways of measuring CPU speed above (clock frequency, MIPS, FLOPS) are insightful when comparing CPUs with identical instruction sets. However, these measures may be misleading when comparing the clock frequencies of CPUs with different instruction sets since the CPUs may differ in their CPI quantities. A common alternative for comparing the speed of CPUs with different instruction sets is to compare the time it takes to finish executing certain benchmark programs. Note, however, that this measurement depends on factors that are not related to the CPU such as the speed of the memory and motherboard communication channels. Such a comparison makes sense when comparing end-to-end systems but may be less appropriate when evaluating a CPU in isolation.

Traditionally, scaling up program speed was focused on increasing the clock frequency or improving the memory access speed. While we continue to see such improvements in recent years, the rate of these improvements is slowing down. A consequence of this slow-down in the rate of improvements is that it's becoming much more important to scale up computation by parallelizing the computation over a large number of computers and/or using GPU-accelerated computing.

2.4.1 Logical and Physical Clocks

The concept of time in computing is as important as the role it plays in other aspects of life. For any nontrivial program, understanding the order of events (e.g., reading and writing data) is key to verifying its correctness (see Sect. 13.3.2 for an

example); such order is enforced using a logical clock that keeps track of logical time.[1] The more prominent uses of time in computing have to do with capturing and representing physical time as we—humans—use it (e.g., capturing the time when a payment was made); physical clocks provide methods to obtain and maintain physical time in computer systems. A typical computer has a real-time clock (RTC), which is an integrated circuit that keeps track of physical time even when the computer is powered-off (in which case, a battery installed on the motherboard keeps it running). When the operating system loads up, it initializes a system clock using the value of the RTC, and then proceeds to maintain physical time using a system timer.

2.4.2 Clock Drift

Nontrivial programs require not only a precise system clock but also an accurate one; measurements of time on different systems should be as close as possible to each other (precision) and as close as possible to true time (accuracy). Moreover, they interact with other systems, which have their own independent clocks. To ensure that interconnected systems have accurate clocks, they talk to a time server to get the coordinated universal time (UTC). Nowadays, the vast majority of computers synchronize their clocks over the Internet using the network time protocol (NTP). For example, the Windows operating system uses time synchronization services to update both the RTC and the system clock.[2] The National Institute of Standards and Technology (NIST) uses atomic clocks to provide a time synchronization service that serves—at the time of writing this book—about 16 billion requests per day; NIST is the source of truth for UTC in the USA in addition to serving UTC to the entire Internet.

Atomic clocks are extremely accurate; in February 2016, scientists from the Physikalisch-Technische Bundesanstalt (PTB), in Germany, built an atomic clock that has a measurement uncertainty of 3×10^{-18}. Before this engineering feat, such accuracy had only been a mere theoretical prediction.[3] The accuracy of system clocks in typical computers is lower than those of the time servers with which they synchronized due to unreliable network routes. A time server that receives simultaneous requests from various clients will reply with identical NTP timestamps, but the time taken for these responses to travel over the network—via unreliable routes whose latencies cannot be accurately predicted—causes the clocks on those clients to diverge and become less accurate; such changes are known as clock drift.

[1] http://amturing.acm.org/p558-lamport.pdf.

[2] https://support.microsoft.com/en-us/kb/232488.

[3] http://link.aps.org/doi/10.1103/PhysRevLett.116.063001.

Clock drift is a serious problem in distributed systems; one that cannot be solved but only mitigated. Software developers must be cognizant of its perils, and understand its scope and symptoms. Differences between clocks—even on the same device—can cause hard-to-find bugs in software that doesn't account for such differences. For example, the CPU's Time Stamp Counter (TSC) stores the number of clock ticks since the system was reset, providing software developers a cheap way—a single assembly instruction called RDTSC—to create a high-precision timer; however, that was only valid when the clock frequency was fixed. Modern CPU clocks adjust their frequencies to cool down and save power when desirable; a multi-core processor can adjust the clock frequency of each core independently,[4] causing each core's TSC to change independently. Legacy software that uses RDTSC in timer implementations on a modern multi-core processor may witness time moving backward as a subsequent read may have a smaller TSC value (because it got executed on a slower core); such bugs can be catastrophic! Luckily, they are easily fixed by using a monotonically nondecreasing clock implementation like the steady clock class in C++11 (the C++ standard released in 2011—see Chap. 4 for additional background).

A more noticeable example of such issues is clock drift in distributed systems used for financial services, where it can cost millions of dollars in losses. For example, imagine two clients connected to two different servers to sell a huge number of stocks at the exact moment the trading window opens; how much of a difference would the drift between the two clocks make? Let's work it out; we used an atomic clock time server (http://www.time.is) to calculate—thrice—a typical clock drift on a relatively fast Internet connection (105 Mbps and both machines are located in California); the average clock drift was $(77 + 81 + 140)/3 \approx 99.3$ms. At the time of writing, the New York Stock Exchange (NYSE) can process over $320,000$ orders per second; so in that slim time window of 99.3ms, the NYSE can process over $31,776$ orders! That's why the NYSE offers co-location services to other companies that host their trading systems in its data centers to minimize network latency—every microsecond counts.

2.5 The Graphics Processing Unit

The graphics processing unit (GPU), as the name suggests, is responsible for processing graphics that a computer renders on its display. Chip makers have been investing in GPU technologies to meet the increasing demands of gamers, graphic designers, 3D animators, and—surprisingly—data scientists; the latter group got interested in GPUs recently to accelerate general-purpose computations—unrelated to rendering graphics—because of their incredible performance in data parallelism (e.g., executing the same instruction in parallel on the elements of a massive

[4] http://www.intel.com/content/www/us/en/support/processors/000007073.html.

vector). To illustrate how different CPUs and GPUs are, consider the following: A typical laptop computer manufactured in 2018 has thousands of GPU cores made specifically to perform parallel computations efficiently; compared to a few CPU cores, each is tasked with processing sequential instructions.

To harvest the power of GPUs in general-purpose applications, software developers can use hardware-specific languages to program the GPU (like NVIDIA's CUDA, which stands for Compute Unified Device Architecture); or an open standard like OpenCL, which supports programming GPUs—and CPUs—made by various chip makers. The recent advancements in general-purpose computing on GPUs have been extremely beneficial in many fields like data science, machine learning, financial technology, and supercomputers.[5] Section 10.13 discussed more details about the use of GPUs in parallel computing.

2.6 Binary Representations

We describe below conventional ways to encode integers, real numbers, and strings as bit sequences. Using a binary encoding, the CPU is able to interpret memory bits as numbers or strings and it's also able to execute arithmetic and other operations. Understanding how computers encode numbers is important in order to avoid potential pitfalls as a result of rounding, overflow, and underflow (see Sect. 2.6.4). In addition, a programmer who understands standard numeric encoding can construct a custom encoding to trade-off speed vs. accuracy depending on the application at hand.

2.6.1 Binary Representation of Integers

The conventional way to encode a nonnegative integer z as a sequence of bits $b_1, b_2, \ldots, b_k$ is as follows:

$$z = \sum_{i=1}^{k} b_i 2^{k-i}, \qquad b_i \in \{0, 1\}. \tag{2.2}$$

Using the above formula, encoding a sequence of k bits can encode any integer in the range $\{0, 1, \ldots, 2^k - 1\}$. The value of k depends on the CPU, but a common value is $k = 32$.

We can encode with a byte ($k = 8$) any number between 0 and 255. For example,

[5] http://images.nvidia.com/content/tesla/pdf/Apps-Catalog-March-2016.pdf.

$$
\begin{aligned}
&00000000 \quad \text{represents} \quad 0\\
&00000001 \quad \text{represents} \quad 1 \cdot 2^0 = 1\\
&00000010 \quad \text{represents} \quad 1 \cdot 2^1 = 2\\
&00000011 \quad \text{represents} \quad 1 \cdot 2^1 + 1 \cdot 2^0 = 3\\
&00000100 \quad \text{represents} \quad 1 \cdot 2^2 = 4\\
&00011110 \quad \text{represents} \quad 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 = 30\\
&11111111 \quad \text{represents} \quad 1 \cdot (1 + 2 + 4 + \cdots + 128) = 255.
\end{aligned}
$$

There are two popular extensions of this encoding for representing signed integers: the sign and magnitude representation and the two's complement representation.

The sign and magnitude encoding of z uses the first bit to determine $\text{sign}(z)$ (typically 0 for positive and 1 for negative) and the remaining bits to determine $|z|$ (the absolute value of z) using the encoding (2.2). This encoding uses k bits to represent any integer in the range

$$\{-2^{k-1} + 1, \ldots, 0, \ldots, 2^{k-1} - 1\}.$$

Note that the number 0 has two different representations, one with a positive sign and one with a negative sign.

In the sign and magnitude representation, the number 42 is represented as 00101010 and the number -42 is represented as 10101010 (assuming $k = 8$).

The two's complement representation for an unsigned integer z uses the first bit to determine $\text{sign}(z)$ and the remaining bits to determine $|z|$ as follows.

- For positive integers z, the representation is identical to the sign and magnitude representation above.
- For negative numbers, the representation is the same as the sign and magnitude representation, only that all bits except for the first bit are flipped and then incremented by one.

Using this encoding, a sequence of k bits can represent integers in the range

$$\{-2^{k-1}, \ldots, 0, \ldots, 2^{k-1} - 1\}.$$

In contrast to the sign and magnitude representation, zero has only a single encoding: 00...0. The two's complement is more popular than the sign and magnitude representation since it facilitates the implementation of arithmetic subtraction.

In the two's complement representation with eight bits, we have

$$00000000 \quad \text{represents} \quad +0$$

00000001	represents	$+1$
11111111	represents	-1
00000010	represents	$+2$
11111110	represents	-2
00101010	represents	$+42$
11010110	represents	-42
10000000	represents	-128.

2.6.2 Binary Representation of Real Numbers

There are two different approaches to representing real numbers with bits: fixed point and floating point.

In the fixed point representation, the bit sequence $b_1, \ldots, b_k$ is interpreted as the corresponding integer, for example using two's complement representation, multiplied by 2^{-r} for some $r > 0$. This encoding can represent real numbers in the range $R = [-2^{k-1}/2^r, (2^{k-1}-1)/2^r]$. The representation is approximate rather than precise, as it cannot distinguish between two very close real numbers. As k increases the approximation quality increases. As r increases the range R of the possible numbers that can be represented decreases but the representation accuracy within that range increases. Note that the representation accuracy is uniform across the range R (the density of the represented values in different regions of R is uniform).

The floating-point representation differs from the fixed point representation in that the quality of the approximation in representing real numbers is nonuniform inside the achievable range R. The sequence of bits in this representation is divided into three binary sub-sequences: a single sign bit b, a sequence of exponent bits $e_1, \ldots, e_k$, and a sequence of mantissa bits $m_1, \ldots, m_l$. The three groups of bits combine to represent the number

$$(-1)^b \cdot M \cdot 2^E,$$

where M is the number encoded by the mantissa bits $m_1, \ldots, m_l$, and E is the number encoded by the exponent bits $e_1, \ldots, e_k$.

Many computers have two versions of this representation: a single precision representation using a total of 32 bits and a double precision representation using a total of 64 bits. Double precision floating-point representation can capture a wider range of possible values and with higher accuracy than single precision floating-point representation. The precise number of mantissa bits and exponent bits and their encoding depends on the floating-point standard being used. See for example

http://en.wikipedia.org/wiki/IEEE_754-1985 for a description of the popular IEEE 754-1985 standard.

Floating-point representation approximates real numbers in a typically wide range of numbers $[a, b]$, where $a < 0$ and $b > 0$, with better approximation quality for numbers that are small in absolute value and worse approximation quality for numbers that are large in absolute value. In other words, unlike fixed point representation, the density of floating-point representations differs in different ranges of R. This gives floating-point representation more flexibility in representing both very small values (in absolute values) very accurately and very large numbers (in absolute values) less accurately. For this reason, the floating-point representation is more popular than the fixed precision representation.

2.6.3 Encoding Strings as Bits

The American Standard Code for Information Interchange (ASCII) encodes the letters a-z, A-Z, 0-9, and other keystrokes such as colon, semicolon, comma, period, plus, and minus as integers in the range 0-255, represented by 8 bits according to the unsigned integer representation described in Sect. 2.6.1. The ASCII mapping appears in many websites, like wikipedia.org/wiki/ASCII for example. Concatenating bytes in ASCII representation leads to a convenient representation of text strings.

The ASCII encoding of A and B are 65 and 66 respectively. The binary encoding of the string AB is the following sequence of sixteen bits or two bytes.

$$01000001\ \ 01000010.$$

Unicode is an alternative to ASCII that can represent a wider range of characters including Arabic, Chinese, Cyrillic, and Hebrew letters. The current unicode mapping is available at http://unicode.org/charts.

2.6.4 Rounding, Overflow, and Underflow

Rounding, overflow, and underflow are three important phenomena that follow from the binary representations described above.

Rounding occurs when a real number x cannot be precisely matched a fixed- or a floating-point representation. The resulting rounding approximation $\text{fp}(x)$ is typically either the nearest fixed-point or floating-point representation of x, or a truncated version of x obtained by dividing and dropping the remainder.

A roundoff example in R code appears below (see Chap. 4 for a description of the R programming language). The symbol # below denotes comment and the symbol ## prefixes the output of the code below.

```
# example: roundoff of 1/3=0.33333333333333333333333..
# to a nearby floating-point
print(1/3, digits=22) # print 22 digits of fp(1/3)
## [1] 0.3333333333333333148296
```

Overflow occurs when the number x has a big absolute value that is outside the range of possible binary representations. In the case of integers, this occurs when the number of bits in the representation is too small to represent the corresponding number. In the case of floating-point representation, this occurs when the number of exponent bits is too small to represent the corresponding number. When overflow occurs the number is replaced by either the closest binary representation or is marked as an overflow and considered unavailable.

An overflow example using R code appears below.

```
print(10^100)   # no overflow
## [1] 1e+100
print(10^500)   # overflow, marked by Inf value
## [1] Inf
```

Underflow occurs when the number x is closer to 0 than any of the possible binary representations. In the case of floating-point representation, this occurs when the number of exponent bits is too small (the negative exponent is not low enough). In most cases, the number x is then replaced by zero.

An underflow example using R code appears below.

```
print(10^(-200),digits=22) # roundoff, but no underflow
## [1] 9.999999999999999821003e-201
print(10^(-400),digits=22) # underflow
## [1] 0
```

We conclude with a few practical observations.

- If we represent the floating-point representation of a number x as $\text{fp}(x)$, we may have $\text{fp}(x - y) \neq \text{fp}(x) - \text{fp}(y)$. The same conclusion applies to other arithmetic operations such as addition, multiplication, and division.
- Subtracting the floating-point representation of two similar numbers x, y results in a loss of approximation accuracy. Specifically, some of the mantissa bits of $\text{fp}(x)$ cancel out with the corresponding mantissa bits of $\text{fp}(y)$. For example,

$$0.1232215 \cdot 10^k - 0.1232214 \cdot 10^k = 0.0000001 \cdot 10^k,$$

rather than the preferred representation of $1 \cdot 10^{k-6}$.

In the extreme case where x and y are close enough to have identical floating-point representations, we may have $fp(x) - fp(y) = 0$ even though $\text{fp}(x - y) \neq 0$ may provide a good approximation for $x - y$.

- Comparing whether two real numbers are identical is problematic due to roundoff errors. It is preferable to consider instead the distance of the absolute value of the difference from zero.

  For example, suppose we want to compare whether a binary representation of $\sqrt{3} \cdot \sqrt{3}$ is the same (or very similar) as the binary representation of 3. A precise comparison may fail as

$$\text{fp}(\sqrt{3}) \cdot \text{fp}(\sqrt{3}) \neq \text{fp}(3),$$

  while an approximate comparison

$$|\text{fp}(\sqrt{3}) \cdot \text{fp}(\sqrt{3}) - \text{fp}(3)| < \epsilon$$

  may be useful for some small $\epsilon > 0$.

  The following C++ program demonstrates this example.

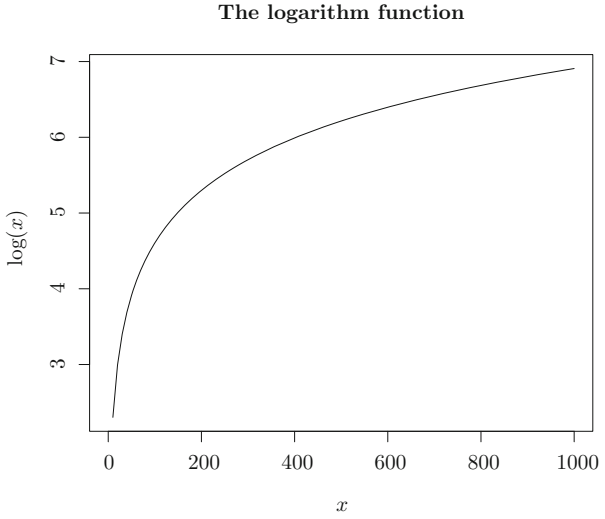
```
int main() {
  cout << (sqrt(3)*sqrt(3) - 3)                  << endl
       << (sqrt(3)*sqrt(3) == 3)                 << endl
       << (fabs(sqrt(3)*sqrt(3)-3) < 0.0000001) << endl;
  return 0;
}
```

  The program prints the following result showing that an exact comparison (second line) fails while an approximate comparison works (third line).

```
-4.440896e-16
0
1
```

- A common trick for avoiding overflow and underflow is to work in a logarithmic scale, rather than the original scale. Since logarithm compresses large positive numbers (see figure below where a long interval on the right side of the x axis gets mapped to a short interval on the y axis), $\text{fp}(\log(x))$ will not overflow in many cases where $\text{fp}(x)$ overflows. Similarly, since logarithm stretches numbers close to zero (see figure below where a short interval on the left side of the x axis gets mapped to a long interval on the y axis), $\text{fp}(\log(x))$ will not underflow in many cases where $\text{fp}(x)$ underflows.

```
# The R code below graphs log(x) as a function of x
curve(log, from = 0, to = 1000, xlab = "$x$",
      ylab = "$\\log(x)$", main = "The logarithm function")
```



To maintain correctness of arithmetic operations, multiplications need to be replaced with sums due to the relation

$$\log \prod_i x_i = \sum_i \log(x_i)$$

$$\prod_i x_i = \exp\left(\sum_i \log(x_i)\right).$$

Once all the calculations are completed on the log-scale without underflow or overflow, the result may be converted to the original scale using the exponent function, or in case where the final result triggers underflow or overflow, the result may be kept in log-scale. Note that this method does not work for negative numbers since logarithm is defined only for positive numbers.

For example, the following R code shows how to avoid underflow by keeping the result in logarithm scale (we use below the logarithm property $\log(a^b) = b \log a$).

```
print(3^(-800), digits = 22) # underflow
## [1] 0
print(log(3^(-800)), digits = 22) # log of underflow
## [1] -Inf
# avoiding underflow, keep result in log-scale
print(-800 * log(3), digits = 22)
## [1] -878.88983093448780437 03
```

The example below shows how to avoid underflow when multiplying several terms, some of which are very small. Results may be returned in the log-scale or in

the original scale. The practice of keeping track of a product of small numbers on a log-scale is very useful when computing probabilities over a large sample space.

```
print(3^(-600) * 3^(-100) * 3^(150), digits = 22) # underflow
## [1] 0
# avoiding underflow, returning results in log-scale
print(log(3) * (-600 - 100 + 150), digits = 22)
## [1] -604.2367587674604010317
# avoiding underflow, returning results in original scale
print(exp(log(3) * (-600 - 100 + 150)), digits = 22)
## [1] 3.83098017177260867676 1e-263
```

2.7 Assembly Language

The role of the CPU is to execute a program, composed of a sequence of simple instructions.

Definition 2.7.1 The set of possible instructions that the CPU can execute is called the instruction set or assembly language of the CPU. The binary encoding of these instructions is called the machine language of the CPU. The Assembler is a program that converts the assembly language instructions into machine language.

We will examine the CPU as it processes instructions one at a time. The instruction processing cycle proceeds along the following stages:

1. The contents of the memory pointed to by the program counter (Definition 2.3.1) are written to a part of the CPU, known as the instruction register.
2. The contents of the instruction register are decoded.
3. The CPU executes the instruction.
4. The program counter is incremented by the number of bytes corresponding to the instruction.

2.7.1 Memory Addresses

Most modern computers are byte-addressable, which means that each memory byte has its own sequential address. The address of the first byte is 0, the second is 1, and so on. Memory addresses are encoded in binary form using the binary encoding for unsigned integers in Sect. 2.6.1 using k bits.

Note that in order to be able to express all bytes in memory we have the following constraint

$$B \leq 2^k,$$

where B is the number of bytes in memory. For example, 32-bit computers encode addresses using $k = 32$ bits, implying that an instruction cannot point to memory addresses beyond $2^{32} - 1$ or 4 gigabytes. This motivated the move from 32-bit computers to 64-bit computers that encode addresses using $k = 64$ bits and can refer to memory larger than 4 GB.

The sequential memory addresses from 0 to $B - 1$ are sometimes referred to as physical memory addresses. Programmers typically refer to addresses that are relative to the program counter (Definition 2.3.1), or to virtual memory addresses. It is the task of the operating system and the compiler to translate these addresses into physical memory.

2.7.2 Instruction Set

The list below outlines the most common assembly language instructions.

- Read the content of a memory address and write it to another memory address.
- Read the content of a memory address and write it to a register.
- Read the content of a register and write it to a memory address.
- Add or subtract two signed or unsigned integers located in memory or registers. Write the result to memory or to a register.
- Operate logical AND or OR on two sequences of bits located in memory or registers and write the result to memory or to a register. For example

```
00110101 AND 00001111 = 00000101.
```

- Add, subtract, or multiply the values encoded by two sequences of bits in memory, and write the result to memory.
- Shift bits to the left or right of a sequence of bits located in memory or a register and write the result to memory or a register. For example,

```
SHIFT-LEFT(00010001)  = 00100010.
```

- Set the program counter (Definition 2.3.1) to a new memory address, transferring program execution to a different code segment.
- Set the program counter to a new memory address if two memory addresses contain identical numbers, or alternatively if the number in the first memory address is larger than the number in the second memory address.

While all the instructions above constitute simple operations, a carefully planned sequence of such operations may result in a powerful computer program. The precise set of instructions and their binary encoding differs from one CPU to the next.

Below is a partial list of instructions of a typical CPU. We will explore a few simple programs in this assembly language.

- `MOVM A1 A2`: read data in memory address `A1` and write it to memory address `A2`.
- `MOVR A1 R1`: read data in memory address `A1` and write it to register `R1`.
- `MOVD D A1`: write data `D` to memory address `A1`.
- `ADD A1 A2 A3`: read data in memory addresses `A1` and `A2`, interpret them as unsigned integers, add them, and store the result in `A3`.
- `SUB A1 A2 A3`: read data in memory addresses `A1` and `A2`, interpret them as unsigned integers, subtract them, and store the result in `A3`.
- `MUL A1 A2 A3`: read data in memory addresses `A1` and `A2`, interpret them as unsigned integers, multiply them, and store the result in `A3`.
- `INC A1`: read data in memory addresses `A1`, interpret it as unsigned integers, add one to it, and store the result back in `A1`.
- `JMP A1`: set the program counter to address `A1`.
- `CMJ A1 A2 A3`: set the program counter to address `A3` if the data in addresses `A1` and `A2` are equal.
- `CMN A1 A2 A3`: set the program counter to address `A3` if the data in addresses `A1` and `A2` are different.

We assume, for the sake of simplicity, that both data and addresses are stored as a single byte. Since every instruction above has at most three arguments, we encode each instruction using four bytes: one to represent the instruction type itself (0 for `MOVM`, 1 for `MOVR`, 2 for `MOVD`, and so on until 7 for `CMJ`), and the remaining three bytes to represent potential arguments (note that all numbers must be encoded in binary form).

The instruction `MOVD 13 3` may be encoded as follows:

```
00000010 00001101 00000011 00000000
```

with the first byte encoding 2 for the `MOVD` operation (as the third instruction in the sequence of possible instruction), the second byte encoding 13, the third byte encoding 3, and the last byte encoding zero (unused). Assuming that the instruction above is stored starting at the address byte 128, the memory content of the four bytes starting at 128 is as follows.

```
decimal  binary
address  address            content      interpretation
-------------------------------------------------------
128      10000000           00000010     MOVD
129      10000001           00001101     13
130      10000010           00000011     3
131      10000011           00000000     0
```

The following command sequence increments the content of memory address 10 by one and then starts executing the instructions stored at memory address 128.

```
INC 10
JMP 128
```

Assuming the two instructions are stored in memory starting at address 128, the corresponding binary encoding appears below.

```
decimal  binary
address  address            content      interpretation
-------------------------------------------------------
128      10000000           00000100     INC
129      10000001           00001010     10
130      10000010           00000000     0
131      10000011           00000000     0
132      10000100           00000111     JMP
133      10000101           10000000     128
134      10000111           00000000     0
135      10001000           00000000     0
```

This program will execute an infinite loop that repeatedly adds one to the contents of memory address 10 (at some point an overflow will occur).

As the CPU executes the program above, the memory content will change as follows: We assume that memory address 10 contains the unsigned integer 7.

The initial memory content is

```
address  address (binary)   content      interpretation
-------------------------------------------------------
10       00001010           00000111     7
.
.
128      10000000           00000100     INC
129      10000001           00001010     10
130      10000010           00000000     0
131      10000011           00000000     0
132      10000100           00000111     JMP
133      10000101           10000000     128
134      10000111           00000000     0
135      10001000           00000000     0
.
.

Program counter: 128
```

The memory content is modified after the first instruction is executed (note the change in the value of the program counter indicating the address containing the next instruction).

```
address  address (binary)   content      interpretation
-------------------------------------------------------
10       00001010           00001000     8
.
.
128      10000000           00000100     INC
129      10000001           00001010     10
130      10000010           00000000     0
131      10000011           00000000     0
132      10000100           00000111     JMP
133      10000101           10000000     128
134      10000111           00000000     0
135      10001000           00000000     0
.
.

Program counter: 132
```

The memory content after the execution of the next instruction appears below.

```
address  address (binary)   content      interpretation
-------------------------------------------------------
10       00001010           00001000     8
.
.
128      10000000           00000100     INC
129      10000001           00001010     10
130      10000010           00000000     0
131      10000011           00000000     0
132      10000100           00000111     JMP
133      10000101           10000000     128
134      10000111           00000000     0
135      10001000           00000000     0
.
.

Program counter: 128
```

After the execution of the next instruction the memory content is as follows.

```
address  address (binary)   content      interpretation
-------------------------------------------------------
10       00001010           00001001     9
.
.
128      10000000           00000100     INC
129      10000001           00001010     10
```

```
130      10000010          00000000     0
131      10000011          00000000     0
132      10000100          00000111     JMP
133      10000101          10000000     128
134      10000111          00000000     0
135      10001000          00000000     0
.
.

Program counter: 132
```

The following assembly program computes 2^4 and writes the result to the display, which is mapped to the memory address 0. We omit below the binary encoding of the instructions and addresses for readability purposes.

```
address instruction     comment
-------------------------------------------------------
128     MOVD 3, 8       store 3 (counter) in memory
132     MOVD 2, 9       store 2 (multiplier) in memory
136     MOVD 2, 10      store 2 (result) in memory
140     MOVD 0, 11      store 0 in memory
144     MOVD 1, 12      store 1 in memory
148     MUL  9, 10, 10  multiply temp result by number 1
152     SUB  8, 12, 8   reduce counter
156     CMN  8, 11, 148 if counter is not 0, repeat
160     MOVM 10, 0      write result to display
                        (address 0)
```

Note that memory addresses in the vicinity of address 128 store the instructions that are being executed (the program), while memory addresses that are being used by the program are in the range 0-10. It's essential to keep these two memory regions separate to prevent a program from overwriting its own instructions.

It is hard to imagine how assembly language programs are capable of the complex behavior we attach to computers. For example, how can a sequence of addition, subtraction, and memory movements implement a computerized chess player, a complex search engine, or a language translation tool. Furthermore, even if a long sequence of assembly language instructions can create such complex behavior, it is hard to see how a programmer can design such a sequence.

The answer to the first question is based on two observations: (a) it is possible to create complex computer behavior with high-level languages like C++, and (b) each C++ instruction is translated into a sequence of assembly language instructions. Taken together, the two observations above imply that a complex C++ program is equivalent to a longer sequence of assembly language instructions. In fact, it is that longer sequence of assembly language instructions that are actually executed by the CPU and not the original C++ program.

The answer to the second question is related to the first answer above. It is extremely hard for a programmer to write complex programs in assembly language.

Fortunately, high-level computer languages like C++, Java, Python, and R exist, and the programmers can concentrate on expressing their program in these languages. A separate program, called the compiler (in the case of C++, for example), converts the high-level code into a long sequence of assembly language instructions that is then executed by the CPU.

The following C++ code prints the result of 2^4.

```
std::cout << pow(2,4);
```

It may be converted by the compiler to a sequence of assembly language instructions similar to the above assembly language examples. Obviously, this single line in C++ is much easier for a programmer to write than the corresponding assembly language code.

Use in High-Level Languages

High-level languages haven't been able to render programming in an assembly language completely obsolete; it's still required in situations when high-level languages don't fit the bill (e.g., real-time systems where every microsecond counts). Many high-level languages support embedding assembly instructions using a compiler feature called the inline assembler. This technique allows developers to hand-craft highly optimized assembly code for critical routines, while enjoying the conveniences of high-level languages otherwise; here's an example in C++ (see Sect. 2.4.2 to learn about the instruction used below):

```
#include<iostream>

int main() {
  uint timeStampCounter = 0;
  asm("rdtsc": "=a" (timeStampCounter));
  std::cout << timeStampCounter << std::endl;
  return 0;
}
```

The code listed above simply executes the RDTSC assembly instruction, which reads the CPU's time stamp counter then stores the higher 32 bits into the EDX register and the lower ones into the EAX register; the value of EAX is then assigned to the `timeStampCounter` variable in C++ and printed out.

2.8 Interrupts

The model described above implies that the CPU executes instructions sequentially. This model is problematic in that it does not enable input and output devices to impact program execution. Interrupts are a mechanism that facilitates the interaction of input and output devices with the CPU.

Interrupts are signals, sent from input and output devices to the CPU, that suspend the execution of the current program. The program counter is set to an address containing the interrupt handler, which is a sequence of instructions designed to handle the input or output device. Once the interrupt signal arrives, the CPU executes the interrupt handler routine and afterwards it resumes the execution of the original program.

The input and output devices typically pass information to and from the CPU through the memory. For example, once a keyboard key is pressed, the device writes the information directly to memory and informs the CPU via an interrupt signal. The CPU then executes the interrupt handler, reads the information containing the keyboard key, decides if additional action is needed, and then resumes execution of the original program.

2.9 The Memory Hierarchy

The speed with which the CPU reads data from memory and writes to memory is critical. A fast CPU with slow memory access would execute most programs slowly as the memory access forms a computational bottleneck.

The term memory hierarchy refers to a sequence of memory devices progressing from larger, slower, and cheaper to smaller, faster, and more expensive. The following list shows the most common devices, sorted by CPU access time (see also Fig. 2.2).

1. Registers
2. Cache memory
3. RAM
4. Solid state disk
5. Mechanical hard disk
6. Optical disk
7. Magnetic tape

Obviously, it is desirable to have the faster devices in the upper memory hierarchy as large as possible and dispense with the slower devices in the lower levels. The problem is that the upper levels in the memory hierarchy are restricted in their size due to manufacturing costs and physical constraints.

Registers are located inside the CPU and are capable of storing several bytes. Due to their proximity to the CPU they have very fast access speed but are very small in terms of how many bits they can hold (for example, 4 bytes).

Cache memory is located in the periphery of the CPU and is typically capable of storing thousands or millions of bytes. The cache memory is substantially larger than registers but is also slower to access. There are several levels of cache, denoted L1, L2, and so on, sorted in order of decreased proximity to the heart of the CPU and consequentially of decreased speed and larger sizes. For example, a typical 2017

laptop computer may have 64 KB L1 cache, 256 KB L2 cache, and several MB of L3 cache.

The RAM is a sequence of chips located outside of the CPU. The location outside of the CPU explains its slower access time and also its potentially larger size. A typical 2018 laptop computer has between four and sixteen GB.

Hard disks are located outside of the CPU and have slower access than RAM. Solid state disks are faster than mechanical disks since they do not require the mechanical alignment of the disk head with the appropriate location. Disks are larger than RAM per dollar cost, with solid state disks holding less per dollar cost than mechanical hard disks. A typical 2017 laptop computer may have 1-2 TB (terabyte) of mechanical disk space or 512 GB of solid state disk space.

At the bottom of the memory hierarchy are the optical disk and magnetic tapes. Both require mechanical action to operate and thus have relatively slow access speed. On the other hand, their technology allows large storage spaces for relatively low cost. These devices are typically used to archive data that is not frequently used or to back-up data.

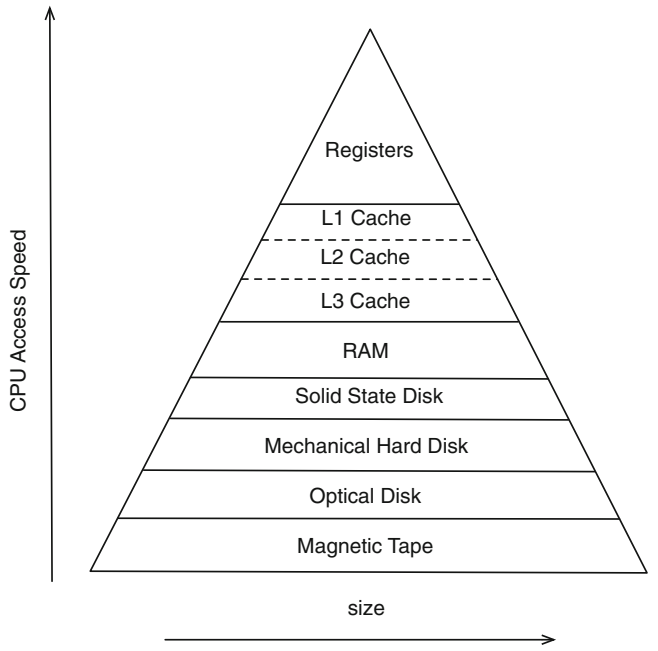


Fig. 2.2 The memory hierarchy on a typical personal computer. The y axis represents CPU access speed and the x axis represents typical size due to manufacturing costs and physical constraints

Effective allocation of information to the memory hierarchy components depends on the usage frequency of that information. For example, contents that are rarely accessed may be safely kept at the bottom of the memory hierarchy (optical disks

and magnetic tapes can be used for backing up data and are thus used only if the original data is accidentally deleted or corrupted). On the other hand, contents that are frequently accessed by the CPU should be kept at top hierarchy levels such as the cache. In this case, the total memory access time will be relatively low.

The usefulness of the memory hierarchy rests on the assumption that we can predict what data or memory addresses will be referenced more frequently. The two principles below motivate this assumption.

Temporal Locality: If a particular memory location is referenced, then it is likely that the same location will be referenced again in the near future.

Spatial Locality: If a particular memory location is referenced, then it is likely that nearby memory locations will be referenced in the near future.

We focus in this section on the relationship between the cache and the RAM that is managed by the computer hardware. The relationship between the RAM and the disk (virtual memory) is managed by the operating system software and is described in the next chapter.

2.9.1 Cache Structure

The cache memory is divided into units called rows. Each row is a sequence of bits divided into three groups: tag bits, data bits, and flag bits. The data bits contain the portion of the RAM memory whose address is indicated by the tag bits. The flag bit indicates whether the cache row is clean (replicates the contents of the RAM) or dirty, in which case its content is more up to date than the corresponding RAM address. See Fig. 2.3 for an illustration of a cache and corresponding physical memory.

When the CPU attempts to read or write a memory address, the cache checks whether it contains the relevant memory address by comparing the address to the tags in the cache rows. A cache hit occurs if the requested address matches the tag field of one of the cache rows. Otherwise we have a cache miss.

A memory read request may result in a cache hit or a cache miss. If there is a hit, the appropriate cache row is passed to the CPU for processing. If there is a miss, the appropriate memory content is read from the RAM and written to the cache, overwriting one of the existing rows, and then passed on to the CPU.

A memory write may similarly result in a cache hit or a cache miss. If there is a cache hit, the cache gets updated with the new memory content. In some cache strategies, the RAM gets updated immediately as well, ensuring that the cache contents are up to date with the RAM. In other cache strategies, the RAM is not updated immediately, resulting in an up-to-date cache and an outdated RAM. In this case, the flag bit of the appropriate cache row is set to one, indicating lack of synchronization between the cache row and the RAM address (see Fig. 2.4). As

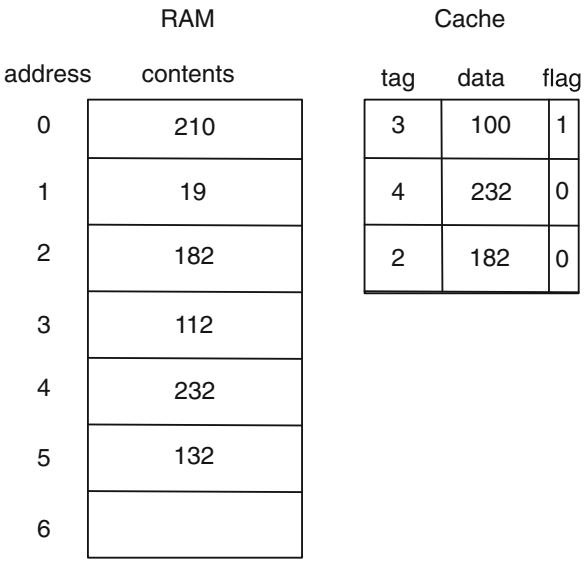


Fig. 2.3 The rows of the cache memory are divided into three groups of bits. The data bits contain the portion of the RAM memory whose address is indicated by the tag bits. The flag bit indicates whether the cache row is clean (replicates the contents of the RAM) or dirty, in which case its content is more up to date than the corresponding RAM address

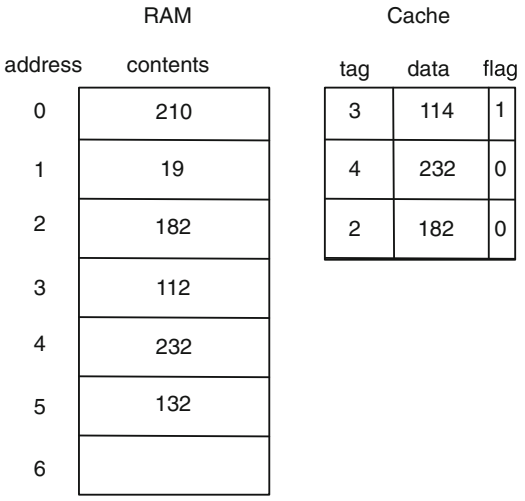


Fig. 2.4 Cache and memory corresponding to Fig. 2.3 following a CPU write operation to memory address 3. To save time, the RAM is not updated resulting in an updated cache but outdated RAM, denoted by a set flag bit

long as the modified cache row remains in the cache there is no compelling need to update the RAM (the CPU will deal with the updated cache row, rather than the outdated RAM). A problem arises, however, when a cache row with a flag bit of 1 is vacated from the cache to make space for a new cache row due to a read miss. In this case, the RAM is updated before the corresponding cache row is overwritten (Fig. 2.5).

2.9.2 *Direct Mapping and Associativity*

There are a number of different strategies to determine which cache row can hold which memory address.

In the case of direct mapping, each memory address may fit in precisely one cache row. An example of a direct mapping function is the modulo operation: the cache row is the remainder when the memory address is divided by the number of cache rows. Since the RAM is much bigger than the cache, each cache row is eligible to hold many memory addresses.

Recall that when the CPU accesses a memory address the cache checks whether the requested address is at one of the cache rows. The direct mapping scheme simplifies this process, since the tag of precisely one cache row needs to be checked in order to determine if there is a hit or a miss. A disadvantage of direct mapping is that it may result in frequent cache misses. For example, a program where the CPU repeatedly accesses two memory addresses mapped to the same cache row will result in repeated cache misses (the contents of the cache row will alternate between the contents of the two addresses). In this case the cache hit-to-miss ratio is low, regardless of the number of cache rows.

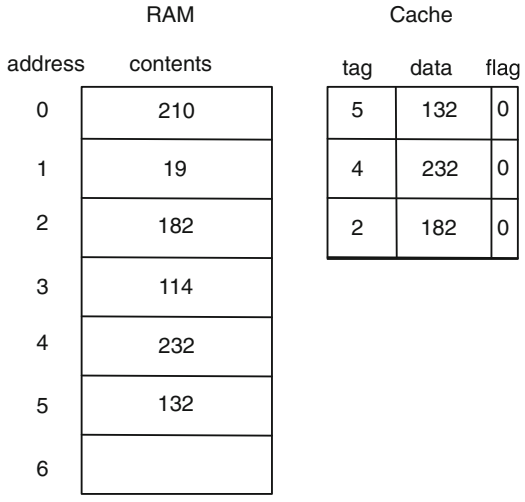


Fig. 2.5 Cache and memory corresponding to Fig. 2.4 following a CPU read operation at memory address 5. Assuming that memory address 5 gets mapped to the first cache row, the updated information needs to be written back to RAM as the new RAM overwrites the existing cache row

In contrast to direct mapping, fully associative mapping allows mapping any memory address to any cache row. The advantage over direct mapping is that the entire cache is used in any program, resulting in a high hit-to-miss ratio. A disadvantage is that checking whether the cache contains a specific memory address requires comparing the address to the tag bits of every single cache row.

A more general strategy that includes both direct mapping and fully associate mapping as special cases is k-associative mapping. In k-associative mapping, each memory address may be mapped to k cache rows. If $k = 1$, k-associativity reduces to direct mapping and if $k =$ number of cache rows, k-associative mapping reduces to fully associative mapping. As k increases, the hit-to-miss ratio increases, but the time it takes the cache to determine whether it has specific memory addresses increases. The optimal value of k (in terms of program execution speed) depends on the size of the cache: a large cache leads to a small optimal value of k and small cache leads to a high optimal value of k.

2.9.3 Cache Miss

In the case of a cache miss on a read request, the corresponding memory address is stored in a cache row, overwriting existing content. In the case of k-associative mapping with $k > 1$, there are several cache rows where the new memory content may be stored. Selecting the best row (in terms of minimizing program execution time) requires predicting which of the possible k rows will not be accessed by the CPU in the near future.

The simplest cache miss strategy, called least recently used (LRU), stores the memory content in the cache row that contains the information that was least recently accessed by the CPU (out of the k possible rows). The underlying assumption in this case is temporal locality. More complex strategies may lead to improved prediction of which of the cache rows will not be required in the near future, and consequentially lead to less cache misses and faster execution time.

2.9.4 Cache Hierarchy

The description thus far assumed a single memory cache. In most cases, however, there are multiple cache levels (see Fig. 2.2) ordered from smallest and fastest (for example, L1) to larger and slower (for example, L3). The relationship between each two successive levels is similar to the relationship described above between the cache and the RAM. For example, a memory access may first trigger an attempt to read it from L1. If an L1-cache miss occurs, an attempt is made to read it from L2, and so on. The different cache levels may have different k-associativity and cache miss policies.

2.10 Multicores and Multiprocessors Computers

A significant trend during the past forty years has been the fabrication of faster and faster CPUs by manufacturing smaller scale transistors. This trend, called Moore's Law, has resulted in exponential growth in the number of transistors per chip and in the CPU clock rate (Definition 2.4.1). This trend has slowed down around the second decade of the twenty-first century as physical limitations make it increasingly hard to further minimize CPU components significantly. This has led to the popularity of multicore and multiprocessor technology as a tool to further speed up computer systems.

Multicore technology fits multiple CPU cores into a single CPU, with each core capable of executing a sequence of instructions independently of the other cores. The cores may have independent cache, shared cache, or partially shared. For example, each core may have its own L1 cache, but all cores may share the larger and slower L2 and L3 cache levels (see Fig. 2.6). A typical 2017 personal computer has 4–8 cores. It is expected that the number of cores in a typical desktop or laptop computer will continue to increase as multicore technology advances.

Multiprocessor technology fits multiple single-core or multicore CPUs into a single computer (see Fig. 2.7). Note that in both the multicore and multiprocessor cases, the RAM is shared between all cores and processors, creating potential synchronization difficulties. Multicore computers are cheaper, more energy efficient, and result in faster communication between computer components than multiprocessor computers. On the other hand, multiprocessor technology can scale up more easily than multicore technology leading to dozens or hundreds of processors. Scaling up the number of cores in a CPU is more difficult due to physical constraints.

As the number of processors and cores increases, parallel programming gains importance. Naive parallelism decomposes the computation to different parts and

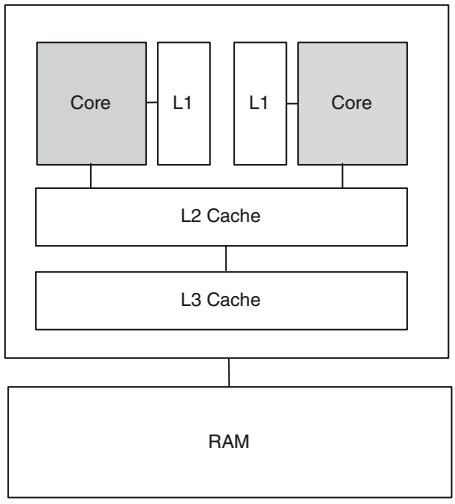


Fig. 2.6 An example of a single processor multicore system. The two cores have independent L1 caches but share the L2 and L3 caches and the RAM

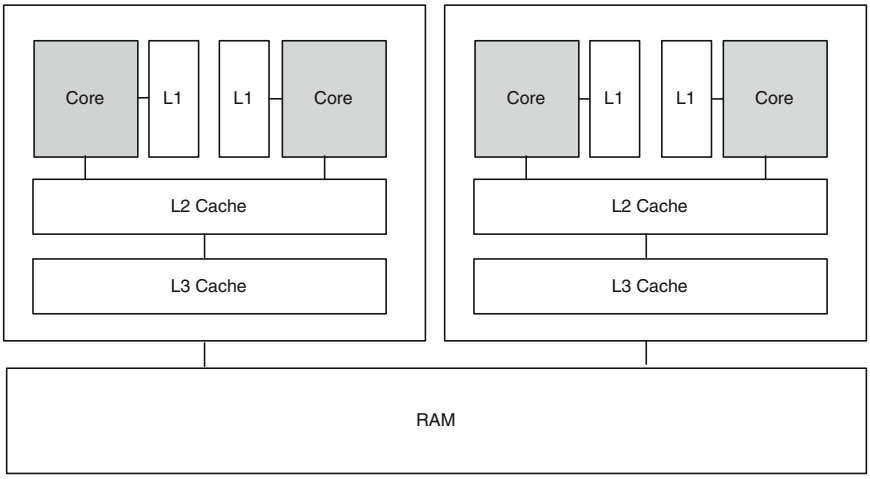


Fig. 2.7 An example of a multiprocessor multicore system. Both processors have independent cache but share the same RAM

assigns each part to a different core or processor. In some cases, the program is not easily decomposable as some part of the computation may depend on other parts of the computation. In these cases, it may still be possible to speed up the computation via parallelism, but steps need to be taken to ensure program correctness or alternatively that a reasonably accurate approximation is used.

2.11 Notes

Our discussion has been partial, focusing on concepts rather than details, and on issues that are relevant for computer programming and algorithms. Specifically, we have avoided discussing sophisticated CPU features such as pipeline and out of order execution, and sophisticated caching strategies such as branch prediction. With a few exceptions, we have avoided specifying precise figures such as the number of nanoseconds it takes the CPU to access RAM. These figures change significantly each year and the precise values are not as important as the underlying principles.

There are many textbooks containing additional details on computer hardware, with two popular choices being (Patterson and Hennessy, 2011) and (Hennessy and Patterson, 2011). Websites containing technical specification of hardware components are another useful source. Don Knuth's classic book (Knuth, 1997) and its update (Knuth, 2005) describe in detail an assembly language of an imaginary digital computer. Detailed descriptions of assembly languages of specific CPUs are available in technical manuals issued by the hardware manufacturer.

References

D. A. Patterson and J. L. Hennessy. *Computer Organization and Design*. Morgan Kaufmann, fourth edition, 2011.

J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, fifth edition, 2011.

D. E. Knuth. *Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison Wesley, third edition, 1997.

D. E. Knuth. *The Art of Computer Programming, Volume 1, Fascicle 1: MMIX - A RISC Computer for the New Millennium*. Addison Wesley, 2005.

Chapter 3
Essential Knowledge: Operating Systems

In Chap. 2, we discussed the CPU and how it executes a sequence of assembly language instructions. This suggests the following model: a programmer writes a computer program as a sequence of assembly language instructions, loads them into memory, and instructs the CPU to execute them one by one by pointing the program counter at the relevant memory address. Unfortunately, there are many problems with this scheme: only one program can run at any particular time, one programmer may overwrite information important for another programmer, and it is hard to reuse instructions implementing common tasks. The operating system mediates between the computer hardware and programmers, and resolves difficulties such as the ones mentioned above. This chapter describes the concepts of an operating system, then delves into the Linux and Windows operating systems in some detail as concrete examples; in addition, it explores command-line interfaces like bash, Command Prompt, and PowerShell, which are essential for developers to know intimately.

The main roles of the operating system (OS) are to:

1. manage concurrent execution of multiple processes,
2. manage presence of multiple programmers and multiple users,
3. manage disk storage using the file system,
4. manage memory allocation and access,
5. facilitate access to input and output devices,
6. protect against malicious or careless acts,
7. provide tools for reusing software in multiple programs, and
8. provide applications that help users accomplish many important tasks.

Throughout this chapter we use the Linux and Windows operating systems to illustrate operating system concepts. Most Linux commands and examples also work on macOS and Windows 10 (build 14316 or higher) via the Windows

© Springer Nature Switzerland AG 2018

G. Lebanon, M. El-Geish, *Computing with Data*,
https://doi.org/10.1007/978-3-319-98149-9_3

Subsystem for Linux (WSL) developer feature.[1] Most PowerShell commands should also work with little to no modification on Linux and macOS, thanks to the release of PowerShell Core in 2018.

3.1 Windows, Linux, and macOS

The three most popular operating systems for desktop and laptop computers at the early twenty-first century are Microsoft Windows, Linux, and macOS.

The Windows operating system is manufactured and sold by Microsoft, and is currently the most popular operating system for personal computers. It is intuitive and easy to use for users interested in running applications such as browsers, email clients, word processors, spreadsheets, and games.

Linux is an operating system that is especially popular for high performance servers, cell phones, and embedded devices. Unlike Windows, Linux is freely available and is open source, implying that no single organization owns it or sells it. Instead, diverse communities of programmers implement, maintain, and improve various flavors of it known as distributions.

The most popular distribution—for personal computers—is Ubuntu, which is available for free at http://www.ubuntu.com. Linux is known for being less intuitive and harder to install and use than Windows and macOS. Despite this reputation, substantial progress has been made over the years and recent versions of the major Linux distributions are easy to install and use. Most Linux distributions feature graphic interfaces that are similar to Windows, for example Gnome (http://www.gnome.org), KDE (http://www.kde.org), and Unity (http://unity.ubuntu.com).

Besides being free, a major advantage of Linux over Windows is that it provides substantial flexibility for programmers and it includes a large variety of programming tools. This makes Linux a more convenient software development platform than Windows; that said, in 2016, Microsoft added a new feature for Windows developers to support running Linux tools natively on Windows 10 (build 14316 or higher) and in 2018 released a version of PowerShell called PowerShell Core for Linux and macOS users. Apple's macOS is similar to Linux in many ways, but it is not freely available. Most Linux commands and examples work on the macOS as well, though there are some exceptions. In a nutshell, you may run bash and/or PowerShell Core on Linux, macOS, and/or Windows—it's a new world!

In this chapter and in the rest of this book we illustrate computing concepts mainly using the Linux operating system (unless otherwise stated). Most examples should work also on Windows and macOS, though some minor differences may occur.

[1] See https://docs.microsoft.com/en-us/windows/wsl/install-win10 for details.

3.2 Command-Line Interfaces

Command-line interfaces provide fast and powerful apparatuses to perform tasks that range from sorting a file to building and deploying a web service. The vast majority of software engineers depend on a command-line interface (CLI) to perform day-to-day operations, automate mundane tasks, etc. Getting acquainted with command-line interfaces—and the tools they provide—is a great skill to have when computing with data. In this section, we go over various command-line interfaces and showcase their capabilities.

3.2.1 The Linux Terminal and Bash

Introduction

The Linux terminal is a text-based application that lets users and programmers interact with different aspects of the operating system, including process management, file system, and user management.

The terminal may run in different modes, known as shells. Each shell has a different way of translating the user commands into operating system tasks. The default shell in most cases is the Bourne Again Shell (abbreviated bash). Zshell (abbreviated zsh), an alternative shell that has recently gained popularity, is compatible with bash and includes additional features. The examples in this chapter should work with either bash or zsh.

To start the Linux terminal program, launch the terminal or xterm application using the graphical interface. In some cases the Linux terminal appears right away, for example when logging-in to a remote server via the `ssh` program or when booting up a Linux computer without the graphic interface. To exit a terminal type `exit` at the terminal prompt and hit the return key or close the window of the terminal application.

Each terminal line contains one or more commands, separated by semicolons. As the return key is pressed, the terminal executes the commands and displays the corresponding output. More specifically, as each command that corresponds to an executable file[2] is executed, the shell finds its respective executable file in the file system (see Sect. 3.6), and executes the instruction sequence in that file.

The bash shell ignores all characters between a # symbol and the end of the line. This provides a useful way to annotate Linux commands with comments. The first Linux command we examine is `echo`, which displays its arguments followed by a newline character. For example, the following command displays the two arguments `hello` and `world` in the Linux terminal. Note how the comment following the # character is ignored.

[2]Some commands, like the `exit` command, are built into the bash executable.

```
# this is a comment
# the prefix ## indicates an output line
# the echo command displays its arguments
# three commands separated by a semicolon
echo hello world; echo hello; echo world
## hello world
## hello
## world
```

Above and elsewhere in this section, we prefix each output line by ## followed by a blank space; this facilitates copying the example above and pasting it into a terminal (the text following ## is ignored and thus may be included in the copied commands).

Many terminal commands accept one or more flags (options), indicated by a dash or two consecutive dashes followed by a text string. For example, the `-n` flag instructs `echo` to avoid printing a newline character at the end.

```
echo hello; echo world; echo -n hello; echo -n world
## hello
## world
## helloworld
```

Some flags can be combined together to indicate a combination of their respective effects:

```
# cat concatenates and prints files
# -e displays $ at the end of each line
# -n displays the line number
# "cat -en" and "cat -e -n" are equivalent
cat -en hello.txt
##     1 hello$
##     2 world$
```

In some cases, options are followed by arguments; they are also known as named arguments, which can be specified on the command-line in any order, since they are identified by their names instead of position. Unnamed arguments, also known as positional arguments, need to be specified in the order the command expects. For example, the command below sends the file `foo.txt` to the printer named `printer101`; the string `foo.txt` is an unnamed argument of the command `lp` and the string `printer101` is an argument of the option `-d` (printing and files will be described later on in this chapter):

```
# printer101 is the value of the destination option
# foo.txt is the value of the first unnamed argument
lp -d printer101 foo.txt
```

In some cases, the arguments of flags are optional; if the arguments are not explicitly supplied, they assume default values. All kinds of command options covered above can be combined together, unless otherwise specified by the command.

The commands `man` and `info` display detailed description of Linux commands. It is customary to denote in this description optional arguments using square

brackets. To scroll up or down when viewing the `man` and `info` descriptions press the up or down arrow keys, and to quit the viewer type `q`.

For example, the following commands display the `man` and `info` information for the `echo` command. We display below only the first few lines of the description pages.

```
man lp

## NAME
##       lp - print files
##
## SYNOPSIS
##       lp  [  -E ] [ -U username ] [ -c ]
## (truncated for brevity)
```

Flags typically have different meaning in different commands. But in some cases, flags preserve their meanings across multiple commands. Some examples of flags that preserve their meaning across multiple commands are listed below.

Flag	Effect
`--help`	display command description
`--version`	display command version

The Linux example below demonstrates the `-version` flag across two different Linux command.

```
# usage of --version flag across multiple commands
man --version
## man 2.6.1
ls --version
## ls (GNU coreutils) 8.13
```

Variables

To set a variable, we assign it a value with the assignment operator =. Most character strings are legal variable names, provided special characters such as `$` or `&` are avoided. Linux is case sensitive, and thus `apple` and `Apple` represent different variables. When the `$` symbol prefixes a variable name, the value of the variable is referenced.

```
a=3; echo a; echo $a
## a
## 3
```

Environment variables, typically written with uppercase characters, are special variables that influence the behavior of the shell. For example, the `SHELL` variable holds the name of the shell that is currently active.

```
# type the active shell program
echo $SHELL
## /bin/bash
```

The command `export` modifies the value of an environment variable. The difference between it and the simpler assignment operator above is that `export` assigns a global variable, making it accessible to other active shells; `printenv` is useful for printing the values of all global environment variables, such as the shell and the user name.

The following command modifies the value of the `PS1` global environment variable, which determines the appearance of the prompt. We assume below that the initial prompt is a `>` symbol. For more details on how to set `PS1` see the PROMPTING section of the `man bash` document or online documentation.

```
> export PS1="% "
% export PS1="$ "
$ export PS1="\\u@\h \\W>"  # set prompt to user@host dir>
joe@myLaptop ~>
```

Pipes

A Linux pipe, denoted by the character `|`, is used to join a sequence of commands where the output of the first command becomes the input of the second command, the output of the second command becomes the input of the third command, and so on.

The following example shows a piped combination of the `echo` command, and the `wc` command, which prints the number of lines, words, and characters in its input (see `man wc` for more information and for a description of its optional flags).

```
echo a sentence with 8 words and 42 characters
## a sentence with 8 words and 42 characters


# count lines, words, characters in the following sentence
echo a sentence with 8 words and 42 characters | wc
##       1       8      42
```

We can feed the output of the piped combination above to another `wc` command.

```
echo a sentence with 8 words and 42 characters | wc | wc
##       1       3      25
```

Appending additional `| wc` at the end of the command above will not modify the displayed output. Do you see why that is the case?

```
echo a sentence with 8 words and 42 characters | wc | wc | wc
##       1       3      25
```

Scripting

Loops

The bash shell allows substituting a variable for a command, using the back quote symbol `` ` ``. The command surrounded by back quotes is executed and its output is substituted in its place. The example below uses the command `seq`, which creates a string of numbers between its two arguments (type `man seq` for more details).

```
seq 1 5
## 1
## 2
## 3
## 4
## 5


echo seq 1 5
## seq 1 5


echo `seq 1 5`
## 1 2 3 4 5
```

Bash shell supports three types of loops: for-loops, while-loops, and until-loops.

For-loops set a variable in each iteration to a different word within a string containing multiple words. The number of iteration is equal to the number of words in the corresponding text string. The `for` statement is followed by a semicolon and then `do` followed by a potential command or command sequence. Another semicolon and the `done` keyword mark the end of the loop.

```
for i in `seq 1 5`;  do echo the current number is $i; done
## the current number is 1
## the current number is 2
## the current number is 3
## the current number is 4
## the current number is 5
```

Fig. 3.1 Logical conditions in Linux, C/C++/R, and mathematical notation

bash	C/C++/R	math notation
`-lt`	`<`	$<$
`-gt`	`>`	$>$
`-le`	`<=`	$\leq$
`-ge`	`>=`	$\geq$
`-eq`	`==`	$=$
`-ne`	`!=`	$\neq$

Parentheses start a new sub-shell, execute the commands inside the parentheses, and return to the original shell with the corresponding output.

```
for i in `seq 1 5`;  do (echo the current number is $i| wc);
done
##         1       5      24
##         1       5      24
##         1       5      24
##         1       5      24
##         1       5      24
```

While-loops iterate until the condition specified inside square brackets is achieved. In the example below, we use the command `let` for performing arithmetic on shell variables and use `-lt` to refer to the logical less-than condition. See Fig. 3.1 for bash keywords corresponding to other standard logical operators. These keywords are needed since the symbols < and > are reserved in Linux for input and output redirection (see Sect. 3.8).

```
i=0; while [ $i -lt 5 ]; do  echo $i; let i=i+1; done
## 0
## 1
## 2
## 3
## 4
```

Until-loops are similar to while-loops, but they iterate until the corresponding condition is achieved.

```
i=0; until [ $i -ge 5 ]; do echo $i; let i=i+1; done
## 0
## 1
## 2
## 3
## 4
```

Conditional Logic

The `if` statement executes the sequence of commands between the `if` and `fi` keywords if the `if` condition holds. An optional `else` keyword prefixes a command that executes if the condition does not hold.

```
a=4 ; b=3 ; if [ $a -eq $b ] ; then echo 1 ; else echo 0 ; fi
## 0
```

Logical operators can be used to rewrite the above example:

```
a=4 ; b=3 ; [ $a -eq $b ] && echo 1 || echo 0
## 0
```

The following expressions are commonly used with bash:

Expression	Description
`[ -a PATH ]`	checks if the specified path exists
`[ -e PATH ]`	same as the above
`[ -d PATH ]`	checks if the specified path exists and is a directory
`[ -f PATH ]`	checks if the specified path exists and is a file
`[ -h PATH ]`	checks if the specified path exists and is a symbolic link
`[ -r PATH ]`	checks if the specified path exists and is readable
`[ -s PATH ]`	checks if the specified path exists and not empty
`[ -w PATH ]`	checks if the specified path exists and is writable
`[ -x PATH ]`	checks if the specified path exists and is executable
`[ -z STRING ]`	checks if the length of the specified string is zero
`[ STRING ]`	checks if the length of the specified string is not zero
`[ X == Y ]`	checks if the two strings are equal (case-sensitive)
`[ X != Y ]`	checks if the two strings are not equal (case-sensitive)
`[ X < Y ]`	checks if `X` ranks before `Y` lexicographically
`[ X > Y ]`	checks if `X` ranks after `Y` lexicographically
`[ ! X ]`	checks if `X` is false
`[ X -a Y ]`	checks if both `X` and `Y` are true
`[ X -o Y ]`	checks if either `X` or `Y` is true

A Few More Tips

Brace expansion is a useful mechanism for generating a collection of strings with a specific pattern. The brace `{X,Y}` expands to `X` and to `Y` and the brace `{X..Y}` expands to all characters in between `X` and `Y`.

```
echo {b,c}; echo a{b,c}d; echo a{a..m}d
## b c
## abd acd
## aad abd acd add aed afd agd ahd aid ajd akd ald amd
```

The shell records the executed commands for future recall. This is convenient when typing long commands containing a typo that need to be fixed, or for repeated execution of common commands. The `history` command displays all recorded commands, sorted in chronological order. An optional numeric argument `k` shows only the `k` most recent commands. Most terminals support navigating through the command history using the up-arrow and down-arrow keys. More specifically, an up-arrow keystroke brings up the most recent command. Another up-arrow keystroke brings up the second most recent command, and so on.

The exclamation mark `!` is a convenient way to recall the most recent command starting with a certain prefix. For example `!ec` re-executes the last command that started with the phrase `ec`.

Another useful way to search through the command history is the key combination CTRL+r followed by a string. This brings up the most recent command matching the specific string as a sub-string (not necessarily a prefix substring). In contrast to the exclamation mark technique mentioned above, the CTRL+r key combination lets the user modify the recalled command before it is executed.

A convenient way to abbreviate long commands is by defining an alias using the command `alias X=Y`. In this case, whenever the command `X` is executed, the shell substitutes it with the command `Y`. Prefixing a string with a backslash escapes any existing aliases and executes the typed command as is. The command `unalias` removes an alias.

```
date
## Mon Feb  6 15:50:34 EST 2015
alias date="date -u"  # modify format with optional flag -u
date     # alias in effect
## Mon Feb  6 20:52:39 UTC 2015
\date    # escape alias, original command in effect
## Mon Feb  6 15:52:52 EST 2015
unalias date  # remove alias
date
## Mon Feb  6 15:53:47 EST 2015
```

3.2.2 Command Prompt in Windows

The Windows operating system provides multiple text-based command-line applications (akin to the terminal in Linux) that enable advanced users to interact—mainly by typing commands—with different aspects of the operating system and other applications.

In 1981, Microsoft released an operating system for personal computers called Microsoft Disk Operating System (MS-DOS); it had a command-line interface (CLI), and no graphical user interface (GUI). Thanks to MS-DOS, Microsoft became known to not only programmers, but also to users of personal computers. Microsoft built early versions of Windows (up to Windows 98) as a GUI for MS-DOS[3]; more recent versions of Windows have been shipping with an command-line interpreter, which supports MS-DOS commands, called Command Prompt.

To start Command Prompt, open the Start menu and search for cmd; the search results should contain an entry for Command Prompt since its executable file is called cmd.exe, which replaces COMMAND.COM[4] in MS-DOS. A command-line interface (CLI) is the only user interface available out of the box on Windows Server

[3] http://windows.microsoft.com/en-us/windows/history.

[4] .COM as in "command," which is a file extension for text files that contain a batch of executable commands; it has nothing to do with .com as in "commercial," the commonly used top-level Internet domain name. Don't mistake COM files for web bookmarks; that's a security hazard that recent versions of Windows try to mitigate by notifying you when a program requests administrator-level permission to run.

Core editions, which are provided as a stripped-down alternatives of Windows Server editions to improve performance, security, service footprint, and feature selectivity.[5] As Windows Server Core (vis-à-vis a full installation of Windows Server) sounds like the wiser choice for large-scale systems, we believe that it pays off to be a proficient Command Prompt user, since there's no Windows Explorer shell when you log-in to the operating system on Server Core machines.

Command Prompt is a command-line interpreter; each Command Prompt line contains one or more commands, separated by ampersands (unlike the Linux terminal, which uses semicolon as a delimiter, Command Prompt uses semicolon as a delimiter for command parameters—like a blank space). As the return key is pressed, Command Prompt executes the commands and displays the corresponding output. Similar to the Linux terminal, as each command that corresponds to an executable file[6] is executed, Command Prompt finds its respective executable file in the file system (see Sect. 3.6), and executes the instruction sequence in that file.

Unlike Linux, file names and commands in Windows (and macOS) are case-insensitive for the most part[7]; For example, the following commands are interchangeable: `whoami`, `WhoAmI`, and `whOamI`; Command Prompt finds the executable file that corresponds to a command using a case-insensitive search. Moreover, Command Prompt correctly displays files and directories that coexist in the same directory and have names that only differ in case. That said, the standard Command Prompt commands—that ship with Windows out of the box—fail to process such files and directories when passed as parameters. In the examples below, we will demonstrate commands typed in lowercase to maintain consistency with the Linux terminal's examples.

The `rem` command records comments in a batch (script) file and in Command Prompt. To demonstrate, let's build a hello world example using the `echo` command, which displays its arguments followed by a newline character; the following command displays the two arguments `hello` and `world` in Command Prompt. Note how the comment following the `rem` command is ignored:

```
rem This is a comment
rem The prefix ## indicates an output line
rem The echo command displays its arguments
echo hello world
## hello world
##
```

Above and elsewhere in this section, we prefix each output line by `##` followed by a blank space. This facilitates copying the example above and pasting it into

[5] https://msdn.microsoft.com/en-us/library/dd184075.aspx.

[6] Some commands, like the `exit` command, are built into Command Prompt.

[7] The Hierarchical File System Plus (HFS+), which is widely used by macOS, is case-insensitive; while the New Technology File System (NTFS), which is commonly used by recent versions of Windows, is case-sensitive; however, the vast majority of applications that run on Windows treat file names with case-insensitivity. For more details, see https://support.microsoft.com/en-us/kb/100625.

Command Prompt (the text following `rem` is recorded as a comment and thus may be included in the copied commands). We recommend to enable the QuickEdit Mode, from the Command Prompt properties dialog, to paste commands with a simple secondary click with the mouse.

To run multiple commands on the same line, chain them using ampersand as a delimiter:

```
echo hello & echo world
## hello
## world
##
```

The vast majority of commands provide flags (options) that the user can specify, instructing a command to execute a variation of its default behavior. Flags to commands are akin to those used to be waved at train drivers back in the day to change tracks. Here's an example using the `dir` command, whose default behavior is printing the current directory's content:

```
rem The current directory is C:\test
rem The "." entry is a reference to the current directory
rem The ".." entry is a reference to the parent directory
dir
##  Volume in drive C has no label.
##  Volume Serial Number is D0A0-D665
##
##  Directory of C:\test
##
## 04/30/2016  01:23 PM    <DIR>          .
## 04/30/2016  01:23 PM    <DIR>          ..
## 04/30/2016  01:22 PM                42 bar.txt
## 04/30/2016  01:22 PM                 3 foo.txt
##                2 File(s)             25 bytes
##                2 Dir(s)  50,272,501,760 bytes free
##
```

A variation of the `dir` command prints the contents of the current directory without any additional information:

```
rem Options are specified using a slash
rem The /b option instrcuts dir to use a bare format
dir /b
## bar.txt
## foo.txt
##
```

Some options can be combined together to indicate a combination of their respective effects; here's an example that lists all directory contents, including hidden files and directories, in a bare format:

```
rem The /a option instrcuts dir to show all contents
rem The /b option instrcuts dir to use a bare format
dir /a /b
## bar.txt
```

```
## foo.txt
## hidden.txt
##
```

In some cases, options can have arguments; for example, the list of directory contents can be sorted using the sort order option:

```
rem The /a option instrcuts dir to show all contents
rem The /b option instrcuts dir to use a bare format
rem The /o:s option instrcuts dir to sort by file size
dir /a /b /o:s
## hidden.txt
## foo.txt
## bar.txt
##
```

Sort order arguments can also be combined (e.g., `dir /o:ns` to sort by file name and size), or omitted altogether; when the arguments of options are optional and their values are not explicitly supplied, they assume default values.

So how can one tell what options a command supports? There's an option for that, which displays the help message of almost every command out there:

```
rem The /? option instrcuts tree to show its help message
tree /?
## Graphically displays the folder structure of a drive
## or path.
##
## TREE [drive:][path] [/F] [/A]
##
##    /F   Display the names of the files in each folder.
##    /A   Use ASCII instead of extended characters.
##
```

The command `help` has a similar effect; it prints the help message of a given command:

```
help help
## Provides help information for Windows commands.
##
## HELP [command]
##
##     command - displays help information on that command.
##
```

Like the example above, arguments don't have to be named; optional arguments are denoted by square brackets; otherwise, they are required. Named arguments can be specified on the command-line in any order, since they are identified by their names instead of position. Unnamed arguments, also known as positional arguments, need to be specified in the order the command expects. All kinds of command options covered above can be combined together, unless otherwise specified by the command.

Variables

Variables are crucial for scripting and writing commands that work on any Windows computer; for example:

```
rem To read a variable, use %VARIABLE_NAME%
echo %TMP%
## C:\Users\joe\AppData\Local\Temp
##
```

Although variable names are case-insensitive, it's customary to write them `CAPITALIZED_WITH_UNDERSCORES` (all characters are capitalized and words are separated by underscores; also known as `SCREAMING_SNAKE_CASE`). `TMP` is a built-in environment variable that points to the current user's temporary folder. When the operating system launches cmd.exe (the executable that is Command Prompt), it loads it with a local copy of the environment variables. Command Prompt can manipulate (read, write, and/or delete) any of these variables, but such changes only take effect within the cmd.exe process.

```
rem Use the set command to set, read, and remove a variable
set X=13
##
set X
## X=13
##
set X=
##
set X
## Environment variable X not defined
##
```

Executing the `set` command without any arguments lists all Command Prompt variables and their respective values.

The following example shows how to set the value of the `PROMPT` environment variable, which changes the text of the prompt:

```
C:\test>set PROMPT=%USERNAME%@%COMPUTERNAME% $P$G

joe@myLaptop C:\test>
```

Alternatively, you can use the `prompt` command; the help message shows details on how to customize the prompt.

Pipes

Pipes in Command Prompt are similar to those in the Linux world, which we discussed earlier in this chapter. Here's an example that shows a piped combination of the `dir` and `sort` command, which we used to list the content of a directory in a reverse alphabetical order:

```
dir /b | sort /r
## Videos
## Searches
## Saved Games
## Pictures
## NTUSER.DAT
## Music
## Links
## Favorites
## Downloads
## Documents
## Desktop
## Contacts
```

Scripting

Environment Scope

Command Prompt commands can be run as a batch using a batch file with the extension `.bat` or individually in the scope of the current Command Prompt environment. A batch file may include the command `SETLOCAL`, typically as the first command, to indicate that changes to the environment in said batch file after calling said command are local to that file and don't affect the parent Command Prompt environment. Conversely, the command `ENDLOCAL`—as its name suggests—marks the end of the file's local scope and restores the previous environment's settings; an implicit call of `ENDLOCAL` is made at the end of a batch file for any outstanding `SETLOCAL` call issued by said file.

Loops

For-loops are commonly used on Command Prompt to process the output of other commands, contents of a folder, etc. We use the `FOR` command to accomplish such tasks. It's important to note that the loop variable in `FOR` loops is case-sensitive; it's also prefixed with a single `%` when running the command on Command Prompt as opposed to using `%%` as a prefix when run in the context of a batch file. An integer counter can also be used for controlling the number of iterations; here we show an example that prints even numbers between 0 and 10 (inclusive):

```
for /L %i in (0, 2, 10) do @echo %i
## 0
## 2
## 4
## 6
## 8
## 10
##
```

In the above example, we print each value of `%i` without printing its respective `echo` command. Removing the `@` symbol will cause the `echo` command to repeat on the Command Prompt window for each iteration. Besides using a loop counter, another common use-case of the `FOR` command is looping over multiple values:

```
FOR %a IN (eggs milk bread) DO @echo %a
## eggs
## milk
## bread
##
```

Instead of typing the data to enumerate inline, they can be fed to the loop as lines in a text file:

```
(echo eggs & echo milk & echo bread) > lines.txt
for /f %i in (lines.txt) do @echo %i
## eggs
## milk
## bread
##
```

Multiple files can be specified in the filename set, whose combined content is going to be processed in order as if they are concatenated. A `FOR` loop is particularly useful in parsing text files; here's an example that reads comma-separated values:

```
echo eggs,milk,bread > data.csv
for /f "delims=, tokens=1-3" %i in (data.csv) do ^
@echo %i & @echo %j & @echo %k
## eggs
## milk
## bread
##
```

In the above example, we broke the command over two lines using the hat (`^`) symbol; in the for-loop, we specified the delimiters used to tokenize a line in the file (in this case, only the comma) and which tokens to read using a one-based index (from 1 to 3). Subsequently, variable %i (which was explicitly declared in the for statement) is assigned the first token of each line parsed, while %j and %k (which were implicitly declared) get the values for the second and third tokens, respectively.

Instead of reading the content of text files, a `FOR` loop can parse an immediate string (the output of a command to run) using single quotes to specify the command (instead of the input fileset); for example, we can parse the output of the `set` command to print out environment variable names:

```
FOR /F "delims==" %i IN ('set') DO @echo %i
## ALLUSERSPROFILE
## APPDATA
## CommonProgramFiles
## CommonProgramFiles(x86)
## CommonProgramW6432
## COMPUTERNAME
```

```
## ComSpec
## (truncated for brevity)
##
```

Alternatively, to execute a command and capture its output in memory to use as if it's an input file, specify the `usebackq` option and use a back-quoted string for the command line to execute:

```
FOR /F "usebackq delims==" %i IN (`set`) DO @echo %i
## ALLUSERSPROFILE
## APPDATA
## CommonProgramFiles
## CommonProgramFiles(x86)
## CommonProgramW6432
## COMPUTERNAME
## ComSpec
## (truncated for brevity)
##
```

We strongly suggest reading more about the `FOR` command by running `help for` or `for /?` in your Command Prompt.

Conditional Logic

Like bash, Command Prompt supports `IF` commands and `&&` operators to allow for conditional execution of commands. The example below shows a simple use-case of the `IF` command:

```
set X=10
IF %X% EQU 10 echo X is 10
## X is 10
##
```

Other numeric comparators can be used: `NEQ` for not equal, `LSS` for less than, `LEQ` for less than or equal, `GTR` for greater than, and `GEQ` for greater than or equal. To check for string equality, we use the `==` operator instead, which can be combined with the NOT option to check for inequality; here's an example:

```
set Y=bar
IF NOT %Y%==foo echo Y is not foo
IF NOT a==A echo case-sensitive
IF /i a==A echo now equal
## Y is not foo
## case-sensitive
## now equal
##
```

There are many useful checks the `IF` command can perform; the example below demonstrates a few of them:

```
IF NOT DEFINED SOME_VAR echo SOME_VAR is not set
IF NOT EXIST some_file.txt echo.>some_file.txt
IF NOT EXIST some_file.txt (echo.>some_file.txt) ^
ELSE echo file already exists
IF %ERRORLEVEL% EQU 0 echo last command succeeded
## SOME_VAR is not set
## file already exists
## last command succeeded
##
```

To learn more, we recommend reading the help article of the `IF` command by running `help if` or `if/?` in your Command Prompt.

3.2.3 PowerShell

PowerShell is a command-line shell that provides a richer feature set for power users. Thanks to built-in support for the .NET Framework, PowerShell can be a more appealing shell to .NET developers when compared to Command Prompt. At the first glance, PowerShell may look similar to other command-line shells since you can interact with it the same way: by typing commands and/or running scripts. Text-based shells—the ones we covered so far in this book—take text in and print text out; PowerShell takes in and returns back .NET objects—what you see on the command-line is a text representation of said objects.

To start PowerShell on Windows, open the Start menu and search for powershell; the search results should contain an entry for Windows PowerShell and another for Windows PowerShell ISE (an Integrated Scripting Environment); we will use the former in the examples below since it provides a user interface similar to that of the other shells we already covered in this book. You can also start PowerShell interactively or to run a PowerShell script within Command Prompt using the `powershell` command. On Windows Server Core editions, you may need to first install PowerShell using the Server Configuration utility (sconfig) from Command Prompt.[8]

PowerShell Core is a version of PowerShell that's made for Linux and macOS; to read about the differences between the two, see this article bit.ly/2Fruvsq; for setup instructions, follow the guide at bit.ly/2jyTv7I.

A PowerShell command is called a cmdlet (pronounced "command-let"); a cmdlet accepts input objects, executes the command, and returns objects either to the output stream or to the next cmdlet in the pipeline. Passing objects around—rather than text in other shells—removes the need to parse input and/or format output for the most part, especially for the purpose of inter-command communication, which helps scripts reduce the number of input/output bugs. A pipeline of cmdlets

[8] https://technet.microsoft.com/en-us/magazine/ff476070.aspx.

combined together can harness the power of such feature to execute a complex workflow of commands in a single line.

Unlike Command Prompt, PowerShell cmdlets are not executables—they are instances of .NET classes. This level of strongly typed consistency makes it easier for developers to use—and develop—cmdlets, knowing that their cmdlets will work harmoniously with other cmdlets (whether they shipped with Windows or were provided by fellow developers). In addition to cmdlets, PowerShell supports the execution of functions, scripts (`.ps1` files), and executable files. The built-in support for .NET comes in handy when examining the behavior of a .NET method in action; it's much faster to launch PowerShell and test said method interactively than doing so in an integrated development environment (IDE) like Microsoft Visual Studio.

Names of executables (cmdlets, functions, files, etc.) are case-insensitive in PowerShell. Cmdlets names almost always follow the `Verb-Noun` naming convention; in the examples below, we will demonstrate cmdlets typed in Pascal case to maintain consistency with PowerShell's naming rules.

Like bash, PowerShell ignores all characters between a # symbol and the end of the line. PowerShell—since version 2.0—added support for block comments which are denoted by <# and #> and can span multiple lines. The example below includes comments that describe the `Write-Host` cmdlet in addition to comments that present its output:

```
# This is a single-line comment
<#
  This is a block comment...
  It can span multiple lines
#>
# The prefix ## indicates an output line
# The Write-Host cmdlet writes output to a PowerShell host
# Host here refers to the process that's hosing PowerShell
Write-Host hello world
## hello world
```

Above and elsewhere in this section, we prefix each output line by ## followed by a blank space; this facilitates copying an example and pasting it into a PowerShell window (the text following ## is ignored and thus may be included in the copied text). Unlike Command Prompt, PowerShells handles the keyboard shortcut Ctrl+v for pasting text into its window.

To run multiple cmdlets on the same line, chain them using semicolon as a delimiter:

```
Write-Host hello; Write-Host world
## hello
## world
```

The vast majority of cmdlets provide flags (switches) that the user can specify, instructing a command to execute a variation of its default behavior. Another railroad analogy comes to mind here as cmdlets switches are similar to railroad

switches used by train operators to change tracks. Switches in PowerShell are indicated by a dash followed by the switch's name. Here's an example using the `Write-Host` cmdlet with the -NoNewline switch, which specifies that a newline character is not printed at the end:

```
# The first cmdlet below has the NoNewLine switch turned on
Write-Host -NoNewLine hello; Write-Host world
## helloworld
```

Most cmdlets also provide named parameters, which can be specified on the command-line in any order since they are identified by their names instead of position. Unnamed parameters, also known as positional parameters, need to be specified in the order the cmdlet expects. Parameters—named and unnamed—and/or switches can be combined together, unless otherwise specified by the cmdlet. Here's an example that combines named and unnamed parameters:

```
# The separator below is added between printed objects
Write-Host -Separator ", " hello world
## hello, world
```

The value we supplied to the `Separator` parameter above is actually an object of type String; to demonstrate the power of passing objects in PowerShell, we supply an array of integers to the `Write-Host` cmdlet to be formatted and printed below:

```
# 0..9 creates an array of integers from 0 to 9
# Parentheses are required to evaluate the expression correctly
Write-Host -Separator ";" (0..9)
## 0;1;2;3;4;5;6;7;8;9
# The + operator below concatenates ranges
Write-Host -Separator ";" (0..9 + 8..0)
## 0;1;2;3;4;5;6;7;8;9;8;7;6;5;4;3;2;1;0
```

In addition to its powerful built-in operators, PowerShell supports executing .NET code; to illustrate, we obtain the value of the separator used in the above examples from a field of type `char` in the `System.IO.Path` class:

```
# The namespace prefix System is optional in PowerShell
Write-Host -Separator ([IO.Path]::PathSeparator) (0..9)
## 0;1;2;3;4;5;6;7;8;9
```

We can go one step further and simply execute a .NET code snippet that joins the array of integers using the path separator and returns a string object that's equivalent to the output above. The PowerShell environment takes care of printing it out to the screen for us:

```
[String]::Join([IO.Path]::PathSeparator, 0..9)
## 0;1;2;3;4;5;6;7;8;9
```

Cmdlets return objects, whose methods can be called as in the example below:

```
# This is also an example of a multi-line entry
# When an entry is incomplete, you'll see this prompt: >>
# To complete the entry, press Enter after the last input
# Get-Date returns a DateTime object
(Get-Date).
AddDays(1).
ToUniversalTime().
ToLongDateString().
ToUpper()
## FRIDAY, MAY 13, 2016
```

Working with objects in PowerShell facilitates operations that could be cumbersome and error-prone in text-based shells, like extracting a specific data field from an array of results. To draw a quick comparison with a text-based shell, we use the `dir` command, which is a built-in command in Command Prompt:

```
rem The current directory is C:\test
rem The "." entry is a reference to the current directory
rem The ".." entry is a reference to the parent directory
dir
##  Volume in drive C has no label.
##  Volume Serial Number is D0A0-D665
##
##  Directory of C:\test
##
## 04/30/2016  01:23 PM    <DIR>          .
## 04/30/2016  01:23 PM    <DIR>          ..
## 04/30/2016  01:22 PM                42 bar.txt
## 04/30/2016  01:22 PM                 3 foo.txt
##                2 File(s)             25 bytes
##                2 Dir(s)  50,272,501,760 bytes free
##
```

Now let's work on extracting the file size field from the text result above; we use a for-loop to process the output text line-by-line, tokenize each line, specify which tokens are passed to the loop's body as parameters, exclude unwanted lines, and finally printing out the desired data:

```
rem ^ allows us to escape special characters
for /f "tokens=4,5" %i in ('dir c:\test') ^
do @if exist %j if %i neq ^<DIR^> echo %j %i
## bar.txt 22
## foo.txt 3
##
```

Not only is it complicated to come up with such script to accomplish a trivial task, but it's also extremely error-prone due to the subtleties of the syntax, which we don't find appropriate to explain here. In fact, the script above is bug-ridden, but we leave the task at hand to eager readers as an exercise to hone their Command Prompt scripting skills.

Now contrast that with how it's done in PowerShell:

```
# dir is an alias for the Get-ChildItem cmdlet
# The output of dir is piped as input to Select-Object
dir | Select-Object Name, Length
##
## Name           Length
## ----           ------
## bar.txt            42
## foo.txt             3
```

In PowerShell, cmdlets and parameters can have aliases. Many cmdlets exist in PowerShell to provide functionalities similar to those of Command Prompt commands; maintaining the same names helps users to migrate easily from Command Prompt to PowerShell. Aliases can be also used to save time when typing long cmdlet names, though it's more expressive to use a descriptive name like `Get-ChildItem` in script files to enhance readability.

Unlike bash and Command Prompt, where some command options—by convention—have the same name and behavior across various commands (e.g., `-help` and `/?` respectively), PowerShell provides a set of strongly typed common parameters that can be used with any cmdlet; here are a few examples:

Parameter	Alias	Description
`-Confirm`	`-cf`	Asks for confirmation before taking an action
`-Debug`	`-db`	Shows debug messages (for developers)
`-ErrorAction`	`-ea`	Determines what to do when an error occurs
`-Verbose`	`-vb`	Shows additional messages that provide more information
`-WhatIf`	`-wi`	Shows messages describing a dry run of the cmdlet
`-?`		Shows a help message describing the cmdlet and its parameters

The `Get-Help` cmdlet—also known as `help` or `man`—shows what a cmdlet does and how it can be used. Here's a simple example of a help message:

```
# The output below is truncated for brevity
Get-Help Out-Null
##
## NAME
##    Out-Null
##
## SYNOPSIS
##    Deletes output instead of sending it down the pipeline.
##
##
## SYNTAX
##    Out-Null [-InputObject [<PSObject>]] [<CommonParameters>]
## (truncated for brevity)
```

To ensure that PowerShell is showing the latest version of help files, run the `Update-Help` cmdlet as an administrator to download and install the latest version available.

Pipes

Piping in PowerShell has the same syntax like Command Prompt and the Linux terminal; however, one major difference is that what gets piped is an object. The pipeline is at the heart of PowerShell and its design philosophy; cmdlets are intentionally scoped to achieve small tasks—as the name suggests—yet they are powerful when combined together. PowerShell was previously known as Monad (which can be confused with the functional programming construct); it was named after Gottfried Leibniz's famed philosophical work: The Monadology.[9] Leibniz's Monad is the simplest substance without parts; Monads, combined together, create everything in existence. The following example is the epitome of why Monad was a perfect name:

```
# Many aliases are used below for brevity
dir | group extension | Sort-Object count | where count -gt 1 |
select count, name | ConvertTo-Json
## [
##     {
##         "Count":  2,
##         "Name":  ".ini"
##     },
##     {
##         "Count":  3,
##         "Name":  ".xml"
##     },
##     {
##         "Count":  7,
##         "Name":  ".log"
##     },
##     {
##         "Count":  9,
##         "Name":  ".exe"
##     }
## ]
```

Variables

Sometimes piping isn't enough; we need variables to store, manipulate, and read data. Variables in PowerShell are objects; variable names have to start with a `$` as the example below demonstrates:

[9] http://people.uvawise.edu/philosophy/phil206/Leibniz.html.

```
# Assign the value 13 to $var then read it
$var = 13
$var
## 13
```

If the variable name has to contain a `$` or any special character, expect the colon symbol, the following syntax is used:

```
# Assign the value 42 to ${a$b} then read it
${a$b} = 42
${a$b}
## 42
```

The variables listed above are assigned integer values, so they take on the type that corresponds to such values: `Int32`. In the aforementioned cases, a type was implicitly assigned. Said variables are also dynamically typed; they can change types afterwards:

```
$x = 3
$x.GetType().Name
$x = "hello world"
$x.GetType().Name
## Int32
## String
```

PowerShell also supports explicit typing, which provides error-checking when a variable is assigned a value of an incompatible type:

```
# Define $t to be of type [DateTime]
[DateTime]$t = (Get-Date) # Use () to evaluate the cmdlet first
$t = "5/23/2016" # Valid conversion from String
$t
$t = "hello world" # Error
##
## Monday, May 23, 2016 12:00:00 AM
## Cannot convert value "hello world" to type
   "System.DateTime".
## (truncated for brevity)
```

We can take data validation in PowerShell one step further by constraining the range of values a variable can take; this is very convenient for parameter validation or design by contract.[10] Here's an example:

```
[ValidateRange(1, 118)][int]$atomicNumber = 1
$atomicNumber = 119 # Error
## The variable cannot be validated because the value 119
## is not a valid value for the atomicNumber variable.
## (truncated for brevity)
```

[10] https://en.wikipedia.org/wiki/Design_by_contract.

Another useful constraint is creating a read-only variable:

```
# A variable in PowerShell can have a description
Set-Variable Mg -option ReadOnly `
-description "Magnesium" -value 12
$Mg
$Mg = 13 # Error
## 12
## Cannot overwrite variable Mg because it is read-only
## or constant.
## (truncated for brevity)
```

In the above example, we also show how to create a multi-line command using the backquote symbol.

So far we covered how to print the value of a variable using its name directly, but what if we want to add a message that includes the variable's value? One way to do so is by using the string concatenation operator + as in the following example:

```
$x = 1
$y = 2
Write-Host ("x = " + $x + ", y = " + $y)
## x = 1, y = 2
```

As you can see in the example above, it's not the most readable way to format a string. Luckily, PowerShell supports formatting a string by embedding variables within said string, and it evaluates them when the `Write-Host` cmdlet prints the string out:

```
$x = 1
$y = 2
Write-Host "x = $x, y = $y"
## x = 1, y = 2
```

Naturally, the question that should follow this revelation is: how to print a `$x` as a literal (when we don't want PowerShell to evaluate it)? The answer is by escaping it using the grave accent operator as in the example below:

```
$x = 1
$y = 2
Write-Host "`$x = $x, `$y = $y"
## $x = 1, $y = 2
```

It's also worth mentioning that if we just want to print a literal, we can simply do so by using single quotes instead of double quotes (to define a literal string):

```
Write-Host '$x is a variable'
## $x is a variable
```

PowerShell provides a cmdlet call that's equivalent to the `set` command (in bash and Command Prompt) to list all variables in the current shell session:

```
$a = 1
# Get the content of the virtual drive (variable:)
dir variable:
##
## Name                           Value
## ----                           -----
## $                              1
## ?                              True
## ^                              $a
## a                              1
## args                           {}
## ConfirmPreference              High
## (truncated for brevity)
```

The use of virtual drives is a common pattern in PowerShell, which allows us to use other cmdlets that deal with files and paths with logical entities like variables. An example of that is testing for the existence of a variable (which is similar to testing for the existence of a file):

```
Test-Path variable:nonexistent
## False
```

Similarly, a variable can be deleted using the `del` cmdlet:

```
$x = 1
Test-Path variable:x
del variable:x
Test-Path variable:x
## True
## False
```

As you can see from the table that is the result of listing all variables in the current shell session, many variables exist automatically: some are environment variables; others are automatically created and maintained by PowerShell. To see the help page about this topic, simply run `man about_Automatic_Variables` in PowerShell. Below are a few examples of key automatic variables:

Variable	Description
`$?`	Status of last execution (True if succeeded)
`$_`	The current pipeline object
`$Env:X`	The environment variable named `X`
`$Error`	An array of most recent errors ordered like a stack
`$False`	Self-explanatory
`$LastExitCode`	The exit code of the last program execution
`$Null`	Represents an empty value
`$PID`	The current process' identifier
`$PWD`	Path of the current (working) directory
`$True`	Self-explanatory

Scripting

Loops

There are multiple ways to loop through a list of items in PowerShell; for brevity, we use equivalent examples in the list below that print the same output:

- `for`: usually used with a loop counter or to loop through a subset of the items in a list.

```
for ($i = 0; $i -lt 5; $i++) { Write-Host $i }
```

- `while`: typically used with a single condition or to loop through a subset of the items in a list.

```
$i = 0; while ($i -lt 5) { Write-Host $i; $i++ }
```

- `foreach`: a loop statement used to execute a code block for each item in a collection.

```
foreach ($i in 0..4) { Write-Host $i }
```

- `.ForEach`: a method used to execute a code block for each item in a collection.

```
(0..4).ForEach({ Write-Host $_ })
```

- `ForEach-Object`: a loop cmdlet that can be used in a pipeline. When the `foreach` statement is used in a pipeline, PowerShell actually runs this cmdlet instead under the hood. The use of `foreach` is recommended when the collection to loop through is small.

```
(0..4) | ForEach-Object { Write-Host $_ }
```

If you haven't guessed it already, the output for any of the above examples is:

```
## 0
## 1
## 2
## 3
## 4
```

In terms of performance, `.ForEach` method is faster than `foreach`; the latter is typically faster than `ForEach-Object`, which is memory-friendly but slower than the former. Another important note: the collections above are expressions that can be replaced with the output of a cmdlet; here's an example:

```
Get-Item *.* | Group extension |
ForEach-Object { Write-Host $_.Name }
## .ipynb
## .csv
## .pkg
## .py
## .zip
```

Conditional Logic

PowerShell supports `if` statements with the following, familiar syntax:

```
if (<if condition>) {
<if block>
} elseif (<elseif condition>) {
<elseif block>
} else {
<else block>
}
```

Comparison operators in PowerShell work not only with numeric type, but also with other comparable types like dates and strings. String comparison is case-insensitive by default (unlike Command Prompt's string equality, which is case-sensitive by default):

```
if ("A" -eq "a") { Write-Host "case-insensitive" }
## case-insensitive
```

The following comparators are supported:

Operator	Description
`-eq`	Equals
`-ne`	Not equals
`-gt`	Greater than
`-ge`	Greater than or equal
`-lt`	Less than
`-le`	Less than or equal

Other operator that can be useful in evaluating conditions:

Operator	Description
`-Match`	Uses a regular expression to match a pattern
`-NotMatch`	The negative form of `-Match`
`-Like`	Checks string equality allowing the wildcards `*` and `?`
`-NotLike`	The negative form of `-Like`
`-In`	Whether an array contains an element
`-NotIn`	The negative form of `-In`
`-Not`	Negates its operand

There are many commonly used conditions PowerShell can check; the example below demonstrates a few of them:

```
"time flies" -like "an arrow"
"fruit flies" -notlike "*lies"
```

```
-not ("anything" -match ".*")
Test-Path nonexistent.txt
$?
## False
## False
## False
## False
## True
##
```

For more details, see the help article about `if` by running `Get-Help about_If` (you may need to download the help articles first); you can see the help article online at https://bit.ly/2JV18jO.

3.3 The Kernel, Traps, and System Calls

The operating system is a collection of individual programs, each consisting of a sequence of assembly language instructions as described in Chap. 2. The kernel is the most important OS program, running from the time the computer is powered on and until the computer is powered off.

As the CPU executes the kernel program, it alternates between the following two modes:

System Mode: The CPU executes the kernel program; this mode is also known as privilege level/ring 0.

User Mode: The CPU executes a non-kernel program; this mode is also known as privilege level/ring 3.

When the computer is powered on, the CPU assumes the system mode and the kernel program initializes the operating system. Once the kernel completes its initialization tasks, the CPU assumes the user mode and execution is transferred to a non-kernel program by resetting the program counter (see Definition 2.3.1) appropriately.

There are two ways to transfer control from user mode back to kernel mode:

System Call: The non-kernel program issues a system call, requesting the operating system to provide a specific service.

Counter: Expiration of a timer that was reset when the CPU last assumed the user mode.

In the first case above, as a result of the system call the kernel mode resumes execution, handles the system call, and then returns execution back to the user program. In the second case above, the expiration of the timer causes the kernel mode to resumes operation after which execution is transferred to a different program (with a reset timer). The previous program that was executing whenever the timer expired is suspended until its turn arrives again.

As described in Chap. 2, at any single moment the CPU executes a single instruction, but the operating system kernel enables the concurrent execution of multiple programs, called processes.

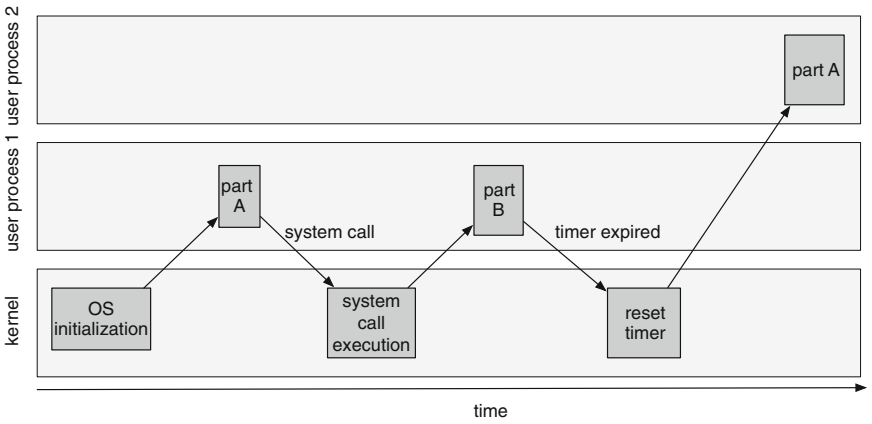


Fig. 3.2 An example of transitions between the kernel and user modes. See text for more details

Definition 3.3.1 A process is a program, or a sequence of assembly language instructions, running in user mode concurrently with the kernel.[11]

Figure 3.2 shows an example of transitioning between the kernel mode and the user mode. The kernel starts executing when the computer boots up, and then transfers execution to process 1. Part A of process 1 ends with a system call, transferring execution back to the kernel mode. The kernel handles the system call and returns execution to process 1. After a while (in CPU time), the timer expires, suspending process 1 and entering the kernel mode. The kernel resets the timer and re-enters user mode, giving process 2 a turn (known as a time slice or quantum).

The mechanism by which system calls transfer execution from user mode to kernel mode is called a trap. A trap is similar to interrupts (see Chap. 2) in that both traps and interrupts transfer execution to a separate program (event handler in the case of interrupts, and kernel mode in the case of traps). An important difference is that interrupts are implemented in the CPU (hardware), while traps are implemented in the operating system kernel (software).

Some examples of popular system calls appear below. The precise set of system calls and their format depends on the operating system.

- Terminate the execution of the current program.
- Request the right to use a portion of the memory of a certain size.

[11] Some definitions also consider the kernel as a process.

- Inform the kernel that a portion of the memory that was allocated to the program is no longer needed.
- Read or write content to the hard disk.
- Write information to an output device, or read information from an input device. In this case the trap triggers a hardware interrupt as well.

3.4 Process Management

A single core CPU can only execute a single instruction at any specific time (see Chap. 2). The alternation between kernel and user modes lets multiple processes run concurrently (see Fig. 3.2). This applies to both interactive programs that wait for user response, such as word processors, pdf viewers, and web browsers, and to noninteractive programs. Running multiple processes concurrently is also useful when the hardware is truly parallel. For example, a computer with l cores can run k processes concurrently even when $k > l$. As a result, concurrent processes may or may not run in parallel.

Definition 3.4.1 The process of transferring execution between one process and another is called a context switch.

Definition 3.4.2 The OS scheduling strategy determines when to perform a context switch, and what process to select next for execution.

A good scheduling strategy should (a) ensure that the CPU or CPU cores are not frequently idle (load balancing), and (b) each process gets an appropriate share of the CPU time. One popular scheduling strategy is round-robin, where each process gets a turn, followed by the next process, and so on. After all processes get their turn, the first process gets another turn, followed by the second process, and so on. Scheduling strategies become more complex when there are multiple cores or processors.

In some cases it is desirable to give some processes priority over other processes. In this case processes with higher priority receive more CPU time than processes with lower priority. For example, it is customary to lower the priority of a computationally intensive process so that concurrent interactive processes (such as a terminal or a web browser) will receive sufficient CPU time to ensure a smooth interactive user experience.

3.4.1 Processes in Linux

Interacting with the Linux terminal may launch a single process or multiple processes; for example, when we have multiple commands separated by pipes.

Definition 3.4.3 A job is a group of processes (Definition 3.3.1) responsible for executing one or more terminal commands connected by pipes.

Definition 3.4.4 A job may run in the foreground, interacting with the user through the terminal, or in the background where it does not interact with the user through the terminal.

For any terminal window, only a single job can run in the foreground, but multiple jobs can run in the background concurrently.

Terminal commands are executed by default in the foreground. The corresponding job interacts with the user by displaying output to the terminal or by reading input from the user keyboard. In particular, the shell waits for the job to finish before it displays another prompt and allows the user to launch a new command.

Appending the `&` symbol at end of a command executes the corresponding job in the background. In this case, the shell immediately displays a new prompt, allowing the user to execute new commands concurrently with the background job. Subsequent commands appended by the `&` symbol will run in the background as well (there can be multiple jobs running in the background).

The CTRL+z keystroke suspends the foreground job and displays a terminal prompt, allowing the user to launch new commands in the prompt. The command `bg` followed by the job number resumes execution of a suspended job in the background (if only one suspended job exists the job number may be omitted) and the command `fg` resumes execution of a suspended job in the foreground. The keystroke combination CTRL+c stops the foreground job without the possibility of resuming it later. The Linux commands `jobs` and `ps` display the current jobs and processes respectively.

The table below shows some common flags and other related commands.

Command	Description
`jobs`	displays jobs launched by current user in current terminal
`ps`	displays active processes launched by the current user, listing process ID, terminal name, CPU time thus far, and the command that launched the process
`ps -u`	same as `ps`, but adds process memory usage, the date the process started, and the user that launched the process
`ps -A`	same as `ps`, but includes all concurrent processes, including processes launched by other users and in other terminals
`X &`	launches the command `X` in the background
`CTRL+c`	stops current foreground job
`CTRL+z`	suspends current foreground job
`fg X`	resumes job `X` in the foreground (argument not needed if there is only a single job)
`bg X`	resumes job `X` in the background (argument not needed if there is only a single job)
`kill X`	kills process `X` (using process ID) or job `X` (using `%` symbol followed by job ID)

The commands `tail` and `head` display the first and last 10 lines of its input, respectively (see Sect. 3.6 for more detail). In the following example, we use the command `tail -f` that displays the last ten lines and waits indefinitely for additional data to be appended.

```
touch a.txt; touch b.txt  # create two empty files
tail -f a.txt &  # launch a never-ending background job
jobs  # display current jobs

## [1]+  Running                 tail -f a.txt &

ps  # display current processes launched by current user

##   PID TTY           TIME CMD
## 11185 ttys005    0:00.00 tail -f a.txt

ps -A | head -n 7 # display all running processes
 (first 7 lines)

##   PID TTY           TIME CMD
##     1 ??         0:10.84 /sbin/launchd
##    10 ??         0:00.79 /usr/libexec/kextd
## (truncated for brevity)

tail -f b.txt &  # launch another never-ending background job
jobs  # displays current jobs

## [1]-  Running                 tail -f a.txt &
## [2]+  Running                 tail -f b.txt &

kill %1  # kill job 1
jobs

## [1]-  Terminated: 15          tail -f a.txt
## [2]+  Running                 tail -f b.txt &

jobs

## [2]+  Running                 tail -f b.txt &

fg  # bring the single current job into foreground

## [CTRL+z keystroke suspends foreground job
## and creates a new prompt]

jobs  # job was suspended by CTRL+z keystroke

## [2]+  Stopped                 tail -f b.txt

bg  # resume single current job in the background
jobs

## [2]+  Running                 tail -f b.txt &
```

The following example starts multiple processes that simply wait for 100 s, first in the foreground and then in the background. In the first case the processes run sequentially: the second process starts after the first process finishes, the third process starts after the second process finishes, and so on.

```
# run five jobs sequentially in the foreground
for i in `seq 1 5`; do  (sleep 100) ; done ;

[ hitting CTRL-Z to suspend the job ]

jobs

## [1]+  Stopped                 ( sleep 100 )

bg  # resume process in the background
jobs

## [1]+  Running                 ( sleep 100 ) &

ps  # shows single active process

##   PID TTY           TIME CMD
## 57140 ttys004    0:00.00 sleep 100
```

In the example below, the processes run concurrently in the background.

```
# run 5 processes in the background concurrently
for i in `seq 1 5` ; do  (sleep 100 &) ; done ;
ps

##   PID TTY           TIME CMD
## 57148 ttys004    0:00.00 sleep 100
## 57150 ttys004    0:00.00 sleep 100
## 57152 ttys004    0:00.00 sleep 100
## 57154 ttys004    0:00.00 sleep 100
## 57156 ttys004    0:00.00 sleep 100

kill 57148  # kill the first process

ps

##   PID TTY           TIME CMD
## 57150 ttys004    0:00.00 sleep 100
## 57152 ttys004    0:00.00 sleep 100
## 57154 ttys004    0:00.00 sleep 100
## 57156 ttys004    0:00.00 sleep 100
```

The `top` command displays properties of the process that are currently running concurrently. The properties include the user who initiated the process, the process runtime, and the process priority. Unintuitively, lower priority numbers in Linux correspond to higher priorities or higher shares of CPU time. Typing `q` exists the

`top` viewer. The command `htop` is an alternative that displays more detailed information (`htop` may not be installed by default, in which case it needs to be manually installed, for example using `sudo apt-get install htop` on Ubuntu Linux).

In the example below, there are four Java processes running in parallel on a computer system with 8 cores. They each were launched by the user `joe` (second column), have a priority 20 (third column), and use 100% of a CPU core (ninth column) and about 1% of the memory (tenth column).

```
x

top

## Tasks: 195 total,   2 running, 193 sleeping,   0 stopped,
## Cpu(s): 69.4%us,  0.3%sy,  0.0%ni, 30.3%id,  0.0%wa,  0.0%hi
## Mem:  99197580k total, 92876600k used,  6320980k free,
## Swap: 100652028k total,    19548k used, 100632480k free
##
##   PID USER PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+ COMMAND
## 26323 joe  20   0 24.2g 1.2g 9396 S  100  1.3   3546:44 java
## 31696 joe  20   0 24.2g 1.3g 9392 S  100  1.4 874:38.32 java
## 29854 joe  20   0 24.2g 971m 9408 S  100  1.0   1962:39 java
## 29419 joe  20   0 24.2g 992m 9396 S  100  1.0   2204:30 java
```

The `nice -n k X` command launches the command `X` with a niceness value of `k`, which is added to the default process priority number of 20. The higher the value of `k`, the less CPU time the process is allocated. Using positive `k` values makes the new process "nicer" than default processes, implying that it will receive a smaller share of the CPU resources. Using negative `k` values makes the new process less "nice," implying that it will receive a larger share of the CPU resources. The `renice -n k X` command modifies the priority of an existing process with process-id `X` by adding to its priority the niceness value `k`. Typically, only a super-user can assign negative values of `k` (see Sect. 3.7 for more information on super-users).

Command	Description
`top`	displays currently running processes, annotated by their CPU, memory usage, and user who launched them (type `q` to quit viewer)
`htop`	more detailed variation of `top`
`nice -n k X`	executes command `X` with modifier `k` added to its priority
`sudo nice -n k X`	executes command `X` with potentially negative modifier `k` added to its priority (super-users only)
`renice -n k X`	modifies the niceness modifier of an existing process with process ID `X` to `k`

3.4.2 Processes in Windows

Interacting with the command-line interfaces may launch a single process or multiple processes; for example, when we have multiple commands separated by pipes. For any shell window, only a single job can run in the foreground, but multiple jobs can run in the background concurrently. Commands are executed by default in the foreground; the foreground job interacts with the user by displaying output to the shell or by reading input from the user keyboard.

For Command Prompt, the `start` command can be used to start a new window to run the given command; by default, the newly created window is left open for the user to interact with after executing the given command. To run a command in a new window that closes after executing said command, the following syntax should be used:

```
start cmd /c "echo poof"
```

To test that effect, you may want to chain a `pause` command at the end:

```
start cmd /c "echo poof && pause"
```

To start a command without creating a new window, set the `/B` flag

```
start /b powershell "sleep 1; echo done"
```

By default, the CTRL+c keystroke[12] sends a signal (SIGINT) to all processes attached to the current shell window.

In PowerShell, the following cmdlets can be used to manage background jobs:

Command	Description
`Get-Job`	Gets background jobs that are running in the current session
`Receive-Job`	Gets the results of background jobs in the current session
`Remove-Job`	Deletes a background job
`Start-Job`	Starts a background job
`Stop-Job`	Stops a background job
`Invoke-Command`	The `AsJob` parameter runs a command as a background job

For more information about background jobs in PowerShell, check the help article at https://msdn.microsoft.com/en-us/library/dd878288.aspx or run `help about_Jobs`

[12]The CTRL+Break combination always sends a signal to terminate a process while applications can choose to change the default CTRL+c behavior; that said, it's getting harder to find keyboards with the Break key these days.

The command `tasklist` displays the currently running processes on the local computer by default, but also can be used to query a remote one for running processes; it also allows many filters, for example:

```
tasklist /FI "PID eq 4"
##
## Image Name PID (truncated for brevity)
## ========== === =================
## System     4   (truncated for brevity)
```

In PowerShell, the cmdlet Get-Process is the equivalent of `tasklist`; to get a specific process, one may use the `Id` and/or `Name` argument. Since the cmdlet returns a process object, it can be used to get more details or manipulate the process using methods like `Kill`. In Command Prompt, the command `taskkill` is used to kill a process using a filter like its process ID (PID), image name, etc.

3.5 Memory Management and Virtual Memory

It's inappropriate for processes to read from or write to arbitrary physical RAM addresses. A few reasons are listed below:

- A process may inadvertently overwrite its own instructions or the OS kernel program.
- Two processes running concurrently may inadvertently access the same physical memory addresses, overwriting each other's information.
- A malicious process may access restricted information and take control of the computer.

The operating system mediates between the processes and the RAM by defining a separate address range for each process, known as virtual memory, that the process can safely read from or write to. The OS ensures that virtual memory allocated to different processes are mapped to disjoint parts of the physical RAM, and that these parts do not overlap with areas of the RAM holding the operating system code or other critical information.

Often, the virtual memory address ranges are mapped to the hard disk, as well as to RAM. Since disk space is much larger than RAM, this lets the operating system allocate more virtual memory for each process.

A significant difficulty associated with mapping virtual memory to the disk is that disk access is much slower than RAM access (see Chap. 2) and a careless allocation policy can considerably slow down the processes. A partial solution is similar to the cache mechanism described in Chap. 2: frequently accessed virtual memory is mapped to RAM and infrequently accessed virtual memory is mapped to disk. Figure 3.3 illustrates this strategy, where contiguous chunks of virtual memory, called pages, are mapped to physical storage in RAM or disk (shaded) depending on their access frequency. If a virtual memory page that is stored only in disk is accessed, the operating system brings it to RAM, potentially overwriting another virtual memory page in RAM that will subsequently reside only in the disk.

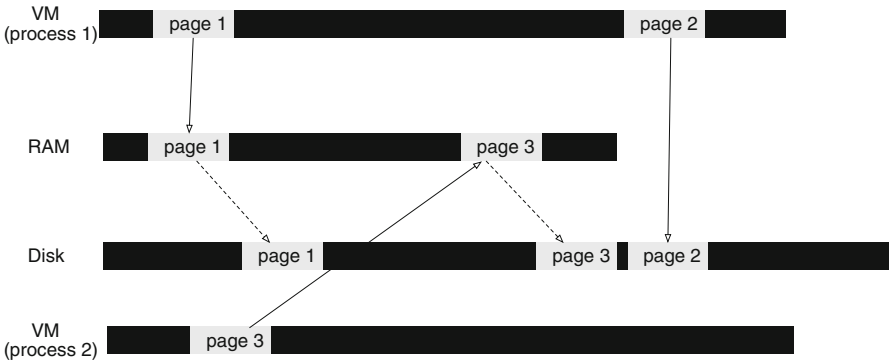


Fig. 3.3 The virtual address memory ranges of two processes is mapped to RAM or disk where it is physically stored. In this example, pages 1 and 3 are frequently accessed and are stored in RAM as well as disk, while page 2 is currently stored only in disk. When the virtual memory in page 2 is accessed, the OS kernel brings the page to RAM, potentially overwriting one of the less frequently accessed virtual memory pages in RAM and making that page reside only in disk

Due to the principles of spatial and temporal locality (see Sect. 2.9), this scheme allows processes to gain access to a large range of virtual memory, with relatively little slowdown. There is substantial similarity between virtual memory and the cache mechanism described in Sect. 2.9. One important difference, however, is that while cache is implemented in hardware by the CPU, the virtual memory is implemented in software by the operating system.

3.6 The File System

The operating system provides an interface between processes and the disk, called the file system. The file system maps physical addresses on the disk to logical units called files that the operating system and its processes access. The interface lets the OS ensure that two concurrent processes do not write to the same file at the same time, and that processes can only access files they are permitted to. In contrast to virtual memory, files persist after the computer is turned off, and reside exclusively on disk.

There are many types of files, but the two most fundamental file types are text and binary. Text files contain strings of characters, usually in ASCII encoding (see Chap. 2). Binary files contain executable programs in the form of sequences of bits corresponding to assembly language instructions. Other types of files such as media files (images, videos, music) and files storing data (in non-ASCII encoding) are usually categorized as binary as well.

Files have three basic properties: size, name, and path. The size of the file is the length of the corresponding byte sequence (or bit sequence divided by 8) on

the disk. The file name is a character sequence, excluding special characters such as $ or &. File names are case sensitive in Linux and macOS and case insensitive in Windows. In many cases, a file name may contain one or more periods, acting as separators between the base file name and file name extensions. File name extensions often denote the type of data the file holds. For example, the extensions `X.txt`, `X.jpeg`, and `X.mp3` (`X` stands here for the base file name) correspond to text, JPEG compression for images, and MPEG compression for audio respectively. Files whose names start with a period are called hidden files.

The operating system offers a convenient mechanism to organize files into groups called directories. Directories are further organized in a hierarchy, where each directory may contain additional directories within it, called sub-directories. A directory containing files or sub-directories is called the parent directory of its files and sub-directories. Thus, every file and directory has one parent directory (unless it is the initial or top-most directory) and may have zero, one, or more sub-directories.

The absolute path associated with a directory is the sequence of directories leading to it starting from the top-most directory, separated by the `/` character. The absolute path associated with a file is the path leading to its parent directory followed by the file name. For example, the path `/a` denotes a file or directory called `a` that reside in the top-most directory. The path `/a/b` denotes a file or directory `b` that resides in the directory whose absolute path is `/a`.

A relative path to a target file or directory is the sequence of directories (separated by `/`) connecting the current directory to the target file or directory. Thus, if the current directory is `/a/b/c`, the relative path `d/e` refers to a directory or file whose absolute path is `/a/b/c/d/e`.

The directory hierarchy usually indicates a structure of specificity: directories represent a common theme and sub-directories within them represent further specialized themes within the theme of the parent directory. Two files or directories with identical names may not reside in the same directory, but may reside in different directories. As a result, absolute paths uniquely identify files and directories in the file system, which are mapped by the OS to specific physical addresses on the disk.

3.6.1 Files in Linux

Hierarchy of Directories

Figure 3.4 shows the standard hierarchy of directories in Linux, annotated with the traditional roles of the different directories (the macOS directory hierarchy is slightly different). The directory `/home` typically holds user home directories, each specified by the corresponding user name. For example `/home/joe` is the home directory of user `joe` and `/home/jane` is the home directory of user `jane`. In many cases, users store their files in their home directory (or its sub-directories), leaving the rest of the file system hierarchy for the operating system and for applications.

At any specific moment, the Linux terminal is associated with a specific directory, called the current directory; the current directory is the base path used to resolve relative paths in the terminal and inside programs that run from it. The Linux command `pwd` displays the absolute path of the current directory. The Linux command `cd`, followed by an absolute or relative path, changes the current directory to the specified one.

The table below displays useful special paths in Linux:

Path	Description
~	home directory of current user
.	current directory
..	parent directory of current directory
~X	home directory of user X

The following command sequence assumes the directory structure of Fig. 3.4.

```
cd /  # change current directory to top-most directory
pwd  # display current path

## /

cd /home  # change directory using absolute path
pwd  # display current path

## /home

cd joe  # change directory using relative path
pwd  # display current path

## /home/joe

cd ../jane  # change directory using relative path
pwd  # display current path

## /home/jane
```

Two convenient commands for examining the directory structure are `df` and `du`. The command `df -h` displays the size of the disk drive, the amount of used space, and the amount of available space. Multiple disk partitions or multiple disk drives are displayed in separate rows. The command `du -sh X` displays in human readable form the size of the directory `X` (including its sub-directories).

```
# display the size of the disk drive and amount of used and
# available space using abbreviations such as Gi for Gigabyte
df -h

## Filesystem     Size   Used  Avail Capacity  Mounted on
## /dev/disk0s2  233Gi  140Gi   93Gi    61%    /
```

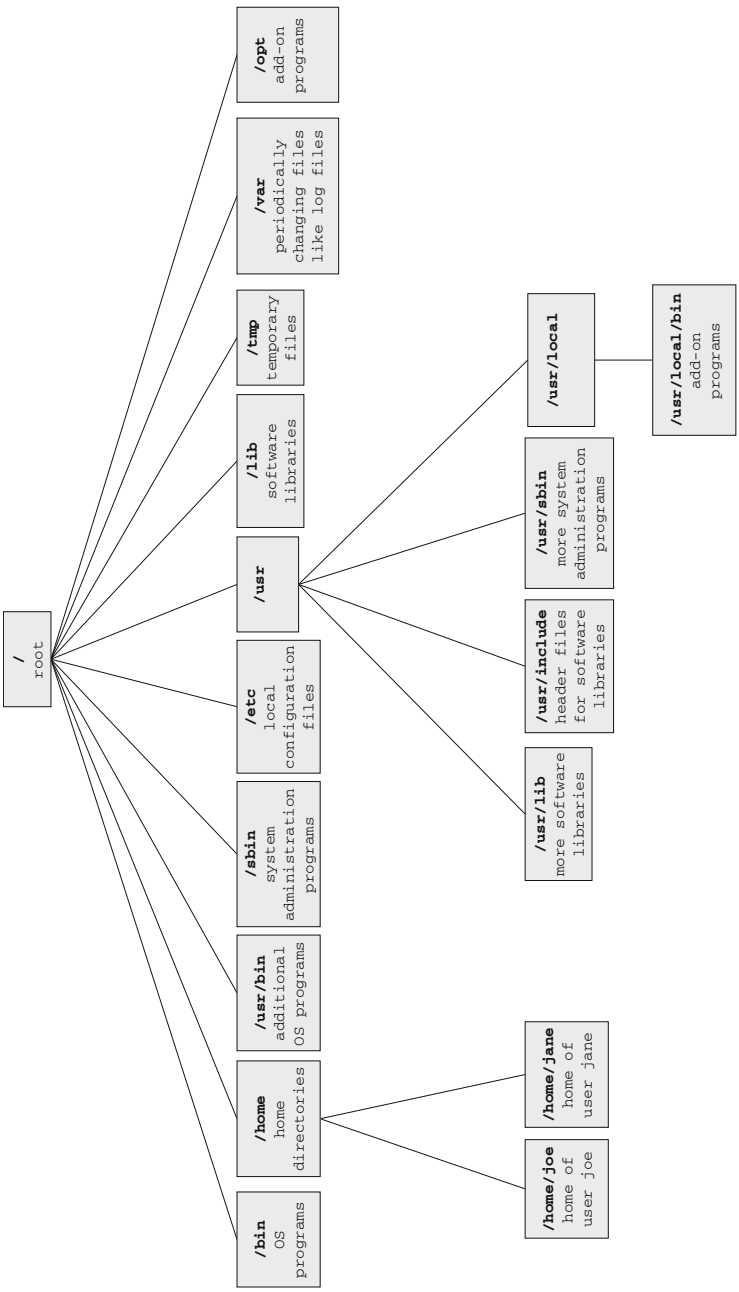


Fig. 3.4 Standard directory structure for Linux systems

```
# display size (21 GB) of directory /home/joe/sw (absolute path)
du -sh ~joe/sw

## 21G /home/joe/sw
```

Displaying Files

The following commands are useful for displaying or examining file contents:

`cat X`	displays contents of file `X`
`less X`	file viewer (scroll with arrow keys, quit with `q`)
`head -n k X`	displays the first `k` lines of the file `X`
`tail -n k X`	displays the last `k` lines of the file `X`
`sort X`	displays the file `X` with sorted lines
`uniq X`	displays the file `X` excluding duplicated lines
`diff X Y`	matches similar files `X`, `Y` and displays unmatched lines
`diff -u X Y`	matches similar files `X`, `Y` and displays for each line whether it is in `X`, in `Y` or in both
`grep Y X`	displays all lines of file `X` containing the pattern `Y`
`grep -w Y X`	same as `grep` but match entire words only

The example below examines the contents of the log file `/var/log/syslog.1`, containing system messages in different lines. Recall that the command `wc` displays the number of lines, words, and characters (see Sect. 3.2.1).

```
head -1 /var/log/syslog.1  # first line

## Feb  5 08:01:24 chance rsyslogd: [origin software="rsyslogd"

tail -1 /var/log/syslog.1  # last line

## Feb  6 07:35:01 chance anacron[7614]: Updated timestamp for

# count number of messages (first column is number of lines)
wc /var/log/syslog.1

## 82  2202 15058 /var/log/syslog.1

# count number of messages on Feb 6
grep "Feb  6" /var/log/syslog.1 | wc

##      30     750    5092

cat /var/log/syslog.1 | grep "Feb  6" | wc # same as above

##      30     750    5092
```

Moving, Copying, and Removing Files and Directories

The following commands are useful for moving, copying, and removing files and directories.

Command	Description
`rm X`	removes the file `X`
`rm -R X`	removes the directory `X` with all its contents
`rmdir X`	removes the empty directory `X`
`mkdir X`	creates a new empty directory `X`
`mkdir -p X/Y/Z`	creates the path of nested directories for `X/Y/Z`
`mv X Y`	moves the file or directory `X` to `Y`
`cp X Y`	copies a file `X` to directory `Y`
`cp -R X Y`	copies a directory `X`, with all its contents, to `Y`

The flag `-R` in the table above indicates that the command applies to a directory recursively, potentially copying or removing all sub-directories and their contents. The `-R` flag should be used with care, as it is easy to accidentally remove or overwrite a major portion of the directory structure. The flag `-i` prompts the user for verification when removing or overwriting existing files. In some cases the commands `cp -i`, `mv -i`, and `rm -i` have the aliases `cp`, `mv`, and `rm` so that using `cp`, `mv`, or `rm` requires confirmation when removing or overwriting existing files. The original behavior that does not require confirmation can be retrieved by removing the aliases using the `unalias` command or by prepending the command with a backslash character.

```
# copy file1 in current dir to file2 in current dir
cp file1 file2
# copy file1 in /tmp to file2 in ~
cp /tmp/file1 ~/file2
# rename file1 as file2 in current dir
mv file1 file2
# move file1 in /tmp to file2 in ~
mv /tmp/file1 ~/file2
# remove file2 in home dir
rm ~/file2
# removes directory tmpFiles and all its contents
rm -R ~/tmpFiles
# same as above but avoids calling an aliased version of rm
\rm -R ~/tmpFiles
```

Wildcard Characters

Linux offers several convenient wildcard characters (also known as globbing patterns) for matching multiple files or directories with a specific pattern. They are listed in the table below:

Symbol	Description
*	any string of characters (including the empty string)
?	any single character (or none)
[X]	any character in the set or range X; e.g., [mca] or [b-y]
[!X]	any character not in the set or range X
{X}	any term the set X; e.g., {*.txt,*.pdf}

One exception to the table above is that wildcards do not match hidden files (files whose names start with a period). To match hidden files, the wildcard should follow a period. Some examples are listed below.

```
# removes all non-hidden files in directory ~/tmpFiles
rm ~/tmpFiles/*
# removes all hidden files in ~/tmpFiles
rm ~/tmpFiles/.*
# copies all files in ~/tmpFiles whose names end with .html
# to directory /tmp
cp ~/tmpFiles/*.html /tmp/
# remove all files ending with .c or .o
rm *.[co]
# remove files whose extension is a lower-case character
rm *.[a-z]
```

Soft Links

In some cases it is desirable to have multiple references (potentially in different directories) to a single physical file or directory. The `ln -s X Y` creates a soft link `Y` pointing to the file or directory `X`. The file itself resides only once in disk, But it may be accessed using the original file reference `X` or using the soft link reference `Y`.

```
ln -s tmp tmpLink # create a link tmpLink to the directory tmp
ls tmp  # display files in ./tmp directory

## file1 file2

# enter the directory tmp by referencing the softlink
cd tmpLink
ls

## file1 file2
```

Listing Directory Contents

The Linux command `ls` lists the contents of the current directory and the Linux command `ls X` lists all files and sub-directories in the directory `X`. The following

flags are supported (see `man ls` for more information on the flags below and for additional flags).

Flag	Description
`-l`	displays more detailed format
`-F`	appends a character denoting file type (files, directories, links, etc.)
`-a`	lists all files, including hidden files
`-R`	includes also files in all sub-directories, recursively
`-r`	displays results in reverse listing order
`-t`	sorts files by modification time
`-h`	displays file size in human readable format

The example below shows some of these flags. Section 3.7 describes the format of the `ls -l` command.

```
ls

## file.txt prog subdir

ls -F

## file.txt prog* subdir/

ls -l

## -rw-r--r--     1 joe   staff  1951 Feb  6 14:06 file.txt
## -rw-r--r-x     1 joe   staff  2467 Feb  6 14:40 prog
## drw-r--r--     2 joe   staff    68 Feb  6 14:43 subdir
```

The PATH Environment Variable

The `PATH` environment variable specifies a list of one or more directories (separated by `:`) that are used to search for executable programs; each process has its own `PATH` setting that is by default inherited from its parent process or the user's settings. When a command is issued using bash, it searches for the first match of the corresponding executable file in a list of directories specified by the `PATH` environment variable. In many cases, the current directory is included in the `PATH` variable explicitly using the `.` notation.

The command `which X` displays the path to the matched file `X`. The command `whereis X` looks for files related to `X` in the standard Linux directory hierarchy, independent of the `PATH` variable.

```
# show path to file matching command ls
which ls
```

```
## /bin/ls

# display PATH variable (note the current directory is the third
# directory in the list below, denoted by the period notation)
echo $PATH

## /bin:/usr/bin:.

# add the directory /home/joe/bin to PATH
export PATH=$PATH:/home/joe/bin
echo $PATH

# /bin:/usr/bin:.:/home/joe/bin
```

Compression

Linux has a number of compressing and archiving commands that are useful for handling large files or a large number of files. Specifically, the command `bzip` and `bunzip` can be used to compress a file into a file of (typically) smaller size. The reduction in size depends on the original file, but is usually significant for text files. The command `tar` can either pack multiple files into a single archive file or unpack an archive file into multiple files. The table below describes the typical usage of these commands.

Command	Description
`bzip2 X`	compresses file `X` into a (potentially) smaller file `X.bz2`
`bzcat X.bz2`	displays contents of compressed file `X.bz2`
`bunzip2 X.bz2`	uncompress the compressed file `X.bz2`
`tar cvf X.tar Y/*`	packs all files in directory `Y` into a single file `X.tar`
`tar -xvf X.tar`	unpacks the tar archive file `X.tar`

The commands `gzip`, `zip`, and `compress` are alternative compression programs to `bzip2`.

The example below compresses, displays, and then uncompresses a text file containing the book Moby Dick. The bzip2 compression provided 63% compression reducing the file size from 52 KB to 18 KB.

```
cat mobyDick.txt | head -n 1  # first line

## Call me Ishmael.

bzip2 -v mobyDick.txt # compress .txt to .txt.bz2
```

```
## mobyDick.txt:  2.762:1,  2.896 bits/byte, 63.80% saved,
## 52342 in, 18950 out.
```

The compressed file is no longer a text file and attempting to view it as an ASCII encoded will not work well. However, the command `bzcat` is able to display the text content of the compressed file without decompressing it first.

```
# first line of compressed file
cat mobyDick.txt.bz2 | head -n 1

## BZh91AY&SY}?lF?u ??)_?p????????...

# using bzcat to display compressed file
bzcat mobyDick.txt.bz2 | head -n 1

## Call me Ishmael.
```

The example below creates an archive containing all files in the directory `tmp`, compresses the archive, and then reverses the process and extracts the files into a directory `tmp2`.

```
# pack all files in tmp/ into a file archive.tar
tar cvf archive.tar tmp/*
# compress the file archive.tar (creating archive.tar.bz2)
bzip2 archive.tar
# uncompress archive.tar.vz2
bunzip2 archive.tar.bz2
# moves archive.tar.bz2 to subdirectory tmp2
mv archive.tar tmp2
# change current directory to tmp2
cd tmp2
# unpack the tar file archive.tar in current directory
tar xvf archive.tar
```

Equivalently, the `gzip` command creates compressed files with the `.gz` extension; you may find the following z-commands, which work with `.gz` file, useful: `zcat`, `zless`, `zmore`, `zgrep`, `zegrep`, `zfgrep`, and `zdiff`; the equivalent bz-command, like `bzless`, work with `.bz2` files.

Bash Initialization File

The hidden file `/.bash_profile` is automatically executed every time the bash shell is launched. It contains assignments for environment variable, alias definitions, and other customization. A simple example file is listed below.

```
# customize the prompt appearance
export PS1="\\u@\h \\W]\\$"
# add current directory to PATH
export PATH=.:$PATH
# avoid overwriting files with output redirection
```

```
set -o noclobber
# prompt user for removing or overwriting files with rm, cp, mv
alias rm='rm -i'
alias cp='cp -i'
alias mv='mv -i'
# store most recent terminal commands in the file .bash_history
export HISTFILE=".bash_history"
# alias for printing last 10 commands
alias h='history | tail'
# alias ls and ll to use my favorite flags by default
alias ls='ls -Ft'
alias ll='ls -Fthalr'
```

In Sect. 3.2.1 we examined the `history` command and up-arrow and down-arrow keystrokes that recall previously executed terminal commands. The operating system accomplishes that by storing the most recently executed terminal commands in a file specified by the environment variable `HISTFILE` (often set to `.history` or `.bash_history`).

Script Files

Shell scripts are text files containing bash commands that are executed when the file name is typed in the Linux terminal. The script file must reside in a directory that is included in the `PATH` variable, or otherwise its full path must be specified when it is executed. The file must have executable permission for the current user (see Sect. 3.7 for more information on file permissions and how to set them) and must start with a line containing # followed by a path to the shell program (see the line below as an example).

```
#!/bin/bash
```

The shell script may be called with one or more arguments, which may be referred to in the script via the variables `$1`, `$2`, and so on (one variable for each argument). Below is an example of a script that accepts two variables when it is called and prints them.

```
cat printTwoArgs

## #!/bin/bash
## echo this is a bash script file that accepts two arguments
## echo $1 $2

chmod a+x printTwoArgs  # add executable permission to scriptFile
./printTwoArgs one two  # executing script by listing its path

## this is a bash script file that accepts two arguments
## one two
```

3.6.2 Files in Windows

Hierarchy of Directories

Windows supports logical volume management (LVM) of storage, which means a disk can be partitioned to multiple logical units that are sometimes called logical disks, volumes, or drives. Each volume is assigned a drive letter (unlike Linux, which uses mount points) used to identify the volume's root (e.g., `C:`
is a commonly used drive letter for the volume that holds system folders). Drive letters date back to CP/CMS, an OS that IBM Cambridge Scientific Center created in the 1960s, and was adopted by MS-DOS then Windows. Disk management in Windows allows for dynamic allocation of volumes so that their sizes can be changed after being allocated; users may use the command-line (via the `diskpart` command) or a GUI (via the Disk Management snap-in `diskmgmt`) to perform such operations.

To view the logical volumes, the following query uses WMIC (Windows Management Instrumentation Command-line) to list the names of logical volumes:

```
wmic logicaldisk get name
## Name
## C:
## D:
##
```

In PowerShell, the equivalent command is

```
get-psdrive -psprovider filesystem | select name
##
## Name
## ----
## C
## D
##
##
```

The default prompts in Command Prompt and PowerShell show the current working directory's path (including the root volume). The current directory is the base path used to resolve relative paths in the command line interface and inside programs that run from it. In PowerShell, you may use the `pwd` command (or the special variable `$PWD`) to get the path of the current directory as a `PathInfo` object:

```
pwd
##
## Path
## ----
## C:\Users\Jane
##
##
```

On Command Prompt, the equivalent is running `echo %CD%`. On either command-line interface, the command `cd`, followed by an absolute or a relative path, changes the current directory to the specified one. One peculiarity of Command Prompt is the association between a volume and its current directory; here's an example that shows how to change the current directory properly when dealing with different volumes on Command Prompt:

```
rem switch to d:
d:
cd \
cd c:\windows
echo %CD%
rem to change directory to c:\windows, switch to c:
c:
echo %CD%
## D:\
## C:\Windows
##
```

To workaround this, it's recommended to use the `pushd` command, which is also available for bash, instead to avoid any confusion: it stores the current directory for use by the `popd` command, then changes the current directory to the specified one (including the volume change). PowerShell changes both, the volume and current directory, when the `cd` command is issued.

Each volume contains its own tree of directory hierarchy, which can be viewed using the `tree` command:

```
tree /A
## Folder PATH listing
## Volume serial number is 0123-4567
## D:.
## +---Data
## |   \---Speech
## \---Code
##
##
```

The table below displays useful special paths in Windows:

Path	Description
~	home directory of current user (in PowerShell and bash)
.	current directory
..	parent directory of current directory
~X	home directory of user X

Displaying Files

The following commands are useful for displaying or examining file contents on PowerShell:

Command	Description

`cat X`
`Get-Content X`
`gc X`
	displays contents of file `X`

`Compare-Object X Y` `diff X Y`	compares two sets of objects: `X`, `Y` and displays unmatched items

The following commands are the equivalent on Command Prompt (which still work on PowerShell):

Command	Description
`type X`	displays contents of file `X`
`fc X Y`	compares two sets of objects: `X`, `Y` and displays unmatched items
`diff X Y`	compares two sets of objects: `X`, `Y` and displays unmatched items

Moving, Copying, and Removing Files and Directories

The following commands are useful for moving, copying, and removing files and directories:

Command	Description
`del X`	deletes the file `X` or all files in the directory `X`
`rd /S X`	removes the directory tree `X` (all directories and files in the specified directory in addition to `X` itself)
`rd X`	removes the empty directory `X`
`md X`	creates a new empty directory `X`
`md X\Y\Z`	creates the path of nested directories for `X\Y\Z`
`move X Y`	moves the file or directory `X` to `Y`
`copy X Y`	copies the file `X` to `Y`
`xcopy X Y`	copies files and directory trees from `X` to `Y`

Note that deleting a file or a folder from the command line interface in Windows is different from moving it to the Recycle Bin; you won't find it there upon deletion and hence would be much harder to restore. A more error-prone behavior is using the `copy` command to copy directories (instead of `xcopy`), it will actually create an empty file as the destination; the worst part is it will have an exit code of zero (success) in this case with no indication of error.

Wildcard Characters

The Command Prompt offers two convenient wildcard characters for matching multiple files or directories with a specific pattern. They are listed in the table below:

Symbol	Description
`*`	any string of characters (including the empty string)
`?`	any single character (or none)

Since those two characters are used as wildcard characters, they cannot be used in any file or directory name.

In addition to the above, PowerShell supports `[X]`, which matches any character in the set or range `X`; e.g., `[mca]` or `[b-y]`.

The PATH Environment Variable

The `PATH` environment variable specifies a list of one or more directories (separated by `;`) that are used to search for executable programs; each process has its own `PATH` setting that is by default inherited from its parent process or the user's settings. For example, when issuing the command `foo` using PowerShell, the latter looks for the first match of the executable to launch in the current directory then in the directories specified by the `PATH` environment variable.

Compression

A quick way to compress files in Windows is using the `compact` command. The compression ratio depends on the original file, but is usually significant for text files. Files are compressed in place when the `compact` command is used; here's an example:

```
compact /c foo.txt
##  Compressing files in C:\Users\Geish\Desktop\
## foo.txt                  5967 :      4096 = 1.5 to 1 [OK]
## 1 files within 1 directories were compressed.
## 5,967 total bytes of data are stored in 4,096 bytes.
## The compression ratio is 1.5 to 1.
##
```

Running the `compact` command without options displays the compression status for the given files. The following options are commonly used:

See `compact /?` for more options.

Option	Description
`/C`	compresses the specified files
`/U`	uncompresses the specified files
`/S`	preforms the specified operation recursively (for all files in the given directory tree)
`/F`	force-compresses all files (including already-compressed ones, which are otherwise skipped)

3.7 Users and Permissions

Most modern operating systems enable multiple users to log-in at the same time, run concurrent processes, and ensure that the different users do not interfere with other users' processes and data.

In most cases the set of users is partitioned into regular users and super-users. Regular users are not able to write or modify essential operating system files, modify or update the operating system itself, and access private files belonging to other users that are marked as private. Super-users have permission to read, write, and modify all files in the disk, including files belonging to the operating system or to other users.

The file permission policy specifies which files each regular user is allowed to access. In most cases, permissions can be granted for specific tasks such as reading a file, writing a file (includes removing it or updating it), and executing a file. For example, user 1 may have read and write access to file 1 and file 2 while user 2 may have only read access to file 1 and only execute access to file 2.

A convenient way to specify file permission policies is to grant read, write, or execute permissions to entire groups of users (rather as specifying user permissions on an individual basis). For example, the operators user group may have only execute permission to several files while the programmers user group may have read, write, and execute permissions.

3.7.1 Users and Permissions in Linux

There are two ways to execute in Linux privileged commands that are available only to super-users.

- Prefix the privileged command with `sudo`.
- Execute the `su` command, which opens a new shell that can execute privileged commands where the current user is `root`.

It is important to be very careful when executing privileged commands since they can lead to deleting important user information or corrupting the operating system by modifying or removing its files.

To log-in as a specific user, the user name must be matched with the corresponding password. The Linux command `passwd` modifies the password of the current user.

Passwords are usually recorded in either `/etc/passwd` or `/etc/shadow` in encrypted format. The file `/etc/passwd` is readable to all users as it contains additional nonconfidential user information. The file `/etc/shadow` is usually readable only to super-users. Since the passwords are recorded in encrypted format, no user or super-users may decipher the passwords.

Below are some commands for examining user names and additional information in Linux.

Command	Description
`who`	displays currently logged-in users and when they logged-in
`whoami`	displays the current user's name
`hostname`	displays the computer's name
`finger`	similar to `who`, but displays also how long has passed since the last time a user has been idle
`finger X`	display additional information on user `X`
`w`	displays a list of currently logged-in users, annotated with CPU time used, CPU load, and currently executed programs.

The command `ls -l X` displays detailed information on the file or directory `X`, including its access permission (see Fig. 3.5). The first character displays the type: file, directory, or symbolic link. The next three groups of three letters each display permissions for the owner, user group, and other users. Each group of three letters correspond to read, write, and execute permissions. For example, the file in Fig. 3.5 has read, write, and execute permissions for the owner user `joesmith` (first triplet `rwx`), read and execute permissions for all users in the group `staff` (second triplet `r-x`), and read and execute permissions for all other users (third triplet `r-x`). The last fields are the number of links to the file, the user who owns the file, the group of users assigned to the file, the file size, the last modification date, and the file name.

The command `chmod` modifies the permissions, assuming that the user issuing the command is the owner of the file (or a super-user using the `sudo` prefix). The command `chown` modifies the owner of a file and the command `chgrp` modifies the user group. The example below demonstrates the use of `chmod` and `chown`.

```
ls -a file.txt

## -rw-r--r-- 1 joe  staff   1.9K Feb  6 14:06 file.txt

chmod g+w file.txt # add write permission to group
ls -a file.txt

## -rw-rw-r-- 1 joe  staff   1.9K Feb  6 14:06 file.txt
```

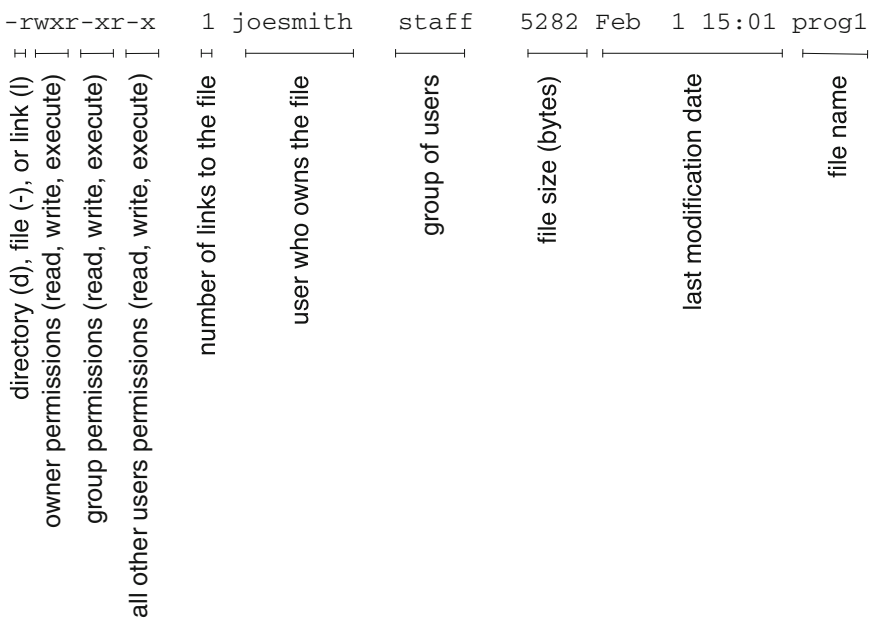


Fig. 3.5 Display format for the `ls -l` command for an executable file named `prog1`, owned by user `joesmith`. See text for more details

```
chmod a+w file.txt # add write permission to all users
ls -l file.txt
## -rw-rw-rw- 1 joe staff 1.9K Feb 6 14:06 file.txt

chmod a-w file.txt # remove write permission from all users
ls -l file.txt
## -rw-rw-r-- 1 joe staff 1.9K Feb 6 14:06 file.txt

chown jane file.txt
ls -l file.txt
## -rw-rw-r-- 1 jane staff 1.9K Feb 6 14:06 file.txt
```

There are also other ways to invoke `chmod` using numeric flag arguments. See `man chmod`, `info chmod`, or the online documentation for more information.

3.7.2 Users and Permissions in Windows

Typically, while running commands in Windows, users are not allowed to run commands that require administrative privileges. To change that, given the logged-in user can escalate her privileges to that of an administrator, right-click the icon for the program to execute (e.g., Command Prompt or PowerShell) and choose "Run

as administrator" from the context menu. To run commands as other users from a command-line interface, you may use the `runas` command like in the example below:

```
runas /user:geish "cmd /k echo hello"
```

The above starts a new Command Prompt that executes the `echo` command then keeps the window open. To know who the current user is, the command `whoami` can be handy:

```
whoami
## my-desktop\geish
##
```

The following commands can be used on Windows to get details about the current computer and users:

Command	Description
`hostname`	displays computer name
`finger`	displays information about a user on a host that's running a Finger service
`net`	displays and manages users, groups, computers, etc.
`net users`	displays a list of user accounts on the current computer

When listing files and sub-directories using the `dir` command, adding the `/q` flag displays the owners of said files and directories. To view or manage permissions, one may use the `icacls` command. In PowerShell, the `Get-Acl` cmdlet shows the security descriptor for a resource (e.g., a file); the `Set-Acl` cmdlet is used to change permissions.

3.8 Input and Output

The operating system provides a convenient mechanism for accessing input and output devices. Specifically, it provides programs that read input from input devices such as keyboard or mouse, and that write output to output devices such the display or printer. A second important role of the operating system is to ensure that multiple concurrent processes do not access the same input or output device at the same time, potentially overwriting the data.

3.8.1 Redirecting Input and Output in Linux

By default, the terminal accepts input from the keyboard and sends output to the display. The terminal offers a convenient mechanism to read input from a file (instead of the keyboard) and to send output to a file (instead of the display). That mechanism, called input-output redirection, uses the < symbol to redirect input and the > symbol to redirect output. The symbol » redirects and appends output to the end of the file (without overwriting existing contents).

The example below appends output from two different commands into a single file using output redirection:

```
echo hello >> output.txt
echo world >> output.txt
cat output.txt
## hello
## world
```

The table below provides some additional information and use-cases:

Command	Effect
`X < Y`	execute command `X`, reading input from file `Y`
`X > Y`	execute command `X`, writing output to file `Y`
`X < Y > Z`	execute command `X`, reading input from file `Y` and writing output to file `Z`
`set -o noclobber`	refuse to overwrite existing files with I/O redirection
`X >| Y`	same as above, but overwrites an existing file even if noclobber variable is set
`X >> Y`	execute command `X`, writing output to end of file `Y` without removing existing content (append)
`tee X`	execute the command `X` and send input to both the display and to file `X`
`tee -a X`	same as above, but append to file

Using input-output redirection in conjunction with pipes is very useful. A couple of examples appear below that use the command `tr X Y` that replaces characters in its input translating the set `X` into corresponding characters in the set `Y`.

```
# create a text file and then append to it its uppercase version
echo this is a text file > a.txt
tr "a-z" "A-Z" < a.txt >> a.txt
cat a.txt
## this is a text file
## THIS IS A TEXT FILE

# convert text file to a sorted list of distinct words
# annotated with their count
```

```
echo this file is a text file > b.txt
tr < b.txt -cs "[:alpha:]" "\n" | sort | uniq -c
##    1 a
##    2 file
##    1 is
##    1 text
##    1 this

tr < b.txt -d ' ' # remove all white spaces
## thisfileisatextfile
```

More details on the `tr` command may be found by typing `man tr` or `info tr` or in online documentation.

The `lp` command sends files to the printer for printing. The command `lpstat` shows the current print queue and the command `cancel` cancels specific printing jobs. The table below lists the typical usage of these commands. Use the `man` or `info` commands (or online documentation) for details and additional flags.

Command	Description
`lp -d X Y`	print file `Y` to printer `X`
`lpstat`	shows print queue of default printer
`cancel id`	removes job `id` from default printer queue

An alternative way to interact with the printer is using the `lpr` (print), `lpq` (display printer queue), and `lprm` (remove print job from printer queue) commands. It is also possible to interact with the printer using the print menu of applications such as web browsers, pdf viewers, or word processors.

3.8.2 Redirecting Input and Output in Windows

Like in Linux, command-line interfaces accept input from the keyboard and send output to the display. They also offer a convenient mechanism to read input from a file (instead of the keyboard) and to send output to a file (instead of the display). That mechanism, called input-output redirection, uses the < symbol to redirect input and the > symbol to redirect output. The symbol » redirects and appends output to the end of the file (without overwriting existing contents).

The example below appends output from two different commands into a single file using output redirection:

```
echo hello >> output.txt
echo world >> output.txt
type output.txt
## hello
## world
```

The `print` command sends a text file to the printer for printing. In PowerShell, the following cmdlets may be used to deal with printing:

Command	Description
`Add-Printer`	Adds a printer to the specified computer
`Add-PrinterDriver`	Installs a printer driver
`Add-PrinterPort`	Installs a printer port
`Get-PrintConfiguration`	Gets a printer's configuration info
`Get-Printer`	Retrieves the list of printers installed
`Get-PrinterDriver`	Retrieves the list of printer drivers
`Get-PrinterPort`	Retrieves the list of printer ports
`Get-PrinterProperty`	Retrieves printer properties
`Get-PrintJob`	Retrieves a list of print jobs
`Read-PrinterNfcTag`	Reads information about printers from an NFC tag
`Remove-Printer`	Removes a printer
`Remove-PrinterDriver`	Deletes printer driver
`Remove-PrinterPort`	Removes the specified printer port
`Remove-PrintJob`	Removes a print job
`Rename-Printer`	Renames the specified printer
`Restart-PrintJob`	Restarts a print job
`Resume-PrintJob`	Resumes a suspended print job
`Set-PrintConfiguration`	Sets a printer's configuration info
`Set-Printer`	Updates the configuration of a printer
`Set-PrinterProperty`	Modifies the printer properties
`Suspend-PrintJob`	Suspends a print job
`Write-PrinterNfcTag`	Writes printer connection data to an NFC tag
`Out-Printer`	Sends output to a printer

3.9 Networking

Many operating systems offer networking services that connect to remote computers. These services are in turn used by applications that provide Internet browsing, email communication, and other networking capabilities.

3.9.1 Working on Remote Linux Computers

The identification of a remote computer connected to the Internet is usually done in terms of its domain name or its IP address. The domain name is a

character sequence separated by periods such as `server1.domain.com` and the IP address is a sequence of numbers separated by periods. Both the domain name and the IP address may characterize a specific computer connected to the Internet. Specifying a specific username is done by prefixing the domain name with the username followed by a `@` symbol. A specific path in the remote file system can be references by appending the path to the domain name. For example, `joe@server1.domain.com/home/joe` refers to user `joe`, the computer `server1.domain.com`, and the directory `/home/joe`.

The `ssh X` command opens a new shell that operates in the remote computer `X` and encrypts the communication between the local and remote computers.

```
ssh joe@server1.domain.com
```

Jobs executed during an `ssh` session on a remote computer are terminated when the user logs out or when the network connection is lost. The command `nohup X &` ensures that the command `X` will continue execution in the background even after the user logs out or the connection is lost.

The `ftp` command transfers files between a local and a remote computer. After authentication, an `ftp` prompt appears that accepts the following commands.

Command	Effect
`put X`	transfer file `X` from local computer to remote computer
`mput X`	transfers multiple files specified by `X` from local computer to remote computer
`get X`	transfer file `X` from remote computer to local computer
`mget X`	transfers multiple files specified by `X` from remote computer to local computer
`ascii`	switch transfer mode to ascii (text files)
`binary`	switch transfer mode to binary (non-text files)
`cd`	change directory on the remote computer
`lcd`	change directory on the local computer
`help`	display list of available ftp commands
`bye`	quit the ftp program

An alternative to `ftp` that uses encrypted communication is the `scp` command.

The `scp` command copies files from a local computer to a remote computer or vice versa. The command `scp X Y` uses a format similar to the `cp X Y` command and may accept wildcards, except that `X` or `Y` may have a domain name prefix specifying the address of the remote computer. If no prefix appears the argument is assumed to refer to the local computer.

```
# copy file /home/joe/file1 from server1.domain.com
# (authenticating as user joe) to local home directory
scp joe@server1.domain.com/home/joe/file1 ~/
# copy entire home directory (including sub-directories)
```

```
# on server1.domain.com to local ~/tmp
scp -R joe@server1.domain.com/home/joe/* ~/tmp/
# copy local files ~/file2.* to home directory on remote computer
scp ~/file2.* joe@server1.domain.com/home/joe/
```

The following table summarizes these commands.

Command	Effect
`ssh X`	creates a secure shell operating on a networked computer `X`
`nohup X &`	launches the command `X` in the background, without stopping execution when the user logs out
`ftp X`	file transfer protocol (ftp) program for transferring files between computer `X` and computer `Y`
`sftp X`	a secure version of ftp
`scp X Y`	secure file copy from location `X` to location `Y` `X` and/or `Y` may be on the same computer or a remote one

3.9.2 Working on Remote Windows Computers

It's hard to compete with the seamless experience of working on a remote Windows computer. The "Remote Desktop Connection" application can be launched from the GUI or using the `mstsc` command. It provides ample options, to list a few:

Option	Description
`/v:<server[:port]>`	Specifies the remote PC
`/admin`	Connects to the session for administering a remote PC
`/f`	Starts Remote Desktop in full-screen mode
`/span`	Matches the remote desktop width and height with the local virtual desktop, spanning across multiple monitors, if necessary
`/prompt`	Prompts you for your credentials when you connect to the remote PC
`/prompt`	Prompts you for your credentials when you connect to the remote PC

The GUI allows for customizing options like tunneling local resources (audio, drives, printers, clipboard, etc.), adaptive remote experience based on network connection speed, etc.

Accessing network locations is also easy using Windows: folders can be explicitly shared with the network or specific users; administrators can access locations they already have access to even if they are not explicitly shared using the Administrative Share feature (e.g., `remote-server | $` to access the `C:` drive remotely).

3.10 Notes

Additional details on the roles and structure of operating systems are available in standard textbooks, such as (Silberschatz et al., 2008). More information on the Linux operating system are available in online documentation and in books that focus on Linux or one of its distributions, such as (Sobell, 2010). A useful resource on the bash shell is (Newham, 2005).

References

A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts*. Wiley, eighth edition, 2008.
M. G. Sobell. *Practical Guide to Ubuntu Linux*. Prentice Hall, third edition, 2010.
C. Newham. *Learning the BASH Shell*. O'Reilly Media Inc., third edition, 2005.

Chapter 4
Learning C++

C++ is a programming language that is especially well suited for computationally intensive programs and for interfacing with hardware or the operating system. In this chapter, we describe C++ starting with low-level features such as variable types, operators, pointers, arrays, I/O, and control flow, and concluding with object-oriented programming and the standard template library. We consider the latest version of C++ at the time of writing: C++17.

C++ evolved from the C language, which was initially developed by Ken Thompson and Dennis Ritchie in 1973 at Bell Labs. C++ was initially designed by Bjarne Stroustrup in the early 1980s and continued to evolve until the most recent 2017 standard. C++ was designed with efficiency and hardware in mind; many modern projects that are written in C++ would cite those two reasons for the choice of the programming language. In other words, the use of C++ nowadays needs to be justified due to its relative complexity (which is caused, at least in part, by its flexibility). For example, at Voicera, we use C++ to perform various tasks of Automatic Speech Recognition (ASR) that need to squeeze each bit of performance. C++ influenced other programming languages, like Java and `C#`, in their design. More importantly, many more languages (like Java, Python, R, and Go) interface with C++ and so they can load native (C++) libraries.

If you're working on a Linux or a macOS computer, chances are you already have the tools required to start building C++ applications. On Windows, and other operating systems, you may download an Integrated Development Environment (IDE) that makes developing C++ relatively easier; we recommend Visual Studio Code from https://code.visualstudio.com/download; it's free and open source. In this chapter, we will show examples that assume you're using the Linux terminal, so your mileage may vary.

© Springer Nature Switzerland AG 2018

G. Lebanon, M. El-Geish, *Computing with Data*,
https://doi.org/10.1007/978-3-319-98149-9_4

4.1 Compilation

Definition 4.1.1 The C++ compiler is a program that converts C++ code to a sequence of assembly language instructions that are executed by the CPU (see Chap. 2). The execution of the compiler program is known as the compilation process.

The compilation process is composed of the following two stages:

1. Compile files containing C++ code (source files), into object code files.
2. Link object code files into a single executable file containing assembly language instructions.

Occasionally, the first stage above is referred to as compilation and the second stage is referred to as linking. In other cases, compilation refers to both stages.

In large projects containing multiple C++ source files, the separation of the compilation process to two stages is particularly useful. If one of the source files is revised, the compilation stage (step 1) has to be repeated for that source file alone (the object files corresponding to the unchanged C++ files are still valid). The linking stage (step 2) needs to be repeated in order to create a new executable. Avoiding the compilation (stage 1) of the unchanged source files can significantly speed up the compilation process.

Since the executable file contains assembly language instructions, it matches a specific operating system and hardware and is not portable. As a result, the compilation process must be repeated for each hardware architecture, and the executable that is the result of a compilation process on one hardware architecture may not execute on a second architecture (or in some cases may execute more slowly).

C++ code is composed of a sequence of case-sensitive statements separated by semicolons and arranged in functions or classes. The function `main` is the entry point to the program, or in other words the CPU executes its code whenever the operating system executes the corresponding executable file.

C++ code is often annotated with comments that provide documentation on the code. Such comments are ignored by the compiler and thus they do not affect the program's functionality. There are two ways to define comments in C++: (a) text following `//` that continues until the end of the line, and (b) text between the symbols `/*` and `*/` (possibly spanning multiple lines).

Consider for example the following C++ program that simply returns the value 0 to the operating system (the returned value is accessible in the Linux operating system through the variable `$?` and in the Windows operating system through the environment variable `ERRORLEVEL`):

```
// this is a comment.
/* this is another comment
   that spans multiple lines. */

int main() {
  return 0;  // return a value 0 to the operating system
}
```

Assuming that the above code is saved as a text file `foo.cpp` (the standard file name extension for C++ is `.cc` or `.cpp`), we can compile and link it in Linux by executing the following terminal command, which calls the standard C++ compiler `g++`:

```
# compile and link foo.cpp into an executable file named foo
g++ foo.cpp -o foo
```

The string following the `-o` flag above indicates the name of the output file created by the compiler. Above and elsewhere we assume that the Linux terminal uses the bash shell or the zsh shell. Note that while C++ comments are denoted by `//` or `/* */`, comments in the Linux terminal are prefixed by a # symbol.

Alternatively, we can execute the two compilation stages separately: compilation only (using the `-c` flag) followed by linking, as in the example below:

```
# compile (but do not link) the C++ code in foo.cpp
# into the object file foo.o
g++ -c foo.cpp -o foo.o
# link the object file foo.o and create an executable file foo
g++ foo.o -o foo
```

The `g++` compiler supports the following compilation flags. Consult the `g++` documentation for more details (for example, using the commands `man g++` or `info g++` or by searching the online documentation).

flag	effect
`-o filename`	specify name of the output file
`-c`	compile but not link
`-g`	maintain information for a future debugging session
`-llib_name`	link with the library file `lib_name`
`-lm`	link with the standard math library
`-O3`	optimized compilation (level 3)
-std=c++0x	compile using the 2011 C++ specification

Optimized compilation typically takes longer but produces executable code that runs faster.

For example, the following shell command compiles multiple C++ files into a single executable, using the C++11 standard, optimized compilation, and linking to the standard mathematical library:

```
g++ -o foo -lm -O3 -std=c++0x foo.cpp
```

After compilation, typing the name of the executable file in the terminal executes the corresponding program (assuming (i) the path of the executable file is either explicitly provided or is in the `PATH` shell environment variable, and (ii) the user has executable permission for the executable file (see Sect. 3.7 for more details on file permissions)).

For example, after running the executable corresponding to the C++ file `foo.cpp` above, the returned value is 0. The Linux commands below execute the executable file and read the value returned by the C++ program (that value is available as the shell variable `$?`). As in other chapters we follow the convention that program or shell output is prefixed by a double comment symbol (the last line below), which makes it easy to copy and paste the program from electronic versions of this book to the command prompt:

```
# compile and link to create an executable file foo
g++ -o foo foo.cpp
# run the executable file foo (in the current directory)
./foo
# display value returned by the compiled program
echo $?

## 0
```

We proceed below to explore the C++ programming language. We start with low-level features such as variables, control flow, functions, and input and output. We continue with object oriented programming, and follow up with the standard template library, both of which are very useful when programming complex programs.

4.2 Types, Variables, and Scope

Variable types in C++ correspond to types of data such as integers, floating-point numbers, ASCII characters (see Chap. 2), or boolean true/false values. Variables are specific instantiation of variable types and thus hold actual values. There may be many variables holding the same variable type, but each distinct variable has a variable name that is used for referencing it. C++ is a strongly typed language in the sense that variables that are defined to be of a specific type cannot be modified to hold a different type.

Classes and structs in C++ are types that can hold more complex information and can be customized by the programmer. In the same way that a variable is an instantiation of a specific type, an object is an instantiation of a specific class or struct. We describe below types and variables and postpone a description of classes and objects to Sect. 4.10.

4.2.1 Types

The type `int` is commonly used to represent integers, while the type `float` is commonly used to represent real numbers using floating-point representation (see Chap. 2 for a description of integers and floating-point representations). The precise range of values that can be represented (and the approximation precision in the case of the floating-point representation) depends on the number of bytes allocated to each type, which is hardware dependent. The typical size of `int` and `float` types is 4 bytes or 32 bits.

The types `long` and `double` are similar to `int` and `float`, respectively, but are usually allocated more bytes and thus may represent larger numbers (and with higher approximation precision in the case of `double` floating-point representation).

The type `char` uses 8 bits to represent numbers between −128 and 127. It is also used to represent text characters such as "a" or "M" using the ASCII encoding (see Chap. 2).

The type `bool` represents two possible values: `true` and `false`. Most numeric types can represent either zero, positive or negative values, but have an `unsigned` version that represents only nonnegative numbers.

type	value	typical size (bits)
`int`	integers	32
`long`	integers	64
`char`	ASCII (text) character	8
`bool`	`true` or `false`	8
`float`	real numbers	32
`double`	real numbers	64
`unsigned T`	nonnegative version of type `T`	same as `T`

4.2.2 Variables

As mentioned earlier, variables are instantiations of types. Each variable is assigned a specific type when it is defined and that type cannot be modified later on.

A C++ variable definition statement includes the variable type followed by the variable name. Several variables of the same type may be defined using comma separators.[1] Variable names are case-sensitive and must avoid certain special characters[2] and existing C++ keywords. It is common to name variables with multiple words by separating the words with an underscore, for example `training_data`, or alternatively by capitalizing the first letter of each word, except perhaps the first word, for example `trainingData`.

The code below defines three variables: a variable `age` of type `int` and two variables `height` and `weight` of type `double`:

```
int age; // age is a variable of type int (integer)
// two double variables
double height;
double weight;
```

C++ code typically resides inside `main()` or some other function or class definition (some exceptions exist as in the case of global variables). However, for simplicity we sometimes omit below the embedding and display isolated code fragments.

The variable definitions above did not assign a value to the variables. Variables may be initialized when they are defined, or later on, using the assignment operator `=`. Uninitialized variables may have unexpected values and should not be used until they are assigned values. It is customary to define variables close to where they are used and assign values to them as soon as possible, and to avoid defining variables that are never assigned values.

Standard algebraic expression may be used to manipulate integers and floating point variables. The addition operator `+`, subtraction operator `-`, and multiplication operator `*` work as expected with the usual rules of operator precedence (for example, `1 + 2 * 3` is 7 rather than 9). Parenthesis may be used to override the standard operator precedence or to make the code more readable:

```
int age = 32; // integer variable holding 32
double height; // unassigned variable
float pi = 3.14; // new float variable
float pi_squared = pi * pi; // new float variable

height = (5 * 30.48) + (10 * 2.54); // assigs a value to height
```

[1] We recommend to declare each variable on its own line to avoid declarations that can cause errors; for example, `int* x, y;` is not the same as `int* y, x;` as the `*` belongs to the variable name and not the type. See Chap. 15 for more style guidelines.

[2] The underscore character is allowed; some C++ compilers allow other special characters, but we recommend against using them to make the code more portable.

Marking a variable as `const` during its definition indicates that its value is fixed and may not change later on. Attempting to modify a `const` variable after it is defined will result in compilation error[3]:

```
int a = 2;
const int b = 3; // more readable
int const c = 5; // alternative form

a = 6;  // ok
b = 6;  // error (b cannot be modified)
```

The 2011 C++ standard (C++11) includes a way of defining variables whose types are implicitly inferred by the compiler, rather than explicitly specified by the programmer. These variables are marked by the `auto` keyword:

```
double height = 58.0;
double weight = 155.2;
auto bmi = weight / (height * height); // inferred type (double)
```

4.2.3 Scope

The scope of a variable corresponds to the portion of code in which it is recognized. Variables defined inside curly braces have scope that is limited to the portion of the curly braces code block that follows the variable definition. Multiple curly braces blocks may be nested. Variables defined outside of any curly braces are considered global variables and have scope throughout the source file.

```
{
  int a = 2;

  a = a + 1; // ok, a is recognized

  {
    int b = a; // ok, a is still in scope
  }

  a = b; // error, b is out of scope and is undefined
}

a = a + 1; // error, a is out of scope
```

A variable may be defined inside a curly braces block that has the same name as a variable defined before the curly braces block. In this case the variable defined inside the curly braces block masks the variable defined before the curly braces block:

[3]The `const_cast` operator can be used to remove constness or volatility in many scenarios.

```
int a = 2;

{
  int a = 3;  // inner a (a=3) masks the outer a (a=2)
  int b = a;  // b is assigned the value 3
}

// inner a is out of scope and outer a is no longer masked
int c = a; // c is assigned the value of the outer a (a=2)
int d = b; // error: b is no longer in scope
```

However, global variables can be accessed using the scope resolution operator `::` as in the following example:

```
int a = 42; // global variable

int main() {
  int a = 13; // local variable
  int b = a + 1; // 14
  int c = ::a + 1; // 43

  return 0;
}
```

4.3 Operators and Casting

4.3.1 Operators

Unary operators take a single argument and return a value, while binary operators take two arguments and return a value. Unary operators appear before or after their argument, while binary operators appear between their two arguments. For example, the unary negation operator applied to the argument 2, `-2`, returns a value that is equal to the binary subtraction operator with a 0 as its first argument and a 2 as its second argument: `0-2`.

In most cases operators do not change the value of their arguments, for example `x+y` does not change the values of the variables `x` and `y` and `-x` does not change the value of `x`. In both cases a new value is returned, but the original values stored in `x` and `y` are unchanged. Notable exceptions are the binary assignment operator `x=y`, which assigns the value of `y` to `x` and the unary operators `++x`, `-x`, which assign to their argument its original value incremented or decremented by one.

The following table lists commonly used operators.

operator	semantics
`x + y`	arithmetic plus
`x - y`	arithmetic minus
`-x`	arithmetic negation
`x * y`	arithmetic multiplication
`x / y`	arithmetic division
`x % y`	remainder after division
`x = y`	assignment of `y` to `x`
`x == y`	equality test
`x != y`	inequality test
`x < y`	arithmetic less than
`x <= y`	arithmetic less than or equal to
`x > y`	arithmetic greater than
`x >= y`	arithmetic greater than or equal to
`++x`	pre-increment `x` by one
`-x`	pre-decrement `x` by one
`x && y`	logical AND (true if x and y are true)
`x || y`	logical OR (true if x or y are true)
`!x`	logical NOT (true if x is false)
`sizeof(x)`	returns size of variable x in bytes
`x, y`	evaluate `x`, and then evaluate and return `y`

The logical operators above return a `bool` value: `true` if the corresponding condition holds and `false` if it does not hold.

When multiple operators co-occur, standard operator precedence and associativity rules apply, for example `a * 2 + 1` is interpreted as first multiply `a` by 2 and then add 1 to the result. Parenthesis can be used to make the code more readable or to override the standard behavior; for example, `a * 2 + 1` is equivalent to `(a * 2) + 1` but `a * (2 + 1)` returns `a` times 3. The code below illustrates the use of operators and parenthesis:

```
bool result;

result = (3 == 3); // result equals true
result = (3 > 3); // result equals false

int a = 3;
int b;

b = ++a; // b and a both equal 4
result = ((4 > 3) && (4 < 5)); // result equals true
result = !(1 > 2); // result equals true
a = 10 % 3; // a equals 1 (remainder after dividing 10 by 3)
b = (a = 5); // assign 5 to a, and then assign that value to b
```

It's worth noting that the assignment `b = (a = 3)` is error-prone; we recommend not to use that pattern because it could be confused with `b = (a == 3)`, which is equivalent to `b = 0` since 0 and `false` are interchangeable in C++.

4.3.2 Type Conversions

Since C++ is strongly typed, the type of a variable cannot be modified after the variable is created. A value of one type, however, can be converted to a compatible value of another type and assigned to a variable of a different type.

Definition 4.3.1 Casting, or type conversion, is the process of converting a value of one type to a compatible value of another type.

When both types are numeric, type conversions produce expected results.[4] For example, converting a `float` value that equals 0.0 to `int` produces the integer 0. Converting an `int` value 3 to a `float` produces the floating-point value 3.0.

Converting a less accurate type to a more accurate type (for example, converting `int` to `long`) does not result in any loss of information. Converting a more accurate type to a less accurate type may result in a loss of accuracy. For example, converting a `float` 3.2 to an `int` produces the integer 3.

We distinguish between explicit type conversions and implicit type conversions. Explicit type conversions are specified by the programmer, while implicit type conversions are triggered by the compiler. An explicit type conversion instructs the compiler to copy a value of one type to a related value in a different type.

Below is an example of explicit casting:

```
int i;
double d = 58.3;

i = (int) d; // converts 58.3 to int
```

Implicit type conversions are triggered by the compiler on a number of occasions. In general, implicit type conversions occur when the expected type is different from the present type.

- Non-boolean types are converted to boolean types when logical conditions are checked. Specifically, zero is converted to `false` and nonzero values are converted to `true`. For example, in the C++ statement `!3` the integer 3 is converted to a `bool true` value, which is then negated producing a `false` value.
- In variable initialization or assignment between related types, the value on the right-hand side is converted to the type of the variable on the left-hand side:

[4] See Sect. 2.6.4 for more details on rounding, overflow, and underflow—all of which may occur in casting.

```
int a = 3.2;  // 3.2 is converted to the integer 3
int b;

b = 3.2;  // same casting as above
b = 3.0 / 2.0; // 1.5 is converted to the integer 1
```

- When an operator that requires both arguments to have identical types receives mixed types, the two types are converted to a common type. If the two types are integer and floating point, the integer is converted to a floating point. If the two types are both integers or both floating points of different accuracy (for example, `int` and `long`), the less accurate type is converted to the more accurate type:

```
int a = 1;
int b = 2;
float f = 2.0f; // without the suffix f, 2.0 is a double

// Below, in the division a/f, the integer a is converted
// to a float resulting in a division of two floats:
// 1.0f/2.0f (which equals 0.5)
float g = a / f;  // g equals 0.5

// Below, there is no type conversion for the division a/b.
// The division of an integer 1 by another integer 2 gives
// the integer 0, which is then converted to a float 0.0
// that is assigned to h
float h = a / b;  // h equals 0.0
```

4.4 References and Pointers

References and pointers in C++ refer to a variable or an object (see Sect. 4.10 for more information on objects). They allow creating multiple entities that refer to a single variable or object in memory. Pointers are also closely associated with arrays, which will be introduced in the next section.

4.4.1 References

References in C++ define alternative names (aliases) for existing variables or objects; they are marked by prefixing the alias with the `&` symbol; for example, `int &refA` is a reference to an `int` variable. A reference does not define a new variable and any attempt to read or modify its value will affect the variable or object to which it refers. References must be initialized at the moment they are defined. There may be many references referring to a single variable or object.

The code below defines several variables and references:

```
int a = 2;
int c = 3;
int &refA = a; // refA is a reference to the variable a
int b = refA; // has same effect as b = a

refA = 5; // has same effect as a = 5

int &refC1 = c;
int &refC2 = c;  // both refC1 and refC2 refer to c
```

References and pointers play an important role in polymorphism, which will be revealed in detail in Sect. 4.10.5.

4.4.2 Pointers

A pointer is a variable containing the memory address where another variable or an object is stored. Pointers are denoted by prefixing the pointer's name with the `*` symbol; for example, `int *p` is a pointer to an `int` variable. The type of a pointer ensures that the compiler knows how to interpret the contents of the memory to which it points:

```
int *pa;
int *pb;  // pa and pb are pointers to int variables
float *fx;
float *fy;  // fx and fy are pointers to float variables
```

After a pointer `x` is defined, the unary `*` operator (as in `*x`) refers to the content of the memory address pointed to by `x` (the pointer type should agree with the type of the data at the corresponding memory in order to interpret it correctly). The unary `&` operator (the address-of operator), `&x`, returns the memory address of the variable or object `x` as in the following example:

```
int a = 2;
int *b; // uninitialized pointer - may contain unexpected address
int *c = &a; // address of variable a is assigned to pointer c
int d = *c; // value at address c is assigned to d (d = 2)
// define a new pointer e that points to the same memory as c
int *e = c;
float f = 3.0f;
float *fp = &f;  // ok
int *a = &f;  // problem: a is of type int* but points to float
```

Programming errors due to pointers are common in C++ and can be hard to discover. A standard example is accessing a prohibited or uninitialized memory location. It is best to avoid defining uninitialized pointers and to ensure that pointers always point to a legitimate memory address.

A double pointer is a variable that holds a memory address whose content is a pointer. Thus, if `pp` is a double pointer to int, then `*(*pp)` or just `**pp` is the

corresponding `int` variable. A double pointer `x` to type `T` is defined by the statement `T **x;` as in the following example:

```
int a = 2;
int *b = &a; // pointer b points to a
int **c = &b; // double pointer c points to b
// c now holds the address of b, which holds the address of a
int *d = *c; // contents of b (address of a) is assigned to d
int e = *(*c); // contents of a (2) is assigned to e
```

C++ also allows triple pointers; in fact, it allows any number of levels of indirection: they are usually used to create multidimensional dynamically allocated arrays (see Sect. 4.5.2), but they are less common than pointers and double pointer.

C++ allows references to pointers but not pointers to references.

4.5 Arrays

An array represents a contiguous portion of memory that can be used to store a sequence of variables or objects. We describe below one-dimensional arrays, followed by multidimensional arrays.

4.5.1 One-Dimensional Arrays

The square brackets operator references specific elements of the array. Square brackets are also used when defining arrays. Some examples appear below:

```
int a[10]; // define an array of 10 integers

a[0] = 1;  // assign 1 to the first element
a[3] = 2;  // assign 2 to the fourth element

int b = a[3]; // assign to b the contents of the fourth element
```

Once an array has been defined in C++, its length cannot be modified. Growing or shrinking an existing array requires defining a new array (that is larger or smaller) and copying elements of the old array to the new array.

There is a close connection between arrays and pointers in C++. An array in C++ may be considered as a constant pointer (whose address cannot be modified) to the first element of the array. Similarly, the bracket notation `x[k]` may be applied to a pointer `x` to refer to the contents of the pointer offset by `k` positions. The precise memory address pointed to by `x[k]` is `x` plus `k` times the size of an array element (the size of an array element depends on the type of elements the array holds, for example `int` or `double`). Some examples appear below:

```
// using an array as a pointer
int a[10];

*a = 3; // assign a value of 3 to the first element
*(a + 2) = 4; // assign a value of 4 to the third element

// using a pointer as an array
int *pA = a + 1; // pA points to the second element of a

pA[1] = 5;  // assign a value of 5 to the third element of a
```

A convenient way to define arrays with predefined values is using the following notation:

```
// define an array of size 10 containing the integers 0, 1,..., 9
int a[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
// array size may be omitted in this case
int a[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
// define an array of size 10 initialized to default
// values (0 for int)
int a[10] = {};
// define an array of size 10 initialized to 1, 0,..., 0
int a[10] = {1};
```

C++ assumes that the array size can be determined in compilation time. Specifically, the array size must be a constant variable or a numeric expression and it cannot be a variable (since the value of a non-const variable is indeterminable during compilation). This significant restriction can be alleviated using dynamic memory allocation (see Sect. 4.11).

```
const int N = 3;
double a[N];  // ok
int m = 3;
double b[m];  // error
```

The operating system ensures that when a program accesses the elements of a defined array, the program does not interfere with other programs or with the operating system. However, a dangerous and unpredictable behavior may occur when a program accesses a memory location that is outside the range of an allocated array. Such software bugs are hard to detect and may lead to erratic runtime behavior and possibly the untimely termination of the program.

```
int a[10] = {};
a[13] = 3; // dangerous bug - possible erratic behavior
```

4.5.2 Multidimensional Arrays

In many cases it is convenient to refer to a collection of values arranged as a two-dimensional table or a higher dimensional array. Such multidimensional arrays are defined using multiple pairs of square brackets. Similarly, multiple pairs of square brackets are used to reference a particular element of the array. As with one-dimensional arrays, the sizes of each dimension must be specified by constant variables or numeric expressions.

```
// a is 3 by 4 table of integers initialized to default values
// (0 in the case of int variables)
int a[3][4] = {};
a[0][0] = 2; // assign 2 to first row, first column element
a[0][1] = 3; // assign 3 to first row, second column element
a[1][2] = 4; // assign 4 to second row, third column element
// b is 3 by 4 by 5
int b[3][4][5] = {};
b[0][0][0] = 2; // first row, first column, first layer element
b[0][1][2] = 2; // first row, second column, third layer element
```

Multidimensional arrays in C++ are really arrays of arrays in the case of two-dimensional arrays or arrays of arrays of arrays in the case of three-dimensional arrays (and so on). Thus, if `a` is a two-dimensional array (for example, defined using `int a[3][4]`), then `a[1]` is an array corresponding to the second row of the table `a`.

4.6 Preprocessor and Namespaces

The preprocessor is a program that runs before the start of the primary compilation stage. It leaves most of the code intact with the exceptions of preprocessor directives, which are statements prefixed by the symbol # (Note that there is no need to include a semicolon at the end of the preprocessor directives; in fact, a warning is generated if there is a semicolon at the end of the directive). The preprocessor removes the preprocessor directives and manipulates the program accordingly.

A popular preprocessor directive is the `#include <X>` statement, which pastes a header file `X` in place of the directive. The program can then use constants, functions, and classes that are defined in the header file. Some notable `#include` statements are `#include <iostream>`, which includes input and output functionality, `#include <string>`, which includes string functionality, and `#include <cmath>`, which include mathematical constants and functions. Here's an example of how to use it:

```
// include the C++ input and output header file
#include <iostream>
```

```
// the statement below prints 3 (cout is an output stream defined
// in the header file iostream)
int main() {
  std::cout << 3;

  return 0;
}
```

The standard notation for including header files that are contributed by the programmer is `#include "X"` as in the following example:

```
// include a header file written by the programmer
#include "my_header_file.h"

// call my_function, which is a function defined in
// the header file my_header_file.h
int main() {
  my_function();

  return 0;
}
```

In either case, the .h header file needs to reside in one of the standard directories containing C++ header files (for example, `/usr/include` or its subdirectories) or the current directory. If the header file is not in the standard directory containing header files or in the current directory, a relative or absolute directory path may be included in the preprocessor directive (for example, `#include "subdir/my_header_file.h"`).

Header files typically contain only definitions, and not the implementation of the definitions. For example, a header file may contain the line `int foo();` without containing the implementation of the function `foo`. The implementation code typically resides in a non-header file that is compiled beforehand and is linked to the program (see Sect. 4.1). This guarantees that the implementation code is not compiled every time the file containing the header file is compiled.

The preprocessor directive `#define X Y` instructs the preprocessor to replace all occurrences of `X` with `Y`. They are sometimes used to name literals, which is an error-prone way to use macros instead of strongly typed constants.

```
#define WIDTH 80

int page_width = WIDTH;
int two_page_width = 2 * WIDTH;

// This code below is similar but harder to comprehend
// and maintain
int three_page_width = 3 * 80;
int four_page_width = 4 * 80;
```

The `#define` directive may also be used to define more complex macros. For example, the code below replaces all occurrences of `arraysize(A)` with the

number of elements of A (assuming that A is an array). The calculation divides the size of the array in bytes by the size of each element.

```
#include <iostream>
#define arraysize(array) (sizeof(array) / sizeof(*array))

int main() {
  int a[10];

  std::cout << arraysize(a); // prints the size of the array (10)

  return 0;
}
```

It is often the case that some header files contain #include directives that are used to include other header files within them. A potential problem arises if one header file is included more than one time in a single file (the compiler may complain that there are multiple definitions of the same function or class). For example, assuming that my_header_file1.h and my_header_file2.h both contain the line #include "my_header_file3.h" in them, the following program will lead to repeated definitions.

```
#include "my_header_file1.h"
#include "my_header_file2.h"
// constants or functions in my_header_file3 are defined twice
```

It's customary to solve this problem using the #ifndef X and #endif preprocessor directives, which instruct the preprocessor to ignore everything between these two directives if X is already defined. In the example above, we can avoid repeated definitions by copying the contents of my_header_file3.h inside the following structure.

```
#ifndef _MY_HEADER_3_GUARD
#define _MY_HEADER_3_GUARD

// copy original contents of my_header_file3.h here

#endif
```

In this way, the definitions within my_header_file3.h will appear a single time, no matter how many times the #include statement appears.

A namespace corresponds to a collection of variables and function names. The using namespace X command instructs the compiler to look at the namespace X when it tries to recognize names of variables or functions. Using namespaces allows multiple developers working independently to define different functions and variables with the same names. A program that uses these variables or functions can then disambiguate between the different definitions by including an appropriate using namespace X statement. An alternative is to prefix the variable or function name with the name of the namespace followed by a :: notation.

A common namespace representing the standard library is `std`. The statement `using namespace std` brings the standard library namespace into scope.

```
// option 1: Prefix cout by the namespace name and ::
std::cout << 3;  // print 3 (cout is defined std namespace)

// option 2: Using namespace std allows dropping the std:: prefix
using namespace std;
cout << 3;
```

4.7 Strings, Input, and Output

The traditional way in C (a precursor of C++) to represent text strings was to use an array of `char` variables, with each `char` variable representing a text letter using ASCII encoding. C++ includes the more sophisticated concept of a string that includes additional functionality. Unlike an array, a string's length may be modified after it is created. Additionally, C++ has functionality for concatenating, comparing, and transforming strings.

Strings may be initialized using a sequence of characters surrounded by double quotes. Specific string elements are referenced using the square bracket array notation.

```
#include <string>
using namespace std;

int main() {
  string s1;  // define an empty string
  string s2 = "hello world"; // define a string with content
  int sz = s2.size(); // assign size of string s2 to variable
  char a = s2[1];  // access second character of s2 ('e')
  string s3 = s2 + s2; // concatenate s2 with itself
  bool b = s2.empty();  // true if s2 is an empty string

  return 0;
}
```

The function `to_string` converts a numeric variable type to a string and the functions `stoi` and `stod` convert strings to integer and double variables, respectively (assuming that the string describes a number).

```
#include <string>
using namespace std;

int main() {
  int a = 123;
  string s = to_string(a);  // assign "123" to s
  int i = stoi(s);  // assign integer 123 to i
  double d = stod(s); // assign floating point 123.0 to d
```

```
  return 0;
}
```

In C++, `cin` reads contents from the standard input, typically character strokes entered by the keyboard when the program is executing. Specifically `cin » x` takes that input, separates it by white spaces, and assigns the first constituent into the variable `x`. Multiple variables may be included in a single `cin` command by separating them with the » symbol.

```
#include <iostream>
#include <string>
using namespace std;

int main() {
  string s1, s2;

  // read a string delimited by white space from standard
  // input into s1
  cin >> s1;

  // read two strings separated by white spaces (first into s1
  // and then into s2)
  cin >> s1 >> s2;

  return 0;
}
```

The input mechanism `cin` can also take numeric variables, in which cases it attempts to convert the inputs to the corresponding type.

```
#include <iostream>
#include <string>
using namespace std;

int main() {
  int c, d;

  // read buffered data from the standard input stream,
  // convert the first two inputs into integers, and
  // store them in c and d
  cin >> c >> d;

  return 0;
}
```

The `getline` function reads an entire line from standard input (a line is considered to be string ending in a newline character).

```
#include <iostream>
#include <string>
using namespace std;
```

```
int main() {
  string s;

  // read a line (possibly containing white spaces)
  // from standard input and assign it to s
  getline(cin, s);

  return 0;
}
```

In C++ `cout` prints strings or numeric variables to standard output, which typically shows in the terminal where the program is executing. Specifically, `cout << x` prints the value of the variable or string `x`. Multiple arguments may be used by separating them with `<<` symbols. The expression `endl` corresponds to the end of line character.

```
#include <iostream>
#include <string>
using namespace std;

int main() {
  string s = "hello world";
  // display hello world followed by an end of line
  cout << s << endl;
  // display hello world followed by ! and end of line
  cout << s << "!" << endl;

  return 0;
}
```

Compiling and executing the program above results in the following output:

```
## hello world
## hello world!
```

4.8 Control Flow

When a program is launched, the statements in the `main` function are executed from top to bottom, unless the control flow is modified using one of the mechanisms below.

4.8.1 If-Else Clauses

An if-else clause evaluates a condition and based on its value determines which of two clauses will be executed. If the condition holds, the first sequence of statements

surrounded by curly braces (if-block) will be executed. If the condition does not hold, the second sequence of statements surrounded by curly braces and coming after the `else` keyword will be executed (else-block). If the sequence of statements contains a single statement the curly braces may be omitted; we recommend to always surround conditional blocks with braces (see Chap. 15 for more style guidelines).

```
int a;
int abs_a;

cin >> a; // read a from terminal
// replace a by its absolute value and store it in abs_a
if (a < 0) {
  abs_a = -a;  // if a is negative, use -a
} else {
  abs_a = a;  // if a is non-negative, use a
}
```

As the example above shows, it is customary to indent the code inside the curly braces compared to the code outside of the curly braces. Omitting the else-clause is equivalent to specifying an empty else clause. For example, the following example achieves the same computation as the one above:

```
int a;
int abs_a;

cin >> a;
abs_a = a;
// negate abs_a if it is negative
if (abs_a < 0) {
  abs_a = -abs_a;
}
```

Alternatively, the ternary operator `? :` can be used to simplify the code:

```
int a;

cin >> a;

int abs_a = a < 0 ? -a : a;
```

The ternary operator (also known as the conditional operator) checks the condition preceding the question mark; if the condition holds, it returns the expression listed before the colon delimiter; otherwise, it returns the expression listed after the colon delimiter.

4.8.2 While-Loops

A while-loop repeats the statements within the curly braces as long as the specified condition is met. The optional `continue` statement skips the remainder of the current iteration and resumes execution from the start of the next iteration (as long as the loop condition is satisfied). The optional `break` statement discontinues the loop and resumes execution at the next command following the loop.

An example of a simple while-loop appears below:

```
// Assigns 4! = 4 * 3 * 2 = 24 to the variable fac_val
// 3 loop iterations will be executed (on fourth iteration the
// condition 1 > 1 fails).
int val = 4;
int fac_val = 1;

while (val > 1) {
  fac_val = fac_val * val;
  val = val - 1;
}
```

The example below demonstrates the `break` statement. Note that the condition `(1)` in `while(1)` indicates that the loop will never end unless a `break` statement is encountered:

```
// similar to previous example with break statement
int val = 4;
int fac_val = 1;

while (1) {
  if (val <= 1) {
    break;
  }

  fac_val = fac_val * val;
  val = val - 1;
}
```

The example below demonstrates the `continue` statement. The code computes $4 \cdot 2 = 8$. During the iteration that multiplies by 3, the `continue` statement is encountered and the remainder of the current iteration is skipped (the next iteration resumes afterwards).

```
// computes = 4 * 2 (iteration that multiplies by 3 is skipped)
int val = 4;
int fac_val_mod = 1;

while (1) {
  if (val == 3) {
    val = val - 1;
    continue;
  }
```

```
  if (val <= 1) {
    break;
  }

  fac_val_mod = fac_val_mod * val;
  val = val - 1;
}
```

4.8.3 For-Loops

The condition of the for-loop is separated into three components, delimited by semicolons. The first component initializes variables at the beginning of the loop execution. The second component is the condition that controls the loop execution (iterations are executed as long as it holds). The third component is executed at the end of each iteration, oftentimes incrementing an iteration variable.

The example below iterates over elements of an array. The first component of the condition `i = 0` initializes an iteration variable `i` to 0 at the start of the loop. The second component `i < 10` implies that iterations will be executed as long as the iteration variable `i` is smaller than 10. The last component `++i` implies that the iteration variable is incremented by one at the end of each iteration:

```
#include <iostream>
using namespace std;

int main() {
  int an_array[10];
  int i;

  // initialize an_array to hold 0, 1, ..., 9
  // 10 iterations are executed, corresponding to
  // i = 0, 1, ..., 9
  for (i = 0; i < 10; ++i) {
    an_array[i] = i;
  }

  // print the values of the array on a single line
  for (i = 0; i < 10; ++i) {
    cout << an_array[i] << ' ';
  }
  cout << endl;

  return 0;
}
```

Note that we could have omitted the curly braces associated with the second for-loop since it contained a single command; this also applies to the first for-loop, but

we recommend to surround the loop's body with curly braces even when there is a single command (see Chap. 15 for more style guidelines).

Compiling the above program and executing it gives the following output:

```
## 0 1 2 3 4 5 6 7 8 9
```

C++11 introduced a variation of for-loops using the syntax `for(x : X)` where `X` is an array or a different collection of variables or objects and `x` is a variable that assumes the value of a different element of `X` every iteration. This revised syntax is easier to read and less likely to result in bugs such as accessing inappropriate memory. For example, the code above can be rewritten as follows with the same effect:

```
#include <iostream>
using namespace std;

int main() {
  int an_array[10];
  int i = 0;

  for (auto &element : an_array) {
    element = i;
    ++i;
  }

  for (auto &element : an_array) {
    cout << element << ' ';
  }
  cout << endl;

  return 0;
}
```

Compiling the above program with the `-std=c++0x` compilation flag (to indicate that we are using C++11) and executing it gives the following output:

```
## 0 1 2 3 4 5 6 7 8 9
```

Note that the iteration variable `element` in both for-loops is specified as `auto`, implying that its type is inferred automatically by the compiler (see Sect. 4.2.2). The symbol `&` in `(auto &element : array)` indicates that `element` is a reference rather than a standard variable, implying that modifying `element` will modify the elements of the array.

For example, modifying the condition of the first for-loop above from `auto &element : array` to `auto element : array` would leave the original array elements unchanged and un-initialized (and consequentially printing uninitialized array elements). Below is an example of such an output (different executions may give different results since uninitialized variables have unpredictable values):

```
0 0 260849664 1 1867269568 32767 0 32767 1867269920 32767
```

In most cases involving for-loops with the new C++ syntax, the use of `auto &` or `const auto &` as the loop variable is preferred over `auto` since there is no need to copy a variable or object residing in memory to a new local variable. The use of `const auto &` is preferred if the for-loop does not modify the array's elements.

While-loops and for-loops can be nested; the example below shows how two nested for-loops are used to initialize the elements of a two-dimensional array to contain consecutive integers:

```
#include <iostream>
using namespace std;

int main() {
  int i;
  int j;
  const int N = 10;
  int a_table[N][N];  // NxN 2D array

  // initialize table to hold the values 0, 1, ..., 99
  for (i = 0; i < N; ++i) {
    for (j = 0; j < N; ++j) {
      a_table[i][j] = i * N + j;
    }
  }

  // print the table
  for (i = 0; i < N; ++i) {
    for (j = 0; j < N; ++j) {
      cout << a_table[i][j] << ' ';
    }
    cout << endl;
  }

  return 0;
}
```

Note that the inner for-loop iterates over columns (iteration variable `j`) and the outer for-loop iterates over rows (iteration variable `i`). Executing the program above produces the following output:

```
## 0 1 2 3 4 5 6 7 8 9
## 10 11 12 13 14 15 16 17 18 19
## 20 21 22 23 24 25 26 27 28 29
## 30 31 32 33 34 35 36 37 38 39
## 40 41 42 43 44 45 46 47 48 49
## 50 51 52 53 54 55 56 57 58 59
## 60 61 62 63 64 65 66 67 68 69
## 70 71 72 73 74 75 76 77 78 79
## 80 81 82 83 84 85 86 87 88 89
## 90 91 92 93 94 95 96 97 98 99
```

4.9 Functions

It is convenient to identify sequences of statements that correspond to specific tasks, and that may be reused multiple times in one or more programs. C++ offers a mechanism to organize such collections of statements, known as functions. When a function is called, the function body is executed, followed by resumed execution of the code that appears after the function call.

For example, executing the code below prints the numbers 1 2 3, with the number 2 printed by the function `print_two`:

```
# include <iostream>
using namespace std;

// function print_two() prints 2 to standard output
void print_two() {
  cout << 2 << endl;
}

int main() {
  cout << 1 << endl;
  print_two();  // execute the function print_two
  cout << 3 << endl;

  return 0;
}
```

Executing the program above displays the following output.

```
## 1
## 2
## 3
```

4.9.1 Return Value

The function may communicate its result to the calling environment in two fundamental ways: (a) using a return value, or (b) by modifying a portion of the memory that is accessible by the calling environment. We describe option (a) below as it is simpler and postpone option (b) to Sect. 4.9.5.

During the function definition, the type of the returned variable appears before the function name. A `void` return type indicates that the function does not return any value (see function `print_two` above).

For example, the function below returns the value of 2, and the code below prints the same sequence 1 2 3 as the code above.

```
# include <iostream>
using namespace std;
```

```
// function return_two returns an int variable that equals 2
int return_two() {
  return 2;
}

int main() {
  cout << 1 << endl;
  cout << return_two() << endl;  // print returned value
  cout << 3 << endl;

  return 0;
}
```

4.9.2 Function Parameters

The statements inside a function may depend on parameters that are passed to the function by the calling environment. In the function definition, the list of parameters appears inside the parenthesis, separated by commas. When the function is called, the specified values are copied to the parameters and can be later referred to inside the function.

For example, the code below has a function that computes the sum of two values that are passed to it as arguments by the calling environment:

```
#include <iostream>
using namespace std;

int add_two(int a, int b) {
  return a + b;
}

int main() {
  // print the value 3 returned by add_two(1, 2)
  cout << add_two(1, 2) << endl;

  return 0;
}
```

4.9.3 Function Definition and Function Declaration

The examples above showed function definitions: the return value, followed by the function name, followed by the list of parameters and the code implementing the function. A function declaration is the same as a function definition, except that it omits the implementation code.

The example below lists a function definition followed by a function declaration:

```
int add_two(int x, int y) { // function definition
  return x + y;
}

int add_two(int x, int y);  // function declaration
```

It is essential that the compiler recognizes functions when they are called. This can be done in two ways: (a) including a function definition in the file containing the function call, or (b) including the function declaration in the file containing the function call and the function definition in another file that is linked during the linking stage (see Sect. 4.1).

In both option (a) and option (b) above, including the function declaration or definition in the file containing the function call is often done using the `#include` preprocessor directive (see Sect. 4.6). This way, the program does not need to contain a long list of function declarations or definitions that may reduce readability, but only a few `#include` statements at the top of the file.

Option (a) is typically used in small programs containing a small number of files. Option (b) is preferred in programs containing a large number of files since it allows recompiling a smaller number of files whenever a small part of the program is modified (for example, if the function definition is modified, the code containing function calls does not need to be recompiled).

The typical way of using option (b) is demonstrated in the example below. The program contains three files: a file `factorial.cpp` that contains the function definition of the function `my_factorial`, a header file `factorial.h` containing the function declaration (`int my_factorial(int);`), and the main program file that appears below:

```
#include "factorial.h"

int main() {
  cout << my_factorial(3) << endl;

  return 0;
}
```

Note that the main program files include the header file that brings the function declaration into the main program. During the compilation stage the compiler recognizes the function but is not aware of its implementation. The linking stage occurs after the compilation of the individual files (see Sect. 4.1) and connects the compiled version of the function definition to the compiled code segments that call the function.

4.9.4 Scope of Function Variables

All variables defined inside a function are local variables in the sense that they are not recognized outside of the function. This is consistent with the rule we saw previously that variables are in scope only within the curly braces block they are defined in.

For example, the code below has two `c` variables—in `multiply_two` and in `main`. The variable `c` in `multiply_two` is not recognized from `main`, and vice versa.

```
#include <iostream>
#include <string>
using namespace std;

// return a * b
int multiply_two(int a, int b) {
  int c = a * b;

  return c;
}

int main() {
  cout << multiply_two(2, 3) << endl; // prints 6

  int b = 4;
  int c = 5;

  cout << multiply_two(b, c) << endl; // prints 20
  cout << c << endl;  // print value of local c (5)

  return 0;
}
```

4.9.5 Pointer and Reference Parameters

We saw how a function can communicate its results back to the calling environment via a return value. Another way for the function to communicate results back to the calling environment is to modify a memory location or an object that is passed to it as an argument. We focus below on modifying memory directly and postpone the issue of objects to Sect. 4.10. Since the memory location that is modified by the function and read by the calling environment needs to be known to both the function and the calling environment, it's convenient to have the calling environment pass a pointer to the function and have the function modify the content of the memory pointed to by that pointer. Below are some examples:

```
#include <iostream>
using namespace std;

// The function below accepts an address and
// an integer n and communicates back by writing
// the first n non-negative even integers in
// the specified address
void write_seq(int *a, int n) {
  for (int i = 0; i < n; ++i) {
    a[i] = i * 2;
  }
}

// The function below accepts an address and
// an integer n and negates the n integers
// that are stored in the specified address
void negate_seq(int *a, int n) {
  for (int i = 0; i < n; ++i) {
    a[i] = -a[i];
  }
}

int main() {
  int a[10] = {};

  write_seq(a,10));  // assign even numbers to the array
  negate_seq(a, 10);  // negate values of a

  for (const auto &element : a) {
    cout << element << " ";
  }
  cout << endl;

  return 0;
}
```

Compiling and executing the code above results in the following output:

```
## 0 -2 -4 -6 -8 -10 -12 -14 -16 -18
```

Note that the memory that is accessed inside the function has to be allocated beforehand. In the code above this was done by the `int a[10]` statement. The memory allocation should be done by the calling environment rather than inside the function since variables defined inside the function are local and go out of scope after the function finishes execution (however, dynamically allocated memory can be used, and the function can return a pointer to it; see Sect. 4.11 for more details about dynamic memory allocation).

Despite its popularity, the above approach has the major pitfall of possibly accessing unallocated memory when the function is called with the wrong parameters; for example, if above we call the function `write_seq(a,-10)` instead of `write_seq(a,10)`. Such bugs cause unpredictable results and may be hard to find.

A safer alternative is to pass to the function parameters that are references to variables that are defined in the scope of the calling environment (see Sect. 4.4 for more details on references). In this case, when the function assigns new values to the parameters, the variables in the calling environment that are referenced are changed as well. These changes persist after the function is executed:

```
#include <iostream>
using namespace std;

// The function modifies variables in the calling environment by
// receiving parameters of type reference, and then modifying
// the variables that are referred to.
void set(int &a, int &b, int &c) {
  a = 1;
  b = 2;
  c = 3;
}

int main() {
  int a = 0;
  int b = 0;
  int c = 0;

  // print initial values
  cout << a << ' ' << b << ' ' << c << endl;
  // modify the local a, b, c by passing references to them.
  set(a, b, c);
  // print values of a, b, c
  cout << a << ' ' << b << ' ' << c << endl;

  return 0;
}
```

Compiling and executing the code above results in the following output:

```
## 0 0 0
## 1 2 3
```

4.9.6 Recursion

So far we have seen examples of `main` calling other functions. It's possible for any function to call another; for example:

```
void bar() {
  cout << "bar" << endl;
}

void foo() {
  cout << "foo" << ' ';
```

```
  bar();
}

int main() {
  // enter foo and print "foo", then enter bar and print "bar"
  foo();

  return 0;
}
```

In fact, it's possible for a function to call itself. This construct, known as recursion, is demonstrated in the examples below. In the first example, a function calls itself, which results in a never-ending sequence of self-calls (until the program crashes as the call stack grows and exceeds its bounds):

```
// recurse() will keep calling itself, going into deeper
// and deeper recursive calls
// (until the program crashes due to call stack overflow)

void recurse() {
  recurse();
}

int main() {
  recurse();

  return 0; // this statement will never be reached
}
```

In the example below, the recursive function contains a stopping condition (`n == 1`), which—when met—stops the recursion. This allows the recursive calls to terminate and return to the calling environment (rather than continuing indefinitely until the program crashes). The code below computes the factorial of a positive integer recursively:

```
#include <iostream>
using namespace std;

int factorial(int n) {
  if (n == 1) {
    return 1;
  }

  return n * factorial(n - 1);
}

int main() {
  cout << factorial(3) << endl; // prints 6

  return 0;
}
```

The code below is equivalent to the code above, except that we add statements that print messages showing when each function is called and with what argument value, and when each function returns and what value it returns. These messages clarify when each function is called and when each function returns to its calling environment:

```
#include <iostream>
using namespace std;

int factorial(int n) {
  cout << "entering factorial(" << n << ')' << endl;

  if (n == 1) {
    cout << "leaving factorial(1) with return value 1" << endl;

    return 1;
  }

  int result = n * factorial(n - 1);
  cout << "leaving factorial(" << n <<
    ") with return value " << result << endl;

  return result;
}

int main() {
  cout << factorial(3) << endl;

  return 0;
}
```

Compiling and executing the code above gives the following output.

```
## entering factorial(3)
## entering factorial(2)
## entering factorial(1)
## leaving factorial(1) with return value 1
## leaving factorial(2) with return value 2
## leaving factorial(3) with return value 6
## 6
```

Calling `factorial(3)` calls `factorial(2)`, which in turn immediately calls `factorial(1)`. At that point, the stopping condition is met and the latter returns to its calling environment,
`factorial(2)`, with a value of 1. Then, `factorial(2)` returns to its calling environment, `factorial(3)`, with a value of 2. Then, `factorial(3)` returns to its calling environment, `main`, with a value of 6. That value is then printed by the main function.

4.9.7 Passing Arguments to Main

C++ allows passing arguments to `main` that may alter the behavior of the program. Since `main` represents the executable program and is called from the operating system, parameters to `main` can be passed through the terminal prompt as follows:

```
# running the program ./my_prog from the Linux terminal
# calls the appropriate main function with parameters a, 1, b, 2
./my_prog a 1 b 2
```

To access the parameters passed to `main`, we consider `main` as a function having the function declaration `int main(int argc, char **argv)`, where `int argc` (argument count) represents the number of arguments and `char **argv` (argument vector) represents an array of character arrays containing the arguments themselves (the first array element, `argv[0]`, represents the name by which the executable was called; for example: "./my_prog").

The program below prints the value of `argc`, and then prints the values of `argv[i]` for values of `i` ranging from 0 to `argc - 1`:

```
#include <iostream>
using namespace std;

int main(int argc, char **argv) {
  cout << "argc is: " << argc << endl;

  for (int i = 0; i < argc; ++i) {
    cout << "argv[" << i << "] is: " << argv[i] << endl;
  }

  return 0;
}
```

Below is an example of calling the appropriate program (we assume that the code above is compiled into an executable file `my_prog`) with four parameters `a`, `1`, `b`, and `2`.

```
./my_prog a 1 b 2

## argc is: 5
## argv[0] is: ./my_prog
## argv[1] is: a
## argv[2] is: 1
## argv[3] is: b
## argv[4] is: 2
```

4.9.8 Overloading Functions

C++ allows multiple functions with the same name but with different sequences of parameter types (the function name followed by the sequence of parameter types is called the function signature). When the function is called, the compiler attempts to determine which function is called based on the list of arguments in the function call.

For example, the code below contains three `print_arguments` functions with different sequences parameters. The `main` function contains three calls to `print_arguments`, each being matched to a different function definition:

```
#include <iostream>
using namespace std;

void print_argument(int a) {
  cout << "One int argument passed: " << a << endl;
}

void print_argument(double a) {
  cout << "One double argument passed: " << a << endl;
}

void print_argument() {
  cout << "No argument passed" << endl;
}

int main() {
  print_argument(); // matches print_argument()
  print_argument(1); // matches print_argument(int)
  print_argument(1.2); // matches print_argument(double)

  return 0;
}
```

Compiling and executing the program above gives the following output:

```
./print_arguments

## No argument passed
## One int argument passed: 1
## One double argument passed: 1.2
```

4.10 Object Oriented Programming

Object oriented programming (OOP) is a way to structure programs around objects, featuring three principles: encapsulation, inheritance, and polymorphism. Nontrivial programs written using the OOP paradigm tend to be simpler; they are easier to read

and modify, and as a result, they are less error-prone compared to those written using the procedural programming paradigm. Recent large scale software development focuses primarily on the OOP paradigm.

We start below by describing structs and classes and then describe the three components of OOP: encapsulation, inheritance, and polymorphism.

4.10.1 Structs

A struct (structure) is a collection of variable types annotated by names. For example, a struct named `Person` may have the named variable types: `int age`, `double height`, and `double weight`. In the same way that a variable is an instantiation of a variable type, an object is an instantiation of a struct. For example, `jane_doe` may be an instantiation of the `Person` struct described above. There may be multiple instantiations of the same struct type.

When an object corresponding to a struct is instantiated, variables corresponding to the variable types are instantiated as well. These variables are called the fields or member variables of the instantiated object.

Structs are defined using the keyword `struct`, and the named variable types composing the struct are listed inside curly braces followed by a semicolon. For example, below is the definition of a struct called `Point` containing two floating-point numbers (that correspond to the point's x and y coordinates):

```
struct Point {
  double x;
  double y;
};
```

After a struct is defined (as in the example above), we can instantiate a corresponding object by typing the struct name followed by the name of the instantiated object. The period operator is used to refer to fields belonging to a specific struct object, for example `p1.x` refers to the field `x` of the struct object `p1`.

```
Point p1; // instantiation of object p1
Point p2; // instantiation of object p2

p1.x = 3;
p1.y = 0;
```

There are several ways in which functions can accept objects as arguments:

Call by Value: The object from the calling environment is copied into a new object that is local to the function. Any modification inside the function to the object will affect only the local copy and not the object that resides in the calling environment.

Calling by value is the default setting, and it occurs when the function signature contains an object rather than a reference or a pointer to one. For example, the following function receives a parameter by value:

```
void by_value(Point p);
```

Call by Reference: The function gains access to a reference to an object defined in the calling environment. Changes the function makes to the object persist after the function is executed. Since there is no need to create a local object and copy the object to the local object, calling by reference is more space- and time-efficient than calling by value.
Calling by reference occurs when the function signature contains the argument name preceded by the `&` symbol, or when the argument is an array. For example, each of the following functions receives a parameter by reference:

```
void by_reference(Point &p);
void by_reference(Point points[]);
```

Call by Reference to a Constant: The function gains access to a reference to an object defined in the calling environment, but it can't modify said object. Since there is no need to create a local object and copy the object to the local object, calling by reference is more space- and time-efficient than calling by value; in addition, it provides a read-only access of the parameter to the function. Reducing mutability (changes to objects) can help prevent bugs and make it easier for code readers to understand the code and maintain it.
Calling by reference to a constant occurs when the function signature contains the argument name preceded by the `&` symbol with a `const` keyword modifying the struct name:

```
void by_reference_to_const(const Point &p);
```

Call by Address: The function gains access to a pointer to an object[5]. Any changes made to the pointed object persist after the function is executed. Since there is no need to create a local object and copy the object to the local object, calling by reference is more space- and time-efficient than calling by value. This type of call is also known as a call by pointer, and it allows for the called function to create the object (as an out parameter) dynamically (see Sect. 4.11 for more details about dynamic memory allocation); we recommend against the use of out parameters because they are less readable (see Chap. 15 for more style guidelines), but it's important to recognize the pattern in others' code.
Calling by address occurs when the function signature contains the argument name preceded by the `*` symbol. For example, the following function receives a parameter by address:

```
void by_address(Point *p);
```

[5]When calling by address, the parameter itself is not copied, but its memory address (pointer) gets copied and passed (by value) to the called function.

Call by Address of a Constant: The function gains access to a pointer to an object[5], but it can't modify the object. Since there is no need to create a local object and copy the object to the local object, calling by address of a constant is more space- and time-efficient than calling by value; in addition, it provides a read-only access of the parameter to the function. Reducing mutability (changes to objects) can help prevent bugs and make it easier for code readers to understand the code and maintain it.
Calling by address of a constant occurs when the function signature contains the argument name preceded by the * symbol, and the argument specification is modified using the `const` keyword either before or after its type:

```
void by_address_of_const(const Point *p); // more readable
void by_address_of_const(Point const *p);
```

When calling by address or by address of a constant, the address itself can be specified as a constant by adding the `const` between the * symbol and the argument name, as in the following examples:

```
void by_const_address(Point* const p);
void by_const_address_of_const(const Point* const p);
```

This means that the copy of the address that the function receives cannot be modified. A nonconstant (mutable) copied address can be reassigned inside the function. Since the address of the object (and not the object itself) is passed by value, any modification inside the function to the address will affect only the local copy of the address and not the address that resides in the calling environment; to illustrate:

```
#include <iostream>
#include <string>
using namespace std;

struct Person {
    string name;
    double age;
};

void by_address_of_const(const Person *p) {
    Person x;

    p = &x;
    cout << "modified copy inside function: " << p << endl;
}

int main() {
  Person person;
  Person *pointer = &person;

  cout << "before function call: " << pointer << endl;
  by_address_of_const(pointer);
```

```
  cout << "after function call: " << pointer << endl;

  return 0;
}
```

A sample run of the program above gives the following output (since the output includes memory locations, expect a different output when you run it):

```
## before function call: 0x7fff517f8ca8
## modified copy inside function: 0x7fff517f8c60
## after function call: 0x7fff517f8ca8
```

Since call by value is wasteful and potentially slow (it creates a local copy of the object), it should be avoided except in cases involving small objects. Instead, it's recommended to use call by reference (if the function is supposed to mutate the object), or call by const reference (if the function is not supposed to mutate the object). The following example demonstrates passing objects by references:

```
#include <iostream>
using namespace std;

struct Point {
  double x;
  double y;
};

// pass by reference - the function is modifying p
void scale_point(Point &p, double scaling_factor) {
  p.x = p.x * scaling_factor;
  p.y = p.y * scaling_factor;
}

// pass by const reference - the function is not modifying p
void print_point(const string &prefix, const Point &p) {
  cout << prefix << '(' << p.x << ',' << p.y << ')' << endl;
}

// pass by const reference; the function doesn't modify p1 or p2
Point subtract_points(const Point &p1, const Point &p2) {
  Point delta;

  delta.x = p1.x - p2.x;
  delta.y = p1.y - p2.y;

  return delta;
}

int main() {
  Point p1;
  Point p2;

  p1.x = 1;
```

```
  p1.y = 2;
  p2.x = 3;
  p2.y = 3;

  print_point("first point: ", p1);
  print_point("second point: ", p2);
  print_point("delta: ", subtract_points(p1, p2));

  // scale second point by a factor of 2
  cout << endl << "scaling the 2nd point to 2x" << endl;
  scale_point(p2, 2);

  print_point("first point: ", p1);
  print_point("second point: ", p2);
  print_point("delta:", subtract_points(p1, p2));

  return 0;
}
```

The program above prints the following output.

```
## first point: (1,2)
## second point: (3,3)
## delta: (-2,-1)
##
## scaling the 2nd point to 2x
## first point: (1,2)
## second point: (6,6)
## delta:(-5,-4)
```

Structs can be nested, for example, the following code defines a new struct `Vector` that contains two structs of the type `Point`. The `length` function computes the length or Euclidean norm of the function.

```
#include <iostream>
#include <cmath>
using namespace std;

struct Point {
  double x;
  double y;
};

struct Vector {
  Point start;
  Point end;
};
//

pass by const reference; the function doesn't modify p1 or p2
Point subtract_points(const Point &p1, const Point &p2) {
  Point delta;
  delta.x = p1.x - p2.x;
  delta.y = p1.y - p2.y;
```

```
  return delta;
}

double length(const Vector &v) {
  Point diff = subtract_points(v.start, v.end);

  return sqrt(diff.x * diff.x + diff.y * diff.y);
}

int main() {
  Vector v;

  v.start.x = 1;
  v.start.y = 0;
  v.end.x = 2;
  v.end.y = 1;

  cout << "length of the vector is: " << length(v) << endl;
}
```

Compiling and running the code above gives the following output (the length of the vector is $\sqrt{1 * 1 + 1 * 1} = \sqrt{2} \approx 1.41421$).

```
## length of the vector is: 1.41421
```

The `->` operator refers to member variables or fields within a struct pointed to by a pointer. For example, if `pp` is a pointer to a `Point` object, `pp->x` is equivalent to `(*pp).x`.

```
int main() {
  // create an array of two points
  Point points[2];

  // initialize the array to (0,1) and (2,3)
  points[0].x = 0;
  points[0].y = 1;
  points[1].x = 2;
  points[1].y = 3;

  // points is a pointer that refers to the start of the array
  // print points[0] using -> syntax
  cout << points->x << ' ' << points->y << endl;

  // print points[1] using . syntax
  cout << points[1].x << ' ' << points[1].y << endl;
}
```

Compiling and executing the above program gives the following output:

```
## 0 1
## 2 3
```

4.10.2 Classes

Classes are similar to structs; both of them may include functions, which are called member functions or methods, but it's more customary for classes to include methods that define various behaviors of an object, while structs represent data transfer objects (DTO), which—as the name suggests—can be very useful when transferring data around. Methods play an integral part in the OOP paradigm as we'll see later in this chapter. To invoke a method after an object corresponding to a specific class is instantiated, the period operator is used with objects and references to objects:
`object_name.method_name`, while the `->` operator is used if we have a pointer to an object: `pointer_name->method_name`. Inside a method, the fields can be accessed by referring to their names or by using the `this->field_name` construct (inside a method, `this` is a pointer that refers to the object that is in context).

Fields and methods that follow a `public:` access specifier are accessible to code that is part of the class definition as well as code that is not part of the class definition. Fields and methods that follow a `private:` access specifier are only accessible to code that is part of the class definition.[6] The key difference between classes and structs in C++ is the default access level for their members when an access specifier is not explicitly specified: members of classes are private by default; members of structs are public by default.

To minimize mutability, we can specify that a specific method does not modify the class fields by including the `const` keyword in the function declaration after the function argument list. Doing so enables calling the method of an object that is defined as `const`.

Methods can also be implemented outside of the class, as long as their declaration appears in the class and the function definition indicates the class membership of the method using `::` (the scope resolution operator). For example, the code below implements a class representing a point in a two-dimensional space. The method `reflect` is implemented outside the class:

```
#include <iostream>
using namespace std;
```

[6] A nonmember function or code in another class can access members of a `friend` class as if they were public. See Sect. 4.10.2 for more details on friendship in C++.

```
class Point {
private:
  // fields are accessible only to methods belonging to point
  double x;
  double y;

public:
  void set_x(double a_x) {
    this->x = a_x; // set the member x to the value a_x
  }

  void set_y(double a_y) {
   this->y = a_y;
  }

  // get value of x, const implies that the method does not
  // modify the object
  double get_x() const {
    return x;
  }

  // get value of y, const implies that the method does not
  // modify the object
  double get_y() const {
    return y;
  }

  // negate x and y (method implemented outside class body)
  void reflect();
};

void Point::reflect() {
  x = -x;  // alternatively, this->x = - this->x
  y = -y;  // alternatively, this->y = - this->y
}

int main() {
  Point p;

  p.set_x(1);
  p.set_y(2);
  cout << '(' << p.get_x() << ',' << p.get_y() << ')' << endl;

  p.reflect();
  cout << '(' << p.get_x() << ',' << p.get_y() << ')' << endl;

  return 0;
}
```

Compiling and executing the code above gives the following output:

```
## (1,2)
## (-1,-2)
```

Once we have a `Point` object in `main`, we can access its methods `set_x`, `set_y`, `get_x`, `get_y`, and `reflect` since they are defined as public members. Accessing the fields or methods in `main` that are marked as private yields a compilation error (for example, including the statement `p.x = 3` in `main()`.)

The Constructor and the Destructor

When a new object is instantiated (for example, the statement `Point p;` in the code above), C++ creates a new object whose fields are initialized to default values. The constructor method offers a way to initialize new objects in a different way, specified by the programmer. One benefit of using a constructor method is that it can accept arguments that are used to initialize the fields. The constructor method has the same name as the class and it does not have any return type.

Using the function overloading concept, we can define multiple constructors, each accepting a different argument list. The appropriate constructor is executed by matching the argument list in the method call with the various overloaded constructor methods. The implicit constructor that initializes all fields to default values is only called if there are no explicit definitions of a constructor function.

For example, we can define the following two constructors for the `Point` class above:

```
class Point {
  ...
public:
  // empty constructor, called by the statement: Point p;
  Point() {}

  // constructor accepting two arguments, called by a
  // statement such as: Point p(1, 2);
  Point (double a_x, double a_y) {
    this->x = a_x;
    this->y = a_y;
  }
  ...
};
```

Compiling and executing the `main` function below, we get the following output:

```
int main() {
  Point p(2,-2);  // constructor with two argument is called

  cout << '(' << p.get_x() << ',' << p.get_y() << ')' << endl;

  return 0;
}


## (2,-2)
```

When defining a constructor, we can also initialize the fields using a colon operator after the argument list. For example, the constructor of the `Point` class above can also be implemented as follows:

```
class Point {
  ...
  // constructor accepting two arguments, called by a
  // statement such as: Point p(1, 2);
  Point(double a_x, double a_y) : x(a_x), y(a_y) {}
  ...
};
```

The destructor method is called whenever an object is destroyed; for example, when we exit a curly braces block in which an object is defined and the object goes out of scope. The name of the destructor method is the ~ symbol followed by the class name. Including a destructor method is optional, but it's particularly important when the object has dynamically allocated memory (see Sect. 4.11). For example, an empty destructor for the `Point` class is explicitly defined below:

```
class Point {
  ...
  ~Point() {} // empty destructor
  ...
};
```

The Copy Constructor

The copy constructor method allows the programmer to define a new object and initialize it using the fields of another object of the same type. It's similar to the constructors we covered so far, but it accepts as an argument a reference to a constant view of the object to copy.

For example, the code below defines a copy constructor for the `Point` class and a main function that defines a new object `p2` based on an existing object `p1` using the copy constructor:

```
class Point {
  ...
  // copy constructor
  Point(const Point& p) {
    x = p.x;
    y = p.y;
  }
  ...
};

int main() {
  Point p1(1, 2); // new object using a constructor
  Point p2(p1);  // new object using the copy constructor
```

```
  return 0;
}
```

If a copy constructor is not explicitly defined, the compiler creates a default copy constructor where the fields are copied from the existing object to the new object. The copy constructor is called when a function parameter is passed by value—to copy said value and pass it to the function being called; it also gets called when the assignment operator is used to copy a value as in `p2 = p1;` which has the same effect as initializing the object using the copy instructor in the example above.

The Converting Constructor

The converting constructor method allows the programmer to define a new object and initialize it using a value of another object of another type. It accepts as an argument the object to convert. For example, the code below defines a converting constructor that converts an integer (that represents the atomic number of an element in the periodic table) to an object of the class `Element`:

```
#include <iostream>
using namespace std;

class Element {
private:
  int atomic_number;

public:
  Element(int an_atomic_number) :
    atomic_number(an_atomic_number) {}

  int get_atomic_number() const {
    return atomic_number;
  }
};

int main() {
  Element na(11);
  Element mg = 12; // alternative form

  cout << mg.get_atomic_number() << endl; // prints 12

  return 0;
}
```

The implicit converting constructor is also called when a function parameter is passed by value—to convert said value and pass it to the function being called as a parameter of the expected type, here's an example:

```
// calling this function with an int parameter causes
// the converting constructor to be called
void print_element(Element x) {
  cout << x.get_atomic_number() << endl;
}

int main() {
  print_element(12); // prints 12

  return 0;
}
```

The implicit conversions in the above examples are not easy to read at first glance; that's why we recommend against using such patterns. To disallow implicit conversion, the `explicit` keyword is used to specify a constructor as an explicit converting constructor:

```
#include <iostream>
using namespace std;

class Element {
private:
  int atomic_number;

public:
  // note the use of the explicit keyword below
  explicit Element(int an_atomic_number) :
    atomic_number(an_atomic_number) {}

  int get_atomic_number() const {
    return atomic_number;
  }
};

int main() {
  Element na(11);
  Element mg = 12; // error

  cout << na.get_atomic_number() << endl; // prints 11

  return 0;
}
```

Operator Overloading

C++ offers a way to define the way operators such as `=`, `+`, and `-` apply to objects. The operator is implemented as a function, whose definition depends on whether the operator is binary or unary and what object type is returned by the operator.

We demonstrate operator overloading by implementing two operators for the `Point` class: the binary addition + operator and the unary stream insertion operator «.

The operator + is a method that accepts one object by reference to a constant. The other object is the object that is in context when the function is called (represented by the `this` pointer). For example, the expression `x + y` reduces to the object `x` calling the operator function + with an argument that is a reference to a constant view of `y`. The operator returns a new object representing the result of the addition operator:

```
class Point {
  ...
  // overload addition operator (+)
  Point operator+(const Point &p) const {
    Point result;

    result.x = this->x + p.x;
    result.y = this->y + p.y;

    return result;
  }
  ...
};
```

Above, the `const` statement after the argument list indicates that the operator does not modify the object that calls the operator function (the left-hand side argument of the addition).

The case of the stream insertion operator « is different in that the operator overloading function is not a method of the `Point` class. As a result, the operator code cannot typically access private fields and must use the `get_x` and `get_y` public methods instead (the case of a friend function is an exception that is described in Sect. 4.10.2).

```
ostream& operator<<(ostream& os, const Point &p) {
  os << '(' << p.get_x() << ',' << p.get_y() << ')';

  return os; // to support operator chaining
}

int main() {
  Point p1(1,2);
  Point p2(2,3);
  Point p3 = p1 + p2;  // calls the overloaded + operator

  // calls the overloaded << operator thrice for points
  // and thrice for spaces and end-of-line delimiters
  cout << p1 << ' ' << p2 << ' ' << p3 << endl;

  return 0;
}
```

The code above gives the following output:

```
## (1,2) (2,3) (3,5)
```

The assignment operator `=` is special: if it's not defined explicitly by the programmer, a default assignment operator is created by the compiler that copies the fields from the source object to the destination object.

Friend Functions and Classes

When a class declares nonmember functions or other classes as friends, it allows them access to all of its members as if they were public. In C++, friendship is all about sharing, and it's not symmetric: if `class A` declares `B` as a friend, `B` can access all of `A`'s members, but not the other way around (unless `B` declares `A` as a friend as well).

For example, the output stream operator overloading function above can be declared as a friend of the `Point` class; in which case, it can access the private fields directly rather than calling the `get_x` and `get_y` methods:

```
class Point {
  ...
  // mark the non-member function overloading << as a friend
  friend ostream& operator<<(ostream &os, const Point &p);
  ...
};

ostream& operator<<(ostream &os, const Point &p) {
  // access private fields since << is a friend of Point
  os << '(' << p.x << ',' << p.y << ')';

  return os;
}
```

Classes have more properties than described above. We continue to explore their properties below as we discuss the three main aspects of object oriented programming: encapsulation, inheritance, and polymorphism.

4.10.3 Encapsulation

The principle of encapsulation states that when there are multiple software components interacting with each other, the code (including variables, functions, and objects) of one software component should not be generally accessible to the code of another software component. The interaction between software components should be limited to a small set of interface methods. A few benefits of encapsulation appear below:

- If the interface is kept unchanged, the implementation of one software component may change without affecting the implementations of the other software components.
- One software component cannot inadvertently corrupt variables that another component relies on (the interface methods can implement a mechanism that ensures fields are set to appropriate values and only in appropriate cases).
- Debugging is easier since bugs can often be localized to specific software components.
- Software development can be conveniently parallelized across multiple development teams, with the different teams agreeing on the interface and then each team working independently on the implementation of their software components.

To illustrate how the encapsulation concept applies to classes, consider a situation where we want to modify the type of the fields `x` and `y` of the `Point` class from `double` to `float`. One reason to do so is to save memory to support very long arrays of `Point` objects. Since `x` and `y` are private fields and are accessible by external code only through public methods, we can modify the variable types without affecting code that uses `Point` objects as long as we keep the interface represented by the public methods unchanged.

4.10.4 Inheritance

Inheritance enables the sharing of attributes and behaviors between two related classes (also known as the is-a relationship; for example, a circle *is a* shape). This applies in situations where there is a simple class, called the base class, and a more complex version of it, called the derived class. The base class may be implemented in full while the derived class only implements the added functionality it offers beyond the base class. In some cases, the terms superclass and parent class are used instead of base class and the terms subclass or child class are used instead of derived class—respectively.

Members of the base class are included in the derived class; that's why the is-a relationship is a subsuming relationship; in some cases, the base class methods are overloaded in the derived class in order to add functionality related to the added complexity of the derived class; in other cases, base class methods are declared but not defined—leaving the implementation details to derived classes (we'll discuss this in more detail in Sect. 4.10.5).

We focus in this chapter on public inheritance in C++, which states that public members of the base class become public members of the derived class and private members of the base class are inaccessible in the derived class. Public inheritance is denoted by following the derived class name with `: public BaseClassName`.

Another property of public inheritance is allowing derived classes to access protected members of the base class (also, protected members of the base class become protected members of the derived class). The keyword `protected` is used

to denote a protected member, which limits access to said member to: the class that declared it, its friends, and classes that inherit it in a fashion that allows such access (e.g., public inheritance).

We demonstrate inheritance by considering the `Point` class as a base class and defining a derived class called `NamedPoint`, which extends `Point` by adding a string field corresponding to the name of the point:

```
class NamedPoint : public Point {
private:
  string name;
public:
  // empty constructor calls point constructor
  NamedPoint() : Point() {}

  // constructor calls point constructor then initializes name
  NamedPoint(double x, double y, string name) :
    Point(x, y), name(name) {}

  string get_name() {
    return name;
  }

  void set_name(string a_name) {
    this->name = a_name;
  }
};

int main() {
  NamedPoint p(0, 0, "origin");

  p.set_name("FIRST POINT");
  p.set_x(2);  // calls a base class method
  cout << p.get_name() << ": (" << p.get_x() << ',' <<
    p.get_y() << ')' << endl;

  return 0;
}
```

The code above gives the following output:

```
## FIRST POINT: (2,0)
```

We can refer explicitly to a method associated with a specific class using `::` (the scope resolution operator). For example, if `A` is the base class and `B` is a derived class, and both classes implement the method `foo()`, `A::foo()` refers to the `foo()` implementation in class `A` and `B::foo()` refers to the `foo()` implementation in class `B`.

4.10.5 Polymorphism

Pointers and references to objects of a type A can refer to objects of type B, if B is a derived class of A. This property allows objects to take many forms, a property known as polymorphism.

We illustrate the usefulness of polymorphism with the example below. Suppose we have a base class Point and a derived class NamedPoint, as in the example above. If we want to store and process a large collection of points, one solution is to create an array of Point objects and an array of NamedPoint objects. But a more convenient solution is to create a single array of type Point references or pointers and have some elements in the array refer to Point objects and others refer to NamedPoint objects.

Continuing the example above, we assume that both Point and NamedPoint implement the method print() differently (printing a NamedPoint requires printing the name of the point, something which is inapplicable to Point objects). The following question arises: if we traverse the array of pointers and call print for each element, which method will be called: Point::print() or NamedPoint::print()? There are two possibilities: (a) Point::print() will be called each time since the array holds Point pointers, or (b) either Point::print() or NamedPoint::print() will be called, depending on which object is pointed to. Case (a) is called static binding and case (b) is called dynamic binding. The static label refers to the fact that the function that is called is determined at compile-time. The dynamic label refers to the fact that the function that is called is determined at runtime.

First, let's examine this code snippet:

```
class Point {
  ...
  void print() {  // Point::print method defined in base class
    cout << '(' << x << ',' << y << ')' << endl;
  }
  ...
};

class NamedPoint : public Point {
  ...
  // NamedPoint::print method defined in derived class
  void print() {
    cout << name << ' ';
    Point::print();
  }
  ...
};

int main() {
  Point p(2,3);
  NamedPoint np(1, 2, "first point");
```

```
  p.print();  // Point::print is called
  np.print();  // NamedPoint::print is called

  return 0;
}
```

The code above prints the following output:

```
## (2,3)
## first point (1,2)
```

The following code demonstrates static binding, which is the default in C++. Note that even though `pp` points to an object of class `NamedPoint`, the method calls `refP.print()` and `pp->print()` actually call `Point::print`:

```
int main() {
  NamedPoint p(0, 0, "origin");
  Point &refP = p;
  Point *pp = &p;

  refP.print(); // static binding calls Point::print()
  pp->print(); // ditto

  return 0;
}
```

The code above gives the following output:

```
## (0,0)
## (0,0)
```

Virtual Functions

In the case of dynamic binding, the method corresponding to the pointed object is chosen at runtime and executed. To enable dynamic binding, the corresponding methods need to be defined as virtual.

Below, we modify the code so that dynamic binding is used. The only modification needed is to add the keyword `virtual` before the definitions of the `print` methods.[7]

```
class Point {
  ...
  // Point::print method defined in base class
  // note the use of the virtual keyword below
  virtual void print() {
```

[7] It's sufficient to declare a method in the base class as virtual for it to be virtual in all derived classes, but it's good to repeat it explicitly to enhance readability.

```
    cout << '(' << x << ',' << y << ')' << endl;
  }
  ...
};

class NamedPoint : public Point {
  ...
  // NamedPoint::print method defined in derived class
  // note the use of the virtual keyword below
  virtual void print() {
    cout << name << ' ';
    Point::print();
  }
  ...
};

int main() {
  NamedPoint p(0, 0, "origin");
  Point &refP = p;
  Point *pp = &p;

  refP.print(); // dynamic binding calls NamedPoint::print()
  pp->print(); // ditto

  return 0;
}
```

The code above gives the following output:

```
## origin (0,0)
## origin (0,0)
```

Pure Virtual Functions and Abstract Classes

Pure virtual methods are virtual methods, defined in a base class, that have no implementation in the base class. They exist simply as a placeholder for defining overloaded methods in derived classes. To define a method as pure virtual simply add a `= 0` following the argument list in the method declaration.

For example, the method `get_area` below in the class `Shape` is a pure virtual method:

```
class Shape {
  ...
  virtual double get_area() = 0;  // pure virtual method
  ...
};
```

Such methods are useful when the implementation only makes sense in derived classes, and since no implementation is specified for a pure virtual method, it's

impossible to instantiate objects of the corresponding class; for this reason, classes with one or more pure virtual functions are called abstract classes.

4.10.6 Static Variables and Functions

Static members (also known as class members) are shared across all instantiated objects of the same class. When one object modifies that static variable, the change is reflected for all current and future objects of the same class. Static members are denoted by the `static` keyword.

Static methods need an object to access non-static fields of said object. Inside the context of a static method, there's no `this` pointer because a static method is a class member, and not an object member. Calling a static method or referring to a static field doesn't require an object; using the syntax `ClassName::x` allows reference to static members even if no object of the corresponding class exists. It's possible to refer to a static member from any instantiated object of the same class, but it's more readable to use the class name instead (to indicate that it's a class member and not an object member).

The example below adds a static field num_objects to the `Point` class. The field is initialized to 0 and is incremented whenever the default constructor is called. Thus, `num_objects` keeps the number of points that were constructed using the default constructor:

```
#include <iostream>
using namespace std;

class Point {
private:
  double x;
  double y;
  static int num_objects;

public:
  Point() {
    // increments counter when a new object is created
    ++num_objects;
  }

  // even though this method does not modify any members,
  // static member functions cannot be qualified as const
  static int get_num_points() {
    return num_objects;
  }
};

// defines the static field num_objects and initializes it to 0
int Point::num_objects = 0;
```

```
int main() {
  Point p1;
  Point p2;
  Point p3;

  cout << "number of point objects: "
       << Point::get_num_points() << endl;

  return 0;
}
```

Compiling and executing the code above gives the following output.

```
## number of point objects: 3
```

Static variables can also be defined inside a function. This usage implies that the variables are kept in memory even if they go out of scope. For example, the following code records the number of times a function has been called:

```
#include<iostream>
using namespace std;

int count_calls() {
  // initialization occurs only once when program is loaded
  static int counter = 0;

  return ++counter; // pre-increments then returns
}

int main() {
  count_calls();
  count_calls();
  cout << "number of calls: "<< count_calls() << endl;

  return 0;
}
```

Compiling and executing the code above gives the following output:

```
## number of calls: 3
```

4.11 Dynamic Memory and Smart Pointers

4.11.1 Dynamic Memory Allocation

Defining a variable (or an array of variables) reserves a portion of memory whose size is known at compilation time. This mechanism, which includes all of the examples we have seen thus far, is known as static memory allocation.

Dynamic memory allocation is a mechanism for allocating memory for an object or an array whose size is not known in compilation time. Memory that is dynamically allocated exists until it is explicitly freed by the programmer.

The C++ `new` operator dynamically allocates memory and the `delete` operator frees the memory. If memory is allocated for an object, the `new` operator calls the appropriate constructor function and the `delete` operator calls the appropriate destructor function. C++ also supports the legacy C functions `calloc` and `malloc` for allocating memory and `free` for freeing memory. We do not describe below these functions as `new` and `delete` supersede `calloc`, `malloc`, and `free`.

In our first example below, we allocate memory dynamically for a double variable, and define a pointer that points to that memory. We later free that memory with the `delete` statement:

```
int main() {
  // pd points to a dynamically allocated double variable
  double *pd = new double;

  // free the allocated memory
  delete pd;

  return 0;
}
```

In the example below we allocate memory for an object of class `Point`:

```
int main() {
  Point *pp1 = new Point; // the default constructor
  Point *pp2 = new Point(1, 2); // an overloaded constructor

  delete pp1;
  delete pp2;

  return 0;
}
```

In the example below we allocate memory for an array of objects:

```
int main() {
  int n = 2;
  // allocates memory for an array of n objects of type point
  Point *pp = new Point[n];

  delete[] pp; // frees up a dynamically-allocated array

  return 0;
}
```

Note that the size of the allocated array is marked by a non-const variable, and thus may not be known at compilation time. For this reason, a static allocation in this case would fail. Also note the use of the `delete[]` operator, which is used to free dynamically allocated arrays.

4.11.2 Smart Pointers

Dynamic memory allocation requires the programmer to free memory when it's no longer used, and to keep memory that is still needed later on in the program. The 2011 C++ standard includes a mechanism known as smart pointers that relieves the programmer from some of these responsibilities.

Smart pointers are similar to standard pointers except that they automatically free the appropriate memory when it is no longer used. There are two types of smart pointers in C++: `shared_ptr`, which allows multiple pointers to point to the same memory, and `unique_ptr`, which allows only a single pointer to point to the allocated memory. To use smart pointers, the header file `memory` should be included in the program.

The smart pointers `shared_ptr` and `unique_ptr` are actually templates. We therefore qualify the pointer type with angle brackets containing the type of variables that they point to. See Sect. 4.12 for more information on templates. Here are a few examples:

```
#include<memory>
using namespace std;

int main() {
  // creates smart pointers to double variables,
  // each holds a value of 4.2
  shared_ptr<double> pd1 = make_shared<double>(4.2);
  shared_ptr<double> pd2(new double(4.2)); // alternatively

  // pd2 is reassigned to a new memory location,
  // older memory content is no longer pointed to
  // and is therefore freed up automatically
  pd2 = make_shared<double>(8.4);

  // several shared_ptr objects may point to the same memory
  shared_ptr<double> pd3 = pd2;

  return 0;
}
```

The smart pointer `unique_ptr` differs from `shared_ptr` in that `unique_ptr` "owns" the memory it points to, so that no other smart pointer can refer to that memory. One exception to that rule is that a new `unique_ptr` smart pointer may be assigned to the memory pointed to by a `unique_ptr` that is being destroyed:

```
#include <memory>
using namespace std;

unique_ptr<int> create_unique_pointer() {
  return unique_ptr<int>(new int(4));
}
```

```
int main() {
  unique_ptr<int> pi1(new int(3));
  // error: two unique_ptr pointers pointing to the same memory
  unique_ptr<int> pi2 = pi1;
  // ok: new unique_ptr takes over as old unique_ptr is destroyed
  unique_ptr<int> pi3 = create_unique_pointer();

  return 0;
}
```

The smart pointer `unique_ptr` can also point to arrays of objects; `shared_ptr` cannot do that (by default). In that case the variable type in the angle brackets needs to be followed by square brackets to indicate that the smart pointer points to an array:

```
#include <iostream>
#include <memory>
using namespace std;

int main() {
  int n = 10;
  // smart pointer to an array of int variables
  unique_ptr<int[]> array(new int[n]);

  for (int i=0; i<n; ++i) {
    array[i] = i;
  }

  for (int i=0; i<n; ++i) {
    cout << array[i] << ' ';
  }
  cout << endl;

  return 0;
}
```

Compiling and executing the code above results in the following output:

```
## 0 1 2 3 4 5 6 7 8 9
```

4.12 Templates

In some cases, a sequence of operations is repeated on variables of different types. For example, a function that sums up the elements of an array needs to be implemented separately for each variable type: `int`, `long`, `double`, etc. Templates allow writing code that applies generically without specifying particular variable or object types. The code that calls the template function or that uses the template class specifies the variable or object type that the generic code will run.

We start below by describing template functions and then proceed to describe template classes.

4.12.1 Template Functions

Template functions implement generic code which expects a type parameter. When a template function is defined, it's preceded by `template <class T>`, where `T` is a placeholder for a generic type that will be specified by the calling environment. The name of the type parameter `T` can be replaced with any other name, but it's conventional to use the uppercase `T` for a type parameter. When the function is called, the placeholder type is converted into a specific variable or object type by appending the function name with the appropriate type inside angle brackets. Template functions can accept more than a single placeholder, in which case the calling environment needs to specify several respective types.

For example, the template function below returns the minimum value among two arguments of the same generic (unspecified) type. It's required, however, that whenever the type is specified, it has an implementation of the comparison operator < (this is the case—by default—for all numeric variable types such as `int`, `double`, etc.):

```
#include <iostream>
using namespace std;

class Element {
private:
  int atomic_number;

public:
  Element(int atomic_number) :
    atomic_number(atomic_number) {}

  int get_atomic_number() const {
    return atomic_number;
  }

  bool operator<(const Element &x) const {
    return this->atomic_number < x.atomic_number;
  }
};

// note the use of the template keyword below:
template <class T>
// note the use of const T& to avoid copying the return value
const T& get_min(const T &a, const T &b) {
  return a < b ? a : b;
}

int main() {
```

```
  // calls the template function, replacing T with int
  cout << get_min<int>(1, 2) << endl;
  // calls the template function, replacing T with double
  cout << get_min<double>(9.1, 2.5) << endl;
  // calls the template function with an inferred type (int)
  cout << get_min(13, 42) << endl;

  Element na = 11;
  Element mg = 12;

  // calls the template function, replacing T with Element
  // note that the replacement type, Element, is inferred
  cout << get_min(na, mg).get_atomic_number() << endl;

  return 0;
}
```

Compiling and executing the program above results in the following output:

```
## 1
## 2.5
## 13
## 11
```

Note that the template function above returns a variable or object of type `T` (or more specifically, `const T&`), the generic type that characterizes its two arguments. Without the mechanism of template functions, the function above would have to be implemented separately for each desired type.

In case where two (or more) generic types are needed, the template function defines how many generic types it accepts; for example, `template <class T, class S>` (as before `T` and `S` can be replaced with other identifiers); additional generic types can be added as needed.

The following example shows a template function that returns `true` if the value of the first parameter `a` precedes that of the second parameter `b` and returns `false` otherwise; the two parameters are allowed to be of different types. It is required that the comparison operator < is supported for the two types; more specifically, an `operator<` function, that accepts an object of type T as the left-hand side operand and an object of type S as the right-hand side operand, must be defined:

```
#include <iostream>
using namespace std;

class Element {
private:
  int atomic_number;

public:
  Element(int atomic_number) :
    atomic_number(atomic_number) {}

  int get_atomic_number() const {
```

```
    return atomic_number;
  }

  bool operator<(const Element &x) const {
    return this->atomic_number < x.atomic_number;
  }
};

template <class T, class S>
bool precedes(const T &a, const S &b) {
  return a < b;
}

int main() {
  // instructs cout to print true and false instead of 1 and 0
  cout.setf(ios::boolalpha);

  cout << precedes<int, double>(1, 42) << endl;
  cout << precedes(9.1, 3) << endl;
  cout << precedes(1.5f, 1.3) << endl;

  Element na = 11;
  Element mg = 12;

  cout << precedes(na, mg) << endl;

  return 0;
}
```

The code above results in the following output:

```
## true
## false
## false
## true
```

4.12.2 Template Classes

Similar to template functions, template classes define classes that support code operating on variables or objects of generic types.

For example, the code below implements a template class `Point` that is similar to the class `Point` defined previously in this chapter, except that the variable types representing the coordinates of the point are left unspecified. When we instantiate an object of the template class, we need to specify the type that will be used instead of the placeholder:

```
#include <iostream>
using namespace std;
```

```
template<class T>
class Point {
private:
  T x;
  T y;

public:
  void set_x(const T &x) {
    this->x = x;
  }

  void set_y(const T &y) {
    this->y = y;
  }

  T get_x() const {
    return x;
  }

  T get_y() const {
    return y;
  }

  void reflect() {
    x = -x;
    y = -y;
  }
};

int main() {
  Point<int> p;

  p.set_x(1);
  p.set_y(2);
  cout << '(' << p.get_x() << ',' << p.get_y() << ')' << endl;

  p.reflect();
  cout << '(' << p.get_x() << ',' << p.get_y() << ')' << endl;

  return 0;
}
```

Compiling and executing the program above results in the following output:

```
## (1,2)
## (-1,-2)
```

As is the case with template functions, we can have several different unspecified types or placeholders separated by commas in the template class definition.

4.13 The Standard Template Library

The standard template library (STL) is a collection of templates that facilitates storing, retrieving, and manipulating data. We focus below on several STL container classes that are very useful in data analysis. Section 4.14 lists several references where more information on STL is available.

A container is a class that stores multiple variables or objects and facilitates the following three basic operations: insertion, removal, and access.

There are three general types of containers in STL:

Sequence Containers: Each element in a sequence container has a specific position in the collection. That position typically depends on the insertion order and is independent of the value of that element

Associative Containers: Each element in an associative container has a specific position in the collection. That position depends on the value of the element and is independent of the insertion order.

Unordered Containers: Each element in an unordered container has a specific position. That position is immaterial as it's neither dependent on the value of the element nor the insertion order; the position may even change without advance warning.

The precise implementation of these three container types is not specified by the C++ language, and may depend on the specific compiler being used. However, sequence containers are usually implemented using arrays or linked lists. Associative containers are usually implemented using binary trees. Unordered containers are usually implemented using hash tables.

4.13.1 Sequence Containers

There are four types of sequence containers in STL: vectors, deques, arrays, and lists.

A vector is similar to a C++ array except that it provides functionality for modifying its size. Vectors provide random access to their elements, implying that the k-element of a vector can be accessed immediately, without going over the entire vector. Appending or removing elements at the end of the array is usually fast as the typical implementation allocates more memory for an initialized vector than required[8]. Inserting an element at the beginning or the middle of a vector is slow, since all or several of the vector elements need to be moved to make space for the new element.

Elements of a vector can be accessed using the square brackets operator. Elements can be appended to the end using the member function `push_back`. The member function `size` returns the size of the vector.

The following example defines a vector of type `int`, inserts 10 elements, and prints them:

```
#include <vector>
#include <iostream>
using namespace std;

int main() {
  vector<int> v;

  for (int i = 0; i < 10; ++i) { // store 1..10
    v.push_back(i);
  }

  for (const auto &x : v) { // print all elements
    cout << x << ' ';
  }
  cout << endl;

  return 0;
}
```

Compiling and executing the program above gives the following output:

```
## 0 1 2 3 4 5 6 7 8 9
```

As is the case with other STL containers, vectors can hold complex objects including objects corresponding to user defined classes. The example below defines a vector where each element is a vector of integers

```
vector<vector<int>> vvi;
```

A deque is similar to a vector, but it features relatively fast append and prepend operations (these operations are usually fast but sometimes slow[8]). Deques implement (in addition to the `push_back` method) the method `push_front`, which prepends an element to the front of the container.

An array is similar to vector except that its size needs to be known at compile-time and it cannot be modified after it's created. It's thus similar to the statically defined C++ array concept, but it also features the member function `size` that returns the number of elements in the array:

```
array<int, 10> array; // defines an array of 10 integers

// initializes values of all array elements
for (int i; i < array.size(); ++i) {
  array[i] = i;
}
```

[8]Occasionally more memory needs to be allocated when an object is appended to the end of a vector. However, these cases are rare and the insertion operation is considered fast in an amortized sense.

A list is an ordered collection of objects that does not support random access; it's usually implemented using a doubly connected linked list. This implies that to access an element in the middle of the list, the list has to be traversed from the beginning until reaching the desired element. The advantage of a list over vector, deque, and array is that it is much faster to add or remove elements in the middle of a list than it is in the case of a vector or deque (it's impossible in the case of an array). The ways a program stores and accesses data (access patterns) play a key role in choosing which containers to use; for example, a linked list could be suboptimal because of its poor locality of reference (see Sect. 2.9).

A `forward_list` is similar to a list, except that it's typically implemented as a singly connected linked list and thus doesn't offer the member functions `size` and `push_back`.

4.13.2 Associative Containers

Associative containers are ordered containers whose ordering reflects the nature of the contained elements rather than the insertion order. Associative containers are usually implemented using a binary tree.

A set is a container that holds multiple elements, where no two elements can be equivalent. A multiset is similar to a set but it may hold multiple equivalent elements. The operators < and == need to be defined for the contained elements so that the container reflects the correct order and so that the set container can enforce the constraint that no two equivalent elements can belong to the same set.

The following example constructs a set of `string` variables and then inserts four strings with one string being inserted twice. It then proceeds to print all elements in the set using a for-loop. Note that the element that was inserted twice appears only once, and that the elements are printed in alphabetic ordering, rather than in the order of insertion:

```
#include <set>
#include <string>
#include <iostream>
using namespace std;

int main() {
  set<string> s;

  s.insert({"one"});
  s.insert({"two"});
  s.insert({"three"});
  s.insert({"one"});

  for (const auto &x : s) {
    cout << x << ' ';
  }
  cout << endl;
```

```
  return 0;
}
```

Compiling and executing the program above gives the following output:

```
## one three two
```

Replacing the `set<string>` above with `multiset<string>` results in the following output:

```
## one one three two
```

The template struct `pair` holds a pair of variables or objects of generic types. The two variables or objects may have different types. For example, `pair<int,double>` holds an `int` variable and a `double` variable; the two variables can be accessed by referring to the public fields `first` and `second`:

```
pair<int,double> p;

p.first = 10;
p.second = 3.0;
```

Maps and multimaps are containers that store objects of type `pair<K,V>`: pairs of two objects—the first object is of the generic type `K` (for key) and the second is of the generic type `V` (for value). The elements in the map container are ordered based on the key component of the pairs (note that `<` and `==` need to be defined for `K`). A map may not have more than one pair with the same key; a multimap may have multiple pairs with the same key.

The usefulness of maps and multimaps stems from being able to look up container elements based on their key values relatively fast. Below is an example of a multimap containing multiple pairs of strings and integers, for example, a list of names followed by orders:

```
#include <map>
#include <string>
#include <iostream>
using namespace std;

int main() {
  multimap<string,int> names;

  // note the use of curly braces to create a pair
  names.insert({"John", 3});
  names.insert({"Peter", 2});
  names.insert({"Jane", 6});
  names.insert({"Jane", 2});

  for (const auto &x : names) {
    cout << x.first << ": " << x.second << endl;
  }
  return 0;
}
```

Compiling and executing the program above results in the following output:

```
## Jane: 6
## Jane: 2
## John: 3
## Peter: 2
```

Note that the elements are printed above in the order specified by the keys (in this case, alphabetically), and that since we used multimap instead of a map, the container has two elements with the same key, `Jane`.

Elements in a map can also be accessed using square brackets surrounding the corresponding key. Since this notation resembles an array notation, maps and multimaps are often called associative arrays (arrays indexed by keys rather than by a sequence of consecutive integers):

```
#include <map>
#include <string>
#include <iostream>
using namespace std;

int main() {
  map<string,int> names;

  names["John"] = 3;
  names["Jane"] = 6;

  cout << "names[\"John\"]: " << names["John"] << endl;
  cout << "names[\"Jane\"]: " << names["Jane"] << endl;

  return 0;
}
```

Compiling and executing the program above gives the following output.

```
## names["John"]: 3
## names["Jane"]: 6
```

4.13.3 Unordered Containers

Unordered containers are similar to associative containers, except that the ordering of the elements is unreliable and in general may be independent of both the insertion order and the contained elements.

Unordered containers are typically implemented using a hash table (with arrays or linked lists for handling key collisions). Thus, it's generally very fast to insert and to look-up specific values in an unordered container.[9]

[9] In some cases, insertion and look-up may be very slow, but they are fast in an amortized sense.

The code below shows how replacing the multimap, in the aforementioned example, with an unordered map simply results in the objects being traversed out of order (in the previous case of an ordered multimap, the ordering was alphabetical: Jane, Jane, John, Peter).

```
#include <unordered_map>
#include <string>
#include <iostream>
using namespace std;

int main() {
  unordered_multimap<string,int> names;

  names.insert({"John", 3});
  names.insert({"Peter", 2});
  names.insert({"Jane", 6});
  names.insert({"Jane", 2});

  for (const auto &x : names) {
    cout << x.first << ": " << x.second << endl;
  }

  return 0;
}
```

Compiling and executing the program above gives the following output (your output might vary):

```
## Jane: 2
## Jane: 6
## Peter: 2
## John: 3
```

4.14 Notes

In conclusion, it's good to remember what C++ is suitable for: It's considered to be one of the most popular programming languages, in particular in cases where computational efficiency is emphasized or access to the hardware and/or the operating system is required.

There are many excellent books specifically devoted to C++: An authoritative and detailed reference book on C++ is (Stroustrup, 2013); another popular textbook is (Lippman et al., 2012). We also recommend (Josuttis, 2012), which contains a nice overview of the STL. All of the above books were updated to include the C++11 standard.

References

B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, fourth edition, 2013.
S. B. Lippman, J. Lajoie, and B. Moo. *C++ Primer*. Addison-Wesley, fifth edition, 2012.
N. M. Josuttis. *The C++ Standard Library: A Tutorial and Reference*. Addison-Wesley, second edition, 2012.

Chapter 5
Learning Java

The Java programming language is heavily used in big data applications like Apache Cassandra and Elasticsearch. In this chapter, we describe Java starting with compilation, types, operators, I/O, control flow, etc., and concluding with object-oriented programming and other features. We cover parallel programming using Java in Chap. 10.

The first version of Java was released by Sun in 1996. The motivation behind the release was to create a new platform-independent language that is especially suited for web programming. Subsequent versions were released by Sun and later Oracle, which acquired Sun; the current version is Java 11 (as of 2018). James Gosling, a former employee of Sun and later Oracle, is widely considered to be the father of the Java programming language. Java is a high-level programming language that is similar to C++ in many ways. The biggest difference between C++ and Java is that while C++ programs are compiled to a sequence of assembly language instructions (also called native code) for execution by the CPU, Java is executed by a separate program called the Java Virtual Machine (JVM). Specifically, the Java compiler converts the Java program into a byte-code file, which is then used by the JVM to execute the program.

There is significant controversy over whether C++ is preferred to Java, or vice versa. Some advantages of Java over C++ are listed below:

1. The JVM can ensure that a Java program has precisely the same functionality across computers with different hardware architectures. The functionality of C++ is hardware dependent.
2. Java does not have pointers, and as a result it avoids runtime errors resulting from accessing invalid memory.

© Springer Nature Switzerland AG 2018

G. Lebanon, M. El-Geish, *Computing with Data*,
https://doi.org/10.1007/978-3-319-98149-9_5

3. Java has automatic garbage collection, which eliminates the need to manually specify when allocated memory is no longer needed.[1]
4. Java has a much larger standard library than C++, including better support for multi-threading, networking, databases, and graphics.

Some advantages of C++ over Java are listed below:

1. Java was invented by Sun and is now owned by Oracle. Many aspects of it are free, but not all. Specifically, Oracle allows the free usage of Java on desktops and server platforms, but may require royalty payments for the usage of Java in embedded systems.
2. The usage of the JVM has computational overhead, and the process of running byte code instructions is often slower than running native code. This implies that in some cases Java programs are slower than C++ programs.

The computational overhead of the Java (as opposed to compiled C++ code) is often cited as a significant weakness of Java. In recent years, however, Java's overhead declined due to the use of advanced just-in-time (JIT) compilation. JIT compilation identifies computational bottlenecks, also known as hot-spots, and compiles them into native code. As a result, the computational efficiency gap between Java and C++ programs has narrowed significantly. In theory, Java programs may even run faster than C++ since the Java JIT compiler has access to optimization techniques that are not available to the C++ compiler. In practice, however, C++ is usually preferred to Java when computational efficiency is paramount.

Java is similar to C++ in many ways. We therefore structure this chapter similar to Chap. 4. We also make references to relevant parts of Chap. 4 in order to avoid duplicating a large part of Chap. 4.

5.1 Compilation

The Java environment is divided into two components: the Java compiler and the Java virtual machine (JVM). In Linux, the Java compiler is available as the program `javac` and the JVM is available through the program `java` (in Linux, both `javac` and `java` are typically placed in the directory `/usr/bin/`). The Java software, known as the Java Development Kit (JDK), is available for download from Oracle's website http://oracle.com/technetwork/java/javase/downloads; the most recent version, at the time of writing, is JDK 11.

The Java compiler program, `javac`, takes as input a Java program and produces a file containing the Java byte code. The `java` program takes as input the byte code file and runs it within the JVM.

[1] C++11 introduced a smart pointer mechanism that provides garbage collection. The Java formalism, however, is more natural and easy to use.

Files containing Java programs typically have a `.java` suffix and files containing Java byte code typically have a `.class` suffix (the `.class` suffix should be omitted when calling the `java` program on a byte code file—see below for an example). It is convenient to add the directory path for `java` and `javac` to the operating system `PATH` variable, so these programs are recognized without specifying the full path (see Chap. 3).

As in C++, Java code is composed of a sequence of case-sensitive statements separated by semicolons and arranged in functions and classes. The function `main` is special in that the CPU executes its code whenever the JVM executes the corresponding byte code file.

As in C++, Java code is often annotated with comments that provide documentation on the code. Such comments are ignored by the compiler and thus they do not affect the program's functionality. As in C++, there are two ways to define comments in Java: (a) text following `//` that continues until the end of the line, and (b) text between the symbols `/*` and `*/` (possibly spanning multiple lines).

The code describing the Java program resides in a `main` function, as in C++. But in contrast to C++, the `main` function needs to be a static member function of a class whose name matches the file containing the Java code. The `main` function in Java does not return any value (marked by a `void` return value).

For example, the following Java code contains a simple program that prints a greeting message in the operating system terminal. Note that `main` is a static member function of the class `HelloWorld`. The function `System.out.println` is a Java function that prints to the console its argument, followed by a new line character.

```
public class HelloWorld {
   public static void main(String[] args) {
      System.out.println("Hello, World!");
   }
}
```

In order to compile and run the program, we need to save the code above in a file whose name matches the class containing the `main` function: `HelloWorld.java`. Running the Java compiler on this file produces the file `HelloWorld.class` that contains byte code. To execute the program we need to call the `java` program, specifying the name of the byte code file as an argument, without its `.class` suffix[2].

```
# Compile print_greeting.java (Java code) into byte code
javac HelloWorld.java
# The byte code now resides in the file HelloWorld.class
# Execute byte code file in JVM using the java program
# (omit .class suffix)
java HelloWorld

## Hello, World!
```

[2]Using Java 11, you can launch single-file programs using the source code file directly.

In many cases a Java program is spread across multiple files. We may have one file with the class containing the `main` function and other files containing related classes. To compile the program, run the Java compiler (`javac`) with arguments corresponding to all of the files (possibly using filename wildcards to avoid writing a long list of filenames—see Chap. 3). To execute the program, run the `java` program with an argument corresponding to the class containing the `main` function.

It is possible to have multiple `main` functions, for example each class may have its own `main` function. One reason for doing so is having a main function representing the primary executable program and other `main` functions corresponding to auxiliary programs that test the code in the corresponding classes. After compilation, the correct `main` function can be executed by supplying the java program with the appropriate byte code file.

We proceed below to explore the Java programming language. We start with low-level features such as variables, control flow, functions, and input and output. We continue with object oriented programming, and follow up with generics and the Java collections, which correspond to the C++ concepts of templates and STL (see Chap. 4).

5.2 Types, Variables, and Scope

Java types and variables are very similar to C++ types and variables. We refer the reader to Sect. 4.2 for more information on types and variables.

A few key differences between Java and C++ are described below.

1. The Java type `char` uses 16 bits to represent unicode characters (unicode is a set of characters than include Asian, Arabic, and other characters in addition to the ASCII characters). The Java type `byte` uses 8 bits to represent integers in the range $-128, \ldots, 127$.
2. The Java type `boolean` represents boolean values. The possible values are `true` and `false` as in C++ `bool` type.
3. In Java, the number of bits each type uses (and consequentially the range of possible values and approximation quality in the case of floating-point representations) is hardware independent. For example, `int` uses 4 bytes to represent integers in a range that is approximately ± 2 billion, while `long` uses 8 bytes. Similarly, `float` uses 4 bytes to represent values in a range that is approximately $\pm 3 \cdot 10^{38}$ (up to 6–7 significant digits) while `double` uses 8 bytes to represent values in a range that is approximately $\pm 1.7 \cdot 10^{308}$ (up to 15 significant digits).
4. Uninitialized Java variables cannot be accessed until they are initialized.

```
public class UninitializedVariableExample {
  public static void main(String[] args) {
    int a = 3;
    System.out.println(a); // ok; prints 3
```

```
        int b;
        b = 3;
        System.out.println(b); // ok; prints 3
        int c;
        System.out.println(c); // error: c is not initialized
    }
}
```

5. Java uses the keyword `final` to represent variables that can only be assigned once (whose references are constant). For example, a variable defined as `final int a = 3` cannot be reassigned later on.

5.3 Operators and Casting

Operators and casting in Java are very similar to C++. We refer the reader to Sect. 4.3 for more information on operators and casting.

One important difference between Java and C++ is that Java is less aggressive in applying implicit casting than C++. As a result, Java is considered more strongly typed than C++. For example, Java integers cannot be implicitly cast to a `boolean` type.

For example, the expression `int a=0; if (a) a=1;` is legal in C++ but illegal in Java since the integer `a` in `(a)` needs to be converted to a boolean variable. Similarly, the Java expression `int a = 3.2;` is illegal as implicit casting is not pursued.

5.4 Primitive and Non-Primitive Types

As mentioned in Sect. 5.2, primitive variables in Java behave very similar to C++ variables.

Java also has classes and objects that are very similar to the corresponding C++ concepts. Classes are collections of variables, called fields, and functions, called methods. An object is an instantiation of a class in the same way that a primitive variable is an instantiation of a type. Once an object of a certain class is defined, its fields and methods can be called using the period operator. For example, if `x` is an object then `x.fieldName` or `x.methodName(...)` refer to the field `fieldname` and the method `methodName` that are associated with the object `x`.

Defining non-primitive objects in Java requires the usage of the `new` statement, as in the case of allocating dynamic memory in C++. Assigning one object to another does not create a copy of the object but instead makes both objects refer to the same memory content, and changing one object will cause the other object to change as well. In other words, Java objects behave like pointers to dynamically allocated memory in C++ (without the need for the C++ pointer operators `*` and `&`).

For example, the code below defines an object of class `Date` and then assigns that object to a new object. Since both objects point to the same memory location, modifying the second object causes the first object to change as well.

```
import java.util.Date;

public class DateExample {
  public static void main(String[] args) {
    Date date1 = new Date(); // define a new object date1
    Date date2 = date1; // point date2 to date1
    // print both objects
    System.out.println(date1);
    System.out.println(date2);
    // Both objects date1 and date2 refer to the same memory
    // Modifying date2 changes date1 as well
    date2.setTime(1);
    // print both objects
    System.out.println(date1);
    System.out.println(date2);
  }
}
```

Compiling and executing the code above give the following output.

```
## Wed Apr 24 18:02:49 PDT 2013
## Wed Apr 24 18:02:49 PDT 2013
## Wed Dec 31 16:00:00 PST 1969
## Wed Dec 31 16:00:00 PST 1969
```

In an analogous C++ program, the assignment `date2 = date1` would have created a new object `date2` that is a copy of `date1` and subsequent modification of `date2` would not modify `date1`.

This pointer-like behavior of Java objects[3] applies to any non-primitive type, including arrays, strings, and user-defined classes.

In addition to primitive types such as `int`, `double`, and `boolean` Java has corresponding wrapper classes `Integer`, `Double`, and `Boolean`. These classes allow defining objects that hold basic numeric types, but have additional functionality such as class methods that can be used. For example the `Integer` class defines methods for converting from string to an `Integer` and vice versa.

```
// creating an Integer object with value 2
Integer n = Integer.valueOf(2);
// call method of Integer to transform value to string
String s = n.toString();
```

Java's auto-boxing functionality allows using primitive types when the corresponding objects are expected. Similarly, the un-boxing functionality allows using objects when the corresponding primitive types are expected.

[3] Java does not have references or pointers.

```
Integer n1 = 2; // auto-boxing
int n2 = n1; // un-boxing
```

5.5 Arrays

Java arrays correspond to the C++ concept of pointers to dynamically allocated memory containing sequences of variables or objects.

Below are some key differences and similarities between arrays in Java and C++.

1. In Java, the type of an array holding variables or objects of type `A` is `A[]`. Thus, the statement `int[] x = new int[10]` defines an array of 10 elements of type `int`.
2. The size of a Java array does not need to be known during compilation.
3. The size of a Java array cannot be modified after it is defined.
4. There is no need to explicitly free the allocated memory when the array is no longer needed. The JVM's garbage collection mechanism takes care of that automatically.
5. The JVM performs range checking on Java arrays, and does not allow accessing elements that are outside of the range of the allocated array. Such a violation in Java will result in a compilation error, a significant improvement over returning the wrong answer or suffering a runtime crash as may happen in C++.
6. If `x` is a Java array, the expression `x.length` returns the size of the array.
7. Assigning one array to another `x = y` has the effect of having both arrays refer to the same memory content.

5.5.1 One-Dimensional Arrays

The statement `A[] x = new A[n]` defines a new array `x` of size `n`, holding primitive variables or objects of type `A`. The array elements can be accessed using the square bracket notation, for example `x[k]` refers to the `k+1` element of the array `x`.

Traversing the array can be done using a for-loop with an index variable (the `length` field can be used to limit the index variable) or using a range for-loop. See Sect. 4.8.3 for more information on C++ for-loops, which are identical to Java for-loops.

```
public class ArrayExample {
  public static void main(String[] args) {
    int n = 10;
    int[] ar = new int[n];
```

```
    for (int i = 0; i < ar.length; i++)
      ar[i] = i;
    for (int e : ar)
      System.out.print(e + " ");
  }
}
```

The program above produces the following output:

```
## 0 1 2 3 4 5 6 7 8 9
```

Adding to the program above the line `ar[12] = 3` yields a compilation error as an element outside the array boundary is accessed.

5.5.2 Multidimensional Arrays

Multidimensional arrays in Java are essentially arrays of arrays. They are represented by a double square bracket notation.

```
public class MultidimensionalArrayExample {
  public static void main(String[] args) {
    int[][] ar = new int[10][10];
    for (int i = 0; i < ar.length; ++i)
      for (int j = 0; j < ar[i].length; ++j)
        ar[i][j] = i * 10 + j;
    for (int i = 0; i < ar.length; ++i) {
      for (int j = 0; j < ar[i].length; ++j) {
        System.out.print(ar[i][j] + " ");
      }
      System.out.print("\n");
    }
  }
}
```

The program above produces the following output:

```
## 0 1 2 3 4 5 6 7 8 9
## 10 11 12 13 14 15 16 17 18 19
## 20 21 22 23 24 25 26 27 28 29
## 30 31 32 33 34 35 36 37 38 39
## 40 41 42 43 44 45 46 47 48 49
## 50 51 52 53 54 55 56 57 58 59
## 60 61 62 63 64 65 66 67 68 69
## 70 71 72 73 74 75 76 77 78 79
## 80 81 82 83 84 85 86 87 88 89
## 90 91 92 93 94 95 96 97 98 99
```

Some of the array elements in a two-dimensional array may be arrays of different lengths (ragged array).

```
public class class RaggedMultidimensionalArrayExample  {
   public static void main(String[] args) {
     int[][] ar = new int[10][];
     for (int i = 0; i < ar.length; ++i)
       ar[i] = new int[i+1];
     for (int i = 0; i < ar.length; ++i)
       for (int j = 0; j < ar[i].length; ++j)
         ar[i][j] = i * 10 + j;
     for (int i = 0; i < ar.length; ++i) {
       for (int j = 0; j < ar[i].length; ++j) {
         System.out.print(ar[i][j] + " ");
       }
       System.out.print("\n");
     }
  }
}
```

The program above produces the following output:

```
## 0
## 10 11
## 20 21 22
## 30 31 32 33
## 40 41 42 43 44
## 50 51 52 53 54 55
## 60 61 62 63 64 65 66
## 70 71 72 73 74 75 76 77
## 80 81 82 83 84 85 86 87 88
## 90 91 92 93 94 95 96 97 98 99
```

5.6 Packages and the Import Statement

A package in Java is a collection of classes. By convention, packages are organized in a hierarchy that reflects the semantic role of the individual packages.

Packages allow multiple developers to work on different programs independently without worrying that their classes and variables have identical names. A class is recognized in a program if it is prefixed by the package name followed by a period. Thus if we have two classes with the same name `A` in two different packages `p1` and `p2`, we can distinguish the two classes by referring to them as `p1.A` and `p2.A`.

For example, we can refer to the class `Date` in the `java.util` package by adding the appropriate prefix whenever the class is referred.

```
java.util.Date date1 = new java.util.Date();
```

This cumbersome prefix notation can be avoided by including an `import` statement followed by the class or package name at the top of the Java program. Import statements may refer to specific classes, in which case the `import`

statement is followed by the name of the package followed by the name of the class (separated by a period). Import statements can also refer to entire packages using the wildcard notation `*`.

For example, the code below uses the class `Date`, defined in the package `java.util`, without the prefix notation.

```
import java.util.Date;
...
Date date1 = new Date();
```

Alternatively, we can import the entire package `java.util`.

```
import java.util.*;
...
Date date1 = new Date();
```

Packages may be created by including the statement `package package_name` at the top of the Java program. To create a hierarchy of packages, the hierarchy must correspond to the constituents of the package name (separated by periods), and files must be placed in file directories with the corresponding directory hierarchy.

Class files can also be stored in a Java archive (JAR) file, which is a compressed archive containing a hierarchy of directories corresponding to a hierarchy of packages. A JAR file is convenient since it is a single file (rather than a directory) and it is stored in a compressed format. The classes and packages in the JAR file will be recognized if the class path environment variable includes the path to the directory containing the JAR file, or if the `java` command uses the flag `-classpath` followed by the appropriate directory.

```
# modifying the class path in bash, using : as separator
# (period corresponds to current directory)
export CLASSPATH=/home/joe/classes:/home/jane/classes:.

# alternatively, the class path can be passed as an
# argument to java
java -cp /home/joe/classes:/home/jane/classes:. program.java
```

5.7 Strings, Input, and Output

The `String` class in Java offers functionality that is similar to the C++ `string` class. One difference is that the Java `String` is immutable i.e., a `String` object cannot be modified after it is initialized.

Java `String` objects cannot use square brackets to refer to specific characters in the string as in C++, but the addition operator + implements concatenation on `String` objects as it does in C++. If + appears between a string and a non-string object, the non-string object is converted to a string (if possible), and concatenated with the string object.

The example below defines a `String` object `greeting`, prints the second character using the member function `charAt`, checks whether `greeting` is equal to the string `"Hello"` (it is), and then concatenates the substring `"Hel"` with `"p"`.

```
public class StringExample {
   public static void main(String[] args) {
    String greeting = "Hello";
    System.out.println(greeting);
    System.out.println(greeting.charAt(2));
    System.out.println(greeting.equals("Hello"));
    System.out.println(greeting.substring(0,3) + "p");
   }
}
```

The program above produces the following output:

```
## Hello
## l
## true
## Help
```

5.8 Control Flow

Control flow in Java is very similar to control flow in C++. Please refer to Sect. 4.8 for more information. See Sect. 5.5 for several concrete examples of using for-loops in Java.

5.9 Functions

Functions in Java are similar to functions in C++. See Sect. 4.9 for relevant information on C++ functions, including recursion and parameter overloading. We describe below three key differences between functions in Java and functions in C++.

The first key difference is that all Java functions are methods (member functions in a class); C++ functions can be methods or stand-alone functions.

The second key difference is in the way arguments are passed. Passing primitive types (for example, `int` or `double`) as function arguments in Java behaves as it does in C++. On the other hand, passing objects as function arguments in Java behaves in the same way as passing pointers to the corresponding objects in C++. For example, if an object in Java is passed to a function, any modification to the argument inside the function will persist after the function terminates.

For example, the code below defines two functions that modify their arguments. The first function accepts a primitive variable of type `int` and the second function

accepts an array. Changes made by the first function to the argument do not persist after the function terminates while changes made by the second function do persist.

```
public class FunctionArgumentExample {
  private static void foo1(int a) {
    a = 1;
  }

  private static void foo2(int[] arr) {
    arr[0] = 1;
  }

  public static void main(String[] args) {
    int a = 3;
    System.out.println("primitive variable a before foo1: " + a);
    foo1(a);
    System.out.println("primitive variable a after foo1: " + a);
    int[] b = new int[1];
    b[0] = 3;
    System.out.println("b[0] before foo2: " + b[0]);
    foo2(b);
    System.out.println("b[0] after foo2: " + b[0]);
  }
}
```

The program above produces the following output:

```
## primitive variable a before foo1: 3
## primitive variable a after foo1: 3
## b[0] before foo2: 3
## b[0] after foo2: 1
```

The third key difference between C++ and Java functions is the way arguments are passed to `main`. In Java the argument of `main` is an array of string objects, `string[] args`, where `args[0]` contains the first parameter, `args[1]` contains the second parameter, and so on.

5.10 Object Oriented Programming

5.10.1 Classes

As in C++, Java classes are a collection of primitive variables and objects (called fields), and functions (called methods). An object is an instantiation of a class, in the same way that a primitive variable is an instantiation of a primitive type. As in C++, the fields and methods of an object are referenced using the period operator. See Sect. 4.10.2 for the details of C++ classes, most of which apply to Java classes as well. Some important differences between classes in C++ and Java are listed below.

- In contrast to C++, in Java we need to prefix each field or method definition by its access modifier (for example, `public` or `private`). C++ uses the labels `public:` and `private:` to label as public or private all subsequent fields and methods.
- Java does not require a semicolon after a class definition.
- In Java, objects must be defined using dynamic memory allocation, for example using the following statement `Point p = new Point()`. In C++, objects can be defined using dynamic memory allocation (`Point *p = new Point()`) or using static memory allocation (`point p`). Since Java has automatic garbage collection, the dynamically allocated memory does not need to be explicitly freed.
- Java uses the method `finalize` instead of the C++ destructor.
- The `this` keyword in Java refers to the current object, rather than a pointer to the current object as it does in C++. Thus the C++ statement `this->fieldName` becomes `this.fieldName` in Java.
- Static fields and methods in Java are accessed using the period operator prefixed with the class name (rather than the C++ `::` operator). Thus, the C++ statement `MyClass::staticField` is equivalent to the Java statement `MyClass.staticField`.
- In Java, fields that are not explicitly initialized by the constructor are initialized to default values. This is in contrast to variables that are not fields (for example, local variables defined in a method), which must be explicitly initialized before their use.

The example below defines a class that is similar to the C++ `Point` class from the previous chapter. Our example contains two classes, both of which exist in a single Java file `PointExample.java`. The first class corresponds to the concept of a point in a two-dimensional space, and the second class holds the `main` function. Note that the file name should match the name of the class containing the `main` function.

```
class Point {
  private double x;
  private double y;

  public Point() { // empty constructor
    x = 0;
    y = 0;
  }

  public Point(int nx, int ny) { // non-empty constructor
    x = nx;
    y = ny;
  }

  public void set_x(double nx) {
    x = nx;
  }
```

```
  public void set_y(double ny) {
   y = ny;
  }

  public double get_x() {
    return(x);
  }

  public double get_y() {
    return(y);
  }

  public void reflect() {
    x = -x;
    y = -y;
  }
}

public class PointExample {
  public static void main(String[] args) {
    Point p = new Point();
    p.set_x(1);
    p.set_y(2);
    System.out.println("(" + p.get_x() + "," + p.get_y() + ")");
    p.reflect();
    System.out.println("(" + p.get_x() + "," + p.get_y() + ")");
  }
}
```

Compiling the above program with the `javac` program creates two byte code files `PointExample.class` and `point.class`. To execute the program, we need to call the JVM with the byte code file corresponding to the class that contains the `main` function: `java PointExample`. This gives the following output:

```
## (1.0,2.0)
## (-1.0,-2.0)
```

Recall that objects in Java behave semantically like pointers to objects in C++. This means that assigning one object to another makes both refer to the same memory and changing the fields through one object will persist when the fields are accessed through the other object. The `clone` method in Java provides a way to create a shallow copy of an existing object.

For example, in the following code both `p1` and `p2` refer to the same object in memory so that when the object is reflected by `p2`, the change appears also when we access `p1`.

```
public class PointExample2 {
  public static void main(String[] args) {
    Point p1 = new Point(1, 2);
    Point p2 = p1;
    p2.reflect(); // note: p1 is also modified
```

```
    System.out.println("(" + p1.get_x() +
        "," + p1.get_y() + ")");
  }
}
```

The program above produces the following output:

```
## (-1.0,-2.0)
```

In C++, there is only one way to initialize the fields, using a constructor. In Java, there are two additional ways for initializing the fields, listed below.

1. Assigning values to fields inside the class definition initializes the fields to the corresponding values before a constructor is executed. If static fields are initialized during their definition, the initialization is carried out only once—when the class is loaded at runtime.
2. An initialization block is a sequence of Java statements inside curly braces that lies inside a class definition. The block is executed whenever an object is constructed. Static fields can be initialized using static initialization blocks, which are prefixed by the keyword `static`.

The code below demonstrates these two alternative field initialization methods.

```
class Point {
  private double x = 0;
  private double y;
  {
    y = 2;  // initialization block
  }
}
```

5.10.2 Inheritance

Inheritance in Java is similar to inheritance in C++. Section 4.10.4 contains details and examples of inheritance in C++. We use the Java terminology of superclass and subclass rather than the C++ terminology of base class and derived class. The important differences between Java and C++ are listed below.

1. Inheritance in Java is marked using the `extends` keyword, rather than the C++ `:` operator.
2. In Java, the keyword `this` can be used to call the constructor or refer to the current object. The keyword `super` can be used to refer to the superclass or call the constructor of the super-class. For example, within a subclass we can refer to a superclass method using `super.methodName()` and call the superclass constructor using `super()`.
3. Denoting a class as final (by adding the prefix `final` before the class definition) prevents other classes from inheriting from it.

4. C++ implements static binding by default, and dynamic binding requires defining the appropriate method as `virtual` (see Sect. 4.10.5). Java implements dynamic binding by default, and there is no need to explicitly specify a method as virtual.
5. If class `A` is the superclass of class `B`, in C++ a pointer of type `A*` can point to an object of class `B`. This allows the use of polymorphism, for example having a single array containing pointers to objects of classes `A` and `B`. Java has no pointers, but it allows an object of a subclass `B` to be assigned to an object of the superclass `A` (or to an element of an array of type `A[]`). If an object of type `A` refers to an object of type `B`, then only the functionality of `A` can be used. Casting the object back to type `B` allows using the full functionality of the subclass `B`.

```
class Parent { // superclass
  int a;
  Parent(int na) { // constructor of Parent
    a = na;
  }
}

class Child extends Parent { // subclass
  int b;
  Child(int na, int nb) {
    super(na); // call to constructor of class Parent
    b = nb;
  }
}

public class InheritanceExample {
  public static void main(String[] args) {
    Parent a = new Parent(3); // define an object of class Parent
    Child b = new Child(1, 3); // define an object of class Child
    // polymorphism: a Parent object refers to a Child object
    Parent c = b;
    Child d;

    if (c instanceof Child)  // check if c points to class Child
      // cast the object referred to by c, to type Child
      d = (Child)c;
  }
}
```

5.10.3 Abstract Classes

Abstract methods are functions that are defined in a class without an implementation. To specify that a method is abstract, prefix the method definition with the `abstract` keyword. A class that has one or more abstract methods is called an abstract class, and the prefix `abstract` must appear before the class definition.

Since abstract classes may contain methods without an implementation, it is illegal to define objects corresponding to abstract classes. If a subclass of an abstract class implements the abstract methods, the subclass is no longer abstract and the corresponding objects may be defined.

5.10.4 Access Modifiers

Access modifiers determine the visibility of fields and methods to different software component. The following list describes three access modifiers, and the default modifier that applies when no access modifier is explicitly selected.

public: Public fields and methods are visible to all.
private: Private fields and methods are visible only to the class itself.
protected: Protected fields and methods are visible only to the class itself, its subclasses, and the package.
default: Protected fields and methods are visible only to the class and the package.

5.11 The Object Class

All classes in Java are implicitly inherited from the `Object` class, and share the methods defined in the `Object` class. Some of the methods of `Object` are listed below:

Expression	Functionality
`x.equals(y)`	returns `true` if the two objects are equal.
`x.hashCode()`	returns an integer hash code for `x`. The hash value is based on the content of the object; two objects that are equal (according to the `equals` method) should get the same hash value.
`x.toString()`	returns a string describing the object `x`. It's used implicitly when an object is printed using `System.out.print`.
`x.clone()`	returns a new object that is a duplicate of `x`. By default, this returns a shallow copy of `x`.

Since these functions are defined for `Object`, a class that is the superclass of all other classes, it may be desirable to overload some of these methods if their default behavior is inappropriate. For example, if `x` is an object whose fields are themselves objects, `clone()` will create a duplicate with a new set of fields that point to the

same memory as the fields of the original objects. In other words, `clone` performs shallow copying and fails to duplicate nested objects within the original object. A customized `clone` may be defined using function overloading to perform deep copying.

5.12 Interfaces

An interface is a collection of method declarations (methods whose implementations are missing). A class implements the interface if it provides an implementation to the methods in the interface. We denote this by appending `implements InterfaceName` after the class name in the class definition. All methods in an interface are by default `public` and should be defined in that way in the implementing class.

For example, the interface below contains the method `foo`. The class `Concretion` implements that interface.

```
interface Interface {
  int foo();
}

class Concretion implements Interface {
  public int foo() {
    return 0;
  }
}
```

A class can implement more than a single interface and each interface can be implemented by more than a single class. The interface mechanism may be used in conjunction with inheritance.

5.13 Generics

Java generics are very similar to templates in C++. See Sect. 4.12 for more details on C++ templates. A few important differences between C++ templates and Java generic appear below.

1. In Java there is no need to include the keyword `template` when defining a generic class.
2. The generic type may by omitted during the dynamic memory allocation of a generic object. For example in `ArrayList<String> books = new ArrayList<>()` we omit the generic type after the `new` statement. The omitted type is inferred automatically.

3. Generic parameter types cannot be primitive variables. For example, the objects `Double` and `Integer` should be used as generic parameters instead of the primitive variables `double` and `int` (see Sect. 5.4 for more information on primitive types and their corresponding objects).
4. A single method within a non-generic class can be defined as generic. The appropriate syntax is to prefix the method name with the angle bracket notation rather than to append it as in C++.

   For example, the code below defines a generic class in Java.

```
class Point<T> {
  private T x;
  private T y;

  public T get_x() {
    return x;
  }

  public T get_y() {
    return y;
  }

  public void set_x(T nx) {
    x = nx;
  }

  public void set_y(T ny) {
    y = ny;
  }

  Point() {}  // empty constructor
}

public class GenericsExample {
  public static void main(String[] args) {
    Point<Double> p = new Point<>();
    p.set_x(3.0);
    p.set_y(1.0);
    System.out.println("x: " + p.get_x() + " y: " + p.get_y());
  }
}
```

   The program above produces the following output:

```
x: 3.0 y: 1.0
```

   The code below defines a generic method.

```
class ClassName {
  ...
  // definition of a generic method
  public static <T> T methodName(T x) {
  ...
```

```
  }
  ...
}

public class Main {
  public static void main(String[] args) {
    ...
    // call generic method
    ClassName.<String>methodName("foo");
    ...
  }
}
```

5.14 Collections

Java collections are analogous to container classes in C++'s STL (see Sect. 4.13). There are multiple generic classes that implement standard collection operations such as insertion, deletion, look-up, and iteration. The classes differ in the precise operation that they implement, and in the way the operations are implemented. For example, some classes are more efficient in insertion while other classes are more efficient in look-up.

Some popular Java collection classes are listed in the table below:

Class	Description
`ArrayList`	Similar to a Java array, with the added capability of modifying its size as elements are added or deleted (similar to vector in C++ STL).
`LinkedList`	Ordered sequence that allows fast insertion and deletion at any location.
`HashSet`	Unordered collection that does not allow duplicates.
`TreeSet`	Ordered version of `HashSet`.
`PriorityQueue`	Ordered collection that allows fast removal of smallest element.
`HashMap`	Stores key-value pairs for fast retrieval based on key values.
`TreeMap`	Ordered version of `HashMap`.

The classes above map to data structures in a straightforward way. `ArrayList` corresponds to an array that has extra memory in case additional elements are inserted (the array is copied to a different location if additional size is needed). `ArrayList` provides random access (fast look-up at an arbitrary position of the array).

`LinkedList` is implemented as a doubly connected linked list and allows easy insertion and deletion at any point inside the sequence. It does not allow random access as it requires traversing the linked list until reaching the required position.

`HashSet` and `HashMap` are implemented as hash tables using linked lists for collision handling. They differ in that `HashMap` stores key-value pairs whereas `HashSet` stores only keys. Both classes do not allow multiple copies with the same keys and are unordered in the sense that they do not allow traversing the elements in order. `HashSet` and `HashMap` provide fast look-up, insertion, and deletion as long as the hash function hashes the data with a uniform key distribution.

`PriorityQueue` is implemented as a heap. It allows fast removal of the smallest element.

`TreeSet` is an ordered version of `HashSet` that allows traversing all elements in order. `TreeMap` is an ordered version of `HashMap` that allows traversing all elements in order. `TreeSet` and `TreeMap` are implemented using binary search trees rather than hash tables.

The code below shows an example of using the `ArrayList` class:

```
import java.util.ArrayList;

public class ArrayListExample {
  public static void main(String[] args) {
    ArrayList<Double> al = new ArrayList<Double>();
    // insert three elements
    al.add(1.0);
    al.add(3.0);
    al.add(2.0);
    // traverse collection and print elements
    for (double d : al)
      System.out.print(d + " ");
    System.out.println("");
    // modify 2nd element
    al.set(1, 3.5);
    // print 2nd element (random access)
    System.out.println(al.get(1));
    // traverse collection and print elements
    for (double d : al)
      System.out.print(d + " ");
    System.out.println("");
  }
}
```

Running the program above produces the following output.

```
## 1.0 3.0 2.0
## 3.5
## 1.0 3.5 2.0
```

The code below shows an example of using the `HashMap` class.

```
import java.util.HashMap;
```

```
public class HashMapExample {
  public static void main(String[] args) {
    HashMap<String,Double> hm = new HashMap<>();
    // insert three elements
    hm.put("John", 1.0);
    hm.put("Mary", 3.0);
    hm.put("Jane", 2.0);
    // print all entries
    System.out.println(hm);
    // remove an entry
    hm.remove("Jane");
    // print all entries
    System.out.println(hm);
    // Look up value based on key
    System.out.println("John: " + hm.get("John"));
  }
}
```

Running the program above produces the following output.

```
## {Jane=2.0, Mary=3.0, John=1.0}
## {Mary=3.0, John=1.0}
## John: 1.0
```

For more information refer to the online documentation or the textbooks in the next section.

5.15 Notes

There are many good textbooks on Java. One example is the two volume set: (Horstamann and Cornell, 2012) (volume 1), which covers basic features, and (Horstamann and Cornell, 2013) (volume 2), which covers advanced features. Another excellent book that offers a deeper, more advanced look into using Java for building big-scale systems is (Bloch, 2008). The online application programming interface (API) at https://docs.oracle.com/javase/11 provides extensive Java documentation (replace 11 in the URL with a different Java version number if needed). The website http://docs.oracle.com/javase/tutorial contains links to many Java tutorials.

References

C. Horstamann and G. Cornell. *Core Java, volume 1*. Prentice Hall, ninth edition, 2012.
C. Horstamann and G. Cornell. *Core Java, volume 2*. Prentice Hall, ninth edition, 2013.
J. Bloch. *Effective Java: Programming Language Guide*. Addison-Wesley, 2001. ISBN 9780201310054

Chapter 6
Learning Python and a Few More Things

Python is one of the most popular programming languages. It's broadly used in programming web applications, writing scripts for automation, accessing data, processing text, data analysis, etc. Many software packages that are useful for data analysis (like NumPy, SciPy, and Pandas) and machine learning (scikit-learn, TensorFlow, Keras, and PyTorch) can be integrated within a Python application in a few lines of code. In this chapter, we explore the programming language in a similar approach to the one we took for C++ and Java. In addition, we explore tools and packages that help accelerate the development of data-driven application using Python.

Python is a general-purpose programming language that emphasizes object-oriented programming, extensibility, and code readability. Like R, Python's syntax makes it possible to express complex programs in fewer lines of codes than C++ or Java.

Python is most often used as an interpreted scripting language. Python code is typically stored in .py files. When the Python program executes a script, it compiles the Python code in the .py file into Python byte code that is stored in .pyc or .pyo files. The CPython interpreter program then executes the Python byte code. The interpreted nature of this process makes Python programs slower than the corresponding C or C++ implementations. Typically, Python programs are also slower than the corresponding Java programs but faster than the corresponding R programs. It is also possible to compile Python code into an executable file that is executed on a machine that does not have the CPython interpreter installed.

Python can be run inside an interactive shell by typing `python` in the command-line shell (e.g., the Linux terminal). Typing `exit()` terminates the interactive shell and returns to the command-line shell. In the interactive mode, programmers can enter commands and see their effects immediately, which accelerates the software development and data analysis processes. Many programmers prefer to use the IPython interactive shell instead of the standard Python interactive shell. An alternative way of running Python interactively is

© Springer Nature Switzerland AG 2018

G. Lebanon, M. El-Geish, *Computing with Data*,
https://doi.org/10.1007/978-3-319-98149-9_6

through the Eclipse plugin PyDev (http://pydev.org), or by using a python IDE such as IDLE (http://docs.python.org/2/library/idle.html) or Enthought's Canopy (https://www.enthought.com/products/canopy). Python modes are also available for the Emacs and Vim editors.

A noninteractive programming process begins with writing the Python code and storing it in a .py file and then calling the python command with the appropriate file name argument, for example `python my_prog.py`. If the line `#!/usr/bin/python` appears at the top of a Python code file, the file can be executed by typing the file name in the Linux or Mac terminal. This assumes that the Python program is placed in the `/usr/bin` directory—you can check the corresponding directory by running `which python`—and that the file containing the Python code has executable permissions (see Chap. 3 for more details on shells and file permissions).

Each line of Python code may hold several Python commands separated by a semicolon (if the line contains a single command the semicolon is optional). Text following a # symbol is considered a comment and is ignored by the Python program.

In this chapter, we prefix lines output by the program by ## followed by a blank space; doing so makes it easy to copy and paste code segments (from electronic versions of this book) into an IPython interactive shell or text file (lines prefixed by ## will be ignored as comments and will not trigger an error). The code segment below demonstrates a short Python program followed by the output generated by the Python program when the code is executed; we will describe the `print` function in more detail later on in this chapter:

```
# this is a comment
print("hello world")
## hello world
```

The Python programs in this book target Python 3.5.1 (the latest version available at the time of writing these lines). Python 2.x is legacy and Python 3.x offers several new features, but it's not entirely backward-compatible[1] with Python 2.x; Sect. 6.11 describes material differences between the two versions. The code examples in this chapter should run as expected using either version (unless otherwise stated).

6.1 Objects

Variable names in Python are untyped and refer to typed objects in memory. This means that a variable name may be reassigned to refer to an object of a different type, but the type of objects in memory is fixed.

[1]Most of the important data analysis extensions have been ported from Python 2.x to 3.x, but not all.

```
# variables are dynamically typed
x = 3
x = "three" # ok in Python
```

In an assignment such as `x = y`, the object referred to by `y` is not copied—instead the assignment makes the names `x` and `y` refer to the same object; this operation is called name binding.[2] Thus, there may be more than a single name referring to the same object in memory (this is similar to the C++ concept of references).

```
x = [1, 2, 3]  # x is a list
y = x  # both x and y refer to the same list
y.append(4)  # appending to y impacts x as well
print(x)
## [1, 2, 3, 4]
```

In Python, every number, string, data structure, function, class, etc., is an object. As in C++ and Java, Python objects have member variables and member methods that can be accessed using the dot operator.

```
s = "three"
# call upper, a member method of the string object
t = s.upper()
print(t)
## THREE
```

A Python module is a code object loaded into the Python program by the process of importing (e.g., a .py source file that is executed when we call `import` followed by the module file name without the .py suffix). In many cases, modules consist of definitions of variables, functions, and classes, rather than code outside of such environments. When importing such modules, the definitions in the module become available for usage, but no additional side effect is executed during the `import` command.

For example, suppose we have a module file `my_vars.py` containing the following code.

```
LENGTH = 24
WIDTH = 48
```

The definitions in this file can be accessed in a different file using the appropriate `import` command:

```
import my_vars
# accessing variables in my_vars module
area = my_vars.LENGTH * my_vars.WIDTH
```

The statement `import module as alias` allows the following syntax:

[2] https://docs.python.org/3/reference/executionmodel.html#naming-and-binding.

```
import my_vars as mv
area = mv.LENGTH * mv.WIDTH
```

The statement `from module_name import x` makes the definition `x` in the current module accessible without the need for its prefix; for example:

```
from my_vars import LENGTH, WIDTH
area = LENGTH * WIDTH
```

Replacing `x` with the wildcard `*` makes all the definitions available for use without the need for a prefix:

```
from my_vars import *
area = LENGTH * WIDTH
```

The function `dir()` displays all currently defined names (e.g., variables, functions, and modules). If supplied with an argument, `dir(X)`, it shows all names defined in the module `X`. The function `help(X)` displays information about the function, class, or module `X`.

Shell commands can be executed from within Python using the function `system("shell command")`, which is defined in the module `os`:

```
import os
os.system("ls -al")  # call shell command ls -al
```

A newer alternative to `os.system` that provides much more flexible functionality is the `call` function, which is defined in the `subprocess` module. Subprocess can be used to spawn new processes, and provides input and output redirection:

```
import subprocess as sp
sp.call(["ls", "-al"])
```

The function `os.chdir(path)` changes the program's current directory. The function `os.getcwd()` returns the current directory.

6.2 Scalar Data Types and Operators

Python's scalar data types are similar to the corresponding types in C++ and Java. A `bool` type holds `True` or `False` values. An `int` type holds integers and a `float` holds floating-point number values. Large `int` values are automatically converted to a `long` type, which represents potentially large integers with arbitrary precision. The `str` type holds strings. The type `None` represents "null" value.

In some cases, objects of a particular type can be converted or cast to objects of a different type. Such casting tries to match the value in the new type to the value in the old type. Casting an object to a new type requires calling a function whose name is identical to the target type:

```
a = 42
b = float(a)
c = str(a)
d = bool(a)
# print the type and value for each object above
for x in a, b, c, d:
  print("type: {}, value: {}".format(type(x), x))
## type: <class 'int'>, value: 42
## type: <class 'float'>, value: 42.0
## type: <class 'str'>, value: 42
## type: <class 'bool'>, value: True
```

In some cases casting fails with an error; for example, casting a string object whose value can't be converted into an integer (in base 10 by default):

```
int("a")
## Traceback (most recent call last):
##   File "<stdin>", line 1, in <module>
## ValueError: invalid literal for int() with base 10: 'a'
```

Python operators are similar to operators in C++ and Java; however, boolean operators in Python are usually expressed using keywords, for example `and`, `or`, and `not`.

```
x = True
y = False
print(x and y)
print(x or y)
print(not x)
## False
## True
## False
```

Figure 6.1 lists a few cases where Python considers operators differently from Java or C++. Division in Python 3.x is different; Sect. 6.11 explains that difference.

Python Operator	Description
`x ** y`	`x` raised to the power `y`
`x / y`	True division: divide `x` by `y` (real floating-point quotient)
`x // y`	Floor division: divide `x` by `y` and omit remainder
`x and y`	`True` if `x` and `y` are interpreted as true
`x or y`	`True` if either `x` or `y` is interpreted as true
`not x`	`True` if `x` is interpreted as false
`x is y`	`True` if `x` and `y` refer to the same object
`x is not y`	`True` if `x` and `y` do not refer to the same object
`x == y`	`True` if `x` and `y` are equal

Fig. 6.1 Python operators that differ from the standard C++ and Java syntax

6.2.1 Strings

Strings in Python can be denoted using single or double quotes. Strings that span several lines can be denoted using three consecutive single or double quotes. Adding the prefix `r` or `R` before a string literal instructs Python to avoid substituting escape characters in the string (such as the newline escape character backslash followed by n).

```
s = ''' this is a string
that spans
multiple lines'''
print('\n line')  # newline followed by "line"
print(r'\n line') # avoid escape character substitution
##
##  line
## \n line
```

Below are a few string properties in Python:

- Python strings are immutable and thus cannot be modified after they are created. It is possible to create a new string that is a modification of an existing string.
- Specific characters in a string can be referred to using the square bracket index notation (indices in Python start from 0).
- Many Python objects can be converted to a string using the `str` function.
- Two python strings can be concatenated using the + operator.
- The string object includes many member functions that are useful for manipulating strings. A few examples are listed below.

```
s = "Python"
print(s[2])  # print third character of string s
print(s + str(123))  # concatenate s with "123"
print(s.replace('P', 'p'))  # replace 'P' in s with 'p'
## t
## Python123
## python
```

- It is convenient to form strings by embedding numeric variables into a string template. A template is a string that contains replacement fields delimited by { }. When the template is formatted using the `format` method called with a list of variables, said variables are embedded in the corresponding markers:

```
import math

print('Formatting in {} is simple and powerful'.format('Python'))
print('Refer to {1} by {0}'.format('index', 'fields'))
print('Use {name} too'.format(name='keyword arguments'))
print('Rounding: pi = {:.3}'.format(math.pi))
print('Attributes: pi = {0.pi:.3}'.format(math))
print('And {0[1][0]} more!'.format(['so', ['much', ['more']]]))
## Formatting in Python is simple and powerful
```

```
## Refer to fields by index
## Use keyword arguments too
## Rounding: pi = 3.14
## Attributes: pi = 3.14
## And much more!
```

In Python 3.5.x, the `format` method is the preferred way to format strings; older alternatives include using the `%` operator as follows (this technique is similar to the `sprintf` function in C):

```
import math

value = '%'
print('Formatting using the %s operator is error-prone.' % value)
print('Values must be specified %s %s.' % ('in', 'order'))
value = ('a', 'tuple')
# Wrap value as (value,); otherwise we get a TypeError
print('If a value is %s, things get complicated!' % (value,))
value = {'data': 'dictionary'}
print('Using a %(data)s key works though.' % value)
print('Rounding: pi = %.3f and e = %.3f.' % (math.pi, math.e))
print('%% has to be escaped when formatting.' % ())
## Formatting using the % operator is error-prone.
## Values must be specified in order.
## If a value is ('a', 'tuple'), things get complicated!
## Using a dictionary key works though.
## Rounding: pi = 3.142 and e = 2.718.
## % has to be escaped when formatting.
```

The way illustrated above is still supported—yet discouraged—in Python 3.x for backward compatibility. Because formatting using the `%` operator is error-prone, a simpler way was introduced in Python 2.4 using the `string.Template` class:

```
from math import pi
from string import Template as T # aliased for brevity

print(T('$id work').substitute(id='Keyword arguments'))
print(T('$id works too').substitute({'id': 'A dictionary'}))
print(T('Note the ${id}').substitute(id='differences'))
print(T('pi = ${id}').substitute(id=pi))
print(T('$$ has to be escaped').substitute())
## Keyword arguments work
## A dictionary works too
## Note the differences
## pi = 3.141592653589793
## $ has to be escaped
```

Python 3.6 brings about yet another mechanism called Literal String Interpolation,[3] which introduces the notion of f-strings (short for formatted strings). Here's an example:

[3] https://www.python.org/dev/peps/pep-0498/.

```
from math import pi

value = 'string interpolation'
print(f'The use of {value} is awesome!') # more readable
print(f"Another PI example: pi = {pi:.3f}")
## The use of string interpolation is awesome!
## Another PI example: pi = 3.142
```

Having many ways to accomplish the same task is confusing, but the reality of Python supporting these different mechanisms forces developers to be familiar with them to be able to read others' code; that's why we recommend further reading about this topic in the Python Language Reference.[4]

6.2.2 *Duck Typing*

In Java and C++, the suitability of an operation for an object is determined by the object's type. For example, a function `foo(x)` will be able to operate on its argument based on whether `x` is of a suitable type specified in the function definition. A C++ function `int foo(float x)` will only work on a `float` argument.[5]

In duck typing, the constraints are not specified in the code in the same sense. Rather, the interpreter tries to execute the code and if it cannot, a runtime error will occur. For example, a function will take an argument of any type and tries to execute the code. Duck typing is called duck typing after the phrase "if it walks like a duck and quacks like a duck, it is a duck."

Below is an example[6] where the function `foo` accepts any objects that can execute the two functions `quack` and `walk` (rather than be constrained to a specific class). As a result, we can call it with both a `Duck` object and a `Fox` object that imitates a `Duck` in that it can execute the functions `quack` and `walk`:

```
class Duck:
  def quack(self):
    print("Quack")
  def walk(self):
    print("Shuffle")

class Fox:
  def quack(self):
    print("Quackkkkk")
  def walk(self):
    print("Shufffle")
```

[4] https://docs.python.org/3/library/string.html#custom-string-formatting.

[5] A conversion may happen to convert a parameter into the expected type. Parameters of a derived class (satisfying the is-a relationship) are also accepted.

[6] The duck typing example is inspired by the example found in the Wikipedia page for Duck Typing.

```
    def kill(self):
      print("Yum!")

def foo(x):
  x.quack()
  x.walk()

donald = Duck()
swiper = Fox()
foo(donald)
foo(swiper)
## Quack
## Shuffle
## Quackkkkk
## Shufffffle
```

There is some similarity between duck typing and interfaces in Java. Duck typing is less constrained since it's not required to specify that a particular interface is being implemented. In addition duck typing can implement only a part of an interface, and if only that part is called, then the program will proceed to execution.

A big disadvantage of duck typing is that it delays error discovery from compile-time to runtime. Runtime errors are generally considered undesirable as they are much harder to prevent and debug.

6.3 Compound Data Types

Python's most popular compound data types are tuples, lists, dictionaries, and sets. We describe these types below. Section 6.9 describes additional compound data types.

6.3.1 Tuples

A tuple is an immutable sequence of objects of potentially different types. Each object can be referred to using the square bracket index notation. Tuples are defined using parenthesis or the built-in function `tuple`:

```
a = (1, 2) # tuple of two integers
b = 1, 2 # alternative way to define a tuple
a[0] = 3 # error: modifying an immutable tuple after its
           creation


a = (1, 2)  # tuple of two integers
b = (1, "one")  # tuple of an integer and a string
c = (a, b)  # tuple containing two tuples
```

```
print(b[1])  # prints the second element of b
print(c[0])  # prints the first element of c
## one
## (1, 2)
```

It's possible to unpack a tuple into individual variables as follows:

```
t = (1, 2, 3)  # tuple of three integers
(a, b, c) = t  # copy first element of t to a, second to b, etc.
a, b, c = t  # alternative syntax to accomplish the same thing
print(a)
print(b)
print(c)
## 1
## 2
## 3
```

Nested tuples can also be unpacked as follows:

```
t = (1, (2, 3))  # tuple containing an integer and a tuple
# copy the first element of t to a, second to (b, c)
(a, (b, c)) = t
a, (b, c) = t
print(a)
print(b)
print(c)
## 1
## 2
## 3
```

Note that the number of values to unpack has to match the number of values the tuple contains; otherwise, an error occurs:

```
a, b, c = t # error: not enough values to unpack
```

6.3.2 Lists

Lists are similar to tuples, but they're mutable, and they're defined using square brackets (or the built-in function `list`):

```
a = [1, 2, 3]  # list containing three integers
b = [a, "hello"]  # list containing a list and a string
b.append(4)  # modify existing list by appending an element
print(a)
print(b)
## [1, 2, 3]
## [[1, 2, 3], 'hello', 4]
```

Existing list elements can be removed and new elements can be inserted at arbitrary positions, but these operations are often inefficient for extremely long lists.

Lists can also be easily sorted and combined. The method `sort()` sorts a list in-place (changes the object calling the method) and the method `sorted(x)` returns a new list that is a sorted version of the passed parameter:

```
a = [1, 2, 3]
a.insert(1, 100)  # insert 100 before index 1
print(a)
a.remove(2)  # remove first occurrence of 2
print(a)
a.sort()  # sort list
print(a)
a.extend([20, 21, 23])  # extend list with an additional list
print(a)
b = sorted(a)
print(b)
## [1, 100, 2, 3]
## [1, 100, 3]
## [1, 3, 100]
## [1, 3, 100, 20, 21, 23]
## [1, 3, 20, 21, 23, 100]
```

The statement `del` deletes a list element referred to by index and the function `len` returns the length of the list. The methods `append`, `insert`, and `remove` allow the list object to append an element, insert an element, and remove an element (referred to by value)—respectively.

```
a = ['hello', ', ', 'world']
print(len(a))
del a[1]
print(len(a))
print(a)
## 3
## 2
## ['hello', 'world']
```

The + operator can be used to create a new list that is the concatenation of the list operands (evaluated left to right):

```
print([1, 2, 3] + [4, 5] + [6])
## [1, 2, 3, 4, 5, 6]
```

6.3.3 Ranges

The built-in function `range` is often used to access a list of evenly spaced integers. The first argument of the `range` function denotes the starting value; the second argument denotes the exclusive end position (the integer following the last selected index, which is the sequence's length by default); and the third argument denotes the step size (defaults to 1).

```
print(list(range(0, 10, 1)))
print(tuple(range(0, 10))) # same as tuple(range(0, 10, 1))
print(tuple(range(10))) # same as tuple(range(0, 10))
print(tuple(range(0, 10, 2))) # step size of 2
## [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
## (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
## (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
## (0, 2, 4, 6, 8)
```

Before Python 3.x, the function `range` was used to create a list of evenly spaced integers, which can be inefficient when looping over a large range of integers as the entire list needed to be generated first. In Python 3.x, the function `range` generates a `range` object that can be enumerated or converted to a list or a tuple.

6.3.4 Slicing

Slicing refers to accessing a slice of a list, tuple, or string using a range of integers representing the corresponding indices; said range is represented using the notation `start:end:step` where `start` denotes the first selected position (default is 0), `end` denotes the exclusive end index, and `step` denotes a potential skip value (default is 1). Some—or all—of `start`, `end`, or `step` may be omitted (if `step` is omitted, the second colon may be left out):

```
a = list(range(10))
print(a[:])  # starting index (0) to end index (9)
print(a[3:])  # index 3 to end (9)
print(a[:3])  # index 0 to 2
print(a[0:10:2])  # index 0 to 9, skipping every other element
## [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
## [3, 4, 5, 6, 7, 8, 9]
## [0, 1, 2]
## [0, 2, 4, 6, 8]
```

Negative values of `step` represent progression from the end towards the beginning and negative values of `start`, `end` represent offset from the end (in the case of `start`) or the beginning (in the case of `end`):

```
a = list(range(10))
print(a[::-1])  # start index (0) to end (9) - backwards
print(a[-3:])  # index 3 from the end (7) to the end (9)
## [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
## [7, 8, 9]
```

The `zip` function can pair up elements from two lists or tuples (like the teeth of a zipper) to create a zip object (a chain of pairs to enumerate):

```
a = [1, 2, 3]
b = ["one", "two", "three"]
print(list(zip(a, b)))
## [(1, 'one'), (2, 'two'), (3, 'three')]
```

It can also be used with zero, one, or many sequences to zip:

```
a = [1, 2, 3]
b = ["one", "two", "three"]
c = ["uno", "dos", "tres"]
print(list(zip()))
print(list(zip(a)))
print(list(zip(a, b, c)))
## []
## [(1,), (2,), (3,)]
## [(1, 'one', 'uno'), (2, 'two', 'dos'), (3, 'three', 'tres')]
```

6.3.5 Sets

A set is an unordered collection of unique immutable objects (yet the set itself is mutable). A set can be defined using curly braces, with objects separated by commas:

```
# duplicity in sets is ignored
s = {1, 2, 3, 2}
print(s)
## {1, 2, 3}
```

The following code demonstrates the set operations of union $A \cup B$, intersection $A \cap B$, and set-difference $A \setminus B$, and the set relations $A \cap B = \emptyset$ (disjoint), $A \subset B$ (subset), and $A \supset B$ (superset):

```
a = set([1, 2, 3]) # make a set from a list
b = set((2, 3, 4)) # make a set from a tuple
print(a | b)  # union
print(a & b)  # intersection
print(a - b)  # set-difference
print(a.isdisjoint(b), a.issubset(b), a.issuperset(b))
## {1, 2, 3, 4}
## {2, 3}
## {1}
## False False False
```

Besides ensuring the uniqueness of values within the same set, it's more efficient to use a set when looking up a value or checking for membership.

6.3.6 Dictionaries

A dictionary is an unordered, mutable compound data type representing a set of (key, value) pairs, where each key may appear at most one time. Each (key, value) pair may be thought of as a mapping from a key to a value, and consequentially the entire dictionary can be thought of as a function mapping keys to values. Keys of a dictionary must be immutable.

Dictionaries are denoted using curly braces with each key-value pair following the format `key: value`, and multiple key-value pairs are separated by commas. Dictionaries are mutable and thus additional key-value pairs may be added or removed after the dictionary is created.

The example below creates a dictionary with two key-value pairs and then prints all keys and all values:

```
# create 3 dicts with three key-value pairs:
# ('Na', 11), ('Mg', 12), and ('Al', 13)
d = dict((('Na', 11), ('Mg', 12), ('Al', 13)))
e = {'Na': 11, 'Mg': 12, 'Al': 13}
f = dict(Na=11, Mg=12, Al=13)
print(d == e == f)  # check equivalence
print(d.keys())  # print all keys (unordered)
print(d.values())  # print all respective values
## True
## dict_keys(['Na', 'Mg', 'Al'])
## dict_values([11, 12, 13])
```

The primary use of a dictionary is to efficiently retrieve a value associated with a specific key. The example below does that, and it then checks whether a specific key exists in the dictionary:

```
# create a dict with two key-value pairs
d = {'Na': 11, 'Mg': 12, 'Al': 13}
print(d['Na'])  # retrieve value corresponding to key 'Na'
print('Li' in d)  # check if 'Li' is a key in dict d
## 11
## False
```

In the example below we create a dictionary object, and then add a new key-value pair and remove an existing key-value pair. We then print the dictionary as a sequence of key-value pairs:

```
# create a dictionary with two key-value pairs
d = {'Na': 11, 'Mg': 12, 'Al': 13}
d['Li'] = 3 # add the key value pair ('Li', 3)
del d['Na']  # remove key-value pair corresponding to key 'Na'
print(d)  # print dictionary
## {'Mg': 12, 'Al': 13, 'Li': 3}
```

Since dictionaries are unordered, iterating over the key-value pairs is not guaranteed to follow the order of insertion or the alphabetic order over the keys.

It's easy to create dictionaries from two lists, one containing keys and the other containing values using the `zip` function (while creates a chain of tuples from multiple lists):

```
keys = ['Na', 'Mg', 'Al']
values = [11, 12, 13]
d = dict(zip(keys, values))
print(d)
## {'Na': 11, 'Mg': 12, 'Al': 13}
```

The following example iterates over all items in a dictionary (we will see the details of the `for` control flow construct later in this chapter):

```
d = dict(Na=11, Mg=12, Al=13)
# iterate over all items in d
for key, value in d.items():
  print(key, value)
## Na 11
## Mg 12
## Al 13
```

A common anti-pattern when reading dictionary values is a double lookup; here's an example of what not to do when iterating over all items in a dictionary:

```
d = dict(Na=11, Mg=12, Al=13)
for key in d: # similar to for key in iter(d):
  print(key, d[key])
## Na 11
## Mg 12
## Al 13
```

Note that `d[key]` performs an unnecessary lookup for each key; we could have iterated over all dictionary items once instead. Another example of a double lookup is checking for the existence of the key in the dictionary and then looking up its value (we will see the details of the `if-else` conditional construct later in this chapter):

```
counts = dict(apples=3, oranges=5, bananas=1)
key = 'apples'
if key in counts: # first lookup
  print(counts[key]) # second lookup
else:
  print(0)
## 3
```

Alternatively, the `dict` class provides a `get(key, default)` method that checks if the key is in the dictionary, in which case it returns its respective value; otherwise, it returns the specified default value:

```
counts = dict(apples=3, oranges=5, bananas=1)
key = 'apples'
print(counts.get(key, 0)) # single lookup
## 3
```

For readers who are familiar with machine learning, the aforementioned example demonstrates the use of a `dict` object as a sparse feature vector where features are referenced by name, and zero-value features are not stored, which makes the `dict` object an efficient way to represent a feature vector.

The previous example also demonstrates the statistical concept of frequency count. Since it's a common use-case of a `dict`, Python provides a `dict` subclass called `Counter`, which is mainly used for frequency-counting of its keys. In order to use the `Counter` class, we need to import it first:

```
from collections import Counter
counts = Counter(apples=3, oranges=5, bananas=1)
print(counts)
## Counter({'oranges': 5, 'apples': 3, 'bananas': 1})
```

`Counter` objects are useful for text processing, here's an example of unigram[7] frequency counting using a `Counter`:

```
from collections import Counter
# split the text into a list of words then count them
counts = Counter('to be or not to be'.split())
print(counts)
## Counter({'to': 2, 'be': 2, 'or': 1, 'not': 1})
```

`Counter` objects can represent sparse vectors, as the count (value) of a nonexistent key is 0:

```
from collections import Counter
# split the text into a list of words then count them
counts = Counter('to be or not to be'.split())
print(counts['question'])
## 0
```

More importantly, the `Counter` class provides rich functionality that's very useful for data queries:

```
from collections import Counter
# split the text into a list of words then count them
counts = Counter('to be or not to be'.split())
# repeat the keys as many times as their respective counts
print(tuple(counts.elements())) # unordered
# get the n most common elements
tuple(counts.most_common(2)) # unordered
## ('to', 'to', 'be', 'be', 'or', 'not')
```

The example below adds and subtracts two counter objects and prints the resultant dictionaries:

[7] https://en.wikipedia.org/wiki/N-gram.

```
from collections import Counter

counts = Counter('to be or not to be'.split())
other = Counter('love or war'.split())
# accumulate two counters together
print(dict(counts + other))
# subtract counts (and only keep positive counts)
print(dict(counts - other))
## {'to': 2, 'be': 2, 'or': 2, 'not': 1, 'love': 1, 'war': 1}
## {'to': 2, 'be': 2, 'not': 1}
```

An alternative for the `dict` class to use as a sparse vector is `defaultdict`, which provides default values for nonexistent keys using a factory function that creates said default value:

```
from collections import defaultdict

# the default value is what int() returns: 0
d = defaultdict(int)
print(d[13])
## 0
```

The `defaultdict` class is useful for grouping elements by their respective keys; a common use-case of this pattern is indexing documents to speed up search; a simple way to index by words is to use each (unique) word as a key in a dictionary and group the set of documents in which said key (word) appeared as its respective value. The example below uses the `set()` function to create a new set when a new key is encountered, and adds the current value being processed to the newly created set; if the key already exists, the respective set is retrieved and the current value being processed is added to the retrieved set:

```
from collections import defaultdict

# documents to index: a list of (title, text) tuples
books = [
  ('hamlet', 'to be or not to be'),
  ('twelfth night', 'if music be the food of love'),
]
# the default value is what set() returns: {}
index = defaultdict(set)
for title, text in books:
  for word in text.split():
    # get or create the respective set
    # then add the title to it
    index[word].add(title)

# pretty-printer (recommended for nested data)
from pprint import pprint
pprint(index)
# query for documents that have the word 'music'
print(index['music'])
## defaultdict(<class 'set'>,
```

```
##              {'be': {'twelfth night', 'hamlet'},
##               'food': {'twelfth night'},
##               'if': {'twelfth night'},
##               'love': {'twelfth night'},
##               'music': {'twelfth night'},
##               'not': {'hamlet'},
##               'of': {'twelfth night'},
##               'or': {'hamlet'},
##               'the': {'twelfth night'},
##               'to': {'hamlet'}})
## {'twelfth night'}
```

Another use-case of dictionaries is building a trie (also known as a prefix tree), which is a data structure that can be used to speed up text search. In the example below, we explore the use of a trie to perform word autocomplete, which is a common feature in most web search engines today. The example may look a bit advanced, but it should be easy to read now, and easy to implement from scratch after finishing this chapter. It's also worth noting that it's not the most efficient way to implement a trie, but it's a good example of how to use a `defaultdict`:

```
from collections import defaultdict

words = { 'apple', 'ban', 'banana', 'bar', 'buzz' }
# a factory function/lambda expression that returns
# a defaultdict which uses it as a factory function
factory = lambda: defaultdict(factory)
trie = defaultdict(factory)
DELIMITER = '--END--'
LEAF = dict()
for word in words:
  level = trie
  for letter in word:
    level = level[letter]
  level[DELIMITER] = LEAF

def autocomplete(root, prefix, suffix=''):
  if DELIMITER in root:
    print(prefix + suffix)
  for key, value in root.items():
    autocomplete(value, prefix, suffix + key)

# query for words that start with 'ba'
autocomplete(trie['b']['a'], 'ba')
## bar
## ban
## banana
```

Hashable Objects

In addition to the requirement that each key may appear at most once, keys are required to be hashable (the `hash` function must be able to take the key as an

argument) and thus have to be immutable. Examples of hashable immutable objects are integers, floats, strings, and tuples. Lists and dictionaries are mutable and therefore cannot serve as keys in a dictionary.

The code example below prints the hash values of a string and a tuple of integers.

```
print(hash("abc"))
print(hash((1, 2, 3)))
## -4156837777623529643
## 2528502973977326415
```

6.4 Comprehensions

Processing data usually entails many steps of transformation and massaging; it typically involves iterating over data records and performing some sort of a mapping operation to create new data, and so on, until the desired data format is obtained. In most programming languages, this usually means a series of loops, and many lines of verbose syntax to accomplish such tasks. Python is different; for many constructs in Python, there's a dull way of doing things, and then there's the Pythonic way. One of the most commonly used Pythonic constructs is comprehensions, which we discuss below in detail.

6.4.1 List Comprehensions

Take, for example, a simple transformation: given a list of words, return a list that contains the most common letter (mode) of each word and its count. The mundane way of performing such transformation looks like the following:

```
from collections import Counter

words = ['apple', 'banana', 'carrot']
modes = []
for word in words:
  counter = Counter(word)
  # most_common(n) returns a list and the *
  # operator expands it before calling append(x)
  modes.append(*counter.most_common(1))
print(modes)
## [('p', 2), ('a', 3), ('r', 2)]
```

Now let's examine the Pythonic way of creating a list, which removes most of the superfluous syntax:

```
from collections import Counter

words = ['apple', 'banana', 'carrot']
print([Counter(w).most_common(1)[0] for w in words])
## [('p', 2), ('a', 3), ('r', 2)]
```

Note that comprehensions don't support iterable unpacking (using the * operator to unpack the list parameter), so we had to extract the singleton element using the [] operator instead. The syntactic sugar of comprehensions allows for creating a list in a concise fashion. At a second glance, one can see the various constructs in the first example were simply rearranged to be more compact in the latter example.

Comprehensions also support selectivity (selecting a sublist of the input list to transform based on a boolean criterion):

```
from collections import Counter

words = ['apple', 'banana', 'carrot']
print([Counter(w).most_common(1)[0] \
       for w in words if len(w) > 5])
## [('a', 3), ('r', 2)]
```

After the `for` clause in a list comprehension, any number of `if` and additional `for` clauses can follow. Combining `for` clauses together mimics the behavior of nested loops. In the example below, given two lists, we calculate their Cartesian product where neither component of the resultant coordinates is zero:

```
x = (0, 1, 2, 7)
y = (9, 0, 5)
print([(a, b) for a in x for b in y if a != 0 and b != 0])
# alternatively
print([(a, b) for a in x if a != 0 for b in y if b != 0])
## [(1, 9), (1, 5), (2, 9), (2, 5), (7, 9), (7, 5)]
## [(1, 9), (1, 5), (2, 9), (2, 5), (7, 9), (7, 5)]
```

6.4.2 Set Comprehensions

Set comprehensions work almost exactly the same way as list comprehensions, except that the result is a set (where repeated elements are not allowed) and it's expressed using curly braces:

```
champions = [
  (2014, 'San Antonio Spurs'),
  (2015, 'Golden State Warriors'),
  (2016, 'The Cleveland Cavaliers'),
  (2017, 'Golden State Warriors'),
]
for team in {c[1] for c in champions}: # unordered
    print(team)
## The Cleveland Cavaliers
## Golden State Warriors
## San Antonio Spurs
```

6.4.3 Dictionary Comprehensions

Similarly, dictionary comprehensions work almost exactly the same way as set comprehensions, except that the result is a dictionary (where a unique key is mapped to a value):

```
from pprint import pprint

champions = [
  (2014, 'San Antonio Spurs'),
  (2015, 'Golden State Warriors'),
  (2016, 'The Cleveland Cavaliers'),
  (2017, 'Golden State Warriors'),
]
pprint({c[0]: c[1] for c in champions})
## {2014: 'San Antonio Spurs',
##  2015: 'Golden State Warriors',
##  2016: 'The Cleveland Cavaliers',
##  2017: 'Golden State Warriors'}
```

Comprehensions handle repetitions the same way in the case of sets (repeated elements) and dictionaries (repeated keys), by treating them as if the input elements were added sequentially. Note in the example below that two tuples share the same key and the respective value of the latter overwrites the former value associated with said key:

```
from pprint import pprint

champions = [
  (2014, 'San Antonio Spurs'),
  (2015, 'Golden State Warriors'),
  (2016, 'The Cleveland Cavaliers'),
  (2017, 'Golden State Warriors'),  # this one wins
]
pprint({c[1]: c[0] for c in champions})
## {'Golden State Warriors': 2017,
##  'San Antonio Spurs': 2014,
##  'The Cleveland Cavaliers': 2016}
```

6.4.4 Nested Comprehensions

Comprehensions can be nested; the expression that evaluates to an element in the result can be a comprehension; this pattern is commonly used to define—and initialize—multidimensional compound data structures:

```
# create a 3x5 matrix of 1's
matrix = [[1 for j in range(5)] for i in range(3)]
```

```
print(matrix)
## [[1, 1, 1, 1, 1], [1, 1, 1, 1, 1], [1, 1, 1, 1, 1]]
```

6.5 Control Flow

Python uses white spaces (tabs or spaces) to structure code instead of braces as in Java, C++, and R. A colon denotes the start of an indented code block, after which all lines must be indented by the same amount. A return to the previous indentation signals that the block corresponding to the indented block is ending and a new block begins.

6.5.1 If-Else

If-else blocks use the following structure. Note the lack of curly braces and that the indentation indicates where each `if`, `elif` (else-if), or `else` clause ends:

```
x = 3
if x > 0:
  # if block
  print("x is positive")
  sign = 1
elif x == 0:
  # elif (else-if) block
  print("x equals zero")
  sign = 0
else:
  # else block
  print("x is negative")
  sign = -1
print(sign)  # new block
## x is positive
## 1
```

The following expressions are evaluated to false (in that they would fail an `if` check) in Python: `False`, `None`, `0`, `''`, `()`, `[]`, and `{}`

The opposite is true: expressions like `True`, `1`, etc. pass an `if` check in Python.

6.5.2 For-Loops

Python for-loops typically take the following form, where the code iterates over the elements of an iterable object (e.g., a tuple, list, dictionary, etc.):

```
a = [1, 2, 3, -4]
for x in a:
  # start of the for block
  print(x)

print(x)  # new block, but x is still in scope!
## 1
## 2
## 3
## -4
## -4
```

Note that as the example above indicates the index variable in the for-loop remains in scope after the loop is concluded.

We can keep track of the iteration number using the `enumerate` function as follows:

```
a = [1, 2, 3, -4]
for i, x in enumerate(a):
  print('a at', i, '=', x)
## a at 0 = 1
## a at 1 = 2
## a at 2 = 3
## a at 3 = -4
```

The example below shows a combination of an outer for-loop block and an inner if-else clause; it computes the sum of absolute values of the elements of a list:

```
a = [1, 2, 3, -4]
abs_sum = 0
for x in a:
  if x > 0:
    abs_sum += x
  else:
    abs_sum += -x

print(abs_sum)
## 10
```

Using the Pythonic way, we can rewrite the example above to be more succinct:

```
a = [1, 2, 3, -4]
print(sum(abs(x) for x in a))
## 10
```

6.5.3 Else as a Completion Clause

The use of `else` as a completion clause (e.g., `for-else`) is one of the least intuitive constructs one may see in Python code. The `else` clause in this context is

executed after the loop terminates without being broken (using a `break` statement). The example below searches a string for the first occurrence of a digit character and prints it if found; otherwise, it prints a message indicating that no digit was found:

```
s = 'Jane is 42 years old'
for c in s:
  if c.isdigit():
    print('Found:', c)
    break
else:
  print('No digit was found!')
## Found: 4
```

The following is an example of when the `else` clause is executed:

```
budget = 13
costs = [5, 3, 2]
for i, cost in enumerate(costs):
  print('Item', i, 'for', cost, 'unit(s) of cost: ', end='')
  budget -= cost
  if budget > 0:
    print('acquired')
  else:
    print('insufficient funds')
    break
else:
  print('Remaining budget:', budget)
## Item 0 for 5 unit(s) of cost: acquired
## Item 1 for 3 unit(s) of cost: acquired
## Item 2 for 2 unit(s) of cost: acquired
## Remaining budget: 3
```

6.5.4 The Empty Statement

The empty statement, `pass`, is another example of Python syntax that one might find confusing at first glance; it does nothing. So why would you need a statement that does nothing? One reason is prototyping code that needs to be valid—syntactically. In other languages where brackets can be used to define blocks, an empty block is simply defined as { }; to achieve that in Python, we need to use the `pass` statement:

```
budget = 13
costs = [5, 3, 2]
for i, cost in enumerate(costs):
  pass # TODO: put off until tomorrow
else:
  print('Remaining budget:', budget)
## Remaining budget: 13
```

Test-driven development (TDD) is another use case for the empty statement; developers write unit tests first, then write the least amount of code required to make said tests pass (not to be confused with passing the control flow to the next statement). In which case, the empty statement may come in handy to create placeholders for blocks of logic that will be added later. For more details about unit testing, see Sect. 11.5.

6.6 Functions

Functions in Python are similar to functions in C++, Java, and R. When a function is called, execution transfers to the function code; and after the function terminates, execution returns to the next line after the function call.[8] As in the case of `if` and `for` blocks, the block of the function definition code is denoted using a colon and indentation in the following lines.

The calling environment may pass arguments to the function, which are tied to the function parameters and may be used in the function code. The arguments may be tied to the parameters by their respective positions in the sequence of passed arguments, or by using their respective names. Parameters can have default values, and that allows them to be omitted from the function call; in which case, the function receives the default values instead. The function terminates when encountering a `return` statement or when the end of the block is reached (in which case `None` is returned).

The code below defines a function called `scale`, which scales each element of the specified iterable (the first parameter called `vector`) by a scaling factor specified by the second argument: `factor`. After the function definition, the code below calls the function multiple times with different arguments:

```
def scale(vector, factor):
  return [x * factor for x in vector]

print(scale([1, 2, 3], 2))
print(scale({1, 2, 3}, 2)) # unordered
print(scale(factor=2, vector=(1, 2, 3)))
# the * operator works with lists and tuples as well,
# but it has different semantics:
print(scale([(1, 2, 3)], 2))
## [2, 4, 6]
## [2, 4, 6]
## [2, 4, 6]
## [(1, 2, 3, 1, 2, 3)]
```

[8] In this chapter, we discuss control flow in a sequential-execution (single-threaded) environment; for details on parallel computing, see Chap. 10.

Note that because of duck typing, the `scale` function will happily run with any element type that supports the `*` operator. The last call in the example above uses a tuple as an element type for the specified `vector` parameter; and multiplying a sequence by an integer `n` results in a new sequence that contains the elements of the input sequence repeated `n` times. In the example below, we add a new parameter, `recursively`, which instructs the function to scale elements of the specified `vector` recursively:

```
from collections import Sequence

def scale(vector, factor, recursively=False):
  if recursively:
      # call scale recursively for sequence elements
      return [scale(x, factor, True) if isinstance(x, Sequence)
              else (x * factor) for x in vector]
  else:
    return [x * factor for x in vector]

print(scale([(1, 2, 3)], 2))  # recursively is False by default
print(scale([(1, 2, 3)], 2, recursively=True))
## [(1, 2, 3, 1, 2, 3)]
## [[2, 4, 6]]
```

In the last call above, the value of the last parameter, `recursively`, is explicitly specified; while in other calls, it's implied (using its default value of `False`). Even though Python doesn't require named parameters to be listed in the same order as that of the function's definition, we recommend following the same order (or as close as possible when some parameters are omitted to use their default values) for better readability. We also recommend against the use of default values to ensure expressiveness of intentions simply by reading the function call (without having to look up its definition); for more details about style guidelines, see Chap. 15.

Functions can be nested, so that the outer function can call the inner function, but the inner function is inaccessible otherwise. This is useful when defining a helper function (the inner one) that only gets called by the outer function. First, let's examine this familiar pattern of using a helper method:

```
from collections import Sequence

def scale_recursively(vector, factor):
  return [scale_element(x, factor) for x in vector]

def scale_element(element, factor):
  if isinstance(element, Sequence):
    return scale_recursively(element, factor)
  else:
    return element * factor

print(scale_recursively([1, [2, 3]], 2))
## [2, [4, 6]]
```

Now, let's rewrite the above example using nested functions:

```
from collections import Sequence

def scale_recursively(vector, factor):
  def scale_element(element):
    if isinstance(element, Sequence):
      return scale_recursively(element, factor)
    else:
      return element * factor
  return [scale_element(x) for x in vector]

print(scale_recursively([1, [2, 3]], 2))
## [2, [4, 6]]
```

Note how more compact the latter example is, and how the `factor` parameter was used inside in the inner function. Also, the code is better encapsulated since other code entities don't know about `scale_element`; it's hidden as an implementation detail of `scale_recursively`. For more details about encapsulation, see Sect. 4.10.3. There's a catch though: the inner function is redefined every single time the outer function is called, because it's local to the outer function. That said, it's commonly used in a function calling technique called currying, which is used to transform a polyadic function call (which takes multiple arguments) into multiple monadic (single-parameter) function calls. Here's an example:

```
def create_line_printer(prefix):
  def print_suffix(suffix):
    print(prefix, suffix)
  return print_suffix

write = create_line_printer('[currying example]')
write('this is useful for behavior reuse')
write('and for testing multiple behaviors')
write = create_line_printer('[a prefix to test]')
write('the call of write(x) did not change')
write('even though the prefix had changed')
## [currying example] this is useful for behavior reuse
## [currying example] and for testing multiple behaviors
## [a prefix to test] the call of write(x) did not change
## [a prefix to test] even though the prefix had changed
```

Note that functions can be passed as arguments and returned as return values. In the previous example, the `print_suffix(suffix)` function object was returned to the callers of `create_line_printer(prefix)`; here's an example of using a function as a parameter:

```
def sort_by_second(pairs):
  def get_second(pair):
    return pair[1]
  return sorted(pairs, key=get_second)

pairs = ((1, 3), (4, 2))
```

```
print(sort_by_second(pairs))
## [(4, 2), (1, 3)]
```

In the above example, the `get_second` function is used as a parameter to the `sorted` function, which uses it to extract the value used for comparison while sorting (the sort key). The next example is equivalent, except that it creates a callable object (you can think of it as a function) that is the result of calling `operator.itemgetter(1)`, which is used to extract the second item of the sequence element (whose index is 1):

```
pairs = ((1, 3), (4, 2))
from operator import itemgetter
print(sorted(pairs, key=itemgetter(1)))
## [(4, 2), (1, 3)]
```

As in other languages, variables (names) defined inside functions are local to the functions and thus not accessible outside of the function. An exception to this rule is when a `global` or `nonlocal` statement is used to alter the scope of a local names. The `nonlocal` statement was added in Python 3.x[9] to allow a variable (or a name binding) to be bound to one with the same name in the nearest enclosing scope, but not the global scope. Let's examine an example of nested functions that keep a running sum:

```
def create_accumulator(seed=0):
  tally = seed
  def accumulate(x):
    tally += x # local to accumulate(x)
    return tally
  return accumulate
```

An attempt to create an accumulator and calling it will result in the following error: `"local variable 'tally' referenced before assignment"` since the `tally` name is local to the `accumulate(x)` function. To reference the nonlocal `tally` name, we need to add a simple statement `nonlocal tally`. Here's the code after the modification, and a few calls to show the intended behavior:

```
def create_accumulator(seed=0):
  tally = seed
  def accumulate(x):
    nonlocal tally # the fix
    tally += x
    return tally
  return accumulate

accumulate = create_accumulator()
print(accumulate(2))
```

[9] http://legacy.python.org/dev/peps/pep-3104.

```
print(accumulate(3))
print(accumulate(5))
## 2
## 5
## 10
```

Similarly, the `global` statement is used to allow a local name to refer to a global identifier, which is required in case a value assignment is desired. Here's an example equivalent to the one listed above:

```
tally = 0
def accumulate(x):
  global tally
  tally += x
  return tally

print(accumulate(2))
print(accumulate(3))
print(accumulate(5))
## 2
## 5
## 10
```

A function can return multiple objects by returning a sequence (e.g., a tuple) containing said objects; the calling environment can then easily unpack the returned sequence:

```
def foo(x, y):
  return x + y, x - y

a, b = foo(1, 2)  # unpack the returned tuple into a and b
print(a)
print(b)
## 3
## -1
```

An alternative way to return multiple objects is to return a dictionary containing the returned objects as values and their descriptions or names as the corresponding keys.

Python supports the concept of variadic functions, which can take a variable number of parameters packed into a tuple; this construct is similar to the `...` syntax in C++ and Java (varargs). Python uses the `*` operator to indicate the varargs parameter:

```
# the syntax *lines makes this function variadic
def print_lines(prefix, *lines):
  for line in lines:
    print(prefix, line)

# call the function with varargs
print_lines('[varargs example]', 'hello', 'world')
## [varargs example] hello
```

```
## [varargs example] world
```

The above example is equivalent to the following code:

```
# no * operator, just an ordinary tuple
def print_lines(prefix, lines):
  for line in lines:
    print(prefix, line)

# call the function with a tuple parameter
print_lines('[tuple example]', ('hello', 'world'))
## [tuple example] hello
## [tuple example] world
```

The other way around is also true: you may call a function that accepts `n` parameters with an expanded sequence that has `n` elements; here's an example that demonstrates the use of the `*` operator to unpack the argument list:

```
def power(base, exponent):
  return base ** exponent

data = [2, 3] # or (2, 3)
print(power(*data))
## 8
```

This pattern is convenient when the data is already formatted as a sequence, so there's no need to read each element and pass it as a parameter to the function being called. In case the data was formatted as a dictionary, the `**` operator can be used to expand the parameters used in a function call:

```
def power(base, exponent):
  return base ** exponent

data = {'base': 2, 'exponent': 3}
print(power(**data))
## 8
```

The `**` operator can also be used in the function definition to indicate that it packs a list of named parameters (names and their respective values) into a dictionary inside the function's body:

```
def print_latest_championship(**data):
  for team, year in data.items():
    print(team + ':', year) # dictionaries are unordered

print_latest_championship(cavaliers=2016, warriors=2017)
## cavaliers: 2016
## warriors: 2017
```

Note that `warriors` in the example above is a keyword argument; it follows strict naming rules compared to keys in a dictionary. For instance, string keys

are simply string objects and may contain spaces, while the name of a keyword argument cannot.

By convention, variadic arguments are called `**kwargs` when they represent keyword arguments and `*args` otherwise; and they can also be used together. Here's an example that combines them along with a regular parameter:

```
def print_latest_championship(prefix, *args, **kwargs):
  for line in args:
    print(prefix, line)
  for team, year in kwargs.items():
    print(prefix, team + ':', year) # dictionaries are unordered

print_latest_championship(
  '[NBA]',
  'Conference Finals',
  'Team: Year',
  cavaliers=2016,
  warriors=2017)
## [NBA] Conference Finals
## [NBA] Team: Year
## [NBA] cavaliers: 2016
## [NBA] warriors: 2017
```

Note that there are many ways to define and call functions in Python. All other things are equal, it's best to pick ones that are more readable, expressive of intent, and easier to maintain (usually, regular parameters passing would do the job).

6.6.1 Anonymous Functions

An anonymous function is similar to a regular function except that it doesn't have a name. It is created using the `lambda` keyword and it's typically used when functions expect arguments that are function objects. In this case, the calling environment can define an anonymous function and pass it on as an argument to the function being called. There is no need to name the function since it may never be used again (outside of it being passed as an argument to another function); it's convenient to use `lambda` expressions for single-expression anonymous functions. The code below contains a simple example with which you should be already familiar:

```
pairs = ((1, 3), (4, 2))
print(sorted(pairs, key=lambda p: p[1]))
## [(4, 2), (1, 3)]
```

A `lambda` expression can be assigned a name and reused:

```
pairs = ((1, 3), (4, 2))
get_second = lambda p: p[1]
print(sorted(pairs, key=get_second))
```

```
print('max:', max(pairs, key=get_second))
## [(4, 2), (1, 3)]
## max: (1, 3)
```

Also, a named `lambda` expression can refer to itself. Now would be a good time to revisit the example of a simple trie used to autocomplete a prefix into a word that belongs to a known list of words:

```
from collections import defaultdict

words = { 'apple', 'ban', 'banana', 'bar', 'buzz' }
# a factory function/lambda expression that returns
# a defaultdict which uses it as a factory function
factory = lambda: defaultdict(factory)
trie = defaultdict(factory)
DELIMITER = '--END--'
LEAF = dict()
for word in words:
  level = trie
  for letter in word:
    level = level[letter]
  level[DELIMITER] = LEAF

def autocomplete(root, prefix, suffix=''):
  if DELIMITER in root:
    print(prefix + suffix)
  for key, value in root.items():
    autocomplete(value, prefix, suffix + key)

# query for words that start with 'ba'
autocomplete(trie['b']['a'], 'ba')
## ban
## banana
## bar
```

The `factory` expression in the above example creates a new default dictionary that uses the same expression to create a new default dictionary (when needed) that uses the same expression, and so on. You may think of such pattern as recursion mixed with lazy initialization. A named `lambda` expression can be recursive the same way any function can be:

```
factorial = lambda x: factorial(x - 1) * x if x else 1
print(factorial(5))
## 120
```

Just like named functions, `lambda` expressions can take multiple arguments:

```
power = lambda x, y: x ** y
print(power(5, 2))
## 25
```

They can also be variadic:

```
from math import sqrt
norm = lambda *args: sqrt(sum(x * x for x in args))
print('norm:', norm(1, -2, 2))
## norm: 3.0
```

Basically, anonymous functions can do most of what a regular function can do, except that they are limited to a single expression. So what if you need to trace a call to an anonymous function by printing its arguments inside the `lambda` expression? Here's a trick that may help:

```
from math import sqrt

norm = lambda *a: 0 if print(a) else sqrt(sum(x * x for x in a))
print('norm:', norm(1, -2, 2))
## (1, -2, 2)
## norm: 3.0
```

Since the `print(args)` call returns `None`, it will always fail the `if` check and the `lambda` expression will evaluate and return the `else` expression. The above example is for demonstration purposes only, and shouldn't be used in real-world code since it's extremely counter-intuitive.

6.7 Classes

As in Java and C++, Python classes contain fields (variables) and methods (functions) and allow inheritance. We illustrate the concept below using a simple example of a class representing a point in a two-dimensional coordinate space:

```
class Point:
  def __init__(self, x, y):
    self.x = x
    self.y = y
  def __del__(self):
    print ("destructing a Point object")

p1 = Point(3, 4)
p2 = Point(1, 2)
print("p1.x = {0.x}, p1.y = {0.y}".format(p1))
print("p2.x = {0.x}, p2.y = {0.y}".format(p2))
## p1.x = 3, p1.y = 4
## p2.x = 1, p2.y = 2
## destructing a Point object
## destructing a Point object
```

The class methods definitions include a list of function arguments, the first of which is `self` that does not need to be passed when the method is called. For example, the initialization method above has three arguments `(self, x, y)`, but when a new object is instantiated only two arguments are used: `(x, y)`.

Inside a class method definition, we refer to a field of the current object using `self.field_name`. Outside of the method definition, fields and methods are called by prefixing the field or method name by the object's name followed by a period.

The methods `__init__` and `__del__` are the constructor method and destructor method that are called when an object is instantiated or when an object is destroyed by the garbage collection mechanism. Above, the initialization method is called with two arguments that are set by default to zero if an explicit value is not passed to the method.

The `Point` class above has the fields `x` and `y`, but they are not explicitly included in the class definition. In Python, fields that are explicitly defined in the class definitions are class fields (similar to static fields); they are shared by all instantiated objects of the class. To refer to said class fields, prefix the field name with the class name followed by a period. The example below shows a class having instance fields `x` and `y`; a class field `num_points` tracking the number of existing objects, and a `__doc__` class field holding the documentation string appearing after the class definition (see Sect. 12.5.1 for more information on such documentation strings).

```
class Point:
  '''Represents a point in a two-dimensional coordinate space.'''
  num_points = 0

  def __init__(self, x, y):
    Point.num_points += 1
    self.x = x
    self.y = y

  def __del__(self):
    Point.num_points -= 1
    print("destructing a Point object")
    print("{} Point objects left".format(Point.num_points))


print(Point.__doc__)
p1 = Point(3, 4)
p2 = Point(1, 2)
print("p1.x = {0.x}, p1.y = {0.y}".format(p1))
print("p2.x = {0.x}, p2.y = {0.y}".format(p2))
print("number of objects:", Point.num_points)
## Represents a point in a two-dimensional coordinate space.
## p1.x = 3, p1.y = 4
## p2.x = 1, p2.y = 2
## number of objects: 2
## destructing a Point object
## 1 Point objects left
## destructing a Point object
## 0 Point objects left
```

Python does not enforce private access control as Java or C++ do. Instead, programmers can prefix fields and methods by a single underscore that—by

convention—implies that the field or method should not be accessed by external code. Prefixing a field or method by a double underscore is a similar solution to a single underscore—but stronger—as it executes name-mangling: accessing the field or method can only be done by prefixing the stated name with an underscore followed by the class name.

Encapsulation as in C++ and Java can still be accomplished by referring to fields using getter and setter functions instead or directly accessing the variables. This can be done using the functions `getattr(object, field)` and `setattr(object, field, value)`.

6.7.1 Inheritance

Class inheritance in Python is indicated by including the base class in parenthesis after the class name in the class definition. Fields and methods belonging to the base class can still be referred to in the subclass. Fields and methods that are added to the subclass but do not exist in the base class represent additional functionality. A subclass can also re-implement a method that is already defined in the base class. This allows the subclass to override the base class functionality with new functionality that is customized for the subclass.

We demonstrate this concept by defining `NamedPoint`, a class derived from the above `Point` class. Note that the constructor of `NamedPoint` calls the base class constructor followed by setting the name field:

```
class Point:
  '''Represents a point in a 2-D coordinate space.'''
  num_points = 0

  def __init__(self, x, y):
    Point.num_points += 1
    self.x = x
    self.y = y

  def __del__(self):
    Point.num_points -= 1
    print("destructing a Point object")
    print("{} points left".format(Point.num_points))


class NamedPoint(Point):
  '''Represents a named point in a 2-D coordinate space.'''
  num_points = 0

  def __init__(self, x, y, name):
    # call superclass constructor
    super().__init__(x, y)
    NamedPoint.num_points += 1
    self.name = name
```

```
    def __del__(self):
      super().__del__()
      NamedPoint.num_points -= 1
      print("destructing a NamedPoint object")
      print("{} named points left".format(NamedPoint.num_points))


np = NamedPoint(0, 0, "origin point")
print("number of named points:", NamedPoint.num_points)
print("x = {0.x}, y = {0.y}, name = {0.name}".format(np))
## number of named points: 1
## x = 0, y = 0, name = origin point
## destructing a Point object
## 0 points left
## destructing a NamedPoint object
## 0 named points left
```

Note that in the example above, the `__init__` method in the subclass overrides the definition of `__init__` in the superclass. Such overriding can be done for other methods as well. In Python 2.X, you may see syntax like the following: `super(NamedPoint, self).__init__(x, y)` that refers to the superclass initializer (which is called by the subclass initializer); Python 3.X replaced such syntax with the simpler `super().__init__(x, y)`.

Even though `NamedPoint` is-a `Point`, we separate the `num_points` class fields so that the counts are independent. Python also supports polymorphism in the sense that it performs dynamic bindings. For example, if we call a method `foo` that is implemented differently by multiple subclasses, the interpreter will call the right method based on the specific subclass that the object is instantiating. In fact, Python has a stronger form of polymorphism than Java or C++ due to its duck typing (see Sect. 6.2.2). The correct method will be called regardless of the inheritance hierarchy (assuming the object provides the method). As a result, polymorphism and inheritance become somewhat less important in Python than in C++ and Java.

6.7.2 The Empty Class

When working with records or data transfer objects (DTO), all that you need is a bunch of data members grouped together into a class. In Python, that can be easily achieved using an empty class definition:

```
class Element:
  '''Represents an element in the periodic table.'''
  pass


na = Element()
na.atomic_number = 11
```

```
na.name = 'Sodium'

print(f'Atomic Number: {na.atomic_number}; Name: {na.name}')
## Atomic Number: 11; Name: Sodium
```

The above example demonstrates the use of an instance object in a way similar to a dictionary of fields. In fact, the current implementation of Python 3.x uses a `dict` object called `__dict__` to store the fields. This is an implementation detail and should not be used directly because it might change in the future.[10]

6.8 IPython

The traditional programming process has two stages: (a) writing code in one or more files, and (b) executing the code. This process is repeated as mistakes are found or additional features are added. In contrast, an interactive programming process consists of simultaneously writing and executing lines of code in an interactive shell. In many data analysis scenarios interactive programming is more effective than the traditional programming process. After the code has been fully developed using the interactive programming process, the code may be saved to a file and executed in a noninteractive manner.

IPython is an enhanced interactive environment that is more convenient than Python's default interactive environment. IPython can be installed separately and executed by typing `ipython` in the Linux or Mac terminal, or as a part of a software bundle (for example, Enthought's Canopy). The website http://ipython.org contains documentation and installation instructions.

IPython features the following important improvements over the standard interactive environment:

1. Pressing the tab key auto-completes variable names and function names.
2. Copying code segments and pasting them in the IPython environment often work better than in the standard Python interactive environment.
3. Prefixing or suffixing a variable or function name with `?` displays useful information. A double question mark `??` may display additional information.
4. Linux commands may be executed from the IPython prompt by prefixing them with `!`. The output of the shell command is displayed in the IPython environment. The prefix `!` may be omitted for some popular shell commands such as `ls`, `pwd`, and `cd`.
5. IPython enables logging command history and searching previous commands using the Readline tool. For example, typing control-R followed by a string finds the last command containing that string.[11]

[10] https://docs.python.org/3/reference/datamodel.html.

[11] See https://en.wikipedia.org/wiki/GNU_Readline for more information on the Readline library.

6. IPython has better support for displaying graphic windows and interacting with them.
7. IPython offers interactive debugging and profiling capabilities.
8. IPython has some additional commands called magic commands, typically prefixed by `%`. For example, the command `%run foo.py` runs the Python source code file `foo.py` in the IPython interactive environment and the commands `%who` and `%whos` display the variables currently defined in the IPython environment (`%whos` also displays the variables' types and values). You can type `%magic` for more magic commands and details on the magic subsystem in IPython.

   We describe below how to debug and profile using IPython.

6.8.1 Debugging

IPython enhances Python's standard debugger pdb with tab completion and syntax highlighting, and it displays better context information. One way to execute the IPython debugger is by typing `%debug` right after an error has occurred. In this case, the IPython debugger starts in the precise position where the error exception was raised. Another way to enter the debugger is by typing the command `%run -d foo.py`, which starts the debugger before executing any line of `foo.py`.

Inside the debugger, the command `n` (next) executes the current line and advances to the next line, the command `s` steps into a function call, the command `c` continues execution from the current position, and the command `b filename.py:line_number` sets a breakpoint at a specific line of the corresponding source file.

An alternative to using IPython's debugger is using Eclipse and the PyDev plugin (http://pydev.org), which provide interactive debugging through the Eclipse IDE.

6.8.2 Profiling

IPython has a number of convenient tools for measuring execution time and more generally profiling code. The IPython command `%time statement` runs the corresponding Python statement and reports execution time. The command `%timeit statement` is similar, but it runs the statement multiple times and reports running time statistics. The command `%time` is appropriate for measuring Python statements that take a significant amount of time to completion, while `%timeit` is appropriate for measuring Python statements that execute in a short amount of time.

The Python profiler can be executed by calling Python from the Linux or Mac terminal with a .py file as an argument and the following flags; the profiler generates a report profiling the execution of the code in the corresponding python file:

```
python -m cProfile -s cumulative file_name.py
```

A similar report can be generated by typing the following command inside IPython: `%run -p -s cumulative file_name.py`. The IPython command `%prun -s cumulative foo()` is similar, but it profiles arbitrary statements or functions rather than entire Python source files (in this case it profiles the execution of the function `foo`).

IPython features additional profiling tools including a line profiler that shows how much time was spent on different lines of code (`lprun` command) and memory profiling (`mprun` and `memit` commands). Consult the online documentation for additional information on these tools.

6.9 NumPy, SciPy, Pandas, and scikit-learn

Python has a large collection of modules that can be imported, some of which are part of the Python standard library and are installed by default. Third party modules in the Python Package Index (PyPI) can be installed using the `pip` command.[12] For example, to install a third party module `X` in PyPI call `pip install X`. In this section we cover four important modules for data analysis: NumPy, SciPy, pandas, and scikit-learn. Chapter 8 describes another module, matplotlib, for generating graphs from data.

NumPy is a Python package that provides (a) a data type `ndarray` for multi-dimensional arrays, (b) tools for vector, matrix, and array arithmetic operators over `ndarray` objects, (c) linear algebra and random number generation functionality, and (d) tools for reading from files and writing to files `ndarray` objects.

SciPy is a Python package that provides optimization algorithms, mathematical functions, signal processing techniques, and sparse matrix objects.

Pandas is a Python package that contains an implementation of an R-like dataframe object and includes a variety of functionality for manipulating dataframes.

Scikit-learn is a Python package for machine learning and data analysis; it can be used for classification, regression, clustering, data preprocessing, etc.

We give a brief overview of the four packages below.

[12] Python 2 (version 2.7.9 or greater) and Python 3 (version 3.4 or greater) ship with the pip tool pre-installed. To manually install the pip tool, follow the instructions at https://pip.pypa.io/en/stable/installing/.

6.9.1 Ndarray Objects

An ndarry is a multidimensional array that generalizes the one-dimensional list into higher dimensions. NumPy and other Python packages feature a wide variety of functions that can operate on ndarrays. Ndarrays have a `shape` variable that is a list of its dimensions (for example, $(2, 3)$ corresponds to a 2×3 matrix) and a `dtype` variable that corresponds to the type of data that it holds. Since an ndarray is an object we can refer to its variables and methods using the dot (`attrgetter`) operator.

The code example below shows how to define a 2x3 ndarray containing numbers, printing it, and casting its values to integers:

```
import numpy as np

# create 2x3 ndarray of floats from a list of lists
m = np.array([[1, 2, 3], [4, 5, 6]], dtype=np.float32)
print("m.shape =", m.shape)
print("m.dtype =", m.dtype)
print("m =\n" + str(m))
m = m.astype(np.int32) # cast the ndarray from float to int
print("m =\n" + str(m))
## m.shape = (2, 3)
## m.dtype = float32
## m =
## [[ 1.  2.  3.]
##  [ 4.  5.  6.]]
## m =
## [[1 2 3]
##  [4 5 6]]
```

The code below defines three ndarrays and prints them:

```
import numpy as np

m1 = np.zeros((2, 3)) # create a 2x3 ndarray of zeros
m2 = np.identity(3) # the 3x3 identity matrix
m3 = np.ones((2, 3, 2)) # create a 2x3x4 ndarray of ones
print("m1 =\n" + str(m1))
print("m2 =\n" + str(m2))
print("m3 =\n" + str(m3))
## m1 =
## [[ 0.  0.  0.]
##  [ 0.  0.  0.]]
## m2 =
## [[ 1.  0.  0.]
##  [ 0.  1.  0.]
##  [ 0.  0.  1.]]
## m3 =
## [[[ 1.  1.]
##   [ 1.  1.]
##   [ 1.  1.]]
```

```
##
##  [[ 1.  1.]
##   [ 1.  1.]
##   [ 1.  1.]]]
```

Arithmetic operations between ndarrays typically operate element-wise. Operations between an ndarray and a scalar repeat the scalar operation across all elements of the ndarray.

The code below defines two ndarrays, multiplies the first one by 2, and then adds them:

```
import numpy as np

m1 = np.identity(3)
m2 = np.ones((3, 3))
print("2 * m1 =\n" + str(2 * m1))
print("m1 + m2 =\n" + str(m1 + m2))
## 2 * m1 =
## [[ 2.  0.  0.]
##  [ 0.  2.  0.]
##  [ 0.  0.  2.]]
## m1 + m2 =
## [[ 2.  1.  1.]
##  [ 1.  2.  1.]
##  [ 1.  1.  2.]]
```

There are two convenient ways to refer to a portion of an ndarray: (a) using the sublist or slicing notation and (b) using a logical condition that characterizes the selected indices. Recall that the sublist notation `start:end` starts from index `start` and continues until index `end - 1` (indices count starts from 0). This works on both the right-hand side and the left-hand side of the assignment operator.

An example showing the sublist notation in the one-dimensional case appears below:

```
import numpy as np

a = np.array(range(9))
print("a[:] =", a[:])
print("a[3] =", a[3])
print("a[3:6] =", a[3:6])
a[3:6] = -1
print("a =", a)
## a[:] = [0 1 2 3 4 5 6 7 8]
## a[3] = 3
## a[3:6] = [3 4 5]
## a = [ 0  1  2 -1 -1 -1  6  7  8]
```

The example below shows the logical condition in the one-dimensional case:

```
import numpy as np
```

```
a = np.array(range(7))
# create an bool (mask) ndarray of True/False
print("a < 3 =", a < 3)
# refer to all elements that are lower than 3
print("a[a < 3] =", a[a < 3])
a[a < 3] = 0  # replace elements lower than 3 by 0
print("a =", a)
## a < 3 = [ True  True  True False False False False]
## a[a < 3] = [0 1 2]
## a = [0 0 0 3 4 5 6]
```

The example below shows the sublist notation in the two-dimensional case. Higher dimensional cases are similar:

```
import numpy as np

m = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])
print("m =\n" + str(m))
print("m[:, 1] =\n", m[:, 1]) # second column
# first two rows of the 2nd column
print("m[0:2, 1] =\n", m[0:2, 1])
# first 2 rows of the first two columns
print("m[0:2, 0:2] =\n" + str(m[0:2, 0:2]))
## m =
## [[0 1 2]
##  [3 4 5]
##  [6 7 8]]
## m[:, 1] =
##  [1 4 7]
## m[0:2, 1] =
##  [1 4]
## m[0:2, 0:2] =
## [[0 1]
##  [3 4]]
```

The example below shows the logical condition in the two-dimensional case. Higher dimensional cases are similar:

```
import numpy as np

m = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])
print("m =\n" + str(m))
print("m < 3 =\n" + str(m < 3))
print("m[m < 3] =\n", m[m < 3])
m[m < 3] = 0
print("m =\n" + str(m))
## m =
## [[0 1 2]
##  [3 4 5]
##  [6 7 8]]
## m < 3 =
## [[ True  True  True]
##  [False False False]
```

```
##  [False False False]]
## m[m < 3] =
##  [0 1 2]
## m =
## [[0 0 0]
##  [3 4 5]
##  [6 7 8]]
```

Referring to a multidimensional ndarray with a single index yields an ndarray obtained by fixing the first dimension to be the provided index and varying all other indices. For example, if `m` is a matrix, then `m[1]` corresponds to the second row of that matrix.

Another way to refer a subset of a two-dimensional ndarray is using a list of two lists of coordinates:

```
import numpy as np

m = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])
print("m =\n" + str(m))
# first row
print("m[0] =\n", m[0])
# first and third rows
print("m[[0, 2]] =\n" + str(m[[0, 2]]))
# third and first rows
print("m[[2, 0]] =\n" + str(m[[2, 0]]))
# refer to the (0, 2) and (1, 0) elements
print("m[[0, 1], [2, 0]] =\n", m[[0, 1], [2, 0]])
## m =
## [[0 1 2]
##  [3 4 5]
##  [6 7 8]]
## m[0] =
##  [0 1 2]
## m[[0, 2]] =
## [[0 1 2]
##  [6 7 8]]
## m[[2, 0]] =
## [[6 7 8]
##  [0 1 2]]
## m[[0, 1], [2, 0]] =
##  [2 3]
```

Assigning a new variable to an existing ndarray (or a part of it) does not create a new copy of the ndarray, but rather creates a new reference to the original ndarray:

```
import numpy as np

a = np.array(range(9))
b = a[0:4] # b refers to elements 0-3
b[1] = 42 # modify second element of b
print(a) # a is modified as well
## [ 0 42  2  3  4  5  6  7  8]
```

6.9.2 *Linear Algebra and Random Number Generation*

NumPy implements many arithmetic operators and linear algebraic concepts for ndarrays. Figure 6.2 lists some of the more popular ones.

Binary Operators	
Operator	**Description**
`**`	raise to a power (element-wise)
`maximum`	maximum of ndarrays (element-wise)
`minimum`	minimum of ndarrays (element-wise)
`dot`	matrix multiplication
Unary Operators	
Operator	**Description**
`fabs`	absolute value
`sqrt`	square root
`exp`	exponentiation
`log`	logarithm
`sign`	sign (-1 for negative, 0 for zero, +1 for positive)
`cos`	cosine
`sin`	sine
`sum`	sum
`mean`	average
`var`	variance
`std`	standard deviation
`max`	maximum
`min`	minimum
`argmax`	index of maximum
`argmin`	index of minimum
`cumsum`	cumulative sum
`cumprod`	cumulative product
`diag`	matrix diagonal
`trace`	matrix trace
`det`	matrix determinant
`T`	matrix transpose
`inv`	matrix inverse
`pinv`	matrix pseudo inverse
`eig`	eigenvalues and eigenvectors
`svd`	singular value decomposition

Fig. 6.2 Arithmetic operators and linear algebraic concepts for ndarrays

Some of these definitions apply specifically for the two-dimensional case (for example, determinant) while others extend to higher dimensions (for example, absolute value). Many of the operators above can operate on specific dimensions, which are passed as optional argument to the `axis` parameter.

The example below demonstrates operating the `mean` function on entire ndarrays, on columns, and on rows:

```
import numpy as np

m = np.array([[0, 1], [2, 3]])
# global average
print("np.mean(m) =", np.mean(m))
# average of columns
print("np.mean(m, axis=0) =", np.mean(m, axis=0))
# average of rows
print("np.mean(m, axis=1) =", np.mean(m, axis=1))
## np.mean(m) = 1.5
## np.mean(m, axis=0) = [ 1.  2.]
## np.mean(m, axis=1) = [ 0.5  2.5]
```

The example below demonstrates linear algebraic operators such as matrix inverse and matrix multiplication:

```
import numpy as np

m = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]]) + np.identity(3)
print("m =\n" + str(m))
m_inv = np.linalg.inv(m)
print("inverse of m =\n" + str(m_inv))
print("m times its inverse =\n" + str(np.dot(m, m_inv)))
## m =
## [[ 1.  1.  2.]
##  [ 3.  5.  5.]
##  [ 6.  7.  9.]]
## inverse of m =
## [[-2.  -1.   1. ]
##  [-0.6  0.6 -0.2]
##  [ 1.8  0.2 -0.4]]
## m times its inverse =
## [[  1.00000000e+00  -5.55111512e-17   1.11022302e-16]
##  [ -2.22044605e-16   1.00000000e+00  -2.22044605e-16]
##  [  0.00000000e+00   8.88178420e-16   1.00000000e+00]]
```

NumPy also has a matrix object, which is a special case of a two-dimensional ndarray. Matrix objects are different from ndarrays in that one-dimensional arrays (vectors) are treated as row or column matrices and can be inserted into a matrix computation with other two-dimensional matrices. In addition, the multiplication operator `*` acts as matrix multiplication:

```
import numpy as np

m = np.mat([[0, 1], [2, 3]])
```

```
print("m =\n" +  str(m))
print("m[:, 0] =\n" + str(m[:, 0])) # column vector
print("m[0, :] =\n" + str(m[0, :])) # row vector
# compute bilinear form v^T * A * v (matrix multiplication)
print("m[0, :] * m * m[:, 0] =")
print(m[0, :] * m * m[:, 0])
## m =
## [[0 1]
##  [2 3]]
## m[:, 0] =
## [[0]
##  [2]]
## m[0, :] =
## [[0 1]]
## m[0, :] * m * m[:, 0] =
## [[6]]
```

Note that if `m` was an ndarray, then `m[:, 0]` and `m[0, :]` would be both one-dimensional ndarray objects rather than matrices with a specific row or column orientation.

NumPy also has set logic functionality:

```
import numpy as np

a = np.array(['a', 'b', 'a', 'b'])
b = np.array(['c', 'd'])

print("a =", a)
print("np.unique(a) =", np.unique(a))
print("np.union1d(a, b) =", np.union1d(a, b))
print("np.intersect1d(a, b) =", np.intersect1d(a, b))
## a = ['a' 'b' 'a' 'b']
## np.unique(a) = ['a' 'b']
## np.union1d(a, b) = ['a' 'b' 'c' 'd']
## np.intersect1d(a, b) = []
```

NumPy also has functions for generating ndarrays containing pseudo-random numbers. These include `permute` for generating a random permutation, `rand` for generating uniformly distributed random variables, and `normal` for generating Gaussian random variables:

```
import numpy as np

 # N(0, 1) Gaussian
print("np.random.normal(size=(2, 2)) =")
print(np.random.normal(size=(2, 2)))
# uniform over [0,1]
print("np.random.uniform(size=(2, 2)) =")
print(np.random.uniform(size=(2, 2)))
# uniform over {1..10}
print("np.random.randint(10, size=(2, 2)) =")
print(np.random.randint(10, size=(2, 2)))
```

```
# random permutation over 1..6
print("np.random.permutation(6) =")
print(np.random.permutation(6))
## np.random.normal(size=(2, 2)) =
## [[ 1.4051513   0.07149204]
##  [-1.55613676  0.596488  ]]
## np.random.uniform(size=(2, 2)) =
## [[ 0.474377    0.25636062]
##  [ 0.64626636  0.50218616]]
## np.random.randint(10, size=(2, 2)) =
## [[4 8]
##  [0 0]]
## np.random.permutation(6) =
## [4 1 0 5 3 2]
```

6.9.3 Sparse Matrices in Python

SciPy contains additional functionality that is useful for signal processing, sparse linear algebra, optimization, integration, and interpolation. SciPy's namespace subsumes NumPy so there is no need to separately import NumPy when SciPy is imported. We focus in this section on SciPy's sparse matrices, which are an extremely useful tool in data analysis, for example when representing text or other categorical features, in machine learning, over a large set of possible values.

A sparse matrix is similar to a generic one, with the exception that most of its values are zero. This makes it possible to easily store very large matrices that have a manageable number of nonzero entries (and perform computation on such matrices).

Python has several sparse matrix classes, which we describe below. Sparse matrix formats are typically split into (a) those that permit efficient modification of the matrix, and (b) those that admit efficient matrix operations. Once the matrix is constructed in full and is not expected to change, it's typically converted to a format in the second category above.

Dictionary Method

The dictionary method (DOK) represents nonzero entries using a dictionary mapping row-column pairs (keys) to values. This format is convenient for incrementally building a matrix, but is inefficient for matrix operations.

List of Lists

List of lists (LIL) format stores the nonzero entries in a list of lists, where the first element of the list contains a list of the nonzero entries in the first row, the second

element of the list contains a list of the nonzero entries in the second row, and so on. Each entry in these lists is a pair of column-index and value. This format is convenient for incrementally constructing a matrix, but is inefficient for matrix operations.

Compressed Sparse Row and Column Representations

Compressed sparse row (CSR) representation stores the nonzero elements using three arrays. Denoting the original $m \times n$ sparse matrix as M, the first array A holds the nonzero entries of M in left-to-right top-to-bottom order. The second array B is defined recursively as follows: $B[0] = 0$ and $B[i] = B[i-1]$ + number of nonzero elements in the $i-1$ row in the original matrix.

The elements $A[B[i]], \ldots, A[B[i+1]-1]$ are the nonzero elements of the i-row of M. The third array C contains the column indices of the nonzero elements of M (using the same order as in A).

The compressed sparse column (CSC) representation is similar to the CSR, but with C keeping the row indices rather than column indices, and B keeping the indices in A where the nonzero values of each column start.

Compressed sparse row and column representations facilitate efficient matrix computation, and typically dictionary or LIL representations are converted to compressed row or column representations after the matrix is created and before the matrix computation begins.

Both CSR and CSC enable fast matrix addition and multiplication by a scalar. CSC provides efficient column slicing but slow row slicing. CSR provides efficient row slicing but slow column slicing. Both representations provide fast element-wise product and matrix multiplication. Both representations are slow to add additional nonzero elements or replace existing nonzero element with zero.

```
from scipy import sparse
from numpy import *

# create a sparse matrix using LIL format
m = sparse.lil_matrix((5,5))
m[0, 0] = 1
m[0, 1] = 2
m[1, 1] = 3
m[2, 2] = 4
print("m =\n" + str(m))
# convert a to CSR format
b = sparse.csr_matrix(m)
print("b + b =\n" + str(b + b)) # matrix addition
## m =
##   (0, 0) 1.0
##   (0, 1) 2.0
##   (1, 1) 3.0
##   (2, 2) 4.0
## b + b =
```

```
##   (0, 0) 2.0
##   (0, 1) 4.0
##   (1, 1) 6.0
##   (2, 2) 8.0
```

6.9.4 Dataframes

Pandas is a package that provides an implementation of a dataframe object that can be used to store datasets. As in R, a dataframe is essentially a two-dimensional array whose rows represent data instances and whose columns represent data attributes or features. In contrast to standard two-dimensional arrays such as ndarrays or matrices, dataframes can have different data types in different columns. For example, the first column may hold strings, the second may hold integers, and the third may hold a string. This example corresponds to a phone book of names (strings), phone numbers (integers), and addresses (strings). As in R, columns may be given names and these names may be used to access the columns making it unnecessary to separately keep meta-data describing the meaning of the different columns.

In the examples below we show how to create a dataframe and access it. The simplest way to create a dataframe is to call the `DataFrame` function with a dictionary whose keys are column names and whose values are lists of equal length:

```
import pandas as pd

data = {
  "names": ["John", "Jane", "George"],
  "age": [25, 35, 52],
  "height": [68.1, 62.5, 60.5],
}
df = pd.DataFrame(data)
print("dataframe content =\n" + str(df))
print("dataframe types =\n" + str(df.dtypes))
## dataframe content =
##    age  height   names
## 0   25    68.1    John
## 1   35    62.5    Jane
## 2   52    60.5  George
## dataframe types =
## age         int64
## height    float64
## names      object
## dtype: object
```

Note that the columns are named appropriately; each column holds objects of a different type, and the rows are numbered starting from zero. Columns can be accessed using brackets and column names or the period operator, and rows can be accessed using the member function `ix`:

```
import pandas as pd

data = {
  "name": ["John", "Jane", "George"],
  "age": [25, 35, 52],
  "height" : [68.1, 62.5, 60.5],
}
df = pd.DataFrame(data)
df["age"] = 35  # assign 35 to all age values
print("age column =\n" + str(df["age"]))
print("height column =\n" + str(df.height))
print("second row =\n" + str(df.ix[1]))
## age column =
## 0    35
## 1    35
## 2    35
## Name: age, dtype: int64
## height column =
## 0    68.1
## 1    62.5
## 2    60.5
## Name: height, dtype: float64
## second row =
## age         35
## height    62.5
## name      Jane
## Name: 1, dtype: object
```

As shown above, the dataframe values may be modified by referring to a column (or a row) on the left-hand side of an assignment operator. If the right-hand side has a list of the appropriate length, the values in that list will overwrite the current values in the dataframe. In a similar way, a new column may be added:

```
import pandas as pd

data = {
  "name": ["John", "Jane", "George"],
  "age": [25, 35, 52],
  "height" : [68.1, 62.5, 60.5],
}
df = pd.DataFrame(data)
df["weight"] = [170.2, 160.7, 185.5]
print(df)
##    age  height    name  weight
## 0   25    68.1    John   170.2
## 1   35    62.5    Jane   160.7
## 2   52    60.5  George   185.5
```

The dataframe object has many useful member methods, many of which apply by default across the dataframe columns, for example `sum`, `median`, and `abs`. Type `dir(df)` where `df` is a dataframe object to see a list of these functions. An optional `axis=1` argument can modify these functions to apply on rows instead:

```
import pandas as pd

data = {
  "name": ["John", "Jane", "George"],
  "age": [25, 35, 52],
  "height" : [68.1, 62.5, 60.5],
}
df = pd.DataFrame(data)
print("medians of columns =\n" + str(df.median()))
print("medians of rows =\n" + str(df.median(axis=1)))
## medians of columns =
## age       35.0
## height    62.5
## dtype: float64
## medians of rows =
## 0    46.55
## 1    48.75
## 2    56.25
## dtype: float64
```

The member method `apply` applies a function passed as an argument to the dataframe rows (or columns if the `axis=1` argument is passed):

```
import pandas as pd

data = {
  "age": [25.2, 35.4, 52.1],
  "height": [68.1, 62.5, 60.5],
  "weight": [170.2, 160.7, 185.5],
}

df = pd.DataFrame(data)
# apply f(x) = x + 1 to all columns
print(df.apply(lambda z: z + 1))
##     age  height  weight
## 0  26.2    69.1   171.2
## 1  36.4    63.5   161.7
## 2  53.1    61.5   186.5
```

The example below shows how NumPy and pandas work with missing values. We insert missing values denoted by `nan` in the NumPy package. When the pandas dataframe is created, these missing values are converted to `NaN` values. Functions typically ignore such missing values, but this default behavior may be modified as shown below. The dataframe member function `describe` computes basic summaries for each column (mean, median, minimum value, maximum value, etc.):

```
import numpy as np
import pandas as pd

data = {
  "age" : [25.2, np.nan, np.nan],
  "height" : [68.1, 62.5, 60.5],
```

```
  "weight" : [170.2, np.nan, 185.5],
}
df = pd.DataFrame(data)
# NA stands for Not Available
print("column means (NA skipped):")
print(str(df.mean()))
print("column means: (NA not skipped)")
print(str(df.mean(skipna=False)))
## column means (NA skipped):
## age        25.20
## height     63.70
## weight    177.85
## dtype: float64
## column means: (NA not skipped)
## age         NaN
## height     63.7
## weight      NaN
## dtype: float64
```

The pandas package has additional for manipulating dataframes, handling missing values, and transforming dataframe values. We cover some of that functionality in Chap. 9.

6.9.5 scikit-learn

scikit-learn is a package that provides machine learning, data mining, and data analysis tools for Python. It requires NumPy and SciPy to run, and uses matplotlib.[13] The scikit-learn package can be used for many machine learning applications like clustering, regression, classification, etc. We will go through a few examples to demonstrate an overview of the scikit-learn package below. For more details about machine learning applications, we recommend two popular textbooks: (Bishop, 2006; Murphy, 2012).

Before getting into the applications of machine learning using scikit-learn, we are going to need data—without which machine learning is futile. Luckily, scikit-learn ships with toy datasets to play with; one of them, the iris dataset, is a classic and we're going to use it in many examples below, so it's important to get familiar with it first:

```
from sklearn import datasets

iris = datasets.load_iris()

print(iris.feature_names) # columns
print(iris.data[:5]) # first 5 rows of the ndarray
```

[13] See Chap. 8 for details about matplotlib.

```
print('...')
print(iris.DESCR) # description
## ['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)',
##     'petal width (cm)']
##
## [[ 5.1  3.5  1.4  0.2]
##  [ 4.9  3.   1.4  0.2]
##  [ 4.7  3.2  1.3  0.2]
##  [ 4.6  3.1  1.5  0.2]
##  [ 5.   3.6  1.4  0.2]]
## ...
## Iris Plants Database
## (truncated for brevity)
```

Note that the Iris data resides in an object whose fields are `DESCR` (dataset description), `data` (an array of arrays of feature values), `feature_names` (descriptive feature names), `target` (class labels in numeric encoding), and `target_names` (descriptive names of class labels). The output above was truncated for brevity; we encourage you to read the documentation and get familiar with a dataset before using it.

Clustering

The goal of clustering is to automatically group similar data together; the machine can learn—from a dataset of samples—how to cluster data based on the samples' features alone and without any supervision from humans—who don't label the data (unsupervised learning).

Using the iris dataset, we try to fit the 150 samples into 3 clusters using the k-means algorithm[14]:

```
import numpy as np
from sklearn import datasets
from sklearn.cluster import KMeans
from sklearn.metrics import *

iris = datasets.load_iris()
# instead of running the algorithm many times with random
# initial values for centroids, we picked one that works well;
# the values below were obtained from one of the random runs:
centroids = np.array([
  [5.006, 3.418, 1.464, 0.244],
  [5.9016129, 2.7483871, 4.39354839, 1.43387097],
  [6.85, 3.07368421, 5.74210526, 2.07105263],
])
predictor = KMeans(n_clusters=3, init=centroids, n_init=1)
predictor.fit(iris.data)
```

[14] http://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html.

```
# range of scores is [0.0, 1.0]; the higher, the better
completeness = completeness_score(iris.target, predictor.labels_)
homogeneity = homogeneity_score(iris.target, predictor.labels_)
accuracy = accuracy_score(iris.target, predictor.labels_)
print("Completeness:", completeness)
print("Homogeneity:", homogeneity)
print("Accuracy:", accuracy)
## Completeness: 0.764986151449
## Homogeneity: 0.751485402199
## Accuracy: 0.893333333333
```

Note that in most clustering problems, we may not know the ground truth since there are no labels (unsupervised learning). The algorithm clustered the data together without knowing what it means to belong to each cluster; in this case, a cluster represents a species in the genus Iris. Without the ground truth (`iris.target`), validation becomes difficult, in fact, Jain and Dubes wrote in their 1988 book (Jain and Dubes, 1988) that "validation of clustering structures is the most difficult and frustrating part of cluster analysis. Without a strong effort in this direction, cluster analysis will remain a black art accessible only to those true believers who have experience and great courage." We encourage readers interested in this topic to read (Jain and Dubes, 1988; Kaufman and Rousseeuw, 1990) to learn more.

Classification

The goal of classification is to label (classify) data; the machine can learn—from a dataset of labeled samples (ground truth)—how to classify new data that weren't used for learning (training); such learning is called supervised because it requires a labeled dataset for training to predict labels of new data.

Using the iris dataset, we implement a binary classifier that predicts whether a sample is an Iris-Versicolor (denoted by the label 1) or not (denoted by the label 0). We randomly pick 80% of the samples (120 samples) as the training dataset, which leaves 20% of the samples (30 samples) for validation and estimating the test (prediction) error. In the example below, we use the Stochastic Gradient Descent (SGD) algorithm to obtain a linear classifier[15] (SVM):

```
import numpy as np
from sklearn import datasets
from sklearn.linear_model import SGDClassifier
from sklearn.metrics import accuracy_score

# constant seed to reproduce the same results every time
np.random.seed(28)
```

[15] http://scikit-learn.org/stable/modules/sgd.html#classification.

```
iris = datasets.load_iris()
# prepare labels for binary classification task
# 1 iff original target is Iris-Versicolor
labels = iris.target == 1
labels = labels.reshape((len(labels), 1))
data = np.append(iris.data, labels, axis=1)
# randomly shuffle data and split to train and test sets
data = np.random.permutation(data)
split = 4 * len(data) // 5
train_data, test_data = data[:split], data[split:]
train_features = train_data[:, :-1]
train_labels = train_data[:, -1]
predictor = SGDClassifier(n_iter=500)
predictor.fit(train_features, train_labels)
test_features = test_data[:, :-1]
test_labels = test_data[:, -1]
test_error = 1 - accuracy_score(test_labels,
predictor.predict(test_features))
print("Test Error: {:.3%}".format(test_error))
## Test Error: 6.667%
```

Note that the measurements above are unreliable because the dataset is very small (only 150 samples); with the right set of data, the classifier should be more reliable.

Regression

The goal of regression is to predict a real-valued label $y \in \mathbb{R}$ for unlabeled data from a training data containing pairs of feature vectors and real valued targets. In contrast to the classification scenario where the model is trained to accurately predict the target values, linear regression is trained to minimize the squared errors between the predicted values and the available labels.

This time we use a dataset that contains data about house prices (continuous values) in Boston; let's take a look at it first:

```
from sklearn import datasets

print(datasets.load_boston().DESCR)
## Boston House Prices dataset
##
## Notes
## ------
## Data Set Characteristics:
##
##     :Number of Instances: 506
##
##     :Number of Attributes: 13 numeric/categorical predictive
##
##     :Median Value (attribute 14) is usually the target
##
##     :Attribute Information (in order):
```

```
##          - CRIM     per capita crime rate by town
##          - ZN       proportion of residential land zoned for
##               lots over 25,000 sq.ft.
##          - INDUS    proportion of non-retail business acres
##               per town
##          - CHAS     Charles River dummy variable (= 1 if tract
##               bounds river; 0 otherwise)
##          - NOX      nitric oxides concentration (parts per 10
##               million)
##          - RM       average number of rooms per dwelling
##          - AGE      proportion of owner-occupied units built
##               prior to 1940
##          - DIS      weighted distances to five Boston
##               employment centres
##          - RAD      index of accessibility to radial highways
##          - TAX      full-value property-tax rate per $10,000
##          - PTRATIO  pupil-teacher ratio by town
## (truncated for brevity)
```

Using the house prices dataset, we implement a linear regression model that predicts the price y of a house using a set of features X. We keep the same data split process of holding out 20% of the samples for validation and estimating the test (prediction) error. In the example below, we use the Stochastic Gradient Descent (SGD)—again—to obtain the linear regression model[16]:

```
import numpy as np
from sklearn import datasets
from sklearn.linear_model import SGDRegressor
from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import StandardScaler

# constant seed to reproduce the same results every time
np.random.seed(448)
houses = datasets.load_boston()
split = 4 * len(houses.data) // 5
X_train, X_test = houses.data[:split], houses.data[split:]
y_train, y_test = houses.target[:split], houses.target[split:]
# linear regression works better with normalized features
scaler = StandardScaler()
scaler.fit(X_train)
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)
predictor = SGDRegressor(loss="squared_loss")
predictor.fit(X_train, y_train)
mse = mean_squared_error(y_test, predictor.predict(X_test))
print("Test Mean Squared Error: ${:,.2f}".format(mse * 1000))
## Test Mean Squared Error: $20,378.09
```

[16] http://scikit-learn.org/stable/modules/sgd.html#regression.

Note that the error here is calculated as:

$$\text{MSE}(y, \hat{y}) = \frac{1}{\text{len}(y)} \sum_{i=0}^{\text{len}(y)-1} (y_i - \hat{y}_i)^2$$

As usual, better data produce a more accurate predictor.

6.10 Reading and Writing to Files

6.10.1 Reading and Writing Data in Text Format

To read from a file or write to a file in Python, the file must first be opened in the appropriate mode using the `open` function (for example, `r` for reading and `w` for writing). The object returned from the open function can then be used to read from the file or write to it in a variety of ways. The `close` function should be called at the end of the reading or writing process. Figure 6.3 lists the most commonly used modes.

Mode	Description
`'+'`	Update (read and write)
`'a'`	Append (write at the end of the file if it exists)
`'b'`	Open in binary mode
`'r'`	Read
`'t'`	Open in text mode
`'w'`	Write (truncate first)
`'x'`	Create (a new file for writing)

Fig. 6.3 File opening modes in Python

A simple way to read a text file is to read it line by line using a for-loop. The code below illustrates this by creating a list of words in a text file. The iterator returned from the file opening function iterates over text lines, and the function `split` converts a string representing a line into a list of strings separated by whitespaces representing words:

```
f = open("mobydick.txt", "r") # open file for reading
words = []
for line in f: # iterate over all lines in file
  words += line.split() # append the list of words in line
f.close()
```

Or, more Pythonically, we can use a list comprehension and automatically close the file using a `with` statement:

```
with open("mobydick.txt") as f: # "rt" is the default mode
  words = [word for line in f for word in line.split()]
```

Building on the example above, we create a set instead of a list to ignore duplicates. Doing so produces a set of distinct words known as the text document vocabulary; we add another constraint by restricting the vocabulary to words of length 10 or longer:

```
with open("mobydick.txt") as f:
  words = {w for line in f for w in line.split() if len(w) > 9}
```

The functions `write` and `writelines` can be used to write contents to a file. The code below demonstrates the use of `write`; note that the file is opened in a writing mode (indicated by the second argument `w` to the `open` function):

```
with open("output.txt", "w") as f:
  f.write("first line\n")
  f.writelines(["second line\n", "third line\n"])
```

6.10.2 Reading and Writing Ndarrays in Binary Format

NumPy offers several functions for reading and writing ndarrays. Specifically, the functions `numpy.load` and `numpy.save` respectively read and write ndarrays in an uncompressed binary format. There is no need to explicitly open and close the files. The standard file name extension for such files is `.npy`.

```
import numpy as np
import os

m = np.array([[1, 2, 3], [4, 5, 6]])
file_name = "matrix.npy"
np.save(file_name, m)
print("File Size in Bytes:", os.stat(file_name).st_size)
loaded = np.load(file_name)
print(loaded)
## File Size in Bytes: 128
## [[1 2 3]
##  [4 5 6]]
```

The function `numpy.savez` is similar but it saves the data in an uncompressed zipped archive format and can take multiple arguments corresponding to several arrays; `numpy.savez_compressed` saves the data in a compressed zipped archive format and can also save multiple arrays. The standard file name extension for zipped binary format is `.npz`. The function `numpy.load` detects the format

automatically and loads the data into a dictionary-like object that holds the loaded arrays:

```
import numpy as np
import os

file_name = "output.npz"
for save in np.savez, np.savez_compressed:
  save(file_name, foo=np.array([1, 2]), bar=np.array([3, 4]))
  arrays = np.load(file_name)
  print("Using {}, {} bytes:".format(
    save.__name__,
    os.stat(file_name).st_size))
  for key, value in arrays.items(): # unordered
    print("\t{}: {}".format(key, value))
  print() # empty line
## Using savez, 394 bytes:
##  foo: [1 2]
##  bar: [3 4]
##
## Using savez_compressed, 350 bytes:
##  foo: [1 2]
##  bar: [3 4]
```

The reduction in file size due to the compression is very modest in the case above due to the small size of the ndarray. It can become much more significant for large ndarrays.

6.10.3 Reading and Writing Ndarrays in Text Format

The function `numpy.loadtxt` reads a text file and converts it to an ndarray (an optional second argument specifies the delimiter). Rows in the text file are copied into rows in the ndarray. The function `numpy.savetxt` saves an ndarray object to a text file. Despite their larger size on disk, text files are sometimes preferred for readability using the Linux or Mac terminal, for example:

```
import numpy as np

def print_file(file_name):
  with open(file_name) as f:
    for line in f:
      print(line, end='')

file_name = "array.txt"
np.savetxt(file_name, np.array([[1, 3], [2, 4]]))
loaded = np.loadtxt(file_name)
print("Using numpy.loadtxt:\n{}".format(loaded))
print("Text File Content:")
print_file(file_name)
## Using numpy.loadtxt:
```

```
## [[ 1.  3.]
##  [ 2.  4.]]
## Text File Content:
## 1.0000000000000000e+00 3.0000000000000000e+00
## 2.0000000000000000e+00 4.0000000000000000e+00
```

The function `numpy.genfromtxt` is similar to `numpy.loadtxt` but it can handle structured arrays and missing values.

6.10.4 Reading and Writing Dataframes

The pandas functions `to_csv` saves a dataframe to a text file using CSV (comma-separated values) format. The function `read_csv` reads a dataframe from a text file.

The code below demonstrates a simple case where we save a dataframe and then load it back. Note that the default behavior when writing to a CSV file is to use the first row of the saved text to represent the names of the columns with the first element of the first row being blank to represent an index column, whose values take the first column for the rest of the file. In the example below, we omit the index column by specifying `index=False`:

```
import pandas as pd

def print_file(file_name):
  with open(file_name) as f:
    for line in f:
      print(line, end='')

data_frame = pd.DataFrame({
  "age": [25.2, 35.4, 52.1],
  "height": [68.1, 62.5, 60.5],
  "weight": [170.2, 160.7, 185.5],
})
file_name = "dataframe.csv"
data_frame.to_csv(file_name, index=False)
loaded = pd.read_csv(file_name)
print("Data Frame:\n{}".format(loaded))
print("Text File Content:")
print_file(file_name)
## Data Frame:
##     age  height  weight
## 0  25.2    68.1   170.2
## 1  35.4    62.5   160.7
## 2  52.1    60.5   185.5
## Text File Content:
## age,height,weight
## 25.2,68.1,170.2
## 35.4,62.5,160.7
## 52.1,60.5,185.5
```

If the text file doesn't start with an appropriate header row describing the column names as its first row, we can still use `read_csv` with the argument `header=False`.

In some cases we need to read data from a text file whose format deviates from the above. One way to deal with this is to write a custom file read function in Python. Another option is to first write a program to convert the text file into an acceptable format (in Python or in a different language), and then call one of the Python functions mentioned above to load it.

6.11 Material Differences Between Python 3.x and 2.x

As noted earlier in this chapter, the Python programs in this book target Python 3.x (since Python 2.x is legacy and Python 3.x offers several new features). Python 3.x is not entirely backward-compatible with Python 2.x. In this section, we describe material differences between the two versions.[17]

6.11.1 Unicode Support

One of the big improvements in Python 3.x is that all string objects are Unicode by default. In Python 2.x, `str` and `unicode` are two separate types, and one had to prefix a string literal with the letter `u` to signify that it's a Unicode literal. To demonstrate (in Python 2.x):

```
# Python 2.x
print(type('')) # str
print(type(u'')) # unicode
## <type 'str'>
## <type 'unicode'>
```

While in Python 3.x:

```
# Python 3.x
print(type('')) # str
print(type(u'')) # also str
## <class 'str'>
## <class 'str'>
```

6.11.2 Print

In Python 2.x, `print` is a statement; In Python 3.x, it became a function. In many cases, the `print()` syntax works correctly in Python 2.x, but it's still a statement

[17] See https://wiki.python.org/moin/Python2orPython3 for more details about the version change.

and not a function. To get the functionality of the `print` function in Python 2.x, one can import it (from the future):

```
# Python 2.x
# imports from the future must come first in the file
from __future__ import print_function
print('hello from the future')
print(print) # the print function is an object
## hello from the future
## <built-in function print>
```

So what difference does it make to use the `print` function vs. the `print` statement with parentheses that evaluate the object passed to the statement? The latter doesn't work when passing multiple arguments because they will be treated as a tuple (in Python 2.x):

```
# Python 2.x
print(13, 42)
## (13, 42)
```

While using the `print` function, the parameters are printed correctly as desired:

```
# Python 2.x
from __future__ import print_function
print(13, 42)
## 13 42
```

The `print` statement syntax is not supported in Python 3.x, so running Python 2.x code that uses it fails with the following error:

```
# Python 3.x
print 13, 42
##  File "<stdin>", line 1
##    print 13, 24
##            ^
## SyntaxError: Missing parentheses in call to 'print'
```

6.11.3 Division

Unlike the case with the `print` statement, porting code that includes division from Python 2.x to 3.x will still work, but it may produce incorrect results—silently! The default behavior of the division operator `/` in Python 2.x is to perform floor (integer) division when both operands are integers (like C++ and Java); for example:

```
# Python 2.x
print(1 / 3) # floor (integer) division
print(1.0 / 3) # true division
## 0
## 0.333333333333
```

After importing the new division operator from Python 3.x, the behavior of the division operator / changes to true division (even for integers), and a new division operator, //, can be used for floor division:

```
# Python 3.x (or 2.x with the import from 3.x)
from __future__ import division
print(1 // 3) # floor (integer) division
print(1 / 3) # true division
```

Changes like this one makes a strong case for testing because even the behavior of a primitive operator like / might change in the future.

6.12 Notes

Python was introduced by Guido van Rossum in 1991 and has quickly gained popularity in a variety of fields, including system administration, data analysis, and web programming. An important part of the Python ecosystem is its package system. NumPy, SciPy, pandas, and scikit-learn are perhaps the most popular packages for data analysis, but many other useful high quality packages exist.

There are many good Python textbooks. One example is (Lutz, 2011) that can be augmented with (Lutz, 2013) for beginners. Such general introductory books are usually broad, but do not cover in detail data analysis packages. A couple of Python textbooks specifically aimed at data analysis are (McKinney, 2013) and (Bird et al., 2009). These latter two books include description of packages that are useful for data analysis. Additional information regarding Python packages is usually best found at the Python Package Index (PyPI).

References

C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

K. P. Murphy. *Machine Learning: A Probabilistic Perspective*. MIT Press, 2012.

Anil K. Jain and Richard C. Dubes. *Algorithms for Clustering Data*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988. ISBN 0-13-022278-X.

Leonard Kaufman and Peter J. Rousseeuw. *Finding groups in data : an introduction to cluster analysis*. Wiley series in probability and mathematical statistics. Wiley, New York, 1990. ISBN 0-471-87876-6. A Wiley-Interscience publication.

M. Lutz. *Programming Python*. O'Reilly Media Inc., fourth edition, 2011.

M. Lutz. *Learning Python*. O'Reilly Media Inc., fifth edition, 2013.

W. McKinney. *Python for data analysis*. O'Reilly Media Inc., 2013.

S. Bird, E. Klein, and E. Loper, editors. *Natural Language Processing with Python*. O'Reilly Media Inc., 2009.

Chapter 7
Learning R

R is a programming language that's especially designed for data analysis and data visualization. In some cases, it's more convenient to use R than C++ or Java, making R a key data analysis tool. In this chapter, we describe similarities and differences between R and its close relatives: Matlab and Python. We then delve into the R programming language to learn about data types, control flow, interfacing with C++, etc.

7.1 R, Matlab, and Python

R is similar to Matlab and Python in the following ways:

- They run inside an interactive shell in their default setting. In some cases this shell is a complete graphical user interface (GUI).
- They emphasize storing and manipulating data as multidimensional arrays.
- They interface packages featuring functionalities, both generic, such as matrix operations, SVD, eigenvalues solvers, random number generators, and optimization routines; and specialized, such as kernel smoothing and support vector machines.
- They exhibit execution time slower than that of C, C++, and Fortran, and are thus poorly suited for analyzing massive[1] data.
- They can interface with native C++ code, supporting large scale data analysis by implementing computational bottlenecks in C or C++.

[1]The slowdown can be marginal or significant, depending on the program's implementation. Vectorized code may improve computational efficiency by performing basic operations on entire arrays rather than on individual array elements inside nested loops.

© Springer Nature Switzerland AG 2018

G. Lebanon, M. El-Geish, *Computing with Data*,
https://doi.org/10.1007/978-3-319-98149-9_7

The three languages differ in the following ways:

- R and Python are open-source and freely available for Windows, Linux, and Mac, while Matlab requires an expensive license.
- R, unlike Matlab, features ease in extending the core language by writing new packages, as well as in installing packages contributed by others.
- R features a large group of motivated contributors who enhance the language by implementing high-quality packages.[2]
- Developers designed R for statistics, yielding a syntax much better suited for computational data analysis, statistics, and data visualization.
- Creating high quality graphs in R requires less effort than it does in Matlab and Python.
- R is the primary programming language in the statistics, biostatistics, and social sciences communities, while Matlab is the main programming language in engineering and applied math, and Python is a popular general purpose scripting and web development language.

7.2 Getting Started

The first step is to download and install a copy of R on your computer. R is available freely for Windows, Linux, and Mac at the R Project website http://cran.r-project.org.

The two most common ways to run R are

- in a terminal prompt, by typing `R` in a Linux or Mac terminal, and
- inside a GUI, by double clicking the R icon on Windows or Mac or by typing `R -g Tk &` on Linux.

Other ways to run R include

- using the Emacs Speaks Statistics (ESS) package from within Emacs,
- using a third party GUI such as R-Studio (freely available from http://www.rstudio.org; see Fig. 7.2 for a screen shot), and
- running R from a client that accesses an R server application.

The easiest way to quit R is to type `q()` at the R prompt or to close the corresponding GUI window.

Like other programming languages, R code consists of a sequence of commands. Often, every command in R is typed in its own line. Alternatively, one line may contain multiple commands, separated by semicolons, which can be confusing

[2]Figure 7.1 demonstrates the rapid growth in the number of packages (see also (Fox, 2009)). All contributed packages go through a quality control process, and enforcement of standards, and have common documentation format.

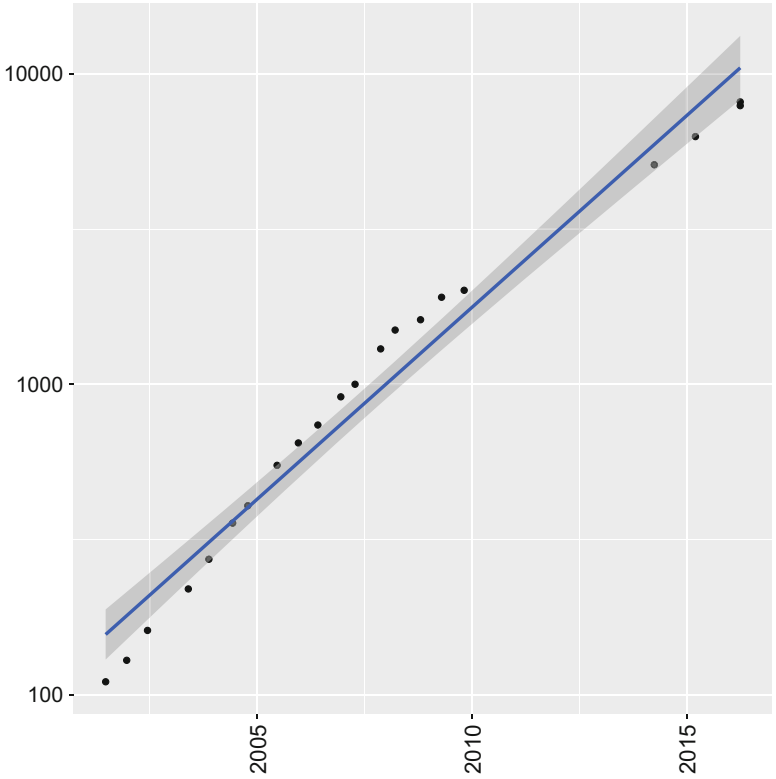


Fig. 7.1 An almost linear growth on the log scale of the number of contributed R packages indicates a growth rate that is close to exponential. See (Fox, 2009) for more details and an interesting description of the open source social movement behind R. The straight line shows linear regression fit. At the time of writing the number of current packages exceeds 8000

sometimes and should be avoided. Comments require a hash sign # and occupy the remainder of the current line.

```
# This is a comment
a = 4   # single statement
a = 4; b = 3; c = b   # multiple statements
```

R is a functional object oriented language in that everything that happens is a function call and everything that exists is an object. For example, `a = b` is equivalent to the function call `'='(a, b)`, and accessing an array element `a[3]` is equivalent to the function call `'['(a, 3)`. R features lazy evaluation in that function arguments or other expressions are only evaluated when (and if) they are actually used.

Since R, unlike C++ or Java, is not strongly typed, we can define a variable without expressing a type and can even change a variable's type within the same session. Thus, the following is quite legal.

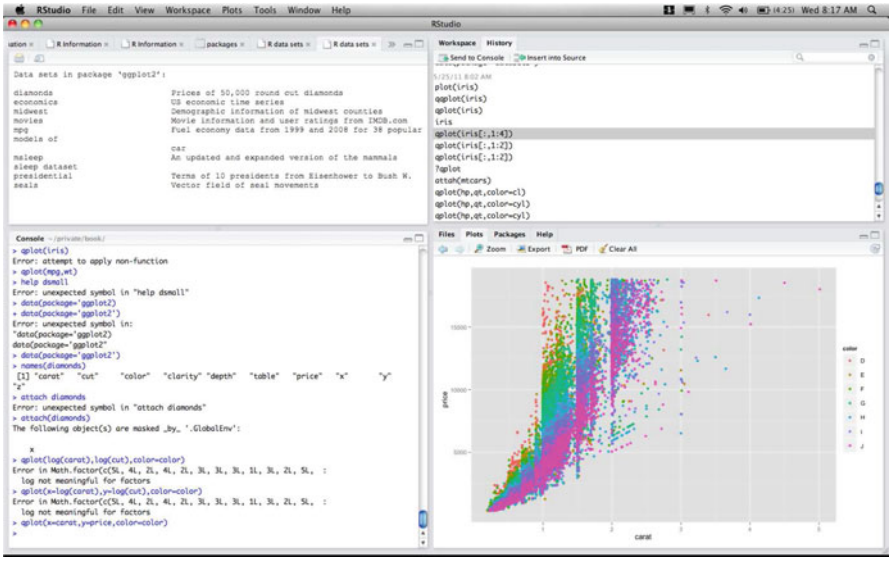


Fig. 7.2 Screen shot of the R development environment R-Studio available from http://www.rstudio.org

```
a = 3.2
a = "string"
```

We can display the value of any variable using the `print()` function or by simply typing its name. Below, we use a double hash symbol ## to prefix any output printed by the R program.

```
a = 4
print(a)
## [1] 4
a  # same thing
## [1] 4
cat(a)  # same thing
## 4
# cat can print multiple variables one after the other
cat("The value of the variable a is: ", a)
## The value of the variable a is:  4
```

As shown above, we set variables with the assignment operator =. An alternative is the operator `<-`, as in `a <- 4` or `4 -> a`. We use the classical = operator since it is more similar to assignment operators in other languages.

Strings in R are sequences of case-sensitive characters surrounded by single or double quotes. To get help on a specific function, operator, or data object, type `help(X)` where `X` is the corresponding string. An abbreviated version of `help(X)` is `?X`. The function `help.search(X)` searches the documentation for the given text string `X` and can be abbreviated by `??X`. The command `example(X)`

shows an example of the use of the function X. The command help.start() starts an html-based documentation within a browser, which is sometimes easier to navigate. In addition to the commands above, searching the web using a search engine often provides useful results.

In R, periods can be used to delimit words within a variable name; for example, my.parallel.clustering is a legitimate name for a variable or a function. The $ operator in R has a similar role to the period operation in C++ and Java.

Here are some important commands with short explanations.

```
x = 3  # assign value 3 to variable x
y = 3*x + 2  # basic variable assignment and arithmetic
ratio.of.x.and.y = x / y  # divide x by y and assign result
ls()  # list variable names in workspace memory
ls(all.names = TRUE)  # list all variables including hidden ones
ls.str()  # print annotated list of variable names
save.image(file = "fname")  # save all variables to a file
save(x, y, file = "fname")  # save specified variables
rm(x, y)  # clear variables x and y from memory
rm(list = ls())  # clear all variables in workspace memory
load(varFile)   # load variables from file back to the workspace
history(15)  # display 15 most recent commands
```

The precise interaction of the command line or R IDE tool (like R-Studio) depends on the operating system and IDE program. In general, pressing the up-arrow key and down-arrow key allows browsing through the command history for previous commands. This can be very useful for fixing typos or for executing slight modifications of long commands. In the Linux and Mac terminal, Control-R takes a text pattern and returns the most recent command containing that pattern.

Upon exiting R with the q() function, the command line prompts the user to save the workspace memory. Saving the workspace memory places all current variables in a file named .RData in the current directory. Launching R automatically uploads that file if it exists in the current directory, retrieving the set of variables from the previous session in that directory. Inside R, the user can change directories or view the current directory using setwd(X) and getwd() respectively (X denotes a string containing a directory path). The function system(X) executes the shell command X.

```
# change directory to home directory
setwd("~")
# display all files in current directory
dir(path = ".", all.files = TRUE)
# execute bash command ls -al (in Linux)
system("ls -al")
```

As stated earlier, R features easy installation of both core R and third party packages. The function install.packages(X) installs the functions and datasets in the package X from the Internet. After installation the function library(X) brings the package into scope, thus making the functions and variables in the package X available to the programmer. This two-stage process mitigates potential

overlap in the namespaces of libraries. Typically, an R programmer would install many packages[3] on his or her computer, but have only a limited number in scope at any particular time. A list of available packages, their implementation and documentation are available at http://cran.r-project.org/web/packages/. These packages often contain interesting and demonstrative datasets. The function `data` lists the available datasets in a particular package.

```
# install package ggplot2
install.packages("ggplot2")
# install package from a particular mirror site
install.packages("ggplot2", repos="http://cran.r-project.org")
# install a package from source, rather than binary
install.packages("ggplot2", type = "source")
library('ggplot2')  # bring package into scope
# display all datasets in the package ggplot2
data(package = 'ggplot2')
installed.packages()  # display a list of installed packages
update.packages()  # update currently installed packages
```

R attempts to match a variable or function name by searching the current working environment followed by the packages that are in scope (loaded using the `library` function) with the earliest match used if there are multiple matches. The function `search` displays the list of packages that are being searched for a match and the search order with the first entry defaulting to the working environment, represented by `.GlobalEnv`. As a result, the working environment may mask variables or functions that are found further down the search path.

The code below demonstrates masking the built-in constant `pi` by a global environment variable with the value 3, and retrieving the original value after clearing it.

```
pi
## [1] 3.141593
pi = 3  # redefines variable pi
pi  # .GlobalEnv match
## [1] 3
rm(pi)  # removes masking variables
pi
## [1] 3.141593
```

The function `sink(outputFile)` records the output to the file `outputFile` instead of the display, which is useful for creating a log-file for later examination. To print the output both to screen and to a file, use the following variation.

```
sink(file = 'outputFile', split = TRUE)
```

We have concentrated thus far on executing R code interactively. To execute R code written in a text file `foo.R` (`.R` is the conventional filename extension for R code) use either

[3]The number of available packages is over 8000 in the year 2015; see Fig. 7.1 for the growth trajectory.

- the R function `source("foo.R")`,
- the command `R CMD BATCH foo.R` from a Linux or Mac terminal, or
- the command `Rscript foo.R` from a linux or R terminal.

It is also possible to save the R code as a shell script file whose first line corresponds to the location of the `Rscript` executable program, which in most cases is the following line.

```
#!/usr/bin/Rscript
```

The file can then be executed by typing its name in the Linux or Mac OS terminal (assuming it has executable permission). This option has the advantage of allowing Linux style input and output redirecting via `foo.R < inFile > outFile` and other shell tricks.

The last three options permit passing parameters to the script, for example using `R CMD BATCH -args arg1 arg2 foo.R` or `Rscript foo.R arg1 arg2`. Calling `commandArgs(TRUE)` inside a script retrieves the command line arguments as a list of strings.

7.3 Scalar Data Types

As we saw in the case of C++ and Java, a variable may refer to a scalar or a collection. Scalar types include numeric, integer, logical, string, dates, and factors. Numeric and integer variables represent real numbers and integers, respectively. A logical or binary variable is a single bit whose value in R is `TRUE` or `FALSE`. Strings are ordered sequences of characters. Dates represent calendar dates. Factor variables represent values from an ordered or unordered finite set. Some operations can trigger casting between the various types. Functions such as `as.numeric` can perform explicit casting.

```
a = 3.2; b = 3  # double types
b
## [1] 3
typeof(b)  # function returns type of object
## [1] "double"
c = as.integer(b)  # cast to integer type
c
## [1] 3
typeof(c)
## [1] "integer"
c = 3L  # alternative to casting: L specifies integer
d = TRUE
d
## [1] TRUE
e = as.numeric(d)  # casting to numeric
e
## [1] 1
f = "this is a string"  # string
```

```
f
## [1] "this is a string"
ls.str()  # show variables and their types
## a :  num 3.2
## b :  num 3
## c :  int 3
## d :  logi TRUE
## e :  num 1
## f :  chr "this is a string"
```

Factor variables assume values in a predefined set of possible values. The code below demonstrates the use of factors in R.

```
current.season = factor("summer",
          levels = c("summer", "fall", "winter", "spring"),
          ordered = TRUE)  # ordered factor
current.season
## [1] summer
## 4 Levels: summer < fall < ... < spring
levels(current.season)  # display factor levels
## [1] "summer" "fall"   "winter" "spring"
my.eye.color = factor(
    "brown",
    levels = c("brown", "blue", "green"),
    ordered = FALSE)  # unordered factor
my.eye.color
## [1] brown
## Levels: brown blue green
```

The value `NA` (meaning Not Available) denotes missing values. When designing data analysis functions, `NA` values should be carefully handled. Many functions feature the argument `na.rm` which, if `TRUE`, operates on the data after removing any `NA` values.

7.4 Vectors, Arrays, Lists, and Dataframes

Vectors, arrays, lists, and dataframes are collections that hold multiple scalar values.[4] A vector is a one-dimensional ordered collection of variables of the same type. An array is a multidimensional generalization of vectors of which a matrix is a two-dimensional special case. Lists are ordered collections of variables of potentially different types. The list signature is the ordered list of variable types in the list. A dataframe is an ordered collection of lists having identical same signature.

To refer to specific array elements use integers inside square brackets. For example, `A[3]` refers to the third element and `A[c(1, 2)]` refers to the first

[4]Formally, a numeric scalar in R is a vector of size 1 and thus it is not fundamentally different from a vector.

two elements. Negative integers inside the square bracket corresponds to a selection of all elements except for the specified positions, for example `A[-3]` refers to all elements but the third one. It is also possible to refer to array elements by passing a vector of boolean values with the selected elements corresponding to the `TRUE` values. For example, `A[c(FALSE, FALSE, TRUE)]` corresponds to the third element. If the boolean vector is shorter than the array length it will be recycled to be of the same length.

Below are some examples of creating and handling vectors and arrays.

```
# c() concatenates arguments to create a vector
x=c(4, 3, 3, 4, 3, 1)
x
## [1] 4 3 3 4 3 1
length(x)
## [1] 6
2*x+1  # element-wise arithmetic
## [1] 9 7 7 9 7 3
# Boolean vector (default is FALSE)
y = vector(mode = "logical", length = 4)
y
## [1] FALSE FALSE FALSE FALSE
# numeric vector (default is 0)
z = vector(length = 3, mode = "numeric")
z
## [1] 0 0 0
q = rep(3.2, times = 10) # repeat value multiple times
q
##  [1] 3.2 3.2 3.2 3.2 3.2 3.2 3.2 3.2 3.2 3.2
w=seq(0, 1, by = 0.1)  # values in [0,1] in 0.1 increments
w
##  [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9
## [11] 1.0
# 11 evenly spaced numbers between 0 and 1
w=seq(0, 1, length.out = 11)
w
##  [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9
## [11] 1.0
# create an array of booleans reflecting whether condition holds
w <= 0.5
##  [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
##  [7] FALSE FALSE FALSE FALSE FALSE
any(w <= 0.5) # is it true for some elements?
## [1] TRUE
all(w <= 0.5) # is it true for all elements?
## [1] FALSE
which(w <= 0.5) # for which elements is it true?
## [1] 1 2 3 4 5 6
w[w <= 0.5] # extracting from w entries for which w<=0.5
## [1] 0.0 0.1 0.2 0.3 0.4 0.5
subset(w, w <= 0.5) # an alternative with the subset function
## [1] 0.0 0.1 0.2 0.3 0.4 0.5
w[w <= 0.5] = 0 # zero out all components smaller or equal to 0.5
```

```
w
##  [1] 0.0 0.0 0.0 0.0 0.0 0.0 0.6 0.7 0.8 0.9
## [11] 1.0
```

Arrays are multidimensional generalizations of vectors; the `dim` attribute specifies the dimension. Matrices correspond to two-dimensional arrays. Referring to a specific array elements can be done by including the coordinates inside the square brackets (separated by commas), for example `A[1,2]` correspond to the element at the first row and second column of `A`. Leaving a specific dimension selection blank inside the square brackets corresponds to selecting the entire dimension. For example, `A[1,]` corresponds to the first row of `A`. As in the case of vectors, negative integers correspond to a selection of all but the selected coordinate.

```
z = seq(1, 20,length.out = 20)  # create a vector 1,2,..,20
x = array(data = z, dim = c(4, 5))  # create a 2-d array
x
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    5    9   13   17
## [2,]    2    6   10   14   18
## [3,]    3    7   11   15   19
## [4,]    4    8   12   16   20
x[2,3]  # refer to the second row and third column
## [1] 10
x[2,]  # refer to the entire second row
## [1]  2  6 10 14 18
x[-1,]  # all but the first row - same as x[c(2,3,4),]
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    2    6   10   14   18
## [2,]    3    7   11   15   19
## [3,]    4    8   12   16   20
y = x[c(1,2),c(1,2)]  # 2x2 top left sub-matrix
2 * y + 1  # element-wise operation
##      [,1] [,2]
## [1,]    3   11
## [2,]    5   13
y %*% y  # matrix product (both arguments are matrices)
##      [,1] [,2]
## [1,]   11   35
## [2,]   14   46
# inner product (both vectors have the same dimensions)
x[1,] %*% x[1,]
##      [,1]
## [1,]  565
t(x)  # matrix transpose
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8
## [3,]    9   10   11   12
## [4,]   13   14   15   16
## [5,]   17   18   19   20
outer(x[,1], x[,1])  # outer product
##      [,1] [,2] [,3] [,4]
```

```
## [1,]    1    2    3    4
## [2,]    2    4    6    8
## [3,]    3    6    9   12
## [4,]    4    8   12   16
rbind(x[1,], x[1,])  # vertical concatenation
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    5    9   13   17
## [2,]    1    5    9   13   17
cbind(x[1,], x[1,])  # horizontal concatenation
##      [,1] [,2]
## [1,]    1    1
## [2,]    5    5
## [3,]    9    9
## [4,]   13   13
## [5,]   17   17
```

We can access multidimensional array elements using a single index. The single index counts elements by traversing the array by columns, then rows, and then other dimensions where appropriate.

```
A = matrix(c(1, 2, 3, 4), nrow = 2, ncol = 2)
A
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
A[3]  # counting by columns A[3]=A[1,2]
## [1] 3
```

Lists are ordered collections which permit positions to hold variables of different types. For example, a list may hold a floating point in its first position, an integer in its second position, and a 3D array in its third position. Lists can also be elements of other lists. If `L` is a list, `L[[i]]` is the `i`-element, and `L[i]` is a list containing the `i` element.

To ease the tracking of the semantics of variables in lists or vectors, we typically assign names to the different positions. This is a form of self-describing data representation: there is no need to keep meta data describing what each position holds since that information is kept in the object itself.

```
L=list(name = 'John', age = 55,
       no.children = 2, children.ages = c(15, 18))
names(L)  # displays all position names
## [1] "name"          "age"
## [3] "no.children"   "children.ages"
L[[2]]  # second element
## [1] 55
L[2]  # list containing second element
## $age
## [1] 55
L$name  # value in list corresponding to name
## [1] "John"
L['name']  # same thing
```

```
## $name
## [1] "John"
L$children.ages[2]  # same as L[[4]][2]
## [1] 18
```

The function `unname(X)` removes names. Existing names can be changed by assigning a vector of strings to `names(X)`.

When using arithmetic operations between arrays of different sizes, the smaller array is extended as needed, with new elements created by recycling old ones. Similarly, storing a value in a non-existing element expands the array as needed, padding with `NA` values.

```
a = c(1, 2)
b = c(10, 20, 30, 40, 50)
a + b
## [1] 11 22 31 42 51
b[7] = 70
b
## [1] 10 20 30 40 50 NA 70
```

A dataframe is an ordered sequence of lists sharing the same signature. A dataframe often serves as a table whose rows correspond to data examples (samples from a multivariate distribution) and whose columns correspond to dimensions or features.

```
vecn = c("John Smith","Jane Doe")
veca = c(42, 45)
vecs = c(50000, 55000)
R = data.frame(name = vecn, age = veca, salary = vecs)
R
##         name age salary
## 1 John Smith  42  50000
## 2   Jane Doe  45  55000
names(R) = c("NAME", "AGE", "SALARY")  # modify column names
R
##         NAME AGE SALARY
## 1 John Smith  42  50000
## 2   Jane Doe  45  55000
```

The core R package `datasets` contains many interesting and demonstrative datasets, such as the `iris` dataset, whose first four dimensions are numeric measurements describing flower geometry, and whose last dimension is a string describing the flower species.

```
names(iris)  # lists the dimension (column) names
## [1] "Sepal.Length" "Sepal.Width"
## [3] "Petal.Length" "Petal.Width"
## [5] "Species"
head(iris, 4)  # show first four rows
##   Sepal.Length Sepal.Width Petal.Length
## 1          5.1         3.5          1.4
```

```
## 2          4.9         3.0          1.4
## 3          4.7         3.2          1.3
## 4          4.6         3.1          1.5
##   Petal.Width Species
## 1         0.2  setosa
## 2         0.2  setosa
## 3         0.2  setosa
## 4         0.2  setosa
iris[1,]  # first row
##   Sepal.Length Sepal.Width Petal.Length
## 1          5.1         3.5          1.4
##   Petal.Width Species
## 1         0.2  setosa
iris$Sepal.Length[1:10]  # sepal length of first ten samples
##  [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9
# allow replacing iris$Sepal.Length with the shorter Sepal.Length
attach(iris, warn.conflicts = FALSE)
mean(Sepal.Length)  # average of Sepal.Length across all rows
## [1] 5.843333
colMeans(iris[,1:4])  # means of all four numeric columns
## Sepal.Length  Sepal.Width Petal.Length
##     5.843333     3.057333     3.758000
##  Petal.Width
##     1.199333
```

The `subset` function is useful for extracting subsets of a dataframe.

```
# extract all rows whose Sepal.Length variable is less than 5
# and whose species is not setosa
subset(iris, Sepal.Length < 5 & Species != "setosa")
##     Sepal.Length Sepal.Width Petal.Length
## 58           4.9         2.4          3.3
## 107          4.9         2.5          4.5
##     Petal.Width    Species
## 58          1.0 versicolor
## 107         1.7  virginica
# count number of rows corresponding to setosa species
dim(subset(iris, Species == "setosa"))[1]
## [1] 50
```

The function `summary` provides a useful statistical summary of the different dataframe columns. R automatically determines whether the variables are numeric, such as `Sepal.Length`, or factors, such as `Species`. For numeric variables, the `summary` function displays the minimum, maximum, mean, median, and the 25th and 75th percentiles. For factor variables, the `summary` function displays the number of dataframe rows in each of the factor levels.

```
summary(iris)
##   Sepal.Length    Sepal.Width
##  Min.   :4.300   Min.   :2.000
##  1st Qu.:5.100   1st Qu.:2.800
##  Median :5.800   Median :3.000
##  Mean   :5.843   Mean   :3.057
```

```
##  3rd Qu.:6.400   3rd Qu.:3.300
##  Max.   :7.900   Max.   :4.400
##   Petal.Length    Petal.Width
##  Min.   :1.000   Min.   :0.100
##  1st Qu.:1.600   1st Qu.:0.300
##  Median :4.350   Median :1.300
##  Mean   :3.758   Mean   :1.199
##  3rd Qu.:5.100   3rd Qu.:1.800
##  Max.   :6.900   Max.   :2.500
##        Species
##  setosa    :50
##  versicolor:50
##  virginica :50
##
##
##
```

With appropriate formatting, we can create a dataframe using a text file. For example, we can load the following text file containing data into a dataframe in R using the `read.table(X, header=TRUE)` function (use `header=FALSE` if there is no header line containing column names).

```
Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
         5.1         3.5          1.4         0.2     setosa
         4.9         3.0          1.4         0.2     setosa
```

```
# read text file into dataframe
Iris=read.table('irisFile.txt', header = TRUE)
# same but from Internet location
Iris=read.table('http://www.exampleURL.com/irisFile.txt',
                header = TRUE)
```

We can examine and edit dataframes and other variables within a text editor or a spreadsheet-like environment using the `edit` function.

```
edit(iris)  # examine data as spreadsheet
iris = edit(iris)  # edit dataframe/variable
newIris = edit(iris)  # edit dataframe/variable but keep
                        original
```

7.5 If-Else, Loops, and Functions

The flow of control of R code is very similar to that of other programming languages. Below are some examples of if-else, loops, function definitions, and function calls.

```
a = 10; b = 5; c = 1
if (a < b) {
    d = 1
} else if (a == b) {
```

```
    d = 2
} else {
    d = 3
}
d
## [1] 3
```

The logical operators in R are similar to those in C++ and Java. Examples include `&&` for AND, `||` for OR, `==` for equality, and `!=` for inequality.

For-loops repeat for a prespecified number of times, with each loop assigning a different component of a vector to the iteration variable. Repeat-loops repeat until a `break` statement occurs. While-loops repeat until a `break` statement occurs or until the loop condition is not satisfied. The `next` statement aborts the current iteration and proceeds to the next iteration.

```
sm=0
# repeat for 100 iteration, with num taking values 1:100
for (num in seq(1, 100, by = 1)) {
    sm = sm + num
}
sm  # same as sum(1:100)
## [1] 5050
repeat {
    sm = sm - num
    num = num - 1
    if (sm == 0) break  # if sm == 0 then stop the loop
}
sm
## [1] 0
a = 1; b = 10
# continue the loop as long as b > a
while (b > a) {
    sm = sm + 1
    a = a + 1
    b = b - 1
}
sm
## [1] 5
```

Functions in R are similar to those in C++ and Java. When calling a function, the arguments flow into the parameters according to their order at the call site. Alternatively, arguments can appear out of order if the calling environment provides parameter names.

```
# parameter bindings by order
foo(10, 20, 30)
# (potentially) out of order parameter bindings
foo(y = 20, x = 10, z = 30)
```

Omitting an argument assigns the default value of the corresponding parameter.

```
# passing 3 parameters
foo(x = 10, y = 20, z = 30)
# x and y are missing and are assigned default values
foo(z = 30)
# in-order parameter binding with last two parameters missing
foo(10)
```

Out of order parameter bindings and default values simplify calling functions with long lists of parameters, when many of the parameters take default values.

```
# myPower(.,.) raises the first argument to the power
# of the second; the former is named bas and has
# a default value of 10; the latter is named pow
# and has a default value of 2
myPower = function(bas = 10, pow = 2) {
    return(bas ^ pow) # raise base to a power
}
myPower(2, 3)  # 2 is bound to bas and 3 to pow (in-order)
## [1] 8
# same binding as above (out-of-order parameter names)
myPower(pow = 3, bas = 2)
## [1] 8
myPower(bas = 3)  # default value of pow is used
## [1] 9
```

Since R passes variables by value, changing the passed arguments inside the function does not modify their respective values in the calling environment. Variables defined inside functions are local, and thus are unavailable after the function completes its execution. The returned value is the last computed variable or the one specified in a `return` function call. Returning multiple values can be done by returning a list or a dataframe.

```
x = 2
myPower2 = function(x) {x = x^2; return(x)}
y = myPower2(x)  # does not change x outside the function
x
## [1] 2
y
## [1] 4
```

It is best to avoid loops when programming in R. There are two reasons for this: simplifying code and computational speed-up. Many mathematical computations on lists, vectors, or arrays may be performed without loops using component-wise arithmetic. The code example below demonstrates the computational speedup resulting from replacing a loop with vectorized code.

```
a = 1:10
# compute sum of squares using a for-loop
c = 0
for (e in a) c = c + e^2
c
```

```
## [1] 385
# same operation using vector arithmetic
sum(a^2)
## [1] 385
# time comparison with a million elements
a = 1:1000000; c = 0
system.time(for (e in a) c = c+e^2)
##    user  system elapsed
##   0.431   0.003   0.443
system.time(sum(a^2))
##    user  system elapsed
##   0.004   0.002   0.007
```

Another way to avoid loops is to use the function `sapply`, which applies a function passed as a second argument to the list, data-frame, or vector that is passed as a first argument. This leads to simplified code, though the computational speed-up may not apply in the same way as it did above.

```
a = seq(0, 1 ,length.out = 10)
b = 0
c = 0
for (e in a) {
    b = b + exp(e)
}
b
## [1] 17.33958
c = sum(sapply(a, exp))
c
## [1] 17.33958
# sapply with an anonymous function f(x)=exp(x^2)
sum(sapply(a, function(x) {return(exp(x^2))}))
## [1] 15.07324
# or more simply
sum(sapply(a, function(x) exp(x^2)))
## [1] 15.07324
```

7.6 Interfacing with C++ Code

R is inherently an interpreted language; that is, R compiles each command at run time, resulting in many costly context switches and difficulty in applying standard compiler optimization techniques. Thus, R programs likely will not execute as efficiently as compiled programs[5] in C, C++, or Fortran.

The computational slowdown described above typically increases with the number of elementary function calls or commands. For example, R code that generates

[5]R does have a compiler that can compile R code to native code, but the resulting speedup is not very high.

two random matrices, multiplies them, and then computes eigenvalues typically will not suffer a significant slowdown compared to similar implementations in C, C++, or Fortran. The reason is that such R code calls routines that are programmed in FORTRAN or C++. On the other hand, R code containing many nested loops is likely to be substantially slower due to the interpreter overhead.

For example, consider the code below, which compares two implementations of matrix multiplication. The first uses R's internal matrix multiplication and the second implements it through three nested loops, each containing a scalar multiplication.

```
n = 100; nsq = n*n
# generate two random matrices
A = matrix(runif(nsq), nrow = n, ncol = n)
B = matrix(runif(nsq), nrow = n, ncol = n)
system.time(A%*%B)  # built-in matrix multiplication
##    user  system elapsed
##   0.001   0.000   0.001
matMult=function(A, B, n) {
    R=matrix(data = 0, nrow = n, ncol = n)
    for (i in 1:n)
      for (j in 1:n)
          for (k in 1:n)
              R[i,j]=R[i,j]+A[i,k]*B[k,j]
    return(R)
   }
# nested loops implementation
system.time(matMult(A, B, n))
##    user  system elapsed
##   1.644   0.010   1.689
```

The first matrix multiplication is faster by several orders of magnitude even for a relatively small $n = 100$. The key difference is that the built-in matrix multiplication runs compiled C code.

Clearly, it is better, if possible, to write R code containing relatively few loops and few elementary R functions. Since core R contains a rich library of elementary functions, one can often follow this strategy. In some cases, however, this approach is not possible. A useful heuristic in this case is to identify computational bottlenecks using a profiler, then re-implement the offending R code in a compiled language such as C or C++. The remaining R code will interface with the re-implemented bottleneck code via an external interface. Assuming that a small percent of the code is responsible for most of the computational inefficiency (as is often the case), this strategy can produce substantial speedups with relatively little effort.

We consider two techniques for calling compiled C/C++ code from R: the simpler `.C` function and the more complex `.Call` function. In both cases, we compile C/C++ code using the terminal command `R CMD SHLIB foo.c`, which invokes the C++ compiler and create a `foo.so` file containing the compiled code. The `.so` file can be loaded within R using the function `dynload('foo.so')` and then called using the functions `.C('foo', ...)` or `.Call('foo',`

`...`) (the remaining arguments contain R vectors, matrices, or dataframes that will be converted to pointers within the C/C++ code).

For example, consider the task of computing $\sum_{j=1}^{n}(a_j + i)^{b_j}$ for all $i, j = 1, \ldots, n$ given two vectors a and b of size n. The C code to compute the result appears below. The first two pointers point to arrays containing the vectors a and b, the third pointer points to the length of the arrays, and the last pointer points to the area where the results should appear. Note the presence of the pre-processor directive `include<R.h>`.

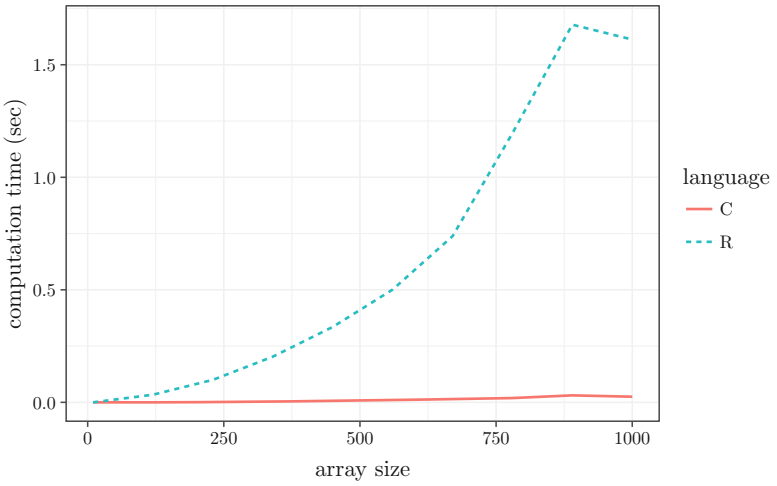
```
#include <R.h>
#include <math.h>

void fooC(double* a, double* b, int* n, double* res) {
  int i, j;
  for (i = 0;i < (*n); i++) {
    res[i] = 0;
    for (j = 0;j < (*n); j++)
      res[i] += pow(a[j] + i + 1, b[j]);
  }
}
```

Saving the code above as the file `fooC.c` and compiling using the terminal command below produce the file `fooC.so` that can be linked to an R session with the `dynload` function.

```
R CMD SHLIB fooC.c
```

```
dyn.load("fooC.so")  # load the compiled C code
A = seq(0, 1, length = 10)
B = seq(0, 1, length = 10)
C = rep(0, times = 10)
L = .C("fooC", A, B, as.integer(10), C)
ResC = L[[4]]  # extract 4th list element containing result
ResC
##  [1] 13.34392 17.48322 21.21480 24.70637
##  [5] 28.03312 31.23688 34.34390 37.37199
##  [9] 40.33396 43.23936
fooR = function(A, B, n) {
    res = rep(0, times = n)
    for (i in 1:n)
        for (j in 1:n)
            res[i] = res[i]+(A[j]+i)^(B[j])
    return(res)
}
ResR = fooR(A, B, 10)
ResR
##  [1] 13.34392 17.48322 21.21480 24.70637
##  [5] 28.03312 31.23688 34.34390 37.37199
##  [9] 40.33396 43.23936
sizes = seq(10, 1000, length = 10)
Rtime = rep(0, 10)
```

```
Ctime = Rtime
i = 1
for (n in sizes) {
    A = seq(0, 1,length = n)
    B = seq(0, 1,length = n)
    C = rep(0, times = n)
    Ctime[i] = system.time(.C("fooC", A, B, as.integer(n), C))
    Rtime[i] = system.time(fooR(A, B, n))
    i = i+1
}
DF = stack(list(C = Ctime, R = Rtime))
names(DF) = c("system.time", "language")
DF$size = sizes
# plot run time as a function of array size for R and
# .C implementations
qplot(x = size,
      y = system.time,
      lty = language,
  color = language,
      data = DF,
      size = I(1.5),
  geom = "line",
      xlab = "array size",
  ylab = "computation time (sec)")
```



The `.Call` external interface offers a higher degree of flexibility than `.C` in terms of passing and returning R arrays or dataframes. The example below shows how to use `.Call` to implement the same functionality as the `.C` example above. The type `SEXP` represents an R object. The `REAL` macro returns a pointer to the corresponding memory for reading or writing to it. A simple example appears below.

```
#include <R.h>
#include <Rinternals.h>
#include <Rmath.h>
#include <math.h>
SEXP fooC2(SEXP aR, SEXP  bR)
{
  int i, j, n = length(aR);
  double *a = REAL(aR), *b = REAL(bR);
  SEXP Rval = allocVector(REALSXP, n);
  for (i = 0; i < n; i++) {
    REAL(Rval)[i] = 0;
    for (j = 0; j < n; j++)
      REAL(Rval)[i] += pow(a[j] + i + 1, b[j]);
  }
  return Rval;
}
```

```
R CMD SHLIB fooC2.c
```

```
dyn.load("fooC2.so")  # load the compiled C code
A = seq(0, 1, length = 10)
B = seq(0, 1, length = 10)
.Call("fooC2", A, B)
```

7.7 Customization

When R starts it executes the function `.First` in the `.Rprofile` file in the user's home directory (if it exists). This is a good place to put user preferred options. Similarly, R executes the function `.Last` in the same file at the end of any R session. The function `options` adjust the behavior of R in many ways, for example to include more digits when displaying the value of a numeric variable.

Below are two very simple `.First` and `.Last` functions.

```
.First = function() {
    options(prompt = 'R >', digits = 6)
    library('ggplot2')
}
.Last = function() {
    cat(date(), 'Bye')
}
```

In Linux or Mac, we can execute R with flags; for example, `R -q` starts R without printing the initial welcome message.

7.8 Notes

R programming books include free resources such as the official introduction to R manual http://cran.r-project.org/doc/manuals/R-intro.html (replace html with pdf for pdf version) and the language reference http://cran.r-project.org/doc/manuals/R-lang.html (replace html with pdf for pdf version). Additional manuals on writing R extensions, importing data, and other topics are available at http://cran.r-project.org/doc/manuals/. Many additional books are available via commercial publishers. Google's programming style guide for R is available at https://google.github.io/styleguide/Rguide.xml and a variation of it written by Hadley Wickham (author of the `ggplot2` package) is available at http://adv-r.had.co.nz/Style.html.

R packages are available from http://cran.r-project.org. Each package features a manual in a common documentation format; many packages feature additional tutorial documents known as vignettes. Navigating the increasing repository of packages can be overwhelming. The Task Views help to aggregate lists of packages within a particular task or area. The list of Task Views is available at http://cran.r-project.org/web/views/. Another useful tool is http://crantastic.org which shows recently contributed or updated packages, user reviews, and ratings. Finally, the freely available R-Journal at http://journal.r-project.org/ contains high quality refereed articles on the R language, including many descriptions of contributed packages.

The Writing R Extensions manual (http://cran.r-project.org/doc/manuals/R-exts.pdf) contains more information on the `.C` and `.Call` external interfaces. The Rcpp package offers a higher degree of flexibility for interfacing C++ code and enables using numeric C++ libraries such as GSL, Eigen, and Armadillo.

Reference

J. Fox. Aspects of the social organization and trajectory of the R project. *The R Journal*, 1/2, 2009.

Chapter 8
Visualizing Data in R and Python

Visualizing data is key in effective data analysis: to perform initial investigations, to confirm or refuting data models, and to elucidate mathematical or algorithmic concepts. In this chapter, we explore different types of data graphs using the R programming language, which has excellent graphics functionality; we end the chapter with a description of Python's matplotlib module—a popular Python tool for data visualization.

8.1 Graphing Data in R

We focus on two R graphics packages: graphics and `ggplot2`. The graphics package contains the original R graphics functions and is installed and loaded by default. Its functions are easy to use and produce a variety of useful graphs. The `ggplot2` package provides alternative graphics functionality based on Wilkinson's grammar of graphics (Wilkinson, 2005). To install it and bring it to scope, type the following commands:

```
install.packages('ggplot2')
library(ggplot2)
```

When creating complex graphs, the `ggplot2` syntax is considerably simpler than the syntax of the graphics package. A potential disadvantage of `ggplot2` package is that rendering graphics using `ggplot2` may be substantially slower.

© Springer Nature Switzerland AG 2018

G. Lebanon, M. El-Geish, *Computing with Data*,
https://doi.org/10.1007/978-3-319-98149-9_8

8.2 Datasets

We use three datasets to explore data graphs. The `faithful` dataframe is a part of the datasets package that is installed and loaded by default. It has two variables: eruption time and waiting time to next eruption (both in minutes) of the Old Faithful geyser in Yellowstone National Park, Wyoming, USA. The code below displays the variable names and the corresponding summary statistics.

```
names(faithful)  # variable names
## [1] "eruptions" "waiting"
summary(faithful)  # variable summary
##    eruptions        waiting
##  Min.   :1.600   Min.   :43.0
##  1st Qu.:2.163   1st Qu.:58.0
##  Median :4.000   Median :76.0
##  Mean   :3.488   Mean   :70.9
##  3rd Qu.:4.454   3rd Qu.:82.0
##  Max.   :5.100   Max.   :96.0
```

The `mtcars` dataframe, which is also included in the `datasets` package, contains information concerning multiple car models extracted from 1974 Motor Trend magazine. The variables include model name, weight, horsepower, fuel efficiency, and transmission type.

```
summary(mtcars)
##       mpg             cyl
##  Min.   :10.40   Min.   :4.000
##  1st Qu.:15.43   1st Qu.:4.000
##  Median :19.20   Median :6.000
##  Mean   :20.09   Mean   :6.188
##  3rd Qu.:22.80   3rd Qu.:8.000
##  Max.   :33.90   Max.   :8.000
##       disp             hp
##  Min.   : 71.1   Min.   : 52.0
##  1st Qu.:120.8   1st Qu.: 96.5
##  Median :196.3   Median :123.0
##  Mean   :230.7   Mean   :146.7
##  3rd Qu.:326.0   3rd Qu.:180.0
##  Max.   :472.0   Max.   :335.0
##       drat             wt
##  Min.   :2.760   Min.   :1.513
##  1st Qu.:3.080   1st Qu.:2.581
##  Median :3.695   Median :3.325
##  Mean   :3.597   Mean   :3.217
##  3rd Qu.:3.920   3rd Qu.:3.610
##  Max.   :4.930   Max.   :5.424
##       qsec             vs
##  Min.   :14.50   Min.   :0.0000
##  1st Qu.:16.89   1st Qu.:0.0000
##  Median :17.71   Median :0.0000
##  Mean   :17.85   Mean   :0.4375
##  3rd Qu.:18.90   3rd Qu.:1.0000
```

```
##  Max.   :22.90   Max.   :1.0000
##        am              gear
##  Min.   :0.0000   Min.   :3.000
##  1st Qu.:0.0000   1st Qu.:3.000
##  Median :0.0000   Median :4.000
##  Mean   :0.4062   Mean   :3.688
##  3rd Qu.:1.0000   3rd Qu.:4.000
##  Max.   :1.0000   Max.   :5.000
##       carb
##  Min.   :1.000
##  1st Qu.:2.000
##  Median :2.000
##  Mean   :2.812
##  3rd Qu.:4.000
##  Max.   :8.000
```

The `mpg` dataframe is a part of the `ggplot2` package and it is similar to `mtcars` in that it contains fuel economy and other attributes, but it is larger and it contains newer car models extracted from the website http://fueleconomy.gov. The columns `cty` and `hwy` record city and highway miles per gallon; `displ` records engine displacement in liters; `drv` records the drivetrain (front wheel, rear wheel, or four wheel); `model` records the car model, and `class` records the class of the car (SUV, compact, etc.).

```
names(mpg)
##  [1] "manufacturer" "model"
##  [3] "displ"        "year"
##  [5] "cyl"          "trans"
##  [7] "drv"          "cty"
##  [9] "hwy"          "fl"
## [11] "class"
```

More information on any of these datasets may be obtained by typing `help(X)` with `X` corresponding to the dataframe name when the appropriate package is in scope.

8.3 Graphics and `ggplot2` Packages

The `graphics` package contains two types of functions: high-level functions and low-level functions. High-level functions produce a graph, while low-level functions modify an existing graph. The primary high-level function, `plot`, takes as arguments one or more dataframe columns representing data and other arguments that modify its default behavior (some examples appear below).

Other high-level functions in the `graphics` package are more specialized and produce a specific type of graph, such as `hist` for producing histograms, or `curve` for producing curves. We do not explore many high-level functions as they are generally less convenient to use than the corresponding functions in the `ggplot2` package.

Examples of low-level functions in the `graphics` package are:

- `title` adds or modifies labels of title and axes,
- `grid` adds a grid to the current figure,
- `legend` displays a legend connecting symbols, colors, and line-types to descriptive strings, and
- `lines` adds a line plot to an existing graph.

For example, the following code displays a scatter plot of the columns of a hypothetical dataframe `df` containing two variables `col1` and `col2` using the `graphics` package and then adds a title to the graph.

```
plot(x = df$col1, y = df$col2)
title(main = "figure title")  # add title
```

The two main functions in the `ggplot2` package are `qplot` and `ggplot`. The `qplot` function accepts as arguments one or two data variables assigned to the variables `x` and `y`, for example `qplot(x=df$col1, y=df$col2)` or alternatively `qplot(x=col1, y=col2, data=df)`. Alternatively, we can use the `ggplot` function that accepts as arguments a dataframe and an object returned by the `aes` function which accepts data variables as arguments, for example `ggplot(df, aes(col1, col2))`.

In contrast to `qplot`, `ggplot` does not create a graph and returns instead an object that may be modified by adding layers to it using the + operator. After appropriate layers are added the object may be saved to disk or printed. The layer addition functionality applies to `qplot` as well.

The `ggplot2` package provides automatic axes labeling and legends. To take advantage of this feature the data must reside in a dataframe with informative column names. We emphasize this approach as it provides more informative dataframes column names, in addition to simplifying the R code.

For example, the `ggplot2` package code below is analogous to the `graphics` package code above.

```
# using qplot
qplot(x = col1, y = col2, data = df, geom = "point") +
  ggtitle("figure title")
# same thing but with ggplot2
ggplot(df, aes(x = col1, y = col2)) +
  geom_point() + ggtitle("figure title")
```

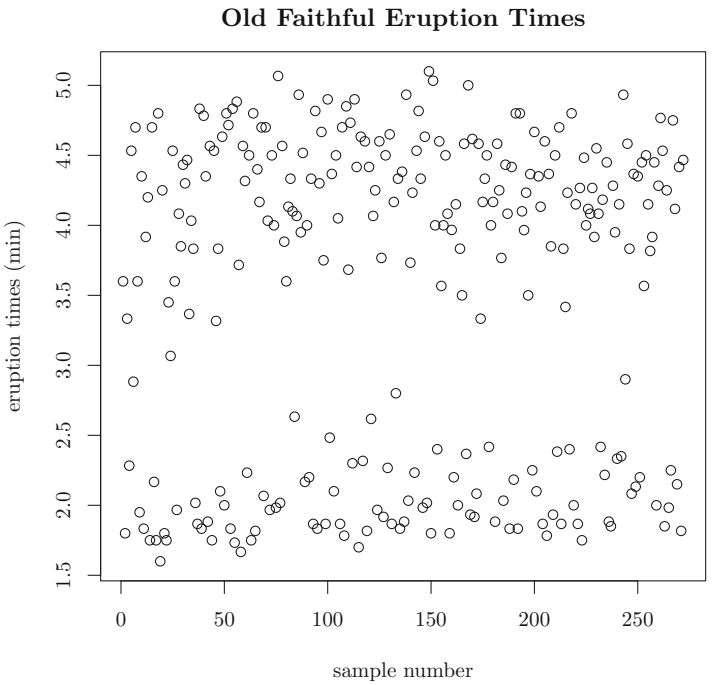
In the following sections we describe several different types of data graphs.

8.4 Strip Plots

The simplest way to graph one-dimensional numeric data is to graph them as points in a two-dimensional space, with one coordinate corresponding to the index of the data point, and the other coordinate corresponding to its value.

To plot a strip plot using the graphics package, call `plot` with a single numerical dataframe column. The resulting x-axis indicates the row number, and the y-axis indicates the numeric value. The `xlab`, `ylab`, and `main` parameters modify the x-label title, y-label title, and figure title.

```
plot(faithful$eruptions,
     xlab = "sample number",
     ylab = "eruption times (min)",
     main = "Old Faithful Eruption Times")
```



We conclude from the figure above that Old Faithful has two typical eruption times—a long eruption time around 4.5 min, and a short eruption time around 1.5 min. It also appears that the order in which the dataframe rows are stored is not related to the eruption variable.

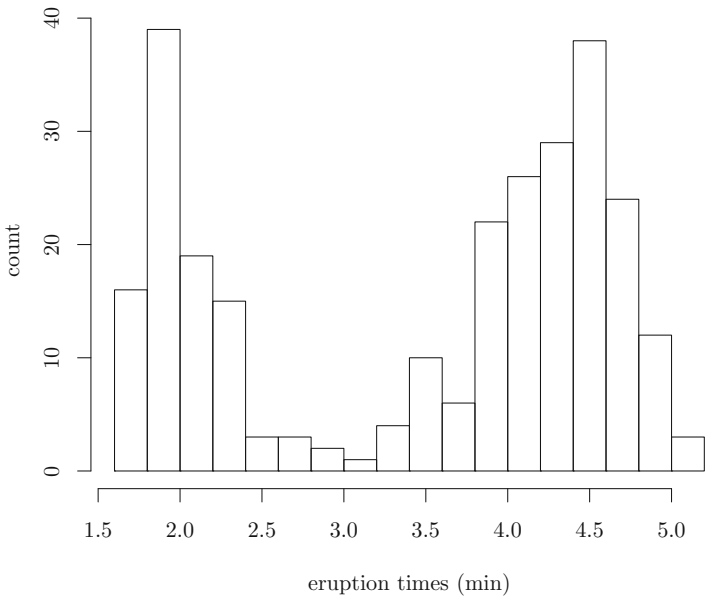
8.5 Histograms

An alternative way to graph one-dimensional numeric data is using the histogram graph. The histogram divides the range of numeric values into bins and displays the number of data values falling within each bin. The width of the bins influences

the level of detail. Very narrow bins maintain all the information present in the data but are hard to draw conclusions from, as the histogram becomes equivalent to a sorted list of data values. Very wide bins lose information due to overly aggressive smoothing. A good bin width balances information loss with useful data aggregation. Unlike strip plots that contain all of the information present in the data, histograms discard the ordering of the data points and treat samples in the same bin as identical.

The `hist(data, breaks = num_bins)` function within the graphics package can be used to display a histogram. The `xlab`, `ylab`, and `main` parameters described in Sect. 8.4 can be added as optional parameters to `hist`. See `help(hist)` for more details on the different parameters, and in particular for assistance on controlling the bin widths. For example, the code below displays a histogram of eruption times of Old Faithful using 20 bins.

```
hist(faithful$eruptions,
     breaks = 20,
     xlab = "eruption times (min)",
     ylab = "count",
     main = "")
```

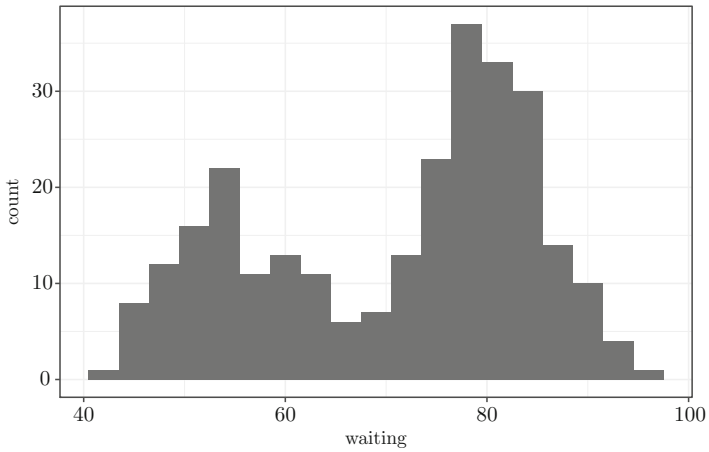


We see a nice correspondence between the above histogram and the strip plot in Sect. 8.4. There are clearly two typical eruption times—one around 2 min and one around 4.5 min.

To graph a histogram with the `ggplot2` package, call `qplot` with two parameters: a dataframe column (assigned to the `x` argument) and a name of the

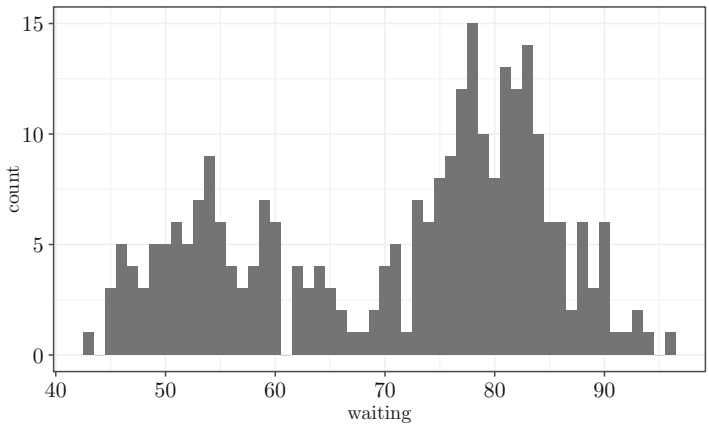
dataframe variable (assigned to the `data` argument). The two axes are automatically labeled based on the names of the variables that they represent. For example, the code below displays a histogram of the waiting time variable using `qplot`.

```
qplot(x = waiting,
      data = faithful,
      binwidth = 3,
      main = "Waiting time to next eruption (min)")
```



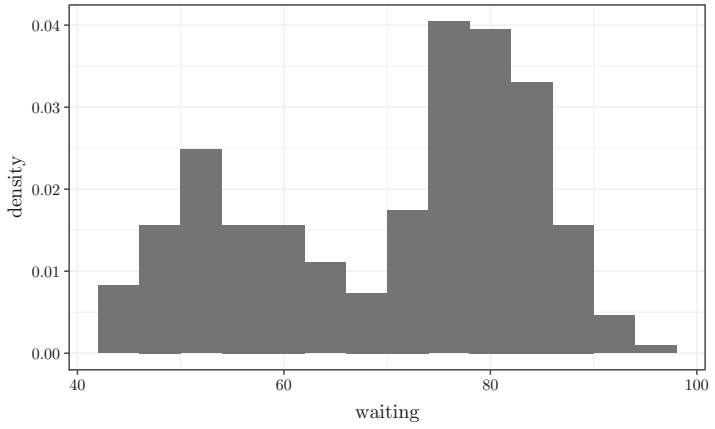
To create a histogram with the `ggplot` function, we pass an object returned from the `aes` function, and add a histogram geometry layer using the `+` operator.

```
ggplot(faithful, aes(x = waiting)) +
  geom_histogram(binwidth = 1)
```



Traditionally, the y axis in a histogram displays counts. An alternative is to display the frequency by surrounding the data variable with `..` on both sides. In this case, the height of each bin times its width equals the count of samples falling in the bin divided by the total number of counts. As a result, the area under the graph is 1 making it a legitimate probability density function (see TAOD volume 1, Chapter 2 for a description of the probability density function). This probability interpretation is sometimes advantageous, but it may be problematic in that it masks the total number of counts.

```
ggplot(faithful, aes(x = waiting, y = ..density..)) +
    geom_histogram(binwidth = 4)
```



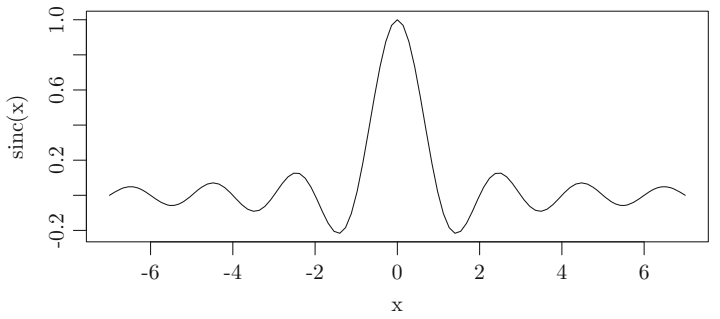
Note that selecting a wider bandwidth (as in the figure above) produces a smoother histogram as compared to the figure before that. Selecting the best bandwidth to use when graphing a specific dataset is difficult and usually requires some trial and error.

8.6 Line Plots

A line plot is a graph displaying a relation between x and y as a line in a Cartesian coordinate system. The relation may correspond to an abstract mathematical function or to relation present between two variables in a specific dataset.

The function `curve` in the graphics package displays mathematical functions. In the example below, the first line defines a new function called `sinc` while the second line plots it. Note the automatic labeling of the axes.

```
sinc = function(x) {
  return(sin(pi * x) / (pi * x))
}
curve(sinc, -7, +7)
```



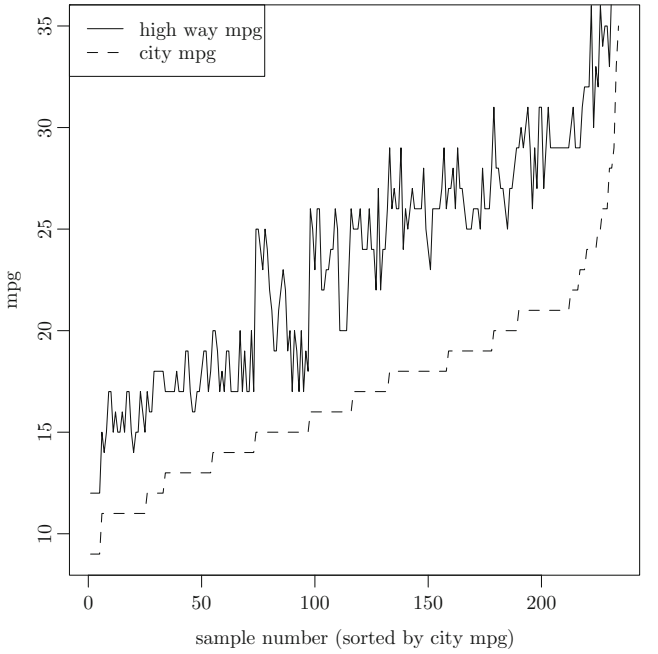
Another option to display a line plot with the graphics package is to use `plot` but with a `type="l"` parameter, as below. The variable `lty` allows us to display different line types (dashed, dotted, etc.). We demonstrate this below by plotting hwy mpg and city mpg as line plots, where the samples are sorted by city mpg.

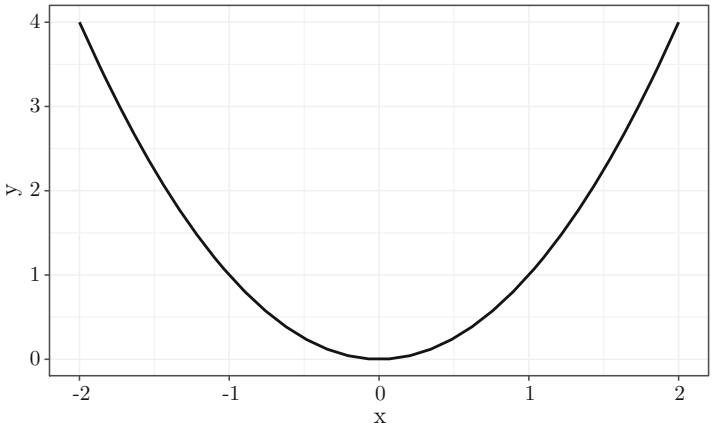
```
S = sort.int(mpg$cty, index.return = T)
#  S$x holds the sorted values of city mpg
#  S$ix holds the indices of the sorted values of city mpg
#  First plot the sorted city mpg values with a line plot
plot(S$x,
     type = "l",
     lty = 2,
     xlab = "sample number (sorted by city mpg)",
     ylab = "mpg")
# add dashed line of hwy mpg
lines(mpg$hwy[S$ix] ,lty = 1)
legend("topleft", c("highway mpg", "city mpg"), lty = c(1, 2))
```

We can conclude from the plot above that (i) highway mpg tends to be higher than city mpg, (ii) highway mpg tends to increase as city mpg increases, and (iii) the difference between the two quantities is less significant for cars with lower fuel efficiency.

The `qplot` function creates a line plot when passed a `geom=line` parameter.

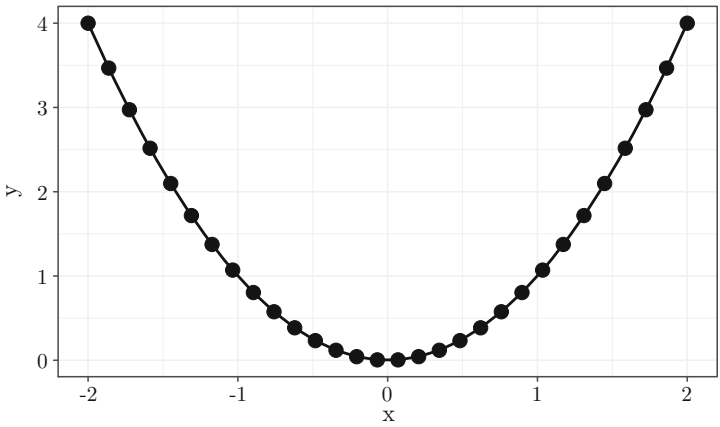
```
x = seq(-2, 2, length.out = 30)
y = x^2
qplot(x, y, geom = "line")
```





Below is a similar example where multiple geometries are present.

```
x = seq(-2, 2, length.out = 30)
y = x^2
qplot(x, y, geom = c("point", "line"))
```



The function `ggplot` creates the same plot by adding a line geometry layer `geom_line()` using the + operator.

```
# new data frame with variables x,y = x^2
dataframe = data.frame(x = x, y = y)
ggplot(dataframe, aes(x = x, y = y)) + geom_line()
       + geom_point()
```

8.7 Smoothed Histograms

An alternative to the histogram is the smoothed histogram. Denoting n data points by $x^{(1)}, \ldots, x^{(n)}$, the smoothed histogram is the following function $f_h : \mathbb{R} \to \mathbb{R}_+$

$$f_h(x) = \frac{1}{n} \sum_{i=1}^{n} K_h(x - x^{(i)})$$

where the kernel function $K_h : \mathbb{R} \to \mathbb{R}$ typically achieves its maximum at 0, and decreases as $|x - x^{(i)}|$ increases. We also assume that the kernel function integrates to one $\int K_h(x)\, dx = 1$ and satisfies the relation

$$K_h(r) = h^{-1} K_1(r/h).$$

We refer to K_1 as the base form of the kernel and denote it as K.

Four popular kernel choices are the tricube, triangular, uniform, and Gaussian kernels, defined as $K_h(r) = h^{-1} K(r/h)$ where the $K(\cdot)$ functions are respectively

$$K(r) = (1 - |r|^3)^3 \cdot 1_{\{|r|<1\}} \qquad \text{(Tricube)}$$

$$K(r) = (1 - |r|) \cdot 1_{\{|r|<1\}} \qquad \text{(Triangular)}$$

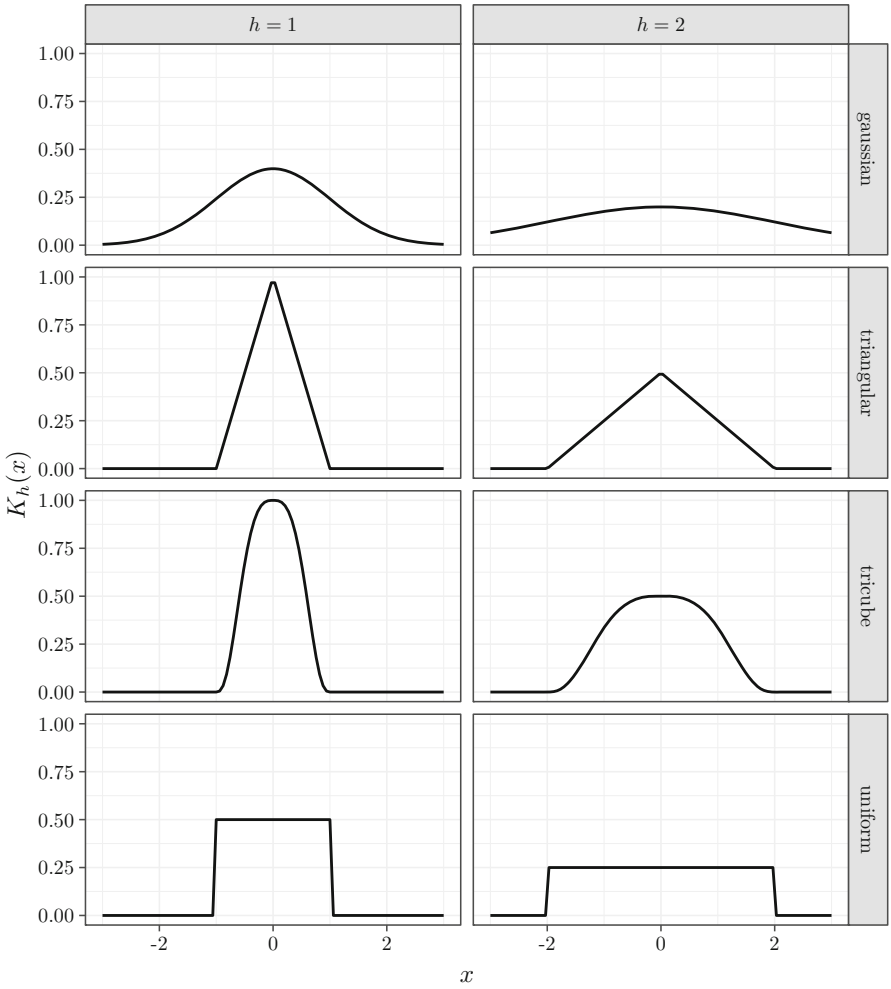
$$K(r) = 2^{-1} \cdot 1_{\{|r|<1\}} \qquad \text{(Uniform)}$$

$$K(r) = \exp(-x^2/2)/\sqrt{2\pi} \qquad \text{(Gaussian).}$$

The following R code displays these kernels for $h = 1$ and $h = 2$. Note how the kernel corresponding to $h = 2$ is twice as wide as the kernels corresponding to $h = 1$. The technique used to display multiple panels in the same graph is called faceting and is described in the next section.

```
x_grid = seq(-3, 3, length.out = 100)
K1 = function(x) {
  ind = abs(x) > 1
  x = x * 0 + 1/2
  x[ind] = 0
  return(x)
}
K2 = function(x) {
  ind = abs(x) > 1
  x = 1 - abs(x)
  x[ind] = 0
  return(x)
}
K3 = function(x) dnorm(x)
K4 = function(x) {
  ind = abs(x) > 1
  x = (1 - abs(x)^3)^3
  x[ind] = 0
  return(x)
}
R = stack(list('uniform' = K1(x_grid),
               'triangular' = K2(x_grid),
               'gaussian' = K3(x_grid),
               'tricube' = K4(x_grid),
               'uniform' = K1(x_grid / 2) / 2,
               'triangular' = K2(x_grid / 2) / 2,
               'gaussian' = K3(x_grid / 2) / 2,
               'tricube' = K4(x_grid / 2) / 2))
head(R)  # first six lines
##   values     ind
## 1      0 uniform
## 2      0 uniform
## 3      0 uniform
## 4      0 uniform
## 5      0 uniform
## 6      0 uniform
names(R) = c('kernel.value', 'kernel.type')
R$x = x_grid
R$h[1:400] = '$h=1$'
R$h[401:800] = '$h=2$'
head(R)  # first six lines
##   kernel.value kernel.type         x     h
## 1            0     uniform -3.000000 $h=1$
```

```
## 2             0     uniform -2.939394 $h=1$
## 3             0     uniform -2.878788 $h=1$
## 4             0     uniform -2.818182 $h=1$
## 5             0     uniform -2.757576 $h=1$
## 6             0     uniform -2.696970 $h=1$
qplot(x,
      kernel.value,
      data = R,
      facets = kernel.type~h,
      geom = "line",
      xlab = "$x$",
      ylab = "$K_h(x)$")
```



Above, we use LaTeX code (text strings containing $ symbols that surround equation code in the example above) to annotate the axes labels or titles with equations.

Identifying the functions $g_i(x) \stackrel{\text{def}}{=} K_h(x - x^{(i)})$ we see that f_h is an average of the g_i functions, $i = 1, \ldots, n$. Since the functions $g_i(x)$ are centered at $x^{(i)}$ and decay with the distance between x and $x^{(i)}$, their average f_h will be high in areas containing many data points and low in areas containing a few data points. The role of the denominator n in f_h is to ensure that the integral of f_h is 1, making f_h a formal estimator of the underlying distribution (see TAOD volume 1, chapter 2).

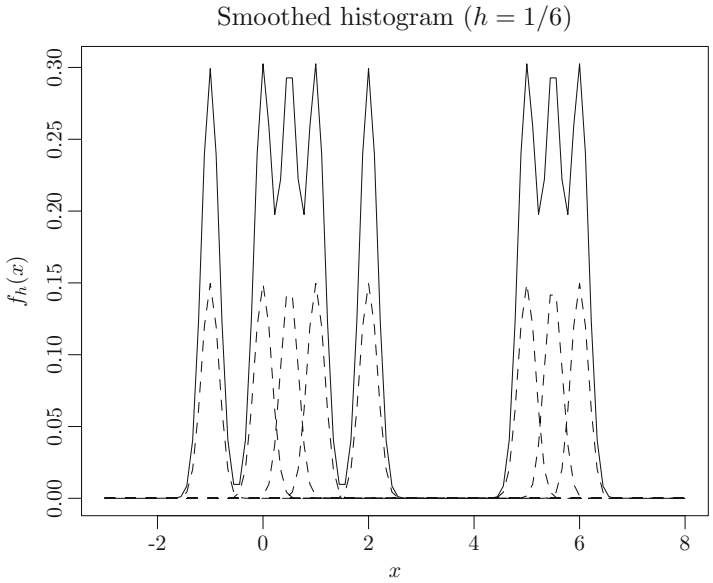
The R code below graphs the smoothed histogram of the data

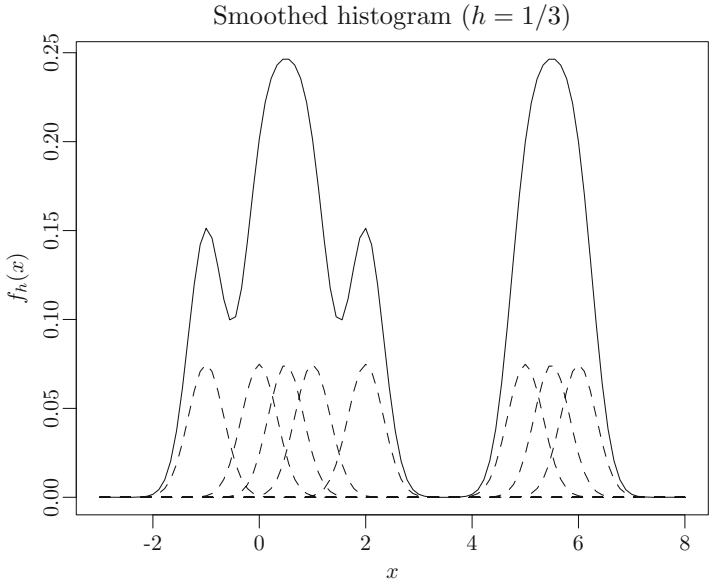
$$\{-1, 0, 0.5, 1, 2, 5, 5.5, 6\}$$

using the Gaussian kernel. The graphs show f_h as a solid line and the g_i functions as dashed lines (scaled down by a factor of 2 to avoid overlapping solid and dashed lines).

In the first graph below, the h value is relatively small ($h = 1/6$), resulting in a f_h close to a sequence of narrow spikes centered at the data points. In the second graph h is larger ($h = 1/3$) showing a multimodal shape that is significantly different from the first case. In the third case, h is relatively large ($h = 1$), resulting in a f_h that resembles two main components. For larger h, f_h will resemble a single unimodal shape.

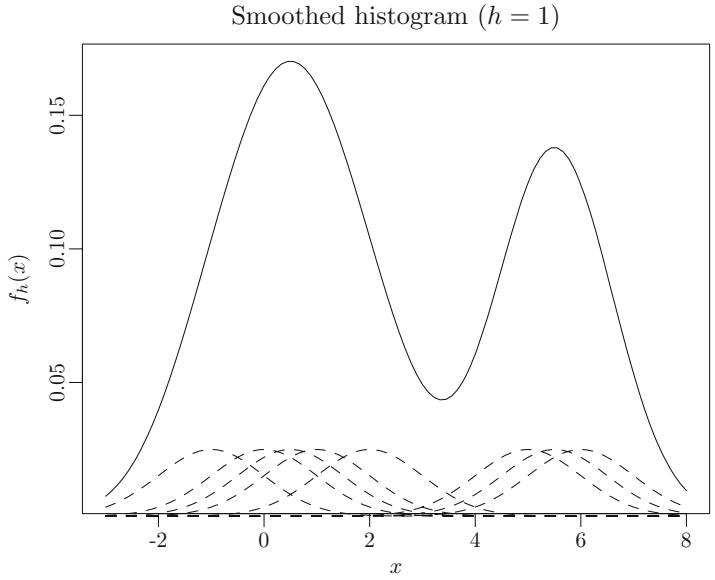
```
data = c(-1, 0, 0.5, 1, 2, 5, 5.5, 6)
data_size = length(data)
x_grid = seq(-3, data_size, length.out = 100)
kernel_values = x_grid %o% rep(1, data_size)
f = x_grid * 0
for(i in 1:data_size) {
  kernel_values[,i] = dnorm(x_grid, data[i], 1/6)/data_size
  f = f + kernel_values[,i]
}
plot(x_grid, f, xlab = "$x$", ylab = "$f_h(x)$", type = "l")
for (i in 1:data_size)
    lines(x_grid, kernel_values[,i]/2, lty = 2)
title("Smoothed histogram ($h=1/6$)", font.main = 1)


f = x_grid * 0
for(i in 1:data_size) {
  kernel_values[,i] = dnorm(x_grid, data[i], 1/3)/data_size
  f = f + kernel_values[,i]
}
plot(x_grid, f, xlab = "$x$", ylab = "$f_h(x)$", type = "l")
for (i in 1:data_size)
    lines(x_grid, kernel_values[,i]/2, lty = 2)
title("Smoothed histogram ($h=1/3$)", font.main = 1)
```






```
f = x_grid * 0
for(i in 1:data_size) {
  kernel_values[,i] = dnorm(x_grid, data[i], 1)/data_size
  f = f + kernel_values[,i]
}
plot(x_grid, f, xlab = "$x$" , ylab = "$f_h(x)$", type = "l")
```

```
for (i in 1:data_size)
    lines(x_grid, kernel_values[,i]/2, lty = 2)
title("Smoothed histogram ($h=1$)", font.main = 1)
```



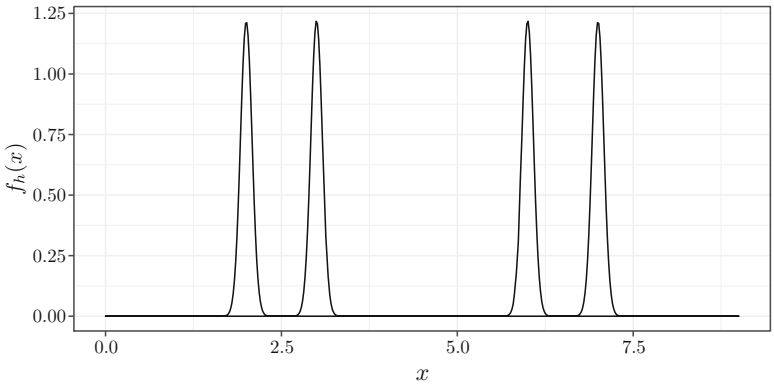
The smoothed histogram is often more intuitive and informative than the non-smoothed histogram. Just as the bin-width selection is crucial for obtaining an informative histogram, selecting the kernel function is very important. Selecting a value h that is too large will have the effect of over-smoothing, causing the resulting $\hat{f}$ to be nearly constant. On the other hand, selecting a value of h that is too small will have an under-smoothing effect, and result in a wiggly and noisy curve.

The `ggplot2` package incorporates smoothed histogram graphing into the `qplot` and `ggplot` functions. The value of h is controlled via the `adjust` parameter that assigns h to be the corresponding multiple of an automatic value determined by R. For example, setting `adjust=1` uses R's automatic h value, while setting `adjust=2` multiplies R's automatic h value by 2.

In the first graph below we use a small h value, causing f_h to exhibit four clearly separated peaks—each corresponding to a separate g_i function. This choice of a small h corresponds to a histogram with a very narrow bin-width.

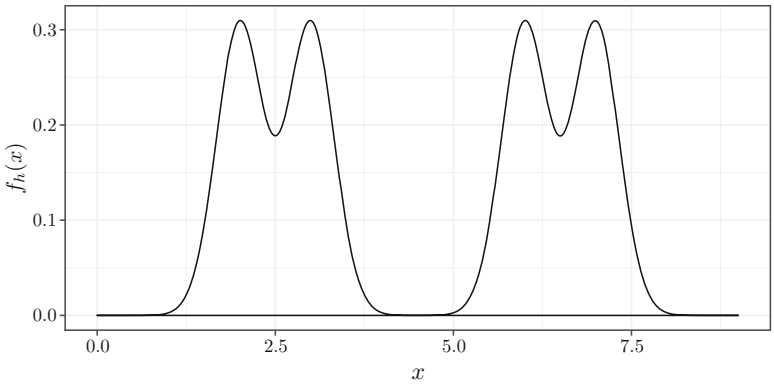
```
qplot(x = c(2, 3, 6, 7),
      y = ..density..,
      geom = c("density"),
      kernel = "gaussian",
      adjust = 0.05,
      xlab = "$x$",
```

```
       ylab = "$f_h(x)$",
       xlim = c(0, 9))
```



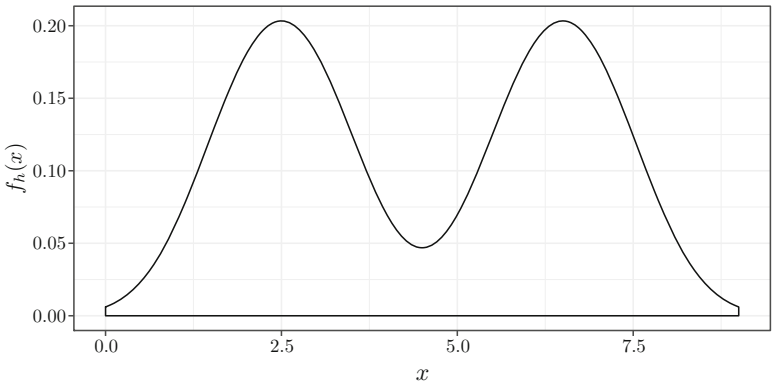
Increasing the value of h by increasing the `adjust` parameter diffuses the g_i functions, causing them to overlap more.

```
qplot(x = c(2,3,6,7),
      y = ..density..,
      geom = c("density"),
      kernel = "gaussian",
      adjust = 0.2,
      xlab = "$x$",
      ylab = "$f_h(x)$",
      xlim = c(0,9))
```



Increasing the value of h further aggregates the four peaks into two peaks, each responsible for a corresponding pair of nearby points.

```
qplot(x = c(2, 3, 6, 7),
      y = ..density..,
      geom = c("density"),
      kernel = "gaussian",
      adjust = 0.5,
      xlab = "$x$",
      ylab = "$f_h(x)$",
      xlim = c(0,9))
```

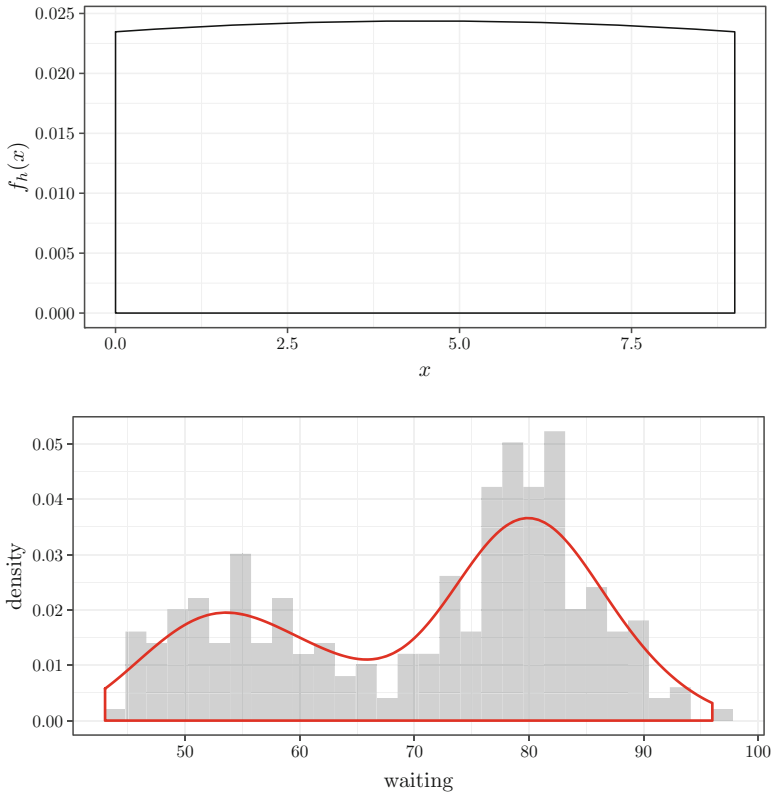


Finally, further increasing the value of h results in nearly constant g_i functions and a nearly constant $\hat{f}$ function. This corresponds to a histogram with very wide bins in which all points fall in the same bin.

```
qplot(x = c(2, 3, 6, 7),
      y = ..density..,
      geom = c("density"),
      kernel = "gaussian",
      adjust = 10,
      xlab = "$x$",
      ylab = "$f_h(x)$",
      xlim = c(0,9))
```

The figure below contrasts a histogram with a smoothed histogram using the `ggplot` function. To enhance the visualization, we made the histogram semi-transparent using the `alpha` argument that takes a value between 0 and 1 indicating the transparency level.

```
ggplot(faithful, aes(x = waiting, y = ..density..)) +
  geom_histogram(alpha = 0.3) +
  geom_density(size = 1.5, color = "red")
```



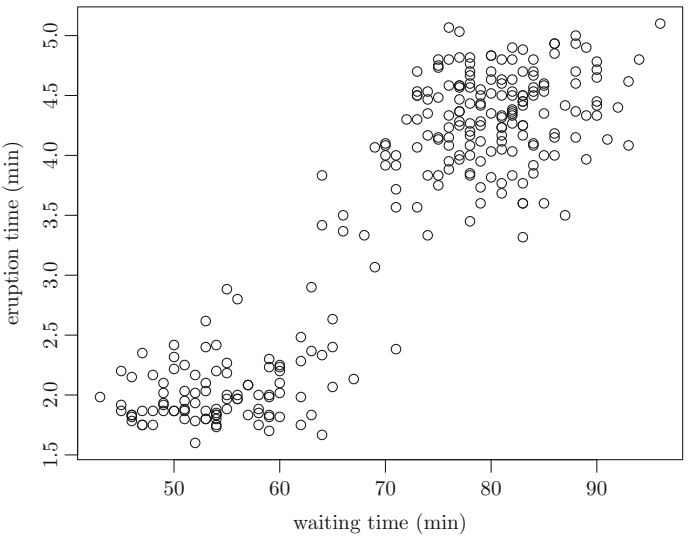
As is apparent from the figure above, smoothed histograms do a better job of uncovering the mathematical relationship between the variable and the frequency of density. This is backed mathematically by the fact that the smoothed histogram is a better estimator for the underlying density than the histogram. On the other hand, histograms feature a more direct relationship between the graph and the underlying data that is sometimes important in its own right.

8.8 Scatter Plots

A scatter plot graphs the relationships between two numeric variables. It graphs each pair of variables as a point in a two-dimensional space whose coordinates are the corresponding x, y values.

To create a scatter plot with the graphics package, call `plot` with two dataframe columns.

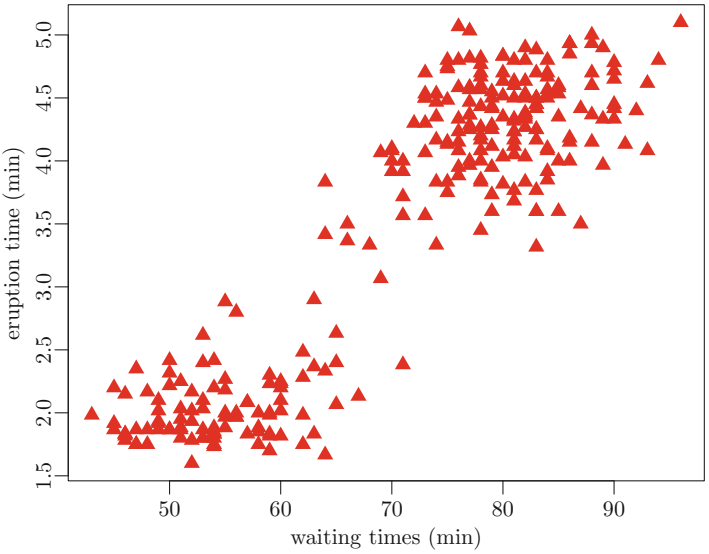
```
plot(faithful$waiting,
     faithful$eruptions,
     xlab = "waiting time (min)",
     ylab = "eruption time (min)")
```



We conclude from the two clusters in the scatter plot above that there are two distinct cases: short eruptions and long eruptions. Furthermore, the waiting times for short eruptions are typically short, while the waiting times for the long eruptions are typically long. This is consistent with our intuition: it takes longer to build the pressure for a long eruption than it does for a short eruption.

The points above are graphed using hollow circular markers. The arguments `pch`, `col`, and `cex` modify the marker's shape, color, and size, respectively. Type `help(pch)` for more information on setting these values.

```
plot(faithful$waiting,
     faithful$eruptions,
     pch = 17,
     col = 2,
     cex = 1.2,
     xlab = "waiting times (min)",
     ylab = "eruption time (min)")
```

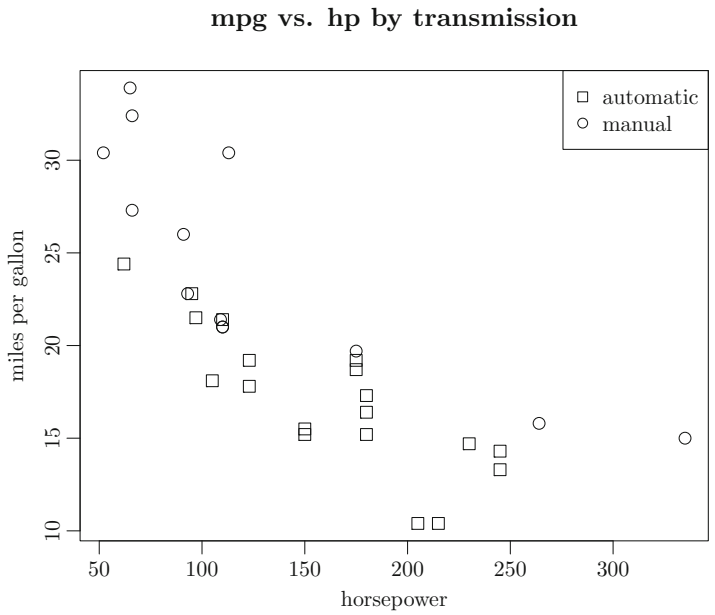


In some cases we wish to plot a scatter plot of one dataframe column vs. another, but distinguish the points based on the value of another dataframe column, typically a factor variable (factors in R take values in an ordered or unordered finite set—see Chap. 7). For example, the plot below shows horsepower vs. mile per gallon of cars within the `mtcars` dataset, but distinguishes between automatic and manual transmission using different symbols. Transmission types are encoded through the `am` variable, which takes values 0 or 1—both legitimate values for the `pch` marker shape argument.

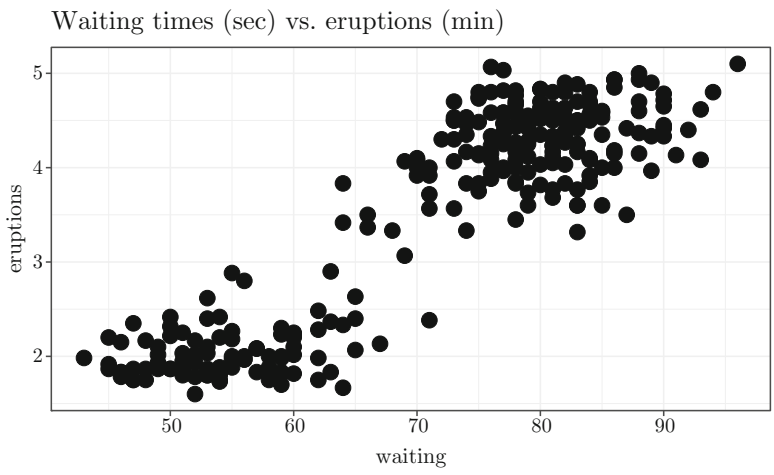
```
plot(mtcars$hp,
     mtcars$mpg,
     pch = mtcars$am,
     xlab = "horsepower",
     cex = 1.2,
     ylab = "miles per gallon",
     main = "mpg vs. hp by transmission")
legend("topright", c("automatic", "manual"), pch = c(0, 1))
```

We draw several conclusions from this graph. First, there is an inverse relationship between horsepower and mpg. Second, for a given horsepower amount, manual transmission cars are generally more fuel-efficient. Third, cars with the highest horsepower tend to be manual. Indeed, the two highest horsepower cars in the dataset are Maserati Bora and Ford Pantera, both sports cars with manual transmissions.

To graph a scatter plot with `qplot`, call it with two dataframe column parameters assigned to the `x` and `y` arguments.

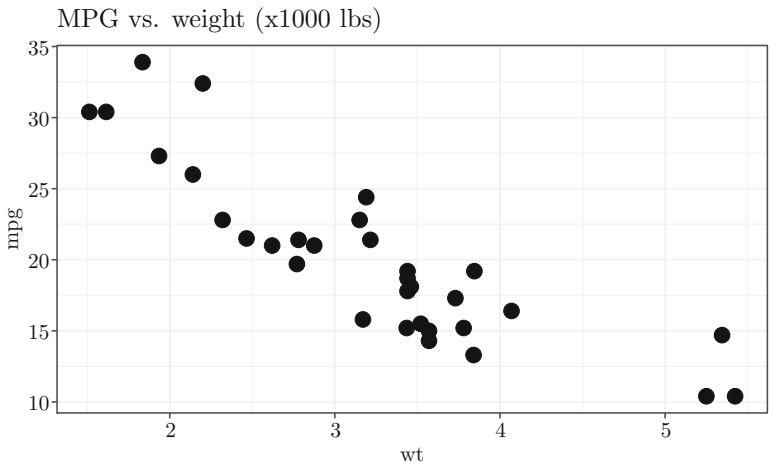



```
qplot(x = waiting,
      y = eruptions,
      data = faithful,
      main = "Waiting times (sec) vs. eruptions (min)")
```



The graph below shows a scatter plot of car weights vs mpg.

```
qplot(x = wt,
      y = mpg,
      data = mtcars,
      main = "MPG vs. weight (x1000 lbs)")
```



We conclude from the figure above that there exists a somewhat linear trend with negative slope between mpg and weight (though that trend decreases for heavier cars).

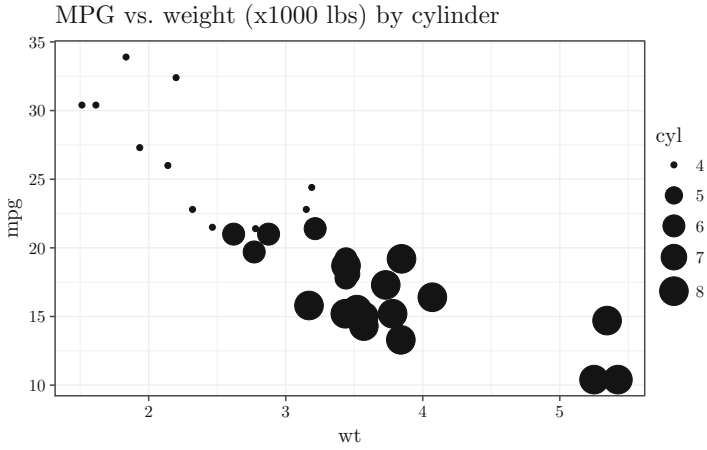
Denoting the number of cylinders by size using the `size` argument allows us to investigate the relationship between the three numeric quantities.

```
qplot(x = wt,
      y = mpg,
      data = mtcars,
      size = cyl,
      main = "MPG vs. weight (x1000 lbs) by cylinder")
```

We conclude from the figure that cars with more cylinders tend to have higher weight and lower fuel efficiency. Alternatively, color can be used to encode the number of cylinders using the argument `color`.

```
qplot(x = wt,
      y = mpg,
      data = mtcars,
      color = cyl,
      main = "MPG vs. weight (x1000 lbs) by cylinder")
```

In many cases the data contains a large amount of noise, and graphing it may focus the viewer's attention on the noise while hiding important general trends. One technique to address this issue is to add a smoothed line curve y_S, which is a weighted average of the original data $(y^{(i)}, x^{(i)})$ $i = 1, \ldots, n$:



$$y_S(x) = \sum_{i=1}^{n} \frac{K_h(x - x^{(i)})}{\sum_{i=1}^{n} K_h(x - x^{(i)})} y^{(i)}. \tag{8.1}$$

The K_h functions above are the kernel functions described in Sect. 8.7.

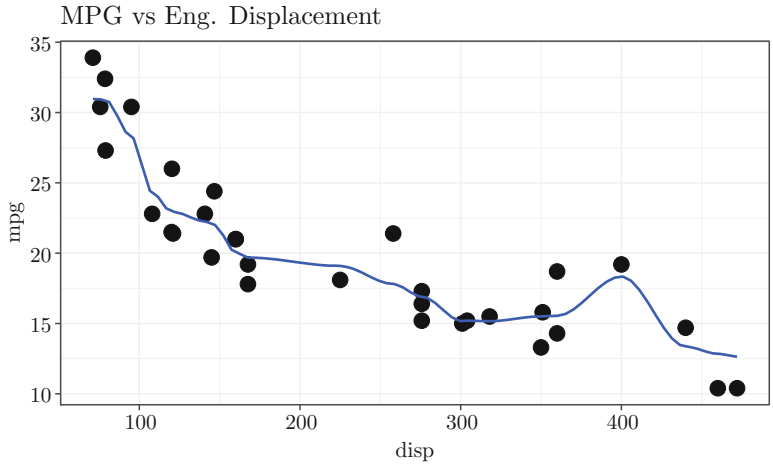
In other words, $y_S(x)$ is an average the $y^{(i)}$ values, weighted in a way that emphasizes $y^{(i)}$ values whose corresponding $x^{(i)}$ values are close to x. The denominator in the definition of y_S ensures that the weights defining the weighted average sum to 1.

Equation (8.1) describes a statistical technique known as locally constant regression that can be used to estimate a relationship between x and y without making parametric assumptions (for example, the assumption of a linear relationship).

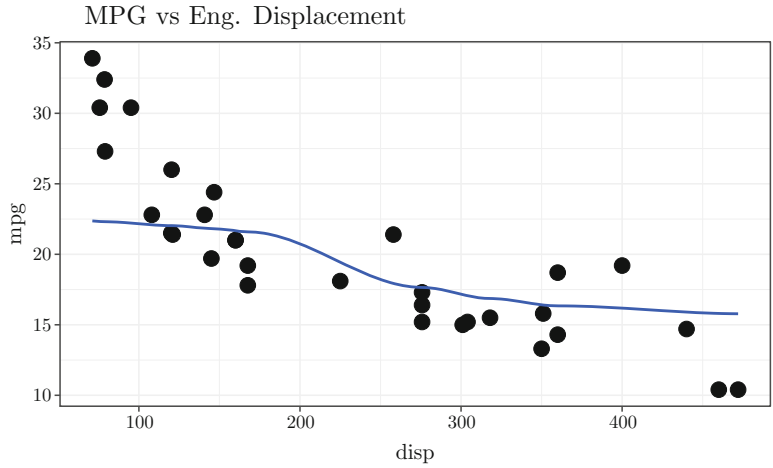
We demonstrate the smoothed scatter plot with several graphs below. The first two graphs explore different values of h, the parameter that influences the spread or width of the $g_i = K_h(x, x^{(i)})$ functions. To adjust h, we modify the `span` argument that has a similar role to the `adjust` parameter in the discussion of the smoothed histogram above.

```
qplot(disp,
      mpg,
      data = mtcars,
      main = "MPG vs Eng. Displacement") +
  stat_smooth(method = "loess",
              method.args = list(degree = 0),
              span = 0.2,
              se = FALSE)
```

Increasing the value of the `span` parameter increases h, resulting in wider g_i functions and a less wiggly curve.

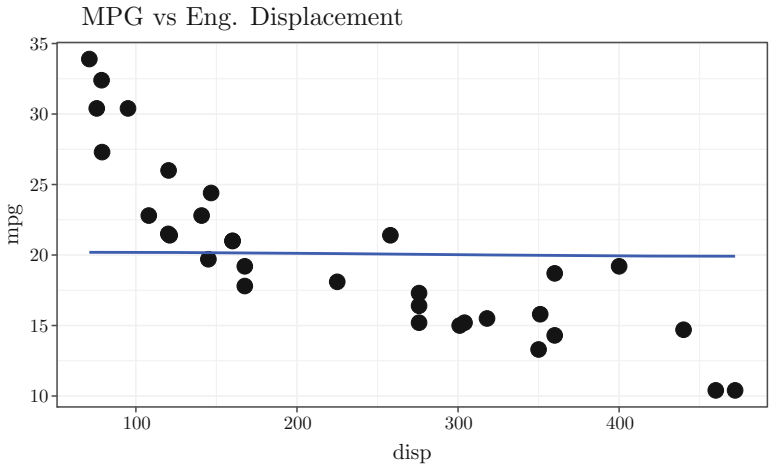



```
qplot(disp,
      mpg,
      data = mtcars,
      main = "MPG vs Eng. Displacement") +
  stat_smooth(method = "loess",
              method.args = list(degree = 0),
              span = 1,
              se = FALSE)
```



Selecting an even larger h results in a nearly constant line.

```
qplot(disp,
      mpg,
      data = mtcars,
      main = "MPG vs Eng. Displacement") +
  stat_smooth(method = "loess",
              method.args = list(degree = 0),
              span = 10,
              se = FALSE)
```

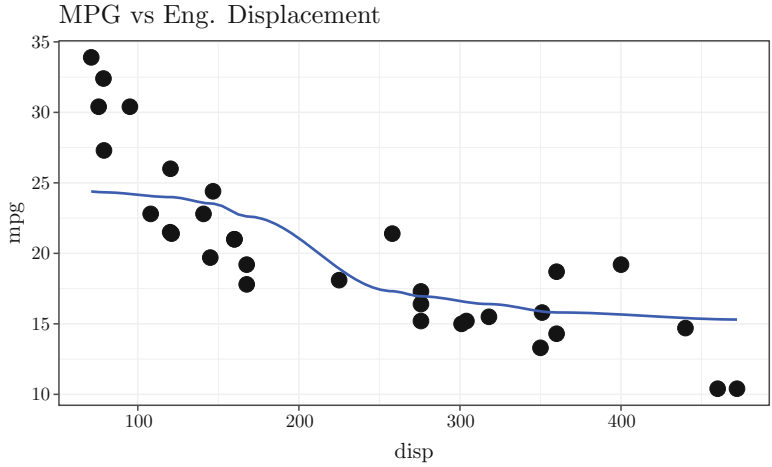


Omitting this parameter reverts to an automatically chosen value.

```
qplot(disp,
      mpg,
      data = mtcars,
      main = "MPG vs Eng. Displacement") +
  stat_smooth(method = "loess",
              method.args = list(degree = 0),
              se = FALSE)
```

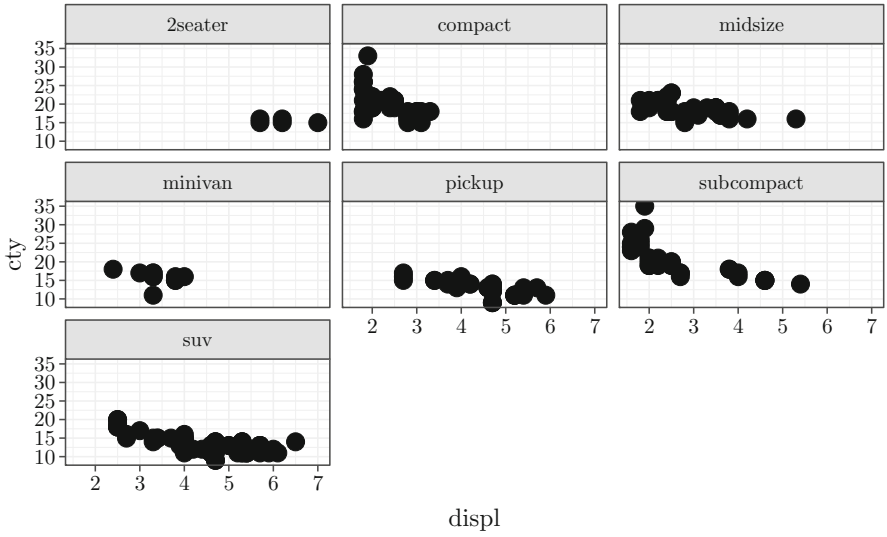
We can conclude from the graph above that mpg decreases as engine displacement volume increases. The trend is nonlinear with a slope that changes as follows: first small slope, then large slope, and then small slope again (in absolute value). This information is readily available from the smoothed y_S but is not easy to discern from the original scatter plot data.

In some cases, we want to examine multiple plots with the same x or y axis in different side-by-side panels. The function `qplot` enables this using the `facet_wrap` function or the `facets` argument.



The `facet_wrap()` function wraps multiple graphs inside a figure but keeps the axes scales constant. For example, the following graph visualizes the relationship between city mpg and engine displacement for different classes of vehicles.

```
ggplot(mpg, aes(displ, cty)) + geom_point() +
  facet_wrap(~class)
```



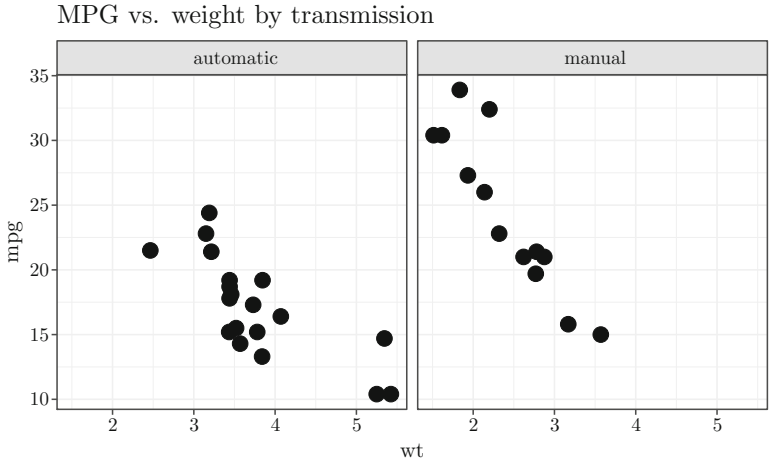
We see that the decay in city mpg is particularly sharp and nonlinear for compact and subcompact vehicles. We can customize the behavior by passing arguments

to `facet_wrap` such as the number of rows and the number of columns. See `help(facet_wrap)` for more details.

The `facet` argument is helpful when we want to arrange the sub-plots in a specific order so that the coordinate of the subplot conveys relevant information. The argument `facet` takes a formula of the form $a \sim b$ and creates multiple rows and columns of panels (a determines the row variable and b the column variable).

In the first example below we graph two scatter plots side by side: mpg vs. weight for automatic transmission cars and manual transmission cars. Note that the two panels are side by side since the `facet` argument is . ~`amf`. The two panels share the same axes' scales, thus facilitating easy comparison. As before, we create new dataframe columns with more appropriate names in order to create more informative axes labeling. Changing the names of existing columns using the function `names` is another option.

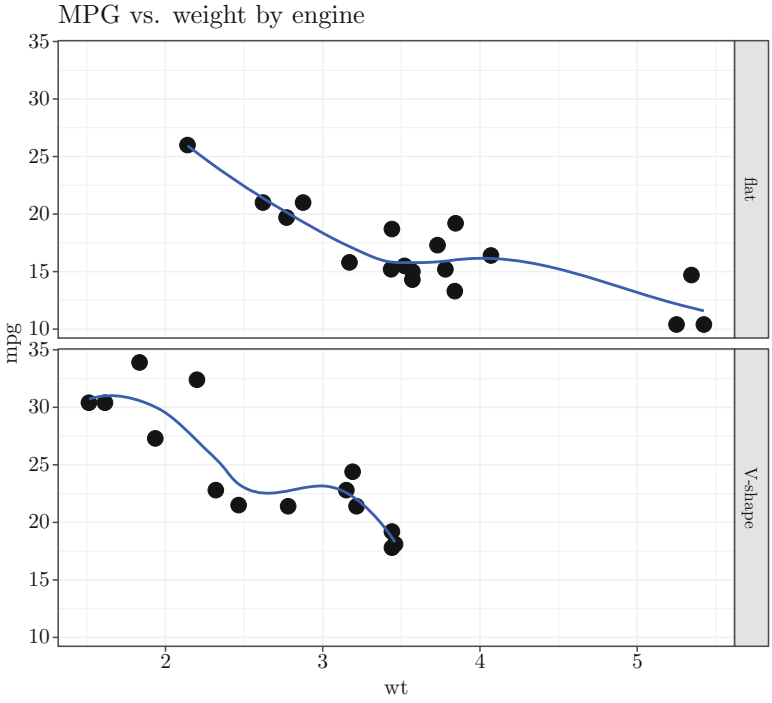
```
# add new dataframe columns with more appropriate names for
# better axes labeling in future graphs
mtcars$amf[mtcars$am==0] = 'automatic'
mtcars$amf[mtcars$am==1] = 'manual'
mtcars$vsf[mtcars$vs==0] = 'flat'
mtcars$vsf[mtcars$vs==1] = 'V-shape'
qplot(x = wt,
      y = mpg,
      facets = .~amf,
      data = mtcars,
      main = "MPG vs. weight by transmission")
```



We conclude from the graph above that manual transmission cars tend to have lower weights and be more fuel efficient.

The graph below plots mpg vs. weight for two panels—one above the other indicating whether or not the engine is a V shape engine.

```
mtcars$amf[mtcars$am==0] = 'automatic'
mtcars$amf[mtcars$am==1] = 'manual'
mtcars$vsf[mtcars$vs==0] = 'flat'
mtcars$vsf[mtcars$vs==1] = 'V-shape'
qplot(x = wt,
      y = mpg,
      facets = vsf~.,
      data = mtcars,
      main = "MPG vs. weight by engine") +
  stat_smooth(se = FALSE)
```

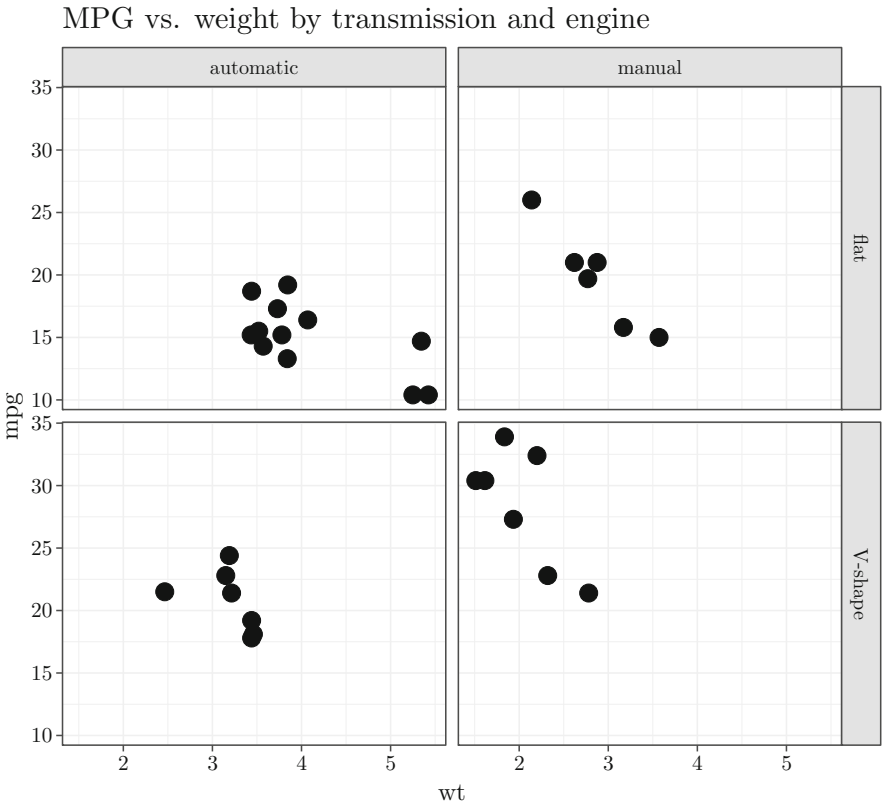


We conclude from the graph above that cars with V shape engines tend to weigh less and be more fuel efficient.

The graph below shows a multi-row and multi-column array of panels. It shows that manual transmission and V engine cars tend to be lighter and more fuel efficient. Automatic transmission and non-V engine are heavier and less fuel efficient.

```
mtcars$amf[mtcars$am==0] = 'automatic'
mtcars$amf[mtcars$am==1] = 'manual'
```

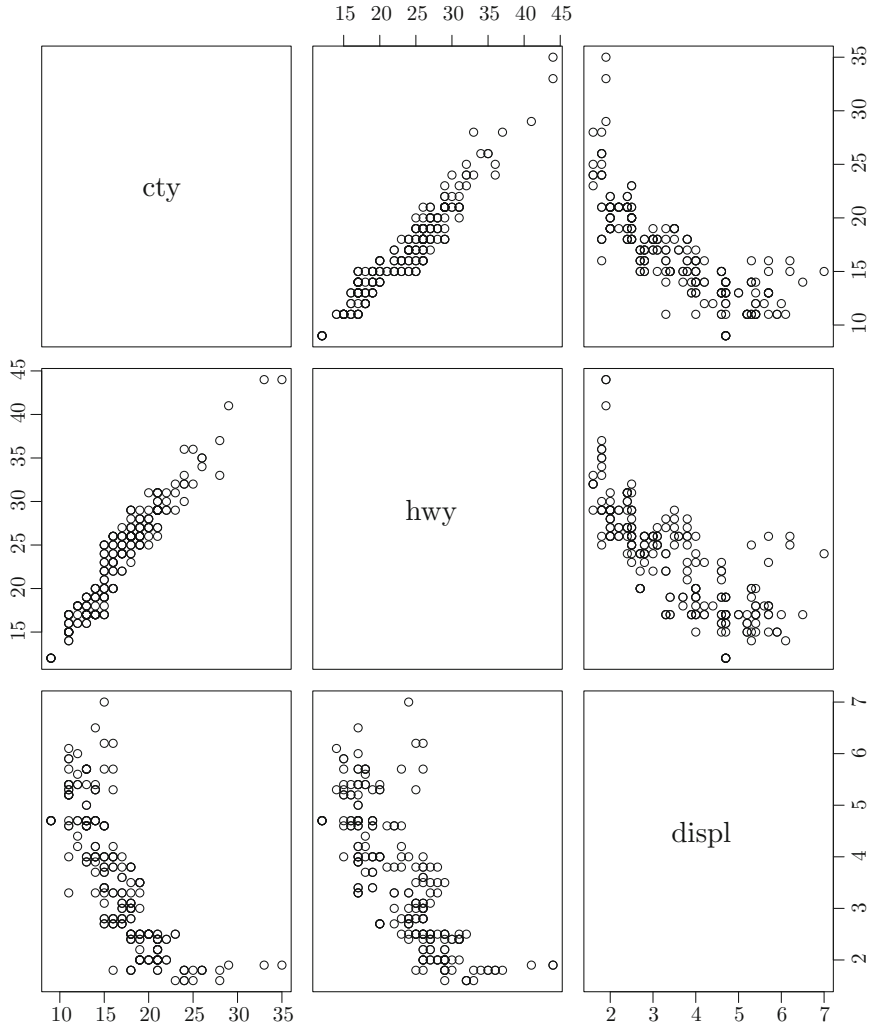
```
mtcars$vsf[mtcars$vs==0] = 'flat'
mtcars$vsf[mtcars$vs==1] = 'V-shape'
qplot(x = wt,
      y = mpg,
      data = mtcars,
      facets = vsf~amf,
      main = "MPG vs. weight by transmission and engine")
```



The function `plot` can create a similar array of panels with synchronized axes scales when it receives an entire dataframe as an argument. We demonstrate this below by exploring all-pairs relationships between city mpg, highway mpg, and engine displacement volume.

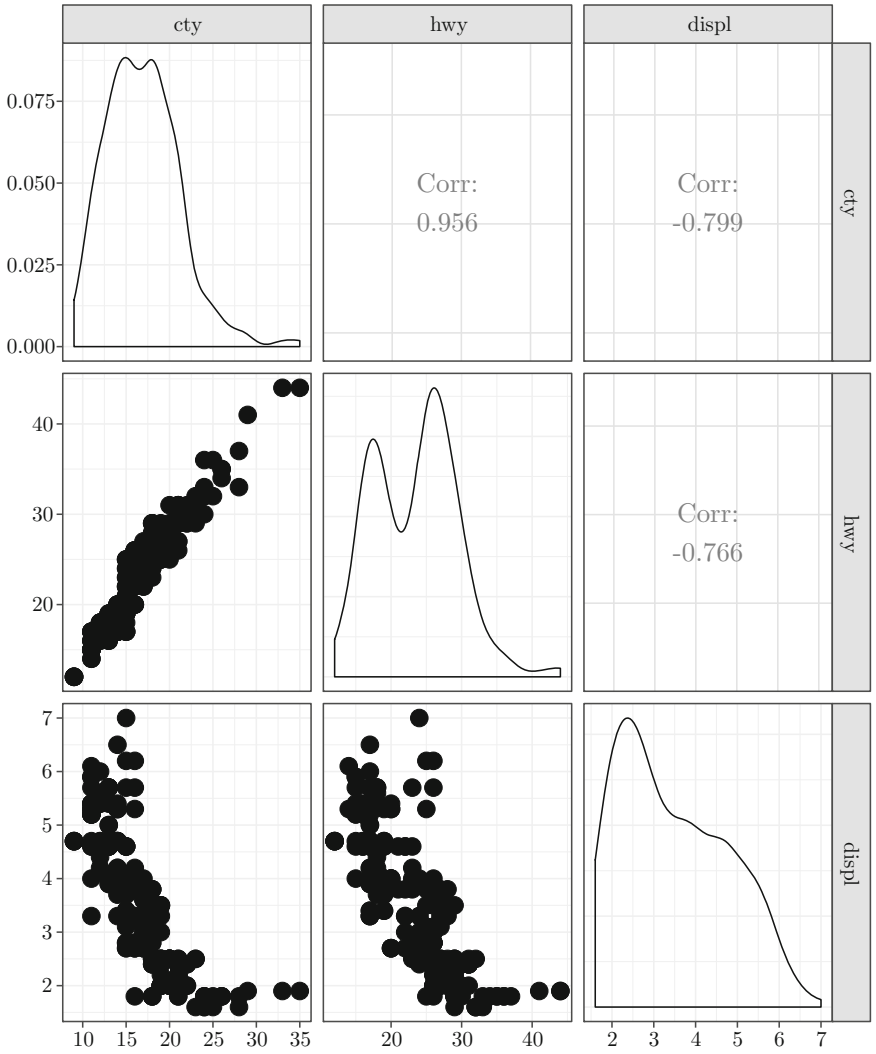
```
# create a new dataframe with three columns: cty, hwy, and disp
DF = mpg[, c("cty", "hwy", "displ")]
plot(DF, main = "City MPG vs. Highway MPG vs. Engine Volume")
```

City MPG vs. Highway MPG vs. Engine Volume



An alternative is the `ggpairs` function from the `GGallly` package. It also displays smoothed histograms of all variables in the diagonal panels and the correlation coefficients in the upper triangle.

```
library(GGally)
ggpairs(DF)
```



We can conclude from the plot above that while highway mpg and city mpg tend to increase together linearly, they are in inverse (nonlinear) relationship to the engine displacement volume.

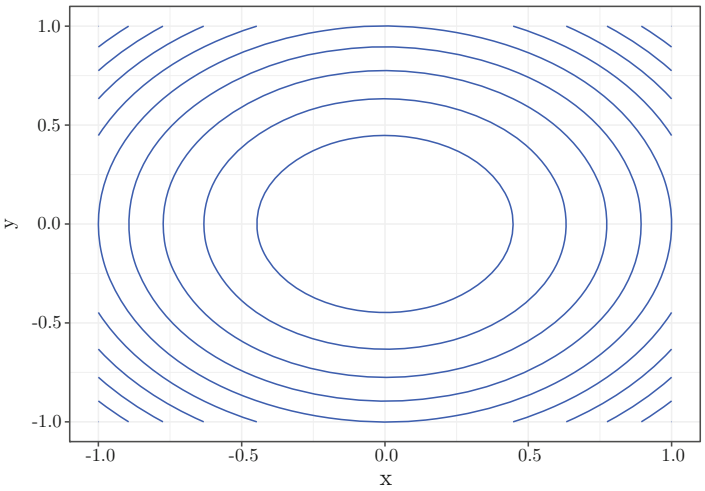
8.9 Contour Plots

The most convenient way to graph a two-dimensional function $f(x, y)$ is by graphing its equal height contours

$$z_c = \{(x, y) \in \mathbb{R}^2 : f(x, y) = c\}$$

for different values of c. To graph such a function with the `ggplot2` package, create a dataframe with columns corresponding to the `x`, `y`, and `z` values. The `x` and `y` columns should feature all possible combinations of the two coordinates over a certain grid. Then call `ggplot` and add the `stat_contour` layer.

```
x_grid = seq(-1, 1, length.out = 100)
y_grid = x_grid
# create a dataframe containing all possible combinations of
  x,y
R = expand.grid(x_grid, y_grid)
# number of rows is 100 x 100 - one for each combination
dim(R)
## [1] 10000     2
# modify column names for clear axes labeling
names(R) = c('x', 'y')
R$z = R$x^2 + R$y^2
head(R)
##             x  y        z
## 1 -1.0000000 -1 2.000000
## 2 -0.9797980 -1 1.960004
## 3 -0.9595960 -1 1.920824
## 4 -0.9393939 -1 1.882461
## 5 -0.9191919 -1 1.844914
## 6 -0.8989899 -1 1.808183
ggplot(R, aes(x = x,y = y, z = z)) + stat_contour()
```



8.10 Quantiles and Box Plots

Histograms are very useful for summarizing numeric data in that they show a rough distribution of values. An alternative that is often used in conjunction with the histogram is the box plot. We start with describing the notion of percentiles that plays a central role in our discussion.

The r-th percentile of a numeric dataset is the point at which approximately $r\%$ of the data lie underneath, and approximately $100 - r\%$ lie above.[1] Another name for the r-th percentile is the $0.r$ quantile.

```
# display 0 through 100 percentiles at 0.1 increments
# for the dataset containing 1,2,3,4.
quantile(c(1, 2, 3, 4), seq(0, 1, length.out = 11))
##   0%  10%  20%  30%  40%  50%  60%  70%  80%
##  1.0  1.3  1.6  1.9  2.2  2.5  2.8  3.1  3.4
##  90% 100%
##  3.7  4.0
```

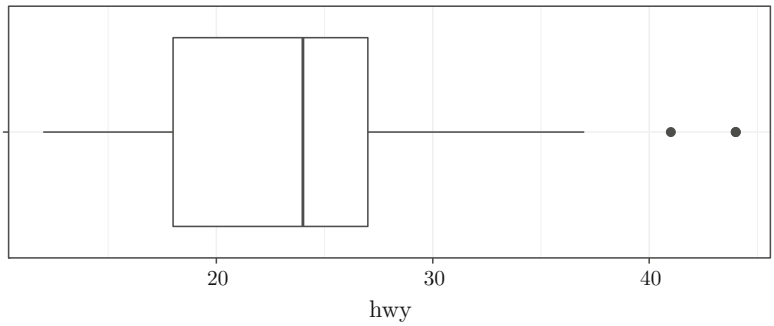
The median or 50th percentile is the point at which half of the data lies underneath and half above. The 25th and the 75th percentiles are the values below which 25% and 75% of the data lie, respectively. These points are also called the first and third quartiles (the second quartile is the median). The interval between the first and third quartiles is called the inter-quartile range (IQR); it's the region covering the central 50% of the data.

A box plot is composed of a box, an inner line bisecting the box, whiskers that extend to either side of the box, and outliers. The box denotes the IQR, with the inner bisecting line denoting the median. The median may or may not be in the geometric center of the box, depending on whether the distribution of values is skewed or symmetric. The whiskers extend to the most extreme point no further than 1.5 times the length of the IQR away from the edges of the box. Data points outside the box and whiskers' range are called outliers and are graphed as separate points. Separating the outliers from the box and whiskers is useful for avoiding a distorted viewpoint where there are a few extreme nonrepresentative values.

The following code graphs a box plot in R using the `ggplot2` package. The `+` operator below adds the box plot geometry, flips the x, y coordinates, and removes the y-axis label.

```
ggplot(mpg, aes("", hwy)) +
  geom_boxplot() +
  coord_flip() +
  scale_x_discrete("")
```

[1] There are several different formal definitions for percentiles. Type `help(quantile)` for several competing definitions that R implements.



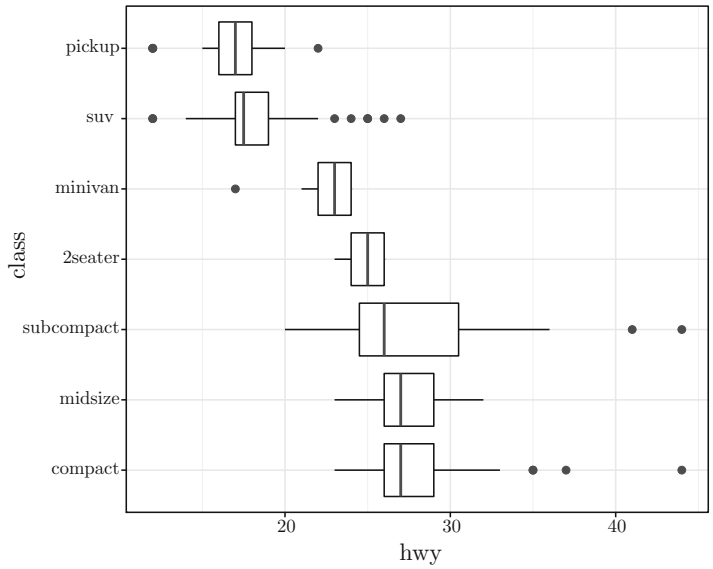
We conclude from this graph that the median highway mpg is around 24, with the central 50% of the data falling within the box that spans the range from 18 to 27. There are two high outliers over 40, but otherwise the remaining data lie within the whiskers between 12 and 37. The fact that the median line is right of the middle of the box hints that the distribution is skewed to the right.

Contrast the box plot above with the smoothed histogram in Page 307 (center panel). The box plot does not convey the multimodal nature of the distribution that the histogram shows. On the other hand, it is easier to read the median and the IQR, which show the center and central 50% range from the box plot.

It is convenient to plot several box plots side by side in order to compare data corresponding to different values of a factor variable. We demonstrate this by graphing below box plots of highway mpg for different classes of vehicles. We flip the box plots horizontally using `coord_flip()` since the text labels display better in this case. Note that we reorder the factors of the `class` variable in order to sort the box plots in order of increasing highway mpg medians. This makes it easier to compare the different populations.

```
ggplot(mpg, aes(reorder(class, -hwy, median), hwy)) +
  geom_boxplot() +
  coord_flip() +
  scale_x_discrete("class")
```

The graph suggests the following fuel efficiency order among vehicle classes: pickups, SUV, minivans, 2-seaters, subcompacts, midsizes, and compacts. The compact and midsize categories have almost identical box and whiskers but the compact category has a few high outliers. The spread of subcompact cars is substantially higher than the spread in all other categories. We also note that SUVs and two-seaters have almost disjoint values (the box and whisker ranges are completely disjoint) leading to the observation that almost all 2-seater cars in the survey have a higher highway mpg than SUVs.



8.11 qq-Plots

Quantile-quantile plots, also known as qq-plots, are useful for comparing two datasets, one of which may be sampled from a certain distribution. They are essentially scatter plots of the quantiles of one dataset vs. the quantiles of another dataset. The shape of the scatter plot implies the following conclusions (the proofs are straightforward applications of probability theory).

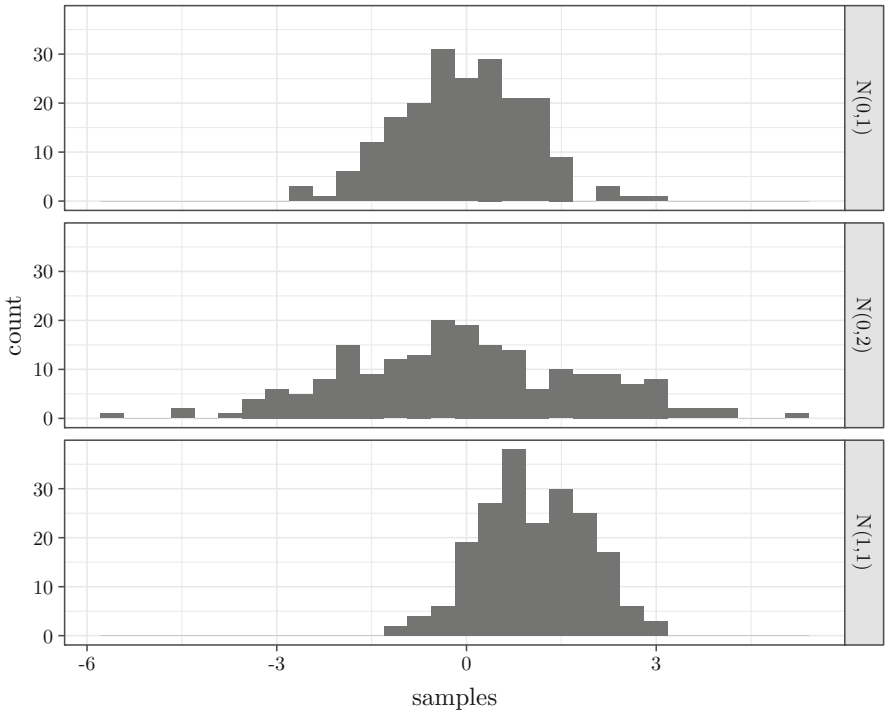
- A straight line with slope[2] 1 that passes through the origin implies that the two datasets have identical quantiles, and therefore that they are sampled from the same distribution.
- A straight line with slope 1 that does not pass through the origin implies that the two datasets have distributions of similar shape and spread, but that one is shifted with respect to the other.
- A straight line with slope different from 1 that does not pass through the origin implies that the two datasets have distributions possessing similar shapes but that one is translated and scaled with respect to the other.
- A nonlinear S shape implies that the dataset corresponding to the x-axis is sampled from a distribution with heavier tails than the other dataset.
- A nonlinear reflected S shape implies that the dataset whose quantiles correspond to the y-axis is drawn from a distribution having heavier tails than the other dataset.

[2]Slope 1 corresponds to 45 degrees incline from left to right.

To compare a single dataset to a distribution we sample values from the distribution, and then display the qq-plots of the two datasets. The quantiles of the sample drawn from the distribution are sometimes called theoretical quantiles.

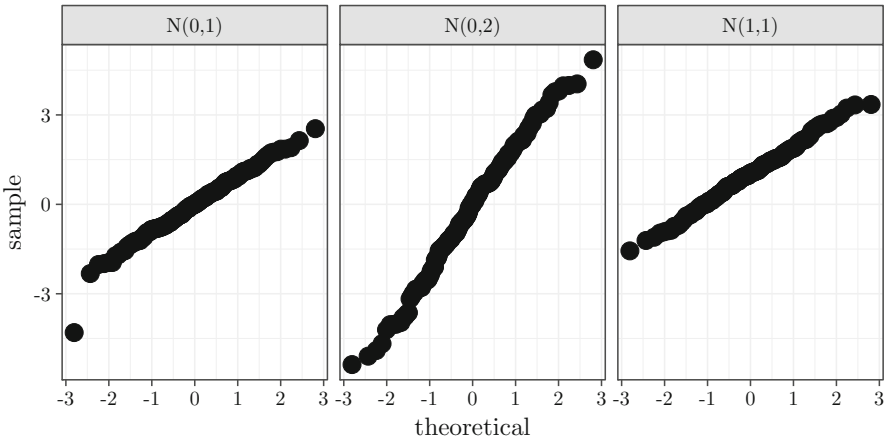
For example, consider the three datasets sampled from three bell-shaped Gaussian distributions $\mathcal{N}(0, 1)$, $\mathcal{N}(0, 1)$, and $\mathcal{N}(0, 2)$ (a precise definition and a discussion of these important distributions appears in TAOD volume 1, Chapter 3). The corresponding histograms appear below.

```
D = data.frame(samples = c(rnorm(200, 1, 1),
                           rnorm(200, 0, 1),
                           rnorm(200, 0, 2)))
D$parameter[1:200]   = 'N(1,1)';
D$parameter[201:400] = 'N(0,1)';
D$parameter[401:600] = 'N(0,2)';
qplot(samples,
      facets = parameter~.,
      geom = 'histogram',
      data = D)
```



We compute below the qq-plots of these three datasets (y axis) vs. a sample drawn from the $\mathcal{N}(0, 1)$ distribution (x axis).

```
D = data.frame(samples = c(rnorm(200, 1, 1),
                           rnorm(200, 0, 1),
                           rnorm(200, 0, 2)));
D$parameter[1:200]   = 'N(1,1)';
D$parameter[201:400] = 'N(0,1)';
D$parameter[401:600] = 'N(0,2)';
ggplot(D, aes(sample = samples)) +
  stat_qq() +
  facet_grid(.~parameter)
```



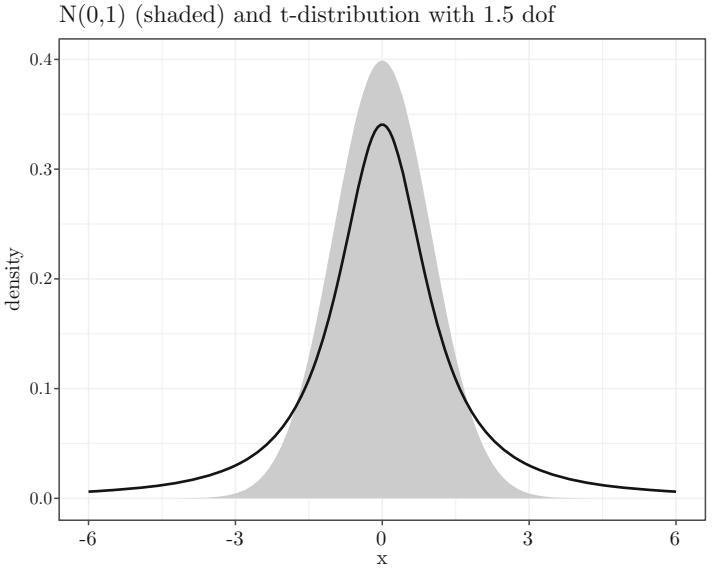
Note how all three plots are linear, implying that the three datasets have distributions similar in shape to the $\mathcal{N}(0, 1)$ distribution up to translation and scaling. In the left panel, we get a linear shape with slope 1 which passes through the origin, since the two datasets were sampled from the same $\mathcal{N}(0, 1)$ distribution. In the middle panel we get a line passing through the origin but with a steeper slope, since the data was more widespread than the $\mathcal{N}(0, 1)$ distribution. In the right panel we get a slope 1 line that does not pass through the origin since the two distributions have identical spread but are translated with respect to each other.

As a final example we show the qq-plot of a sample from a $\mathcal{N}(0, 1)$ distribution and a sample from a t-distribution[3] with 1.5 degrees of freedom. In contrast to the Gaussian $\mathcal{N}(0, 1)$ density, whose tails decrease exponentially, the t density has tails that decrease significantly slower at a polynomial rate. As a result, the two distributions have fundamentally different shapes and their quantiles are nonlinearly related.

The following graph demonstrates the shape of the densities of the two distribution and how the t-distribution has heavier tails than the Gaussian $\mathcal{N}(0, 1)$ distribution.

[3] A formal definition of the t-distribution appears in TAOD volume 1, Chapter 3.

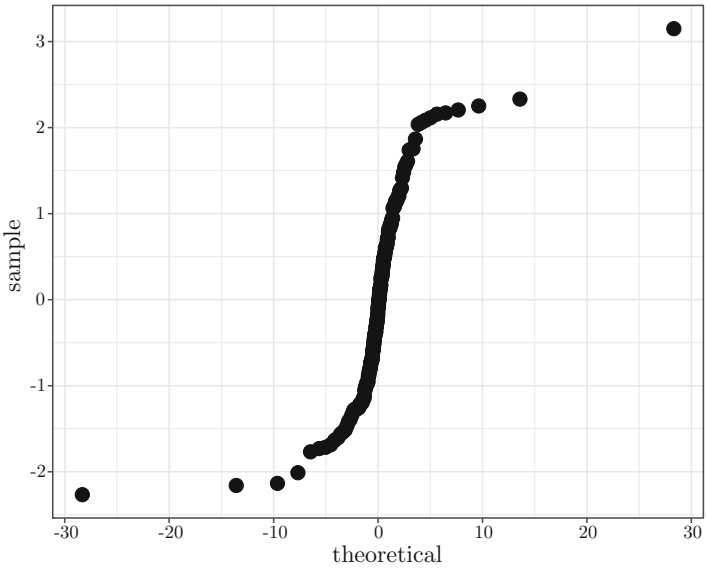
```
x_grid = seq(-6, 6, length.out = 200)
R = data.frame(density = dnorm(x_grid, 0, 1))
R$tdensity = dt(x_grid, 1.5)
R$x = x_grid
ggplot(R, aes(x = x, y = density)) +
  geom_area(fill = I('grey')) +
  geom_line(aes(x = x, y = tdensity)) +
  labs(title = "N(0,1) (shaded) and t-distribution with 1.5
       dof")
```




```
x_grid = seq(-6, 6, length.out = 200)
R = data.frame(density = dnorm(x_grid, 0, 1))
R$samples = rnorm(200, 0, 1)
pm = list(df = 1.5)
ggplot(R, aes(sample = samples)) +
  stat_qq(distribution = qt, dparams = pm)
```

8.12 Devices

By default, R overwrites the current figure with new plots. The function call `dev.new()` opens a new additional graphics windows. The `ggsave` function within the `ggplot2` package saves the active graphics window to a file with the file type (pdf, postscript, jpeg) corresponding to the file name extension.




```
# detects file-type format (PDF) from file name extension
ggsave(file = "myPlot.pdf")
```

To send all future graphics to a single file use one of the following functions: `postscript`, `pdf`, `xfig`, `bmp`, `png`, `jpeg`, or `tiff`. It is essential to call the function `dev.off()` at the end of the graphics session in order to ensure that the graphics file is closed properly. To see a precise list of optional parameters such as font size and compression rate refer to `help(X)` where `X` is one of the device drivers, for example `help("jpeg")`.

The first three drivers (`postscript`, `pdf`, `xfig`) maintain high-resolution graphics using vector graphics. The resulting graphics can be zoomed in to arbitrary precision. Among these formats, pdf is usually preferred, since it is often smaller in file size and since it is accessible by a wide variety of programs.

The latter four drivers (`bmp`, `png`, `jpeg`, `tiff`) produce raster graphics which correspond to pixelized images with fixed resolutions. While vector graphics is generally preferable to raster graphics due to its superior resolution, vector graphics may produce very large files when the graphics contain many objects. In that case a raster graphics file may be preferable due to its smaller file size.

```
# save all future graphics to file myplots.pdf
pdf('myplots', height = 5, width = 5, pointsize = 11)
#  graphics plotting
qplot(...)
qplot(...)
#  close graphics file and return to display driver
dev.off()
```

8.13 Data Preparation

We emphasize in this book graphing data by first creating a dataframe with the appropriate data and informative column names, and then calling `plot`, `qplot`, or `ggplot`. This approach is better than keeping the data in an un-annotated array, graphing the values, and then labeling the axes, legends, and facets appropriately. In the examples above we usually started with a readymade dataframe, but in most data analysis cases the dataframe has to be prepared by the data analyst.

To create a dataframe use the `data.frame` function, for example:

```
R = data.frame(name = vec1, ages = vec2, salary = vec3).
```

If a dataframe already exists, but the variable names are not coherent, change the names using the `names` function to coherent names so that legible axes and legends can be automatically created.

```
names(R) = c("last.name", "age", "annual.income")
```

The functions `rbind` and `cbind` add additional rows or columns to an existing dataframe.

Consider the following example of graphing the line plots of the Gaussian density function (see TAOD volume 1, Chapter 3)

$$f(x) = \mathcal{N}(x\,;0,\sigma) = \exp(-x^2/(2\sigma^2))/(\sqrt{2\pi\sigma^2})$$

with the color and line type corresponding to the value of σ among four different values: 1, 2, 3, and 4. Note that this function is also $K_h(x)$ for the Gaussian kernel described earlier in this chapter.

Our strategy is to first compute a list of four vectors containing y values—one vector for each value of σ. The names of the four list elements identify the σ corresponding to that element. We then use the `stack` function to create the dataframe.

The following code helps illustrate the idea before we move on to the complete example below.

```
A = list(a = c(1, 2), b = c(3, 4), c = c(5, 6))
A
## $a
## [1] 1 2
##
## $b
## [1] 3 4
##
## $c
## [1] 5 6
stack(A)
##   values ind
## 1      1   a
```

```
## 2      2   a
## 3      3   b
## 4      4   b
## 5      5   c
## 6      6   c
```

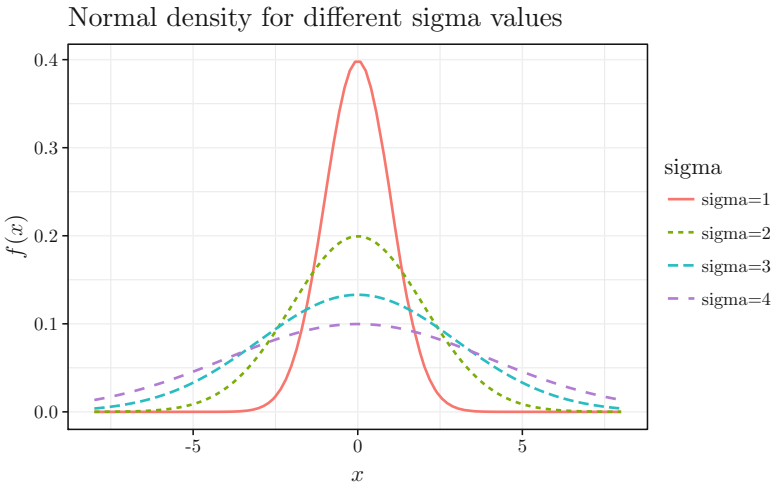
The dataframe above is ready for graphing. The first column contains the values of the variable that is being visualized, and the second column contains a variable that is used to distinguish different graphs using overlays or facets.

The code below provides a complete example.

```
x_grid = seq(-8, 8, length.out = 100)
gaussian_function =
    function(x, s) exp(-x^2/(2*s^2))/(sqrt(2*pi)*s)
R = stack(list('sigma=1' = gaussian_function(x_grid, 1),
               'sigma=2' = gaussian_function(x_grid, 2),
               'sigma=3' = gaussian_function(x_grid, 3),
               'sigma=4' = gaussian_function(x_grid, 4)))

names(R) = c("y", "sigma");
R$x = x_grid
head(R)
##              y   sigma         x
## 1 5.052271e-15 sigma=1 -8.000000
## 2 1.816883e-14 sigma=1 -7.838384
## 3 6.365366e-14 sigma=1 -7.676768
## 4 2.172582e-13 sigma=1 -7.515152
## 5 7.224128e-13 sigma=1 -7.353535
## 6 2.340189e-12 sigma=1 -7.191919
qplot(
      x,
      y,
      color = sigma,
      lty = sigma,
      geom = "line",
      data = R,
      main = "Normal density for different sigma values",
      xlab = "$x$",
      ylab = "$f(x)$")
```

8.14 Python's Matplotlib Module

Despite the fact that R has excellent graphics capabilities, it is sometimes desirable to graph data in another programming language. For example, if the entire data analysis process is in Python, it may make sense to graph the data within Python rather than save the data in Python, load it in R, and graph it in R. We describe below



matplotlib—a popular Python module for graphing data. Matplotlib features two interfaces: (a) the default object oriented programmatic interface, and (b) pylab—a Matlab-like interface. We focus below on pylab as it is simpler and it may be familiar to readers previously exposed to Matlab. To get access to pylab's API start the interactive IPython program using the command `ipython -pylab`.

8.14.1 Figures

The function `figure` opens a new figure and returns an object that may be used to display graphs in that figure. Multiple `figure` functions may be issued, and by default the figure that was opened last is the active figure. The function `close(X)` closes figure `X` (or by default the active figure if the argument is omitted). The function `savefigX` saves the active figure to a file whose filename is `X` (the file type is inferred from the filename extension). For example, the following code opens up two figures, saves the second figure and closes it, and then saves and closes the first figure.

```
import matplotlib.pyplot as plt
f1 = plt.figure()  # open a figure
f2 = plt.figure()  # open a second figure
plt.savefig('f2.pdf')  # save fig 2 - the active figure
plt.close(f2)
plt.savefig('f1.pdf')  # save fig 2 - the active figure
plt.close(f1)
```

8.14.2 Scatter-Plots, Line-Plots, and Histograms

The function `plot(x,y)` displays a scatter plot of the two arrays `x` and `y`, and connects the scatter plot points with lines.

An optional third argument for `plot` is a string containing the color code, followed by marker code that is in turn followed by line-style code. For example, the string `'ro-'` corresponds to color red, circular scatter plot markers, and dashed line. If some of these patterns are omitted the default choice is selected. Omitting the scatter plot markers pattern creates a line plot, for example `'r-'` creates a red line plot. Omitting the line-style pattern creates a scatter plot, for example `'ro'` creates a red scatter plot. Multiple consecutive plot functions add additional features to the current figure.

The functions `xlabel`, `ylabel`, and `title` annotate the x-axis, the y-axis, and the figure, respectively. As in the R examples we can use LaTeX code (text surrounded by $ symbols in the example below) to annotate the axes labels or titles with equations.

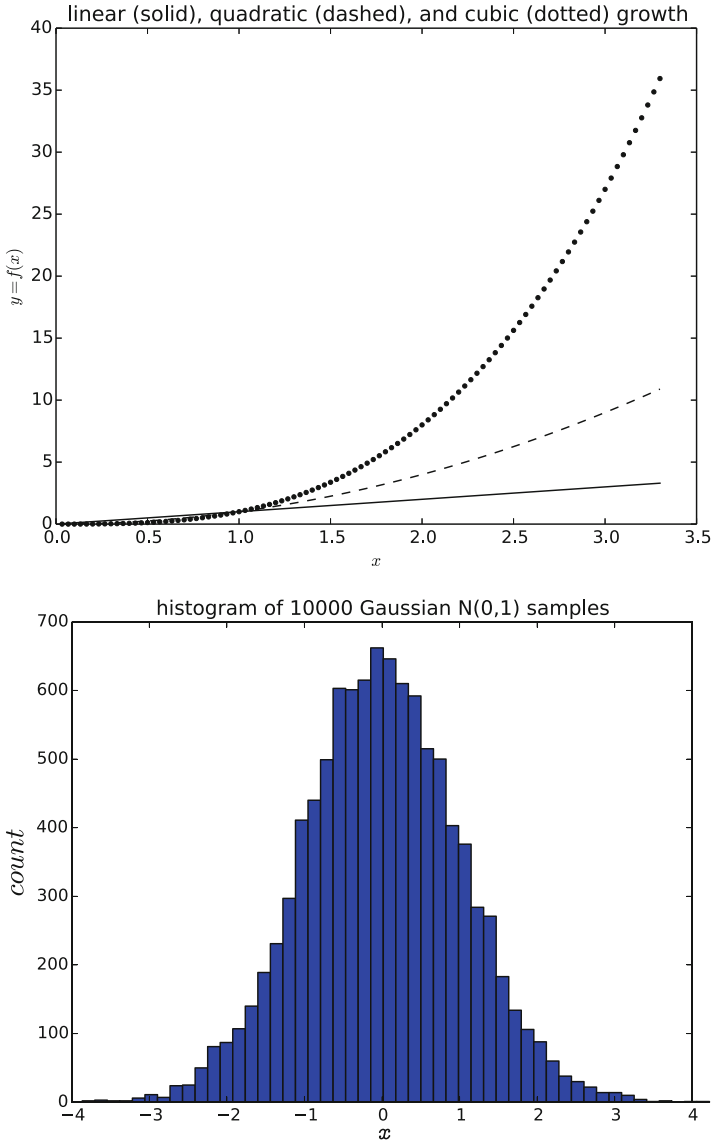
The range of values displayed in the x-axis and y-axis can be modified using the `xlim([min_x, max_x])` and `ylim([min_y, max_y])` functions.

The code below displays three line plots: linear (black solid line), quadratic (black dashed line), and cubic (black dotted line). Since the scatter plot marks patterns are omitted, we get line plots without the scatter plot corresponding to the sampled points.

```
import matplotlib.pyplot as plt
import numpy as np

x_grid = np.arange(1, 100) / 30.0
plt.figure()  # open a figure
plt.plot(x_grid, x_grid, 'k-') # draws a solid line
plt.plot(x_grid, x_grid ** 2, 'k--') # draws a dashed line
plt.plot(x_grid, x_grid ** 3, 'k.') # draws a dotted line
plt.xlabel(r'$x$')
plt.ylabel(r'$y=f(x)$')
plt.title('linear, quadratic, and cubic growth')
```

The function `hist(x,n)` creates a histogram of the data in `x` using `n` bins.

```
import matplotlib.pyplot as plt
from numpy.random import randn

data = randn(10000)
plt.hist(data, 50)
plt.xlabel(r'$x$')
plt.ylabel(r'$count$')
plt.title('histogram of 10000 Gaussian N(0,1) samples')
plt.xlim([-4, 4])
```

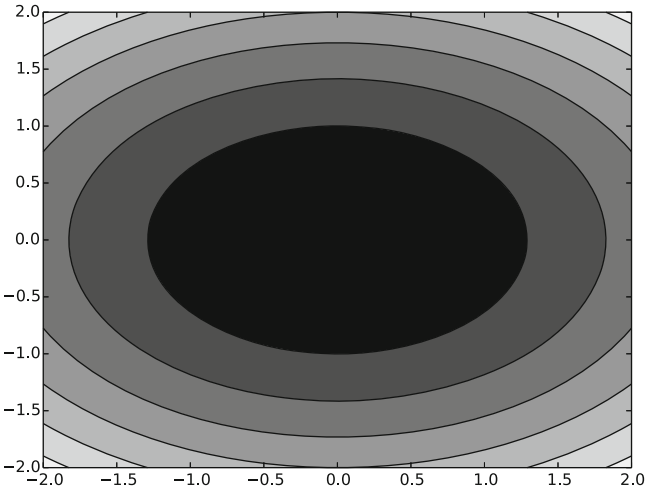


8.14.3 Contour Plots and Surface Plots

Matplotlib can also graph three-dimensional data. We describe below how to create contour plots and surface plots—the two most common 3D graphs.

To create a contour plot or surface plot of a function $f = (x, y)$, we need to first create two one-dimensional grids corresponding to the values of x and y. The function `numpy.meshgrid` takes these two one-dimensional grids and returns

two two-dimensional ndarrays containing the x and y values (the first ndarray has constant columns and the second ndarray has constant rows). We can then create a third ndarray holding the values of $z = f(x, y)$ by operating a two-dimensional function on the two ndarrays.



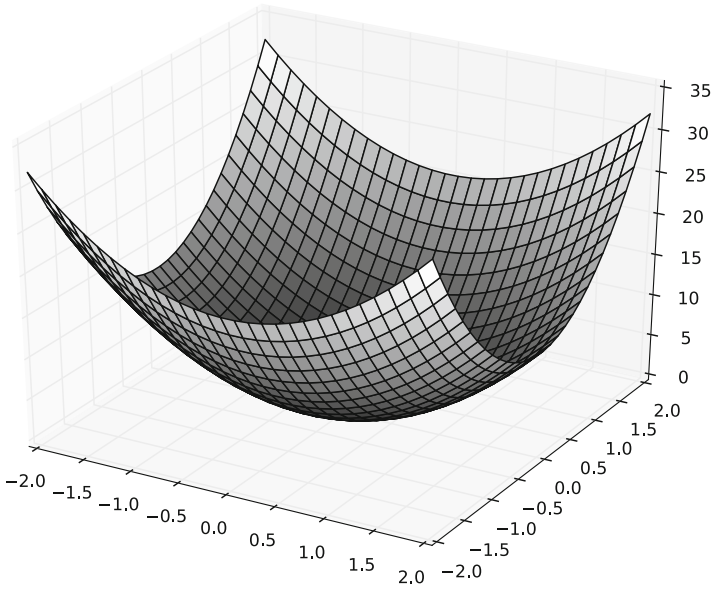
The function `contourf` displays $z = f(x, y)$ as a function of x, y, quantizing the z values into several constant values. The code below shows how to use `contourf` to display a contour plot of the function $z = 3x^2 + 5y^2$.

```
import numpy as np
import matplotlib.pyplot as plt
def f(x, y): return (3*x**2 + 5*y**2)
x_grid = np.linspace(-2, 2, 100)
y_grid = np.linspace(-2, 2, 100)
# create two ndarrays xx, yy containing x and y coordinates
xx, yy = np.meshgrid(x_grid, y_grid)
zz = f(xx, yy)
# draw contour graph with 6 levels using gray colormap
#   (white=high, black=low)
plt.contourf(xx,
             yy,
             zz,
             6,
             cmap = 'gray')
# add black lines to highlight contours levels
plt.contour(xx,
            yy,
            zz,
            6,
            colors = 'black',
```

```
                linewidth = .5)
```

The function `plot_surface` is similar to `contourf` except that it displays a 3D surface plot. The code below shows how to use it. Note that before creating a 3D plot we need to create a 3D axis object using the function `Axes3D` in `mpl_toolkits.mplot3d`.

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
def f(x, y): return (3*x**2 + 5*y**2)
x_grid = np.linspace(-2, 2, 30)
y_grid = np.linspace(-2, 2, 30)
xx, yy = np.meshgrid(x_grid, y_grid)
zz = f(xx, yy)
surf_figure = plt.figure()
figure_axes = Axes3D(surf_figure)
figure_axes.plot_surface(xx,
                         yy,
                         zz,
                         rstride = 1,
                         cstride = 1,
                         cmap = 'gray')
```



8.15 Notes

Refer to the official R manual at cran.r-project.org/doc/manuals/R-intro.html (replace html with pdf for pdf version) and the language reference at http://cran.r-project.org/doc/manuals/R-lang.html (replace .html with .pdf for the PDF version) for more information on R graphics. Detailed information on the `ggplot2` package appears on the website http://had.co.nz/ggplot2/. A comprehensive description of the grammar of graphics which is the basis of the `ggplot2` package is available in (Wilkinson, 2005). Two useful books on how to construct useful graphs are (Cleveland, 1985, 1993). A classic book on exploratory data analysis is (Tuckey, 1977).

The package `lattice` (Sarkar, 2008) is a popular alternative to graphics and `ggplot2`. It often creates graphs faster than `ggplot2`, but its syntax is less intuitive. Many other R packages feature graphics functions for specialized data such as time series, financial data, or geographic maps. A useful resource for exploring such packages is the Task Views http://cran.r-project.org/web/views/.

Python's matplotlib can display additional types of graphs, including bar charts, two-dimensional scatter plots, and three-dimensional surface plots. Please visit http://matplotlib.org for details. Python's pandas module has some graphics functionality that is useful for graphing dataframes. See http://pandas.pydata.org for details. Python also has additional graphics module for specialized graphics, such as interactive graphics.

References

L. Wilkinson. *The Grammar of Graphics*. Springer, second edition, 2005.
W. S. Cleveland. *The Elements of Graphing Data*. Hobart Press, 1985.
W. S. Cleveland. *Visualizing Data*. Hobart Press, 1993.
J. W. Tuckey. *Exploratory Data Analysis*. Addison Wesley, 1977.
D. Sarkar. *Lattice: Multivariate Data Visualization with R*. Springer, 2008.

Chapter 9
Processing Data in R and Python

There is no shortcut to knowledge; and there are no worthwhile data without preprocessing. In the first three sections of this chapter, we discuss situations that necessitate data preprocessing and how to handle them. In the final section we discuss how to manipulate data in general; specifically, how to manipulate data in R using the `reshape2` and `plyr` packages and in Python using the `pandas` module.

In far too many cases, data is available in a form that makes analysis inconvenient. Some frequent situations are listed below:

Missing data: Some of the measurements are not available due to data corruption or difficulty in obtaining the data.

Outliers: Some of the measurements are highly atypical of the data distribution.

Skewed data: The data is highly skewed making its visualization and analysis difficult.

9.1 Missing Data

Data may be missing for a variety of reasons. Perhaps it was corrupted during its transfer or storage. Perhaps some instances in the data collection process were skipped due to difficulty or price associated with obtaining the data. Or perhaps the data was simply unavailable for some other reason.

Denoting n data instances by $\boldsymbol{x}^{(i)}, i = 1, \ldots, n$ (corresponding for example to dataframe rows), missing data implies that for each $i = 1, \ldots, n$ we have a set $A_i \subset \{1, \ldots, n\}$ of indices for which the measurements are missing (A may potentially be the empty set in which case no data is missing). In other words, $x_j^{(i)}$ (the j variable of the i sample) is missing if $j \in A_i$.

Missing data is a general phenomenon. A few specific examples appear below.

© Springer Nature Switzerland AG 2018

G. Lebanon, M. El-Geish, *Computing with Data*,
https://doi.org/10.1007/978-3-319-98149-9_9

- Recommendation systems recommend to users' items from a catalog based on historical user rating. Often, there are a lot of items in the catalog and each user typically indicates their preference on only a small subset of them (for example, by assigning 1–5 stars to previously seen movies).
- In longitudinal studies some of the subjects may not be able to attend each of the surveys throughout the study period. The study organizers may also have lost contact with some of the subjects, in which case all measurements beyond a certain time point are missing.
- In sensor data, some of the measurements may be missing due to sensor failure, battery discharge, or electrical interference.
- In user surveys, users may choose to not respond to some of the questions for privacy reasons.

If the probability of an observation being missing does not depend on observed or unobserved measurements we say that it is missing completely at random (MCAR). For example, in the case of users rating movies using 1–5 stars, we consider ratings of specific movies as dataframe columns and ratings associated with specific users as dataframe rows. Since some movies are more popular than others, the probability of missingness depends on the movie title as well as the movie rating, which violates the MCAR definition.

A more relaxed concept is data missing at random (MAR). This occurs when given the observed data, the probability that data is missing does not depend on the unobserved data. Consider, for example, a survey recording gender, race, and income. Out of the three questions, gender and race are not very objectionable questions, so we assume for now that the survey respondents answer these questions fully. The income question is more sensitive and users may choose to not respond to for privacy reasons. The tendency to report income or to not report income typically varies from person to person. If it only depends on gender and race, then the data is MAR. If the decision whether to report income or not depends also on other variables that are not in the dataframe (such as age or profession), the data is not MAR.

Some data analysis techniques are specifically designed to allow for missing data. In general, however, most methods are designed to work with fully observed data. Below are some general ways to convert missing data to non-missing data.

- Remove all data instances (for example, dataframe rows) containing missing values.
- Replace all missing entries with a substitute value, for example the mean of the observed instances of the missing variable.
- Estimate a probability model for the missing variable and replace the missing value with one or more samples from that probability model.

In the case of MCAR, all three techniques above are reasonable in that they may not introduce systematic errors. In the more likely case of MAR or non-MAR data the methods above may introduce systematic bias into the data analysis process.

9.1.1 Missing Data in R

R represents missing data using the `NA` symbol which stands for not available. This is different from impossible values, represented by `NaN` (not a number). The function `is.na` returns a data structure having `TRUE` values where the corresponding data is missing and `FALSE` otherwise. The function `complete.cases` returns a vector whose components are `FALSE` for all samples (dataframe rows) containing missing values and `TRUE` otherwise. The function `na.omit` returns a new dataframe omitting all samples (dataframe rows) containing missing values.

Many data analysis functions fail on data containing missing values and simply return `NA` value. This is intended as a warning to the programmer that the data contains missing values. Some functions have a `na.rm` argument, which if set to `TRUE` changes the function behavior so that it proceeds to operate on the supplied data after removing all dataframe rows with missing values.

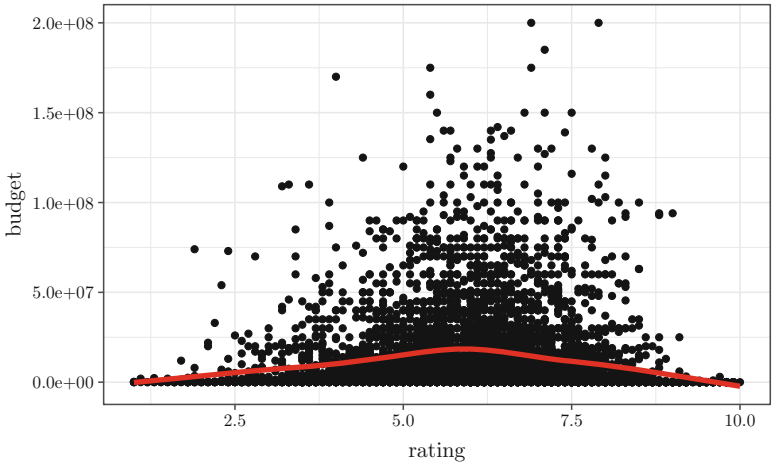
The code below analyzes the dataframe `movies` in the `ggplot2movies` package, which contains 24 attributes (genre, year, budget, user ratings, etc.) for 58,788 movies obtained from the website http://www.imdb.com with some missing values. Type `help(movies)` after loading the package `ggplot2movies` for more information.

```
library(ggplot2movies)
names(movies)
##  [1] "title"       "year"
##  [3] "length"      "budget"
##  [5] "rating"      "votes"
##  [7] "r1"          "r2"
##  [9] "r3"          "r4"
## [11] "r5"          "r6"
## [13] "r7"          "r8"
## [15] "r9"          "r10"
## [17] "mpaa"        "Action"
## [19] "Animation"   "Comedy"
## [21] "Drama"       "Documentary"
## [23] "Romance"     "Short"
# display 20 rows, 6 first columns
movies[9000:9020, 1:6]
## # A tibble: 21 *** 6
##                                 title  year
##                                 <chr> <int>
## 1                          Cat People  1982
## 2  Cat Swallows Parakeet and Speaks!  1996
## 3         Cat That Hated People, The  1948
## 4                  Cat and Dupli-cat  1967
## 5            Cat and the Canary, The  1927
## 6            Cat and the Canary, The  1939
## 7            Cat and the Canary, The  1979
## 8            Cat and the Fiddle, The  1934
## 9          Cat and the Mermouse, The  1949
## 10         Cat from Outer Space, The  1978
```

```
## # ... with 11 more rows, and 4 more
## #   variables: length <int>, budget <int>,
## #   rating <dbl>, votes <int>
mean(movies$length)
## [1] 82.33788
mean(movies$budget)
## [1] NA
mean(movies$budget, na.rm = TRUE)
## [1] 13412513
mean(is.na(movies$budget))
## [1] 0.9112914
```

Below, we create a new dataframe that is identical to the `movies` dataframe with the exception that rows with missing values are removed. Then, we graph movie budget vs. average movie rating to examine the relationship between the two variables. The red line below is a smoothing curve showing the average tendency (see Sect. 8.8 for more details).

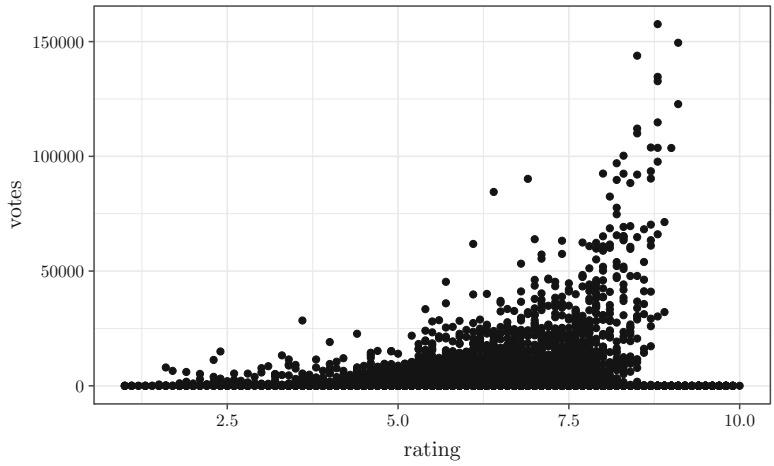
```
moviesNoNA = na.omit(movies)
qplot(rating, budget, data = moviesNoNA, size = I(1.2)) +
  stat_smooth(color = "red", size = I(2), se = F)
```



The graph above shows that there is a high variation in budget, but the general trend is that budget increases with ratings up to a certain point (around 6) and then it decreases. Apparently, high budget movies tend to have decent average IMDB ratings in the range of 5–7.5. The lowest ratings and surprisingly the highest ratings tend to be associated with low-budget movies.

Below we graph total number of IMDB votes vs. average movie rating.

```
moviesNoNA = na.omit(movies)
qplot(rating, votes, data = moviesNoNA, size = I(1.2))
```



The figure above shows that the number of votes (which can be used as a surrogate for popularity) tend to increase as the average rating increases. We also see that the spread in the number of votes increases with the average rating. Finally, it is surprising that the movies featuring the highest average ratings have a very small number of votes. A possible explanation is that when there are only a few votes it is easier to reach very high (or low) average ratings. Consider for example an extreme case where a minor movie is given very high votes by a small circle of fans. In the case of a blockbuster movie with thousands of ratings, opinions vary more and it is much harder to maintain a very high average rating.

When interpreting the graphs above, we should also consider the fact that users tend to see movies that they think they will like, and thus the observed ratings tend to be higher than ratings gathered after showing users random movies.

9.1.2 Missing Data in Python

The `pandas` module in Python provides support for dataframes with missing values. Specifically, `NaN` is used to represent missing values in both floating-point and non-floating-point arrays. In addition, the value `None` can also be used to represent missing values in arrays containing objects.

Pandas provides the following functions for handling missing values: `isnull` returns a boolean array marking whether entries are missing or not; `notnull` returns the boolean negation of `isnull`; `dropna` drops rows or columns containing missing values; `filna` fills-in missing values.

The code below defines a small dataframe with missing values, and demonstrates the `isnull` method.

```
import numpy as np
import pandas as pd
data = pd.DataFrame([[1, 2, np.nan],
                     [3, np.nan, 4],
                     [1, 2, 3]])
print("D:\n" + str(data))
print("pd.isnull(D):\n" + str(pd.isnull(data)))
## D:
##    0    1    2
## 0  1  2.0  NaN
## 1  3  NaN  4.0
## 2  1  2.0  3.0
## pd.isnull(D):
##        0      1      2
## 0  False  False   True
## 1  False   True  False
## 2  False  False  False
```

The code below demonstrates the `dropna` method.

```
import numpy as np
import pandas as pd
data = pd.DataFrame([[1, 2, np.nan],
                     [3, np.nan, 4],
                     [1, 2, 3]])
# drop rows
print("data.dropna():\n" + str(data.dropna()))
# drop columns
print("data.dropna(axis = 1):\n" + str(data.dropna(axis = 1)))
## data.dropna():
##    0    1    2
## 2  1  2.0  3.0
## data.dropna(axis = 1):
##    0
## 0  1
## 1  3
## 2  1
```

Above, we used the optional `axis` argument to drop columns instead of rows. The optional `thresh` argument can be used so that only rows (or columns) with more than a certain number of observations will be retained.

Calling the `fillna` function with a numeric argument replaces missing entries with that value. Passing an optional `dict` object that maps column indices to values replaces missing entries with column-specific constants. Alternatively, the `dict` object can be replaced with an array containing column specific fill-in values. By default, `fillna` returns a new dataframe with filled-in values and does not modify the original dataframe. The optional boolean argument `inplace` can be used to modify that behavior such that the original dataframe is modified.

```
import numpy as np
import pandas as pd

data = pd.DataFrame([[1, 2, np.nan],
                     [3, np.nan, 4],
                     [1, 2, 3]])
print(f'data:\n{data}')
# fill in missing entries with column means
print("data.fillna(data.mean()):")
print(data.fillna(data.mean()))
## data:
##    0    1    2
## 0  1  2.0  NaN
## 1  3  NaN  4.0
## 2  1  2.0  3.0
## data.fillna(data.mean()):
##    0    1    2
## 0  1  2.0  3.5
## 1  3  2.0  4.0
## 2  1  2.0  3.0
```

Many dataframe functions contain an optional `fill_value` argument that can be used to fill-in missing values during the operation of the function.

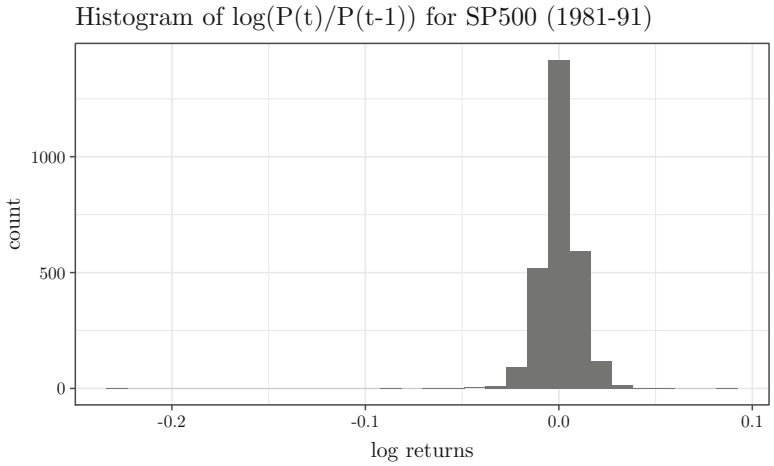
9.2 Outliers

There are two different definitions for outliers. The first considers outliers as corrupted values. That is the case, for example, with human errors during a manual process of entering measurements in a spreadsheet. The second definition considers outliers to be non-corrupt values, but nevertheless are substantially unlikely given our modeling assumptions.

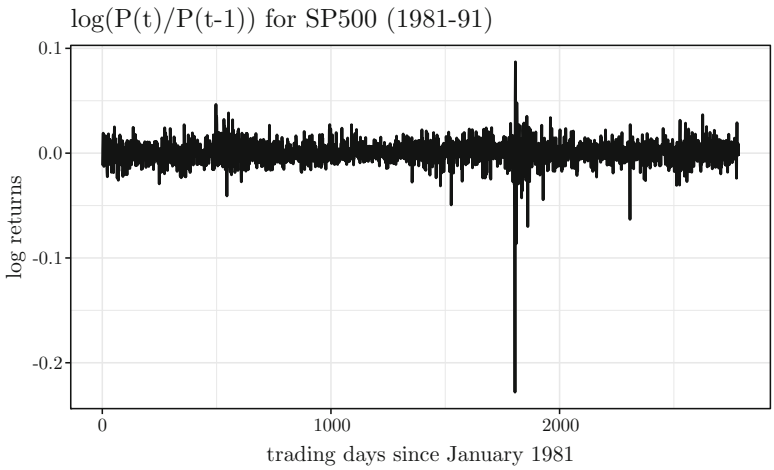
Data analysis based on outliers may result in drastically wrong conclusions. This is pretty clear in the case of corrupted outliers. But it may also be the case with the second definition of outliers, especially when the model used in the data analysis does not account for the extreme observations.

An example for the second outlier definition is the Black Monday stock crash on October 19, 1987, when the Dow Jones Industrial Average lost 22% in one day. The graphs below plot the histogram of log-returns and then the log-returns of the S&P 500 stock index between 1981 and 1991. Log returns is the quantity $\log(P_t/P_{t-1})$, where P_t is the S&P 500 index on day t. A log return close to 0 (return close to 1) indicates no or little daily movement in the index price.

```
library(Ecdat)
data(SP500, package = 'Ecdat')
qplot(r500,
      main = "Histogram of log(P(t)/P(t-1)) for SP500 (1981-91)",
      xlab = "log returns",
      data = SP500)
```




```
qplot(seq(along = r500),
      r500,
      data = SP500,
      geom = "line",
      xlab = "trading days since January 1981",
      ylab = "log returns",
      main = "log(P(t)/P(t-1)) for SP500 (1981-91)")
```



Note how the sharp Black Monday decline barely registers in the histogram of log-returns, but is clearly visible in the log-return graph.

Some models are sensitive to outliers and building such models based on data with outliers can lead to drastically inaccurate predictions. On the other hand, removing outliers is tricky as the resulting model may conclude that future outliers are unlikely to occur.

Robustness describes a lack of sensitivity of data analysis procedures to outliers. An example for a non-robust procedure is computing the mean of n numbers. Assuming a symmetric distribution of samples around 0, we expect the mean to be zero, or at least close to it. But, the presence of a single outlier (very positive value or very negative value) may substantially affect the mean calculation and drive it far away from zero, even for large n.

An example for a robust data analysis procedure is the median, which will not be affected by a single outlier even if it has extreme value. We illustrate this with a hypothetical data of $n + 1$ values $\{x^{(1)}, \ldots, x^{(n)}, e\}$, with median b and mean a. The value $e > \max(x^{(1)}, \ldots, x^{(n)})$ is a single high-valued outlier. Fixing the n observations and increasing e to infinity the median will remain constant at b, while the mean would grow to infinity together with the median. This happens regardless of the value of n. In other words, no matter the size of the dataset, a single extreme outlier will affect the mean in a substantial way but will not affect the median.

Three popular techniques for dealing with outliers are listed below.

Truncating: Remove all values deemed as outliers.

Winsorization: Shrink outliers to border of main part of data. One special case of this is to replace outliers with the most extreme of the remaining values.

Robustness: Analyze the data using a robust procedure.

We assume below that a value is considered an outlier if it is below the α percentile or above the $100 - \alpha$ percentile for some small $\alpha > 0$. In many cases, we assume that the data follows a symmetric distribution, in which case the rule above corresponds to being more than c standard deviations away from the mean. The problem is that since the data contains outliers, the estimates of standard deviation or percentiles based on that data are likely to be corrupted as well. This chicken-and-egg problem can be sidelined by computing the standard deviation or percentiles after removing the most extreme values.

The R code below removes outliers. We sample values from a Gaussian distribution and then overwrite the first entry with a single outlier. We then sort the data, exclude the smallest and largest values, compute the standard deviation, and remove the values beyond a certain multiple of the standard deviation values.

```
original_data = rnorm(20)
original_data[1] = 1000
sorted_data = sort(original_data)
filtered_data = sorted_data[3:18]
lower_limit = mean(filtered_data) - 5 * sd(filtered_data)
upper_limit = mean(filtered_data) + 5 * sd(filtered_data)
not_outlier_ind = (lower_limit < original_data) &
```

```
  (original_data < upper_limit)
print(not_outlier_ind)
##  [1] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
##  [7]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [13]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [19]  TRUE  TRUE
data_w_no_outliers = original_data[not_outlier_ind]
```

The R code below winsorizes data containing an outlier. It uses the function `winsorize` from the `robustHD` package.

```
library(robustHD)
original_data = c(1000, rnorm(10))
print(original_data)
##  [1] 1000.0000000   -0.2674393   -1.6338265
##  [4]    1.8692484    1.1639288   -2.9165898
##  [7]   -0.1578383   -0.6064289   -1.8033750
## [10]   -1.4792288    0.1511776
print(winsorize(original_data))
##  [1]  3.7841721 -0.2674393 -1.6338265
##  [4]  1.8692484  1.1639288 -2.9165898
##  [7] -0.1578383 -0.6064289 -1.8033750
## [10] -1.4792288  0.1511776
```

9.3 Data Transformations

9.3.1 Skewness and Power Transformation

In many cases, data is drawn from a highly skewed distribution that is not well described by one of the common statistical distributions. In some of these cases, a simple transformation may map the data to a form that is well described by common distributions, such as the Gaussian or Gamma distributions (see TAOD volume 1, Chapter 3). A suitable model can then be fitted to the transformed data (if necessary, predictions can be made on the original scale by inverting the transformation).

Power transformations are a family of data transformations for nonnegative values (parameterized by $\lambda \in \mathbb{R}$), defined as follows.

$$f_\lambda(x) = \begin{cases} (x^\lambda - 1)/\lambda & \lambda \neq 0 \\ \log x & \lambda = 0 \end{cases} \qquad x > 0, \quad \lambda \in \mathbb{R}. \tag{9.1}$$

The reason for the algebraic form $(x^\lambda - 1)/\lambda$ rather than the simpler x^λ is that the former choice makes $f_\lambda(x)$ continuous in λ.

The power transformations can also be used to transform negative data by adding a number large enough, so all values are nonnegative and then proceeding according to the definition above (9.1).

Intuitively, the power transform maps x to x^λ, up to multiplication by a constant and addition of a constant. This mapping is convex for $\lambda > 1$ and concave for $\lambda < 1$. A choice of $\lambda < 1$ removes right-skewness (data has a heavy tail to the right) with smaller values of λ resulting in a more aggressive removal of skewness. Similarly, a choice of $\lambda > 1$ removes left-skewness.

One way to select λ is to try different values, graph the resulting histograms, and select one of them. There are also more sophisticated methods for selecting λ based on the maximum likelihood method.

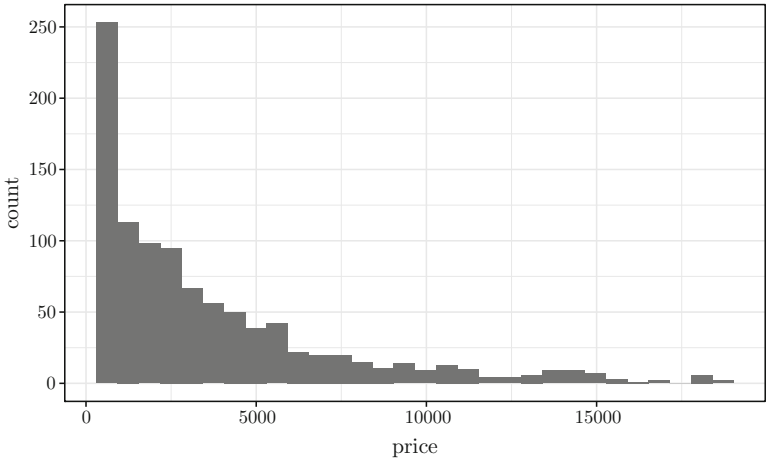
In the example below, we consider the `diamonds` data from the `gpplot2` package, which contains several attributes (cut, carat, price, color, etc.) for 53,940 diamonds.

```
head(diamonds)
## # A tibble: 6 - 10
##   carat       cut color clarity depth table
##   <dbl>     <ord> <ord>   <ord> <dbl> <dbl>
## 1  0.23     Ideal     E     SI2  61.5    55
## 2  0.21   Premium     E     SI1  59.8    61
## 3  0.23      Good     E     VS1  56.9    65
## 4  0.29   Premium     I     VS2  62.4    58
## 5  0.31      Good     J     SI2  63.3    58
## 6  0.24 Very Good     J    VVS2  62.8    57
## # ... with 4 more variables: price <int>,
## #   x <dbl>, y <dbl>, z <dbl>
summary(diamonds)
##      carat               cut
##  Min.   :0.2000   Fair     : 1610
##  1st Qu.:0.4000   Good     : 4906
##  Median :0.7000   Very Good:12082
##  Mean   :0.7979   Premium  :13791
##  3rd Qu.:1.0400   Ideal    :21551
##  Max.   :5.0100
##
##  color         clarity          depth
##  D: 6775   SI1    :13065   Min.   :43.00
##  E: 9797   VS2    :12258   1st Qu.:61.00
##  F: 9542   SI2    : 9194   Median :61.80
##  G:11292   VS1    : 8171   Mean   :61.75
##  H: 8304   VVS2   : 5066   3rd Qu.:62.50
##  I: 5422   VVS1   : 3655   Max.   :79.00
##  J: 2808   (Other): 2531
##      table           price
##  Min.   :43.00   Min.   :  326
##  1st Qu.:56.00   1st Qu.:  950
##  Median :57.00   Median : 2401
##  Mean   :57.46   Mean   : 3933
##  3rd Qu.:59.00   3rd Qu.: 5324
##  Max.   :95.00   Max.   :18823
##
##        x                y
##  Min.   : 0.000   Min.   : 0.000
```

```
##  1st Qu.: 4.710   1st Qu.: 4.720
##  Median : 5.700   Median : 5.710
##  Mean   : 5.731   Mean   : 5.735
##  3rd Qu.: 6.540   3rd Qu.: 6.540
##  Max.   :10.740   Max.   :58.900
##
##        z
##  Min.   : 0.000
##  1st Qu.: 2.910
##  Median : 3.530
##  Mean   : 3.539
##  3rd Qu.: 4.040
##  Max.   :31.800
##
```

We graph the price histogram of a random subset of 1000 diamonds, obtained by sampling rows from the original dataframe.

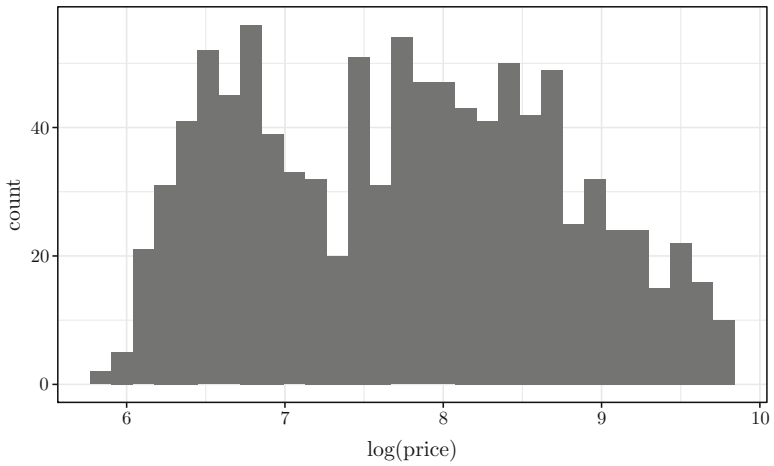
```
diamondsSubset = diamonds[sample(dim(diamonds)[1], 1000),]
qplot(price, data = diamondsSubset)
```



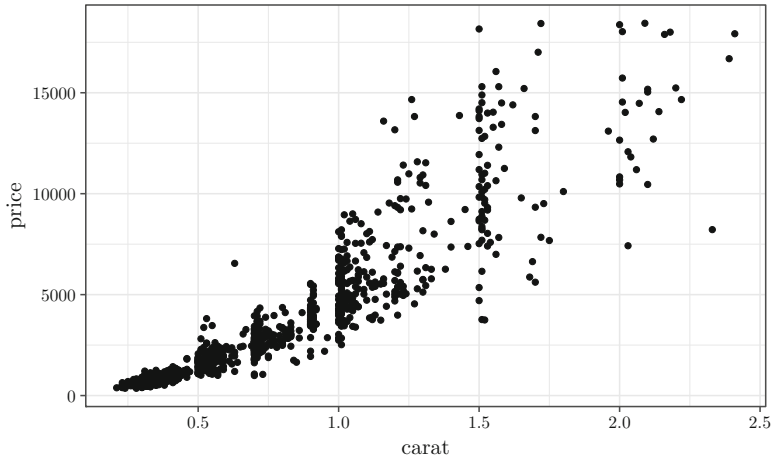
The histogram above is skewed to the right and is not very informative. On the other hand, the transformed histogram below reveals an interesting multi-modal distribution in the log-scale.

```
qplot(log(price), size = I(1), data = diamondsSubset)
```

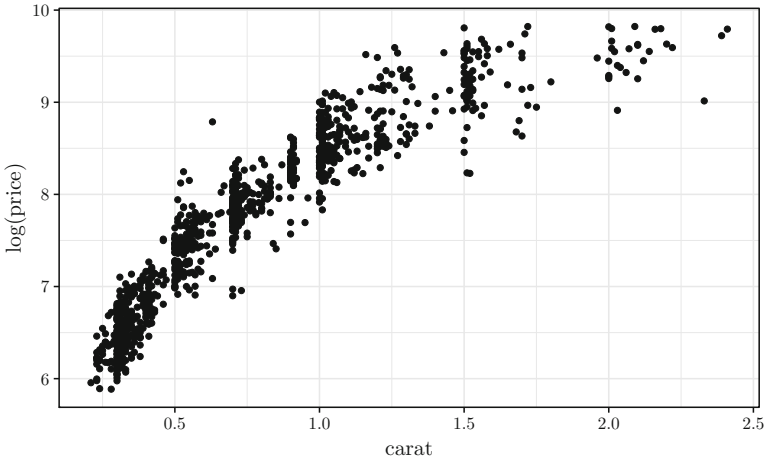
Power transformations are useful also for examining the relationship between two or more data variables. The following plot shows the relationship between diamond price and diamond carat (weight). It is hard to draw much information from that plot beyond the fact that there is a nonlinear increasing trend. Transforming both variables using a logarithm shows a striking linear relationship on a log-log scale.



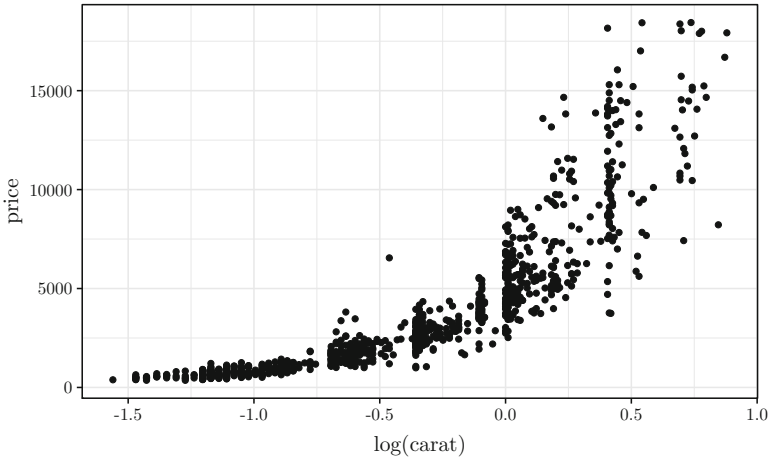

```
qplot(carat,
      price,
      size = I(1),
      data = diamondsSubset)
```




```
qplot(carat,
      log(price),
      size = I(1),
      data = diamondsSubset)
```

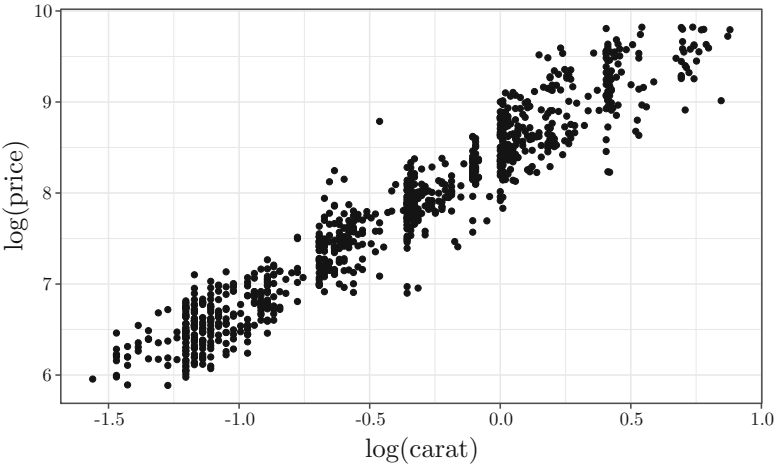



```
qplot(log(carat),
      price,
      size = I(1),
      data = diamondsSubset)
```

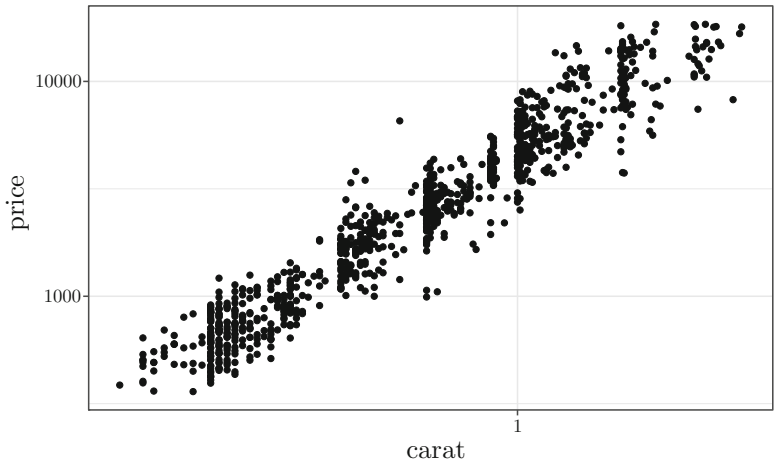



```
qplot(log(carat),
      log(price),
      size = I(1),
      data = diamondsSubset)
```

It is sometimes more readable to display the original un-transformed variables on logarithmic axes.



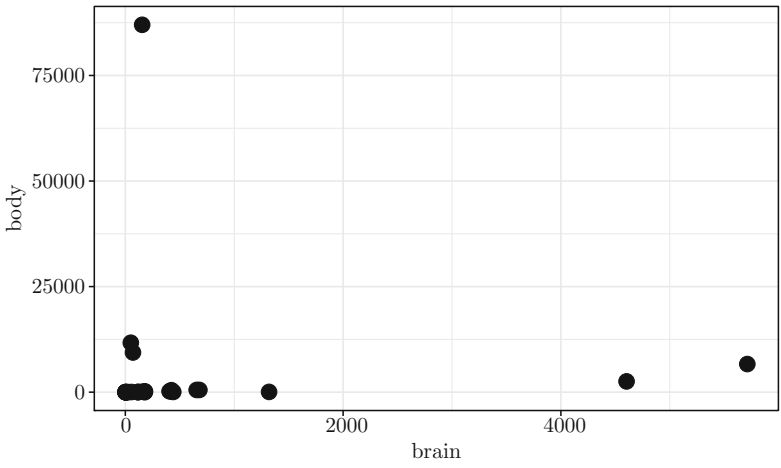

```
qplot(carat,
      price,
      log = "xy",
      size = I(1),
      data = diamondsSubset)
```



In another example we analyze the `Animals` dataset from the `MASS` package. This dataset contains brain weight (in grams) and body weight (in kilograms) for 28 different animal species.

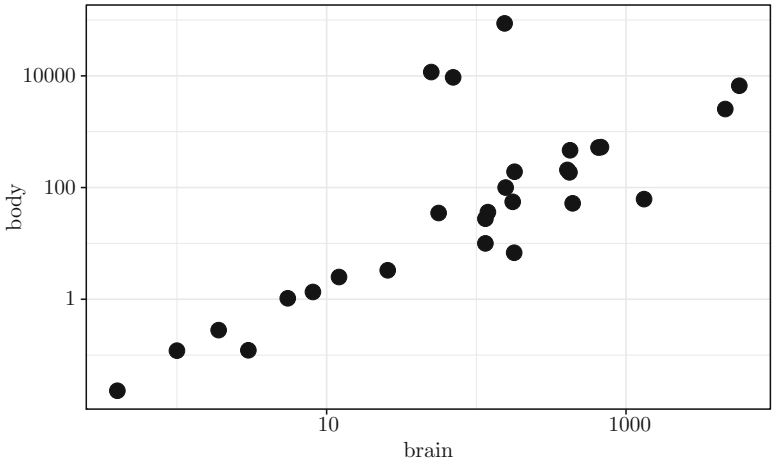
The three largest animals are dinosaurs, whose measurements are obviously the result of scientific modeling rather than precise measurements.

```
library(MASS)
Animals
##                         body  brain
## Mountain beaver        1.350    8.1
## Cow                  465.000  423.0
## Grey wolf             36.330  119.5
## Goat                  27.660  115.0
## Guinea pig             1.040    5.5
## Dipliodocus        11700.000   50.0
## Asian elephant      2547.000 4603.0
## Donkey               187.100  419.0
## Horse                521.000  655.0
## Potar monkey          10.000  115.0
## Cat                    3.300   25.6
## Giraffe              529.000  680.0
## Gorilla              207.000  406.0
## Human                 62.000 1320.0
## African elephant    6654.000 5712.0
## Triceratops         9400.000   70.0
## Rhesus monkey          6.800  179.0
## Kangaroo              35.000   56.0
## Golden hamster         0.120    1.0
## Mouse                  0.023    0.4
## Rabbit                 2.500   12.1
## Sheep                 55.500  175.0
## Jaguar               100.000  157.0
## Chimpanzee            52.160  440.0
## Rat                    0.280    1.9
## Brachiosaurus      87000.000  154.5
## Mole                   0.122    3.0
## Pig                  192.000  180.0
qplot(brain, body, data = Animals)
```



The scatter plot above is hard to comprehend. Transforming both quantities by a power transformation reveals interesting linear trend on the log-log scale for most animals, excluding the three dinosaurs.

```
qplot(brain, body, log = "xy", data = Animals)
```



The log-log plot shows a clear cluster of outliers corresponding to the dinosaurs. Apparently, either dinosaurs had a substantially different brain to body weight relationship than contemporary animals, or the estimates are based on flawed models.

9.3.2 Binning

Below, we discuss representing three different types of variables: numeric, ordinal, and categorical, and computing with them.

- A numeric variable represents real valued measurements, and whose values are ordered in a manner consistent with the natural ordering of the real line. We further expect that the dissimilarity between two measurements a, b is described by the Euclidean distance $|b - a|$. For example, height and weight are numeric variables.
- An ordinal variable represents measurements in a certain range R for which we have a well-defined order relation. Numeric variables are special cases of ordinal variables. For example, the seasons of the year are ordinal measurements.
- A categorical variable represents measurements that do not satisfy the ordinal or numeric assumption. For example, food items on a restaurant's menu are categorical variables.

The distinction above refers to the essence of the variable rather than how it is stored. For example, we may store categorical measurement corresponding to the fruits {`apple, orange, banana`} using the values {0, 1, 2}.

The process of binning (also known as discretization) refers to taking a numeric variable $x \in \mathbb{R}$ (typically a real value, though it may be an integer), dividing its range into several bins, and replacing it with a number representing the corresponding bin. Binning is closely related to rounding and in fact all numbers represented digitally are already discretized (see Chapter 1). However, it is often useful to bin values in order to accomplish data reduction, improve scalability for big-data, or capture nonlinear effects in linear models. A special case of binning is binarization, which replaces a variable with either 0 or 1 depending on whether the variable is greater or smaller than a certain threshold.

For example, suppose x represents the tenure of an employee (in years) and ranges from 0 to 50. A binning process may divide the range $[0, 50]$ into the following ranges $(0, 10]$, $(10, 20]$, $\ldots$, $(41, 50]$ and use corresponding replacement values of $5, 15, \ldots, 45$ respectively. The notation $(a, b]$ corresponds to all values larger than a and smaller or equal to b.

The code below shows how to discretize a variable in Python using the `cut` function in the `pandas` module. The output of the `cut` function is an object that has a `categories` field corresponding to the discretized bins, and a `codes` field corresponding to the index of the bin that replaces each value in the data array. Specifically, below we create an array, define a list of bin ranges and replacement values, call `cut` to replace the original values with the corresponding bins, and then create another array that replaces the values of the original array with the midpoints of the corresponding bins.

```
import numpy as np
import pandas as pd
data = [23, 13, 5, 3, 41, 33]
bin_boundaries = [0, 10, 20, 30, 40 ,50]
bin_values = [5, 15, 25, 35, 45]
cut_data = pd.cut(data, bin_boundaries)
print("labels:\n" + str(cut_data.codes))
binned_data = np.zeros(shape = np.size(data))
for (k,x) in enumerate(cut_data.codes):
    binned_data[k] = bin_values[x]
print("binned_data:\n" + str(binned_data))
## labels:
## [2 1 0 0 4 3]
## binned_data:
## [ 25.  15.   5.   5.  45.  35.]
```

Calling `cut(data,k)` for a positive integer `k` uses `k` equal-length bins whose coverage equals the range of the data (minimum value, maximum value). In some cases it makes sense to select the bins adaptively, i.e., based on the distribution of the data. The Python function `qcut(data,k)` is similar to `cut(data, k)` except that it uses `k` bins with equal probability mass, where the data distribution is estimated based on the data values. For example `cut(data,2)` performs binning

based on two bins whose boundary is the median of the data (see Sect. 8.10 for a definition of the median). Similarly, `cut(data, 4)` uses bins defined by the sample quartiles. This technique is called quantile binning.

```
import numpy as np
import pandas as pd
original_data = [23, 13, 5, 3, 41, 33]
cut_data = pd.qcut(original_data, 2)
print("C.labels:\n" + str(cut_data.codes))
## C.labels:
## [1 0 0 0 1 1]
```

Discretization in R is similar to Python using the R `cut` function.

9.3.3 Indicator Variables

Indicator vectors refer to a technique that replaces a variable x (numeric, ordinal, or categorical) taking k values with a binary k-dimensional vector v, such that `v[i]` (or v_i in mathematical notation) is one if and only if x takes on the i-value in its range. Thus, the variable is replaced by a vector that is all zeros, except for one component that equals one corresponding to the variable value. Often, indicator variables are used in conjunction with binning as follows: first bin the variable into k bins and then create a k dimensional indicator variable. The resulting vector may have high dimension, but it may be easily handled using computational routines that take advantage of its extreme sparsity.

Indicator vectors are useful in data modeling in two cases.

1. Models for numeric or binary data cannot directly model ordinal or categorical data. For example, replacing {`apple, orange, banana`} with {0, 1, 2} and using these values in a model for numeric values would incorrectly assume that banana is greater than orange and that the dissimilarity between orange and apple is identical to the dissimilarity between apple and banana. Using indicator variables can mitigate this problem.
2. In some cases a linear model is used to model numeric variables that exhibit nonlinear relationship. As the linearity assumption of the model is violated, the resulting model may be inaccurate. An alternative is to first transform the numeric values using several nonlinear transformations (for example, multiple power transformations), then bin the transformed data, and finally create indicator vectors to represent the binned values. Training a linear models on the resulting vectors may capture complex nonlinear relationships.
3. It is often much easier to compute with indicator functions since they are binary, and thus replacing numeric variables with indicator vectors may improve scalability.

The Python code below defines a numeric array, bins it, and then create indicator vectors to replace the binned values.

```
import numpy as np
import pandas as pd
data = [23, 13, 5, 3, 41, 33]
indicator_values = pd.get_dummies(pd.qcut(data, 2))
print("indicator_values:\n" + str(indicator_values))
## indicator_values:
##    [3, 18]  (18, 41]
## 0        0         1
## 1        1         0
## 2        1         0
## 3        1         0
## 4        0         1
## 5        0         1
```

In many cases, a machine learning routine accepts a single feature vector that needs to be composed from multiple variables. A common strategy is to transform all numeric variable by binning them and then creating indicator vectors and concatenating all of them into a single long binary vector. Additional non-numeric variables can be concatenated as well. Some variations on this strategy is to concatenate the numeric vectors in their original form and possibly concatenate them with transformed versions of the numeric variables (for example, power transformations).

9.4 Data Manipulation

We discuss below several common operations that manipulate the structure of dataframes.

9.4.1 Random Sampling, Partitioning, and Shuffling

A common operation in data analysis is to select a random subset of the rows of a dataframe, with or without replacement. For example, a subset of size 2 without replacement corresponds to selecting one row, and then selecting another row from the remaining rows. Sampling with replacement may select the same row multiple times.

We demonstrate these tasks using R code. Python's pandas module features similar functionality. The R `sample` function accepts a vector of values from which to sample (typically a vector of row indices), the number of samples, whether the sampling is done with or without replacement, and the probability of sampling different values. By default, the probability of sampling each value is identical: $1/k$ where k is the number of values from which we sample. For example, the R code below samples 3 times from the values 1,2,3,4, without replacements.

```
sampled_row_indices = sample(1:4, 3, replace=FALSE)
print(sampled_row_indices)
## [1] 2 4 3
```

After obtaining the indices that we wish to sample, we form a new array or dataframe containing the sampled rows of the original dataframe.

```
D = array(data = seq(1, 20, length.out = 20), dim = c(4, 5))
D_sampled = D[sampled_row_indices,]
print(D_sampled)
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    2    6   10   14   18
## [2,]    4    8   12   16   20
## [3,]    3    7   11   15   19
```

In some cases, we need to partition the dataset's rows into two or more collection of rows. To do so, we proceed with generating a random permutation of k objects (using `sample(k,k)`), where k is the number of rows in the data, and then divide the permutation vector into two or more parts based on the prescribed sizes. We then create new dataframes whose rows correspond to the divided permutation vector. For example, the R code below partitions an array into two new arrays of sizes 75% and 25%.

```
D = array(data = seq(1, 20, length.out = 20), dim = c(4, 5))
print(D)
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    5    9   13   17
## [2,]    2    6   10   14   18
## [3,]    3    7   11   15   19
## [4,]    4    8   12   16   20
rand_perm = sample(4,4)
first_set_of_indices = rand_perm[1:floor(4*0.75)]
second_set_of_indices = rand_perm[(floor(4*0.75)+1):4]
D1 = D[first_set_of_indices,]
D2 = D[second_set_of_indices,]
print(D1)
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    5    9   13   17
## [2,]    2    6   10   14   18
## [3,]    4    8   12   16   20
print(D2)
## [1]  3  7 11 15 19
```

A related task is data shuffling, which randomly shuffles the dataframe rows.

```
D = array(data = seq(1, 20, length.out = 20), dim = c(4, 5))
print(D)
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    5    9   13   17
## [2,]    2    6   10   14   18
## [3,]    3    7   11   15   19
## [4,]    4    8   12   16   20
```

```
D_shuffled = D[sample(4, 4),]
print(D_shuffled)
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    3    7   11   15   19
## [2,]    4    8   12   16   20
## [3,]    1    5    9   13   17
## [4,]    2    6   10   14   18
```

9.4.2 Concatenations and Joins

Often, data needs to be aggregated from multiple sources into a single object that will be used for visualization and modeling. We demonstrate below how to do it with Python. R features similar capability.

In some cases the two data sources contain different records (dataframe rows) annotated with the same attribute names (column names). In this case aggregating the two sources is simply concatenating the rows of the two dataframes. The Python code below creates two dataframes with identical column names and then concatenates them.

```
import numpy as np
import pandas as pd
data1 = {"ID" : ["2134", "4524"],
         "name" : ["John Smith", "Jane Doe"]}
D1 = pd.DataFrame(data1)
data2 = {"ID" : ["9423", "3483"],
         "name" : ["Bob Jones", "Mary Smith"]}
D2 = pd.DataFrame(data2)
print("D1:\n" + str(D1))
print("D2:\n" + str(D2))
D3 = pd.concat([D1, D2])
print("concatenation of D1, D2:\n" + str(D3))
## D1:
##      ID        name
## 0  2134  John Smith
## 1  4524    Jane Doe
## D2:
##      ID        name
## 0  9423   Bob Jones
## 1  3483  Mary Smith
## concatenation of D1, D2:
##      ID        name
## 0  2134  John Smith
## 1  4524    Jane Doe
## 0  9423   Bob Jones
## 1  3483  Mary Smith
```

When concatenating dataframes with nonidentical columns, new columns are added to the concatenated dataframe with missing values filled in as needed.

```
import numpy as np
import pandas as pd
data1 = {"ID" : ["2134", "4524"],
         "name" : ["John Smith", "Jane Doe"]}
D1 = pd.DataFrame(data1)
data2 = {"ID" : ["9423", "3483"],
         "nick name" : ["Bobby", "Abby"]}
D2 = pd.DataFrame(data2)
print("D1:\n" + str(D1))
print("D2:\n" + str(D2))
D3 = pd.concat([D1, D2])
print("concatenation of D1, D2:\n" + str(D3))
## D1:
##      ID        name
## 0  2134  John Smith
## 1  4524    Jane Doe
## D2:
##      ID nick name
## 0  9423     Bobby
## 1  3483      Abby
## concatenation of D1, D2:
##      ID        name nick name
## 0  2134  John Smith       NaN
## 1  4524    Jane Doe       NaN
## 0  9423         NaN     Bobby
## 1  3483         NaN      Abby
```

Thus far, we assumed that the records in both data sources correspond to distinct entities. In some cases the same entity is described in two data sources that potentially list different attributes. However, in order to determine how to match records in one data source to records in another source we assume that each data source has one or more columns acting as identifiers. The identifiers can be used to match a row in one dataframe with a row in a different dataframe. The operation of merging such data is called a join and the identifier column is called the key.

We create below two dataframes where the identifier attribute is named `ID` in both data sources. An inner join operation retains records that appear in both data sources and all attributes that appear in either the first or the second data source.

```
import numpy as np
import pandas as pd
data1 = {"ID" : ["2134", "4524"],
         "name" : ["John Smith", "Jane Doe"]}
D1 = pd.DataFrame(data1)
data2 = {"ID" : ["6325", "2134"],
         "age" : [25, 35],
       "tenure" : [3, 8]}
D2 = pd.DataFrame(data2)
print("D1:\n" + str(D1))
print("D2:\n" + str(D2))
D3 = pd.merge(D1, D2, on = 'ID', how = 'inner')
print("inner join of D1, D2:\n" + str(D3))
```

```
## D1:
##      ID        name
## 0  2134  John Smith
## 1  4524    Jane Doe
## D2:
##      ID  age  tenure
## 0  6325   25       3
## 1  2134   35       8
## inner join of D1, D2:
##      ID        name  age  tenure
## 0  2134  John Smith   35       8
```

An outer join operation is similar to inner join, except that all records and all attributes are retained, with missing values (denoted by NA) filled in as needed. A left join is similar, except that all common records and attributes are retained plus the records and attributes in the left data source. A right join is similar, except that all common records and attributes are retained plus the records and attributes in the right data source.

```
import numpy as np
import pandas as pd
data1 = {"ID" : ["2134", "4524"],
         "name" : ["John Smith", "Jane Doe"]}
D1 = pd.DataFrame(data1)
data2 = {"ID" : ["6325", "2134"],
         "age" : [25, 35],
         "tenure" : [3, 8]}
D2 = pd.DataFrame(data2)
D3 = pd.merge(D1, D2, on = 'ID', how = 'outer')
print("outer join of D1, D2:\n" + str(D3))
D4 = pd.merge(D1, D2, on = 'ID', how = 'left')
print("left join of D1, D2:\n" + str(D4))
## outer join of D1, D2:
##      ID        name   age  tenure
## 0  2134  John Smith  35.0     8.0
## 1  4524    Jane Doe   NaN     NaN
## 2  6325         NaN  25.0     3.0
## left join of D1, D2:
##      ID        name   age  tenure
## 0  2134  John Smith  35.0     8.0
## 1  4524    Jane Doe   NaN     NaN
```

Above, we assumed that the key attribute acts as a unique identifier in both data sources (any specific key appears at most one time). If we have multiple records with the same key in one or both of the data sources, the join operation forms multiple combinations.

```
import numpy as np
import pandas as pd
data1 = {"ID" : ["2134", "4524", "2134"],
         "name" : ["John Smith", "Jane Doe", "JOHN SMITH"]}
D1 = pd.DataFrame(data1)
```

```
data2 = {"ID" : ["6325", "2134"],
         "age" : [25, 35],
         "tenure" : [3, 8]}
D2 = pd.DataFrame(data2)
D3 = pd.merge(D1, D2, on = 'ID', how = 'outer')
print("outer join of D1, D2:\n" + str(D3))
## outer join of D1, D2:
##      ID        name   age  tenure
## 0  2134  John Smith  35.0     8.0
## 1  2134  JOHN SMITH  35.0     8.0
## 2  4524    Jane Doe   NaN     NaN
## 3  6325         NaN  25.0     3.0
```

We also assumed above that the non-key attribute names are different in both data sources. If they are not, a suffix is introduced in the merged dataframe.

```
import numpy as np
import pandas as pd
data1 = {"ID" : ["2134", "4524"],
         "f1" : ["John Smith", "Jane Doe"]}
D1 = pd.DataFrame(data1)
data2 = {"ID" : ["6325", "2134"],
         "f1" : [25, 35],
         "f2" : [3, 8]}
D2 = pd.DataFrame(data2)
D3 = pd.merge(D1, D2, on = 'ID', how = 'outer')
print("outer join of D1, D2:\n" + str(D3))
## outer join of D1, D2:
##      ID        f1_x  f1_y   f2
## 0  2134  John Smith  35.0  8.0
## 1  4524    Jane Doe   NaN  NaN
## 2  6325         NaN  25.0  3.0
```

9.4.3 Tall Data and Wide Data

Data in tall format is an array or dataframe containing multiple columns where one or more columns act as a unique identifier and an additional column represents value. For example, consider the tall data below representing item sales in a grocery store.

```
2015/01/01  apples  200
2015/01/01  oranges 150
2015/01/02  apples  220
2015/01/02  oranges 130
```

Such data may represent the entire sales records of the store. This format is convenient for adding new records incrementally representing additional sales as they occur, and for removing old records (possibly due to returns in the above case

representing a store). The designation "tall data" reflects the fact that in most cases tall data has many rows but only a few columns.

A disadvantage of tall data format is that it is not easy for conducting analysis or summarizing. For example, we may want to compute the total sales for each day or the average daily sales of a specific item such as apples. Computing these quantities from tall data requires writing a program that will collect all relevant rows, aggregate it appropriately, and produce the desired quantity. This is not only time consuming from a programming perspective, but it may also be time consuming in terms of computing time (if there are many rows).

Wide data represents the same information as tall data, but may represent in multiple columns the information that tall data holds in multiple rows. For example, the wide data version of the grocery data above is listed below.

```
Date       apples  oranges
-------------------------
2015/01/01  200     150
2015/01/02  220     130
```

Wide data is usually simpler to analyze. For example, computing the total sales per day requires summing over each row and computing the average daily per-item sales requires computing averages over columns.

Note that when converting the above data from tall to wide, the first column and the second column represent unique keys and the last column represents the corresponding value. In other words, each (date, item) combination can appear only once, while no such restriction applies to the third column. In converting the tall data to wide, the first column in the tall data became the first column in the wide data, the second column in the tall data became the column names in the wide data, and the third column formed the remaining table entries.

9.4.4 Reshaping Data

In this section we describe how to convert data from tall to wide format and vice versa using the reshape2 package in R (Wickham, 2007). Similar functionality is available in Python's pandas module.

The `melt` function accepts a dataframe in a wide format, and the indices of the columns that act as unique identifiers (remaining columns act as measurements or values). It returns a tall version of the dataframe. The R code below demonstrates this with two different selection of identifier columns.

```
library(reshape2)
# toy (wide) dataframe in the reshape2 package
smiths
##      subject time age weight height
## 1 John Smith    1  33     90   1.87
## 2 Mary Smith    1  NA     NA   1.54
```

```
# columns 2, 3, 4, 5 are measurements, 1 is key
melt(smiths, id = 1)
##      subject variable value
## 1 John Smith     time  1.00
## 2 Mary Smith     time  1.00
## 3 John Smith      age 33.00
## 4 Mary Smith      age    NA
## 5 John Smith   weight 90.00
## 6 Mary Smith   weight    NA
## 7 John Smith   height  1.87
## 8 Mary Smith   height  1.54
# columns 3, 4, 5 are measurements, 1,2 are key
melt(smiths, id = c(1, 2))
##      subject time variable value
## 1 John Smith    1      age 33.00
## 2 Mary Smith    1      age    NA
## 3 John Smith    1   weight 90.00
## 4 Mary Smith    1   weight    NA
## 5 John Smith    1   height  1.87
## 6 Mary Smith    1   height  1.54
```

Note that the tall data produced by `melt` features appropriate column names.

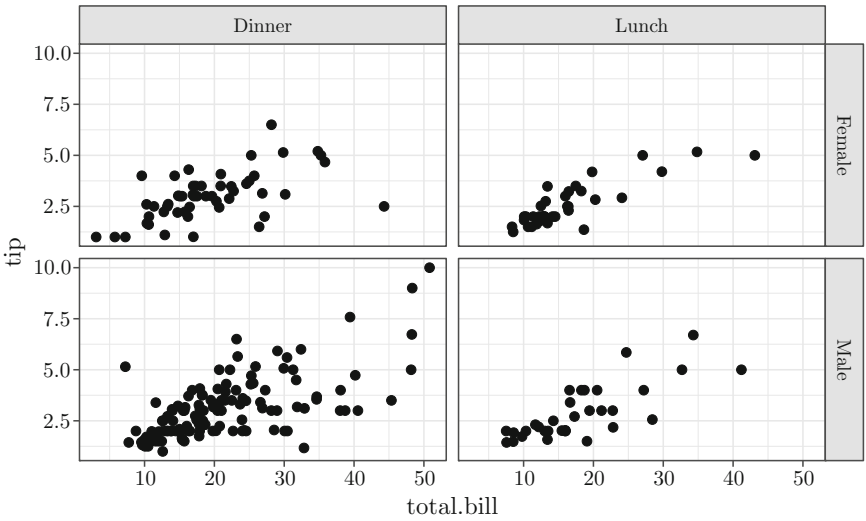
The functions `acast` or `dcast` (the first returns an array and the second a dataframe) represent the inverse of the melt operation. Their argument is a dataframe in wide form and the second is a formula $a \sim b \sim \cdots \sim$ where each of $a, b, \ldots$ represents a list of variables whose values will be displayed along the dimensions of the returned array or dataframe (a for rows, b for columns, etc.). The cast array or dataframe may have at most one value in each cell. If there are more than a single value setting, a third argument `fun.aggregate` executes the corresponding function in order to aggregate the multiple values into a single value.

The example below uses the `tips` dataset from the `reshape2` package. It contains 244 restaurant tips. Dataframe columns include tip, bill, gender of payer, smoker/nonsmoker, day of the week, time of day, and size of party. Type `help(tips)` for more information.

```
tips$total.bill = tips$total_bill
qplot(total.bill,
      tip,
      facets = sex~time,
      size = I(1.5),
      data = tips)
```

We can see from the figure that (1) tip sizes increase with the bill, (2) the variability in tip sizes increase with the bill, (3) bills and tips in dinner are larger than in lunch, (4) there are more dinner tip observations than lunch tip observations, and (5) there are more male payers than female payers.

In the example below we are interested in analyzing tips and bills and the dependence of these variables on the remaining variables. We thus denote the tip and total bill as the measurement variables and the remaining variables as identifiers.




```
library(reshape2)
head(tips)  # first six rows
##   total_bill  tip    sex smoker day   time
## 1      16.99 1.01 Female     No Sun Dinner
## 2      10.34 1.66   Male     No Sun Dinner
## 3      21.01 3.50   Male     No Sun Dinner
## 4      23.68 3.31   Male     No Sun Dinner
## 5      24.59 3.61 Female     No Sun Dinner
## 6      25.29 4.71   Male     No Sun Dinner
##   size total.bill
## 1    2      16.99
## 2    3      10.34
## 3    3      21.01
## 4    2      23.68
## 5    4      24.59
## 6    4      25.29
tipsm = melt(tips,
             id = c("sex","smoker","day","time","size"))
head(tipsm)  # first six rows
##      sex smoker day   time size   variable
## 1 Female     No Sun Dinner    2 total_bill
## 2   Male     No Sun Dinner    3 total_bill
## 3   Male     No Sun Dinner    3 total_bill
## 4   Male     No Sun Dinner    2 total_bill
## 5 Female     No Sun Dinner    4 total_bill
## 6   Male     No Sun Dinner    4 total_bill
##   value
## 1 16.99
## 2 10.34
## 3 21.01
## 4 23.68
```

```
## 5 24.59
## 6 25.29
tail(tipsm)  # last six rows
##        sex smoker  day   time size
## 727 Female     No  Sat Dinner    3
## 728   Male     No  Sat Dinner    3
## 729 Female    Yes  Sat Dinner    2
## 730   Male    Yes  Sat Dinner    2
## 731   Male     No  Sat Dinner    2
## 732 Female     No Thur Dinner    2
##       variable value
## 727 total.bill 35.83
## 728 total.bill 29.03
## 729 total.bill 27.18
## 730 total.bill 22.67
## 731 total.bill 17.82
## 732 total.bill 18.78
# Mean of measurement variables broken by sex.
# Note the role of mean as the aggregating function.
dcast(tipsm,
      sex~variable,
      fun.aggregate = mean)
##      sex total_bill      tip total.bill
## 1 Female   18.05690 2.833448   18.05690
## 2   Male   20.74408 3.089618   20.74408
# Number of occurrences for measurement variables broken by sex.
# Note the role of length as the aggregating function.
dcast(tipsm,
      sex~variable,
      fun.aggregate = length)
##      sex total_bill tip total.bill
## 1 Female         87  87         87
## 2   Male        157 157        157
# Average total bill and tip for different times
dcast(tipsm,
      time~variable,
      fun.aggregate = mean)
##     time total_bill      tip total.bill
## 1 Dinner   20.79716 3.102670   20.79716
## 2  Lunch   17.16868 2.728088   17.16868
# Similar to above with breakdown for sex and time:
dcast(tipsm,
      sex+time~variable,
      fun.aggregate = length)
##      sex   time total_bill tip total.bill
## 1 Female Dinner         52  52         52
## 2 Female  Lunch         35  35         35
## 3   Male Dinner        124 124        124
## 4   Male  Lunch         33  33         33
# Similar to above, but with mean and added margins
dcast(tipsm,
      sex+time~variable,
      fun.aggregate = mean,
```

```
      margins = TRUE)
##      sex   time total_bill      tip
## 1 Female Dinner   19.21308 3.002115
## 2 Female  Lunch   16.33914 2.582857
## 3 Female  (all)   18.05690 2.833448
## 4   Male Dinner   21.46145 3.144839
## 5   Male  Lunch   18.04848 2.882121
## 6   Male  (all)   20.74408 3.089618
## 7  (all)  (all)   19.78594 2.998279
##   total.bill    (all)
## 1   19.21308 13.80942
## 2   16.33914 11.75371
## 3   18.05690 12.98241
## 4   21.46145 15.35591
## 5   18.04848 12.99303
## 6   20.74408 14.85926
## 7   19.78594 14.19005
```

The melt and cast analysis above suggests the following conclusions with respect to the `tips` dataframe.

1. On average, males pay higher total bill and tip than females.
2. Males pay more frequently than females.
3. Dinner bills and tips are generally higher than lunch bills and tips.
4. Males pay disproportionately more times for dinner than they do for lunch (this holds much less for females).
5. Even accounting for (4) by conditioning on paying for lunch or dinner, males still pay higher total bills and tips than females.

A graphical investigation using faceted scatter plots and histograms may reveal similar conclusions. Nevertheless, the above analysis using cast and melt has an advantage over graphical analysis in that a proficient user can very quickly observe properties of the data that are hard to graph (number of measurements for different combination of identifier variables, means of different groups, etc.).

The online package documentation or (Wickham, 2007) provides additional information on the `reshape2` package.

9.4.5 The Split-Apply-Combine Framework

Many data analysis operations on dataframes can be decomposed to three stages:

1. splitting the dataframe along some dimensions to form smaller arrays or dataframes,
2. applying some operation to each of the smaller arrays or dataframes, and
3. combining the results of the application stage into a single meaningful array or dataframe.

Repeatedly programming all three stages whenever we need to compute a data summary may be tedious and can lead to errors. The plyr package in R (Wickham, 2011) automates this process, letting the analyst concentrate on the data analysis task rather than tedious programming.

The plyr package implements the following functions that differ in the type of input arguments they receive and the type of output they provide.

output	array	dataframe	list	discarded
input				
array	`aaply`	`adply`	`alply`	`a_ply`
dataframe	`daply`	`ddply`	`dlply`	`d_ply`
list	`laply`	`ldply`	`llply`	`l_ply`

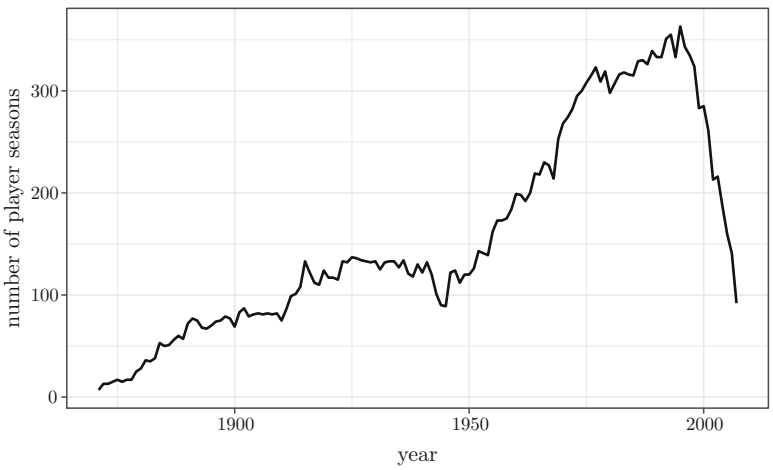
The first argument in each of these functions is the data stored as an array, dataframe, or list depending on the input type in the table above. The second argument of the `a*ply` functions,[1] called `.margins`, determines the dimensions that are used to split the data. A value of 1 implies splitting by rows. A value of 2 implies splitting into columns, and so forth. A combination value may also be used, for example `c(1,2)` splits the data into a combination of rows and columns. The second argument of the `d*ply` functions, called `.variables`, determines the dataframe columns (multiple columns are allowed) that are used to split the data. Since there is only one way to split a list there is no corresponding argument for the `l*ply` functions. For all functions the argument `.fun` determines which function to execute in the apply stage.

We illustrate these functions using the `baseball` dataset from the `plyr` package. This dataset contains variables such as year, team, number of runs, and number of strikeouts for 1228 baseball players. Each row records the performance of a baseball player during one baseball season. In particular, the performance of specific players is spread across multiple rows, one corresponding to each year played. The following example is inspired by the R help listing of the `ddply` function.

```
library(plyr)
head(baseball)
##             id year stint team lg  g  ab  r
## 4   ansonca01 1871     1  RC1    25 120 29
## 44  forceda01 1871     1  WS3    32 162 45
## 68  mathebo01 1871     1  FW1    19  89 15
## 99  startjo01 1871     1  NY2    33 161 35
## 102 suttoez01 1871     1  CL1    29 128 35
## 106 whitede01 1871     1  CL1    29 146 40
##      h X2b X3b hr rbi sb cs bb so ibb hbp sh
```

[1] We use the asterisk in `a*ply` and elsewhere to indicate a collection of functions obtained by substituting the asterisk with other characters.

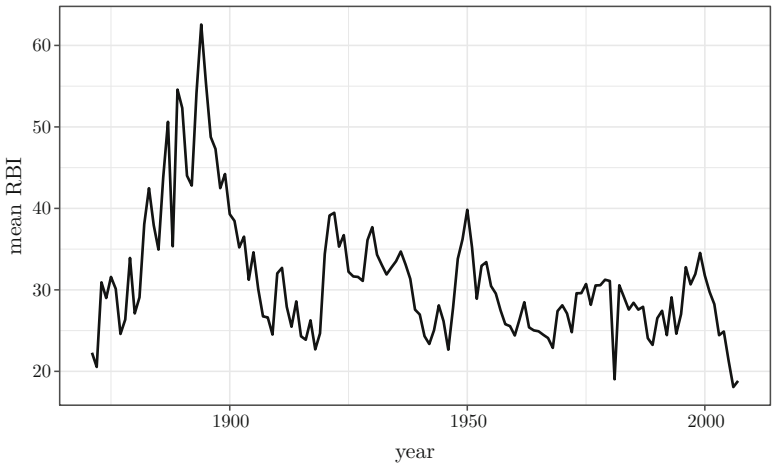
```
## 4   39  11   3  0  16  6  2  2  1  NA  NA NA
## 44  45   9   4  0  29  8  0  4  0  NA  NA NA
## 68  24   3   1  0  10  2  1  2  0  NA  NA NA
## 99  58   5   1  1  34  4  2  3  0  NA  NA NA
## 102 45   3   7  3  23  3  1  1  0  NA  NA NA
## 106 47   6   5  1  21  2  2  4  1  NA  NA NA
##      sf gidp
## 4    NA   NA
## 44   NA   NA
## 68   NA   NA
## 99   NA   NA
## 102 NA   NA
## 106 NA   NA
# count number of players recorded for each year
bbPerYear = ddply(baseball, "year", "nrow")
head(bbPerYear)
##   year nrow
## 1 1871    7
## 2 1872   13
## 3 1873   13
## 4 1874   15
## 5 1875   17
## 6 1876   15
qplot(x = year,
      y = nrow,
      data = bbPerYear,
      geom = "line",
      ylab="number of player seasons")
```



The number of player seasons recorded in the dataset for each year increases dramatically from 1880 to around 1990, but decreases in later years close to the

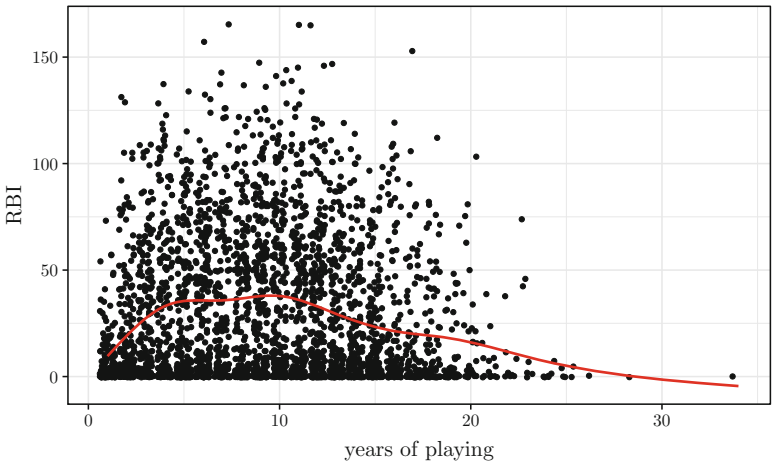
turn of the century. The simple one line command above would be otherwise more complicated.

```
# compute mean rbi (batting attempt resulting in runs)
# for all years. Summarize is the apply function, which
# takes as argument a function that computes the rbi mean
bbMod=ddply(baseball,
            "year",
            summarise,
            mean.rbi = mean(rbi, na.rm = TRUE))
head(bbMod)
##   year mean.rbi
## 1 1871 22.28571
## 2 1872 20.53846
## 3 1873 30.92308
## 4 1874 29.00000
## 5 1875 31.58824
## 6 1876 30.13333
qplot(x = year,
      y = mean.rbi,
      data = bbMod,
      geom = "line",
      ylab = "mean RBI")
```



The graph above shows that the mean of the RBI, which is a batting performance measure, was substantially higher in the late nineteenth century than in other times. Below, we add another variable to the dataframe containing the number of years a player has played baseball up to that point. It is computed as the current year minus the year the player started playing plus one.

```
# add a column career.year which measures the number of years
# passed since each player started batting
bbMod2 = ddply(baseball,
               "id",
               transform,
               career.year = year - min(year) + 1)
# sample a random subset 3000 rows to avoid over-plotting
bbSubset = bbMod2[sample(dim(bbMod2)[1], 3000),]
qplot(career.year,
      rbi, data = bbSubset,
      size = I(0.8),
      geom = "jitter",
      ylab = "RBI",
      xlab = "years of playing") +
  geom_smooth(color = "red", se = F, size = 1.5)
```

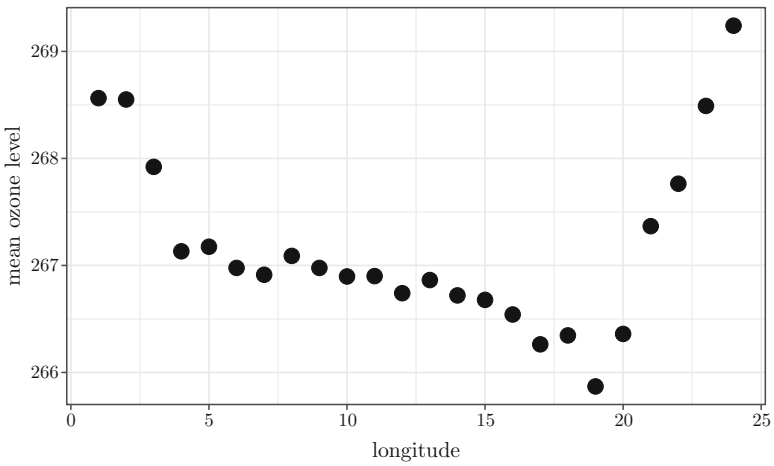


The RBI tends to improve with experience up to 7 or 8 years and then starts to decline (on average). RBI for players with more than 20 years of experience tends to be very low, although a few exceptions exist (outliers at the top-right corner of the figure above).

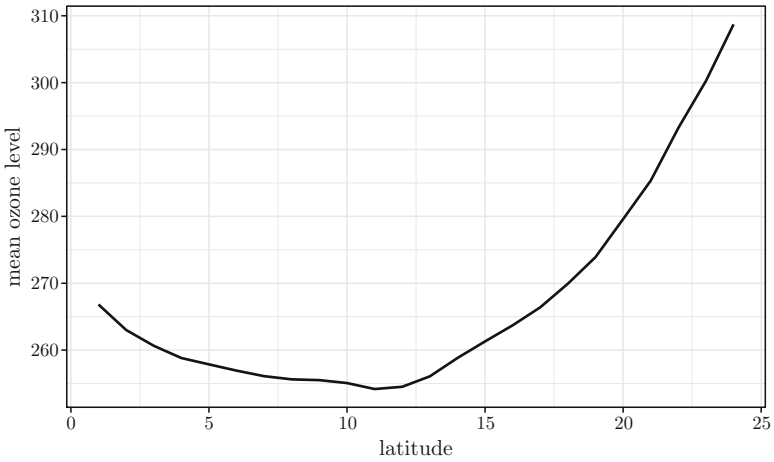
The example below explores the function `aaply` using the `ozone` dataset from the `plyr` package. The ozone dataset contains a 3-dimensional array of ozone measurements varying by latitude, longitude, and time.

```
library(plyr)
dim(ozone)
## [1] 24 24 72
latitude.mean = aaply(ozone, 1, mean)
longitude.mean = aaply(ozone, 2, mean)
time.mean = aaply(ozone, 3, mean)
longitude = seq(along = longitude.mean)
```

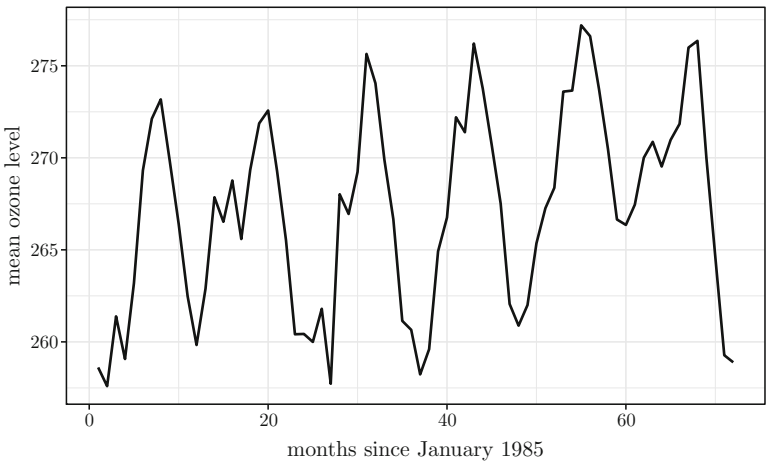
```
qplot(x = longitude,
      y = longitude.mean,
      ylab = "mean ozone level")
```




```
latitude = seq(along = latitude.mean)
qplot(x = latitude,
      y = latitude.mean,
      ylab = "mean ozone level",
      geom = "line")
```



```
months = seq(along = time.mean)
qplot(x = months,
      y = time.mean,
      geom = "line",
      ylab = "mean ozone level",
      xlab = "months since January 1985")
```



From the three figures above, we conclude that ozone has a clear minimum mean ozone level at longitude 19 and latitude 12, and that the ozone level has an interesting temporal periodicity. Not unexpectedly, the periodicity coincides with the annual season cycle (each period is 12 months).

The functions in the `plyr` package are very general. See (Wickham, 2011) or the online package documentation for more details.

9.5 Notes

Our discussion in this chapter of handling missing values, outliers, and skewed data is superficial. An in-depth treatment requires substantial technical prerequisites including probability and statistics. There are several sources containing such an in-depth description. For the topic of missing data refer to (Little and Rubin, 2002). For the topic of robustness refer to (Huber, 1981) or the more recent (Maronna et al., 2006). Power transformations appear in most regression textbooks, for example (Kutner et al., 2004). More information on manipulating data using R appears in (Spector, 2008). Specific details on the `reshape2` and `plyr` packages appear in (Wickham, 2007) and (Wickham, 2011).

References

H. Wickham. Reshaping data with the reshape package. *Journal of Statistical Software*, 21(12), 2007.
H. Wickham. The split-apply-combine strategy for data analysis. *Journal of Statistical Software*, 40(1), 2011.
R. J. A. Little and D. B. Rubin. *Statistical Analysis with Missing Data*. Wiley, second edition, 2002.
P. J. Huber. *Robust Statistics*. Wiley, 1981.
R. Maronna, D. R. Martin, and V. J. Yohai. *Robust Statistics: Theory and Methods*. Wiley, 2006.
M. Kutner, C. Nachtsheim, J. Neter, and W. Li. *Applied Linear Statistical Models*. McGraw-Hill, fifth edition, 2004.
P. Spector. *Data Manipulation with R*. Springer, 2008.

Chapter 10
Essential Knowledge: Parallel Programming

At the heart of any big data system is a plethora of processes and algorithms that run in parallel to crunch data and produce results that would have taken ages if they were run in a sequential manner. Parallel computing is what enables companies like Google to index the Internet and provide big data systems like email, video streaming, etc. Once workloads can be distributed effectively over multiple processes, scaling the processing horizontally becomes an easier task. In this chapter, we will explore how to parallelize work among concurrent processing units; such concepts apply for the most part whether said processing units are concurrent threads in the same process, or multiple processes running on the same machine or on multiple machines. If you haven't already read about process management and scheduling in Sect. 3.4 of Chap. 3, now would be a good time to visit that.

Concurrent computing can be achieved at different layers of a given system: distributed systems with multiple machines, multiple processes on the same machine, multiple threads in the same process, etc. The bigger a system gets, the more parallelism it requires. When selecting what kinds of parallelism to introduce into a system, many factors come into play; for example, the amount of coordination required between the units of execution, the limit of what a single machine can process, etc. Even a machine with a single CPU core, which can only execute one thread at a time, can run multiple threads (and processes) in a fashion that can be perceived as concurrent thanks to an operating system feature known as time-slicing. That said, single-processor machines are a thing of the past nowadays and most modern machines have multiple processors or a processor with multiple cores; this modern architecture is called Symmetric Multi-Processing (SMP). In said architecture, each processor (or core) has its own local cache while the main memory is shared; the operating system manages all of that but it doesn't intrinsically protect against issues that arise due to the way the main memory is shared and caches are accessed.

© Springer Nature Switzerland AG 2018

G. Lebanon, M. El-Geish, *Computing with Data*,
https://doi.org/10.1007/978-3-319-98149-9_10

10.1 Choosing a Programming Language

There are numerous choices that face developers when it comes to parallel computing: whether to use Python or Java, multi-threading or multiprocessing, optimistic or pessimistic concurrency control, etc. In Chaps. 14 and 15, we delve deeper into system design choices and how to make sense out of the cacophony of methods and solutions out there that one may use. In a nutshell, we can follow the same philosophy for parallel computing choices: Start with defining the requirements and goals, device a set of guiding principles, and pick methods that satisfy the requirements and adhere to the chosen principles. Let's take for example the choice of what programming language to use when building a parallel computing system for big data. At Microsoft, while working on a Platform-as-a-Service (PaaS) product for Outlook.com and OneDrive, the key requirement was reducing costs and increasing scalability; the culture of dogfooding inside the company dictated that C# was a natural choice—in fact, it's a great choice given its language-level support for task-based asynchronous programming with the `async` and `await`[1] keywords. At LinkedIn—which is mainly a Java shop—high availability, efficiency, performance, and scale have been top requirements[2]; back then, system architects chose a service-oriented architecture that relied on Java Server Pages to satisfy LinkedIn's requirements. At Voicera, we picked Go to meet our requirements as an early startup in the AI field; we appreciated its built-in concurrency features (like goroutines and channels), its simplicity, being an opinionated language, and its great tools; Go is an apt choice for building distributed systems with many pipelines that keep evolving like the ones we have at Voicera.

So what about Python? We expect the majority of our readers to be familiar with (or at least excited about) Python and want to use it in data computations at scale. Many companies, like YouTube, use Python at scale; that said, Java has been the de facto language for big data systems used by companies like Amazon, LinkedIn, Facebook, Twitter, etc.; Java and JVM languages (like Scala) are widely used and/or supported by projects like ActiveMQ, Apex, Beam, Cassandra, Elasticsearch, Flink, Flume, Hadoop, Hive, Kafka, Lucene, Pig, Samza, Solr, Spark, etc.—and the list goes on. In this chapter, we will be mainly using Java to explore parallel computing concepts that, given they're supported, can be easily translated into other programming languages.

[1] https://docs.microsoft.com/en-us/dotnet/csharp/async.

[2] https://engineering.linkedin.com/architecture/brief-history-scaling-linkedin.

10.2 Processes, Threads, and Fibers

These three constructs (processes, threads, and fibers) are offered by operating systems so that programming languages can provide developers with ways of performing tasks in parallel. A process is a unit of execution that can be described as self-contained; it can be launched with its own memory address space and executed. On the other hand, a thread needs a process to run within its execution context; each process must start at least one thread that's called its main thread. Threads in the same process share its resources (like memory address space) and can intercommunicate without having to use mechanisms external to their host process; inter-process communication (IPC), on the other hand, requires such mechanisms like files on disk, a named pipe, shared memory, etc. IPC can transcend machine boundaries and facilitate parallel computing across machines via networking; processes can communicate over network sockets to schedule work, signal each other, store data into a shared database, etc.

Threads are native to operating systems and many programming languages provide a direct access to system-level threads. That said, some programming languages, like Java, provide what's known as green threads (named after the Green Team at Sun Microsystems that built the threading library in Java), which provide threads that are scheduled by a runtime library or a virtual machine; instead of relying directly on the operating system's native constructs for threading.

Fibers are basically lightweight threads or coroutines that cooperate to execute tasks. Cooperative scheduling means that the code that a fiber executes won't be interrupted until the current fiber yields the CPU; threads on the other hand are scheduled using preemptive scheduling, which means each thread gets a time slice then gets preempted by another thread that wants to execute its own code. Fibers are less common than processes and threads as units of execution and won't be covered in this chapter; we leave exploring more about them to the enthusiastic reader.

10.3 Thread Safety

From a thread safety perspective, resources (e.g., output streams, memory, etc.) are classified as either thread-exclusive, read-only, or lock-protected. The same concepts can apply to multiprocessing with independent processes; each has access to its own dedicated address space (exclusive memory), read-only shared resources (e.g., a read-only file), or writable shared resources (which need to be lock-protected to avoid concurrency issues). Here are a few examples of how to access and use shared resources:

Unsafe Use

- Accessing static variables or heap-allocated memory after it's published (made accessible to other threads).
- Uncontrolled access to shared resources through handles, pointers, or references.
- (Re-)allocating/freeing resources that have global scope (e.g., files) without locking.
- Writing to a shared resource (e.g., an output stream) from multiple threads without locking.

Safe Use

- Accessing local variables, heap-allocated memory before it's published (made accessible to other threads).
- Constants and read-only memory.[3]

The trick with concurrency is understanding what could go wrong; concurrency issues are subtle and hard to reproduce; they are even harder to foresee and plan for. Let's start showing the complexity of concurrency by demonstrating single-threaded logic and then adding concurrency later on:

```
def do_work():
  print('Hello, world!')

do_work()
## Hello, world!
```

Now let's run the same function in a multi-threaded fashion using the `Thread` object in Python:

```
from threading import Thread

for t in (Thread(target=do_work) for _ in range(10)):
  t.start()
## Hello, world!Hello, world!Hello, world!
##
##
## Hello, world!Hello, world!Hello, world!
##
##
## Hello, world!
```

[3]That's different from Java's `final` modifier or C#'s `readonly` modifier, which are only concerned with disallowing reassignment to a reference but don't care about the state of the object's data; for example, calling a method to change the value of one of its data members is allowed regardless of the existence—or lack thereof—of said modifiers for the object.

```
## Hello, world!Hello, world!
##
## Hello, world!
```

Please note that your output may vary because concurrency issues are indeterministic. As you can see, the output is tangled up. That's because multiple threads are writing to a shared resources (the output stream) simultaneously; there's no notion of order or boundaries of a work unit (in this case, a single message to print on its own line). A more practical example would be each thread is calculating some result independently in parallel, then said results are reported when done; here's a simple example in Java[4]:

```
class Task implements Runnable {
  static boolean isDone = false;

  public void run() {
    try {
      System.out.println("Doing work...");
      Thread.sleep(10); // do some work
      isDone = true;
    } catch (InterruptedException ex) {
      System.out.println(ex);
    }
  }
}

class Poller implements Runnable {
  public void run() {
    int iterations = 0;
    System.out.println("Waiting for work to be done...");
    while (!Task.isDone) { // bad way to wait for a signal
      ++iterations;
    }
    System.out.println("Polled " + iterations + " times");
  }
}

public class Main {
  public static void main(String args[]) throws Exception {
    new Thread(new Poller()).start();
    new Thread(new Task()).start();
    Thread.sleep(1000); // ad-hoc wait till threads finish
  }
}
```

The above code results into the following output:

[4]Code examples in this chapter may sacrifice best OOP practices, like encapsulation, for demonstration purposes and brevity.

```
## Waiting for work to be done...
## Doing work...
```

The program then goes into an infinite loop because the polling thread never sees the status update the other thread made. This behavior is due to an infamous concurrency issue that occurs when the thread scheduler allows a thread to execute and read the value of `isDone` while another thread—running in parallel on another processor—is concurrently updating it. In the world of database systems that issue is known as a dirty read. Processors keep cached values of variables in their own independent local caches and only flush the updated value for other processors to read when required.

10.4 Volatility

We can mitigate the issue demonstrated in the previous example by specifying `isDone` as a `volatile` field, which instructs the program to always read its value from the main memory and always write its value to the main memory (and not just the CPU cache). Here's the new code snippet with a single keyword addition, `volatile`, to fix the infinite loop issue:

```
class Task implements Runnable {
  volatile static boolean isDone = false;

  public void run() {
    try {
      System.out.println("Doing work...");
      Thread.sleep(10); // do some work
      isDone = true;
    } catch (InterruptedException ex) {
      System.out.println(ex);
    }
  }
}

class Poller implements Runnable {
  public void run() {
    int iterations = 0;
    System.out.println("Waiting for work to be done...");
    while (!Task.isDone) { // bad way to wait for a signal
      ++iterations;
    }
    System.out.println("Polled " + iterations + " times");
  }
}

public class Main {
  public static void main(String args[]) throws Exception {
    new Thread(new Poller()).start();
```

```
    new Thread(new Task()).start();
    Thread.sleep(1000); // ad-hoc wait till threads finish
  }
}
```

Now the program produces the following output (and terminates afterwards):

```
## Waiting for work to be done...
## Doing work...
## Polled 24038033 times
```

The volatile visibility guarantee comes at a cost though: a performance penalty due to ignoring the CPU cached value and accessing the main memory every time the field is accessed; that's why it should be used judiciously. More importantly, declaring a shared resource as `volatile` can be misleading as it may give a false sense of thread safety; it's crucial to know that the mere act of sprinkling the keyword `volatile` on fields doesn't guarantee thread safety and definitely doesn't protect against concurrency issues. Let's take the following simple task as an example:

```
class Task implements Runnable {
  volatile static int sum = 0;
  public void run() { ++sum; }
}

public class Main {
  public static void main(String args[]) throws Exception {
    Task task = new Task();
    for (int i = 0; i < 10000; i++) {
      new Thread(task).start();
    }
    Thread.sleep(1000); // ad-hoc wait till threads finish
    System.out.println("sum: " + Task.sum);
  }
}
```

The above program erroneously produces the following result:

```
## sum: 9998
```

10.5 Synchronization

The difference between the last two scenarios is that the latter demonstrates concurrent threads that need to perform two operations (load and store) on the same field. In such cases, the volatile value will be written to the main memory and read from the main memory; however, each thread can read the same value at the same time and operate on said value. For example, two threads A and B

can read the value of `sum` as 0 when they first start, each processor increments `sum` to become 1 independently and simultaneously, then each of them race to write the new value to the main memory, which ends up being incorrect. This race condition happens when multiple threads compete to perform conflicting operations on the same shared resource and they end up stepping on each other's toes. To prevent this from happening, we require locking; the thread scheduler locks the specified set of statements—or more precisely, instructions—in a critical section that only one thread can enter to execute it at a time; other threads have to wait until the current thread finishes executing said critical section. A very simple way to introduce such locking mechanism to our program, but not necessarily the most efficient, is by declaring the method that requires access to the shared resource as `synchronized`, which is equivalent to declaring the entire method as a critical section; in other words which are commonly used in parallel computing, the operation is now atomic. Here's the new code snippet with a single keyword addition, `synchronized`, to fix the race condition:

```
class Task implements Runnable {
  static int sum = 0;
  synchronized public void run() { ++sum; }
}

public class Main {
  public static void main(String args[]) throws Exception {
    Task task = new Task();
    for (int i = 0; i < 10000; i++) {
      new Thread(task).start();
    }
    Thread.sleep(1000); // ad-hoc wait till threads finish
    System.out.println("sum: " + Task.sum);
  }
}
```

The above program now consistently produces the following correct result:

```
## sum: 10000
```

10.5.1 Ineffectual Synchronization

To illustrate how subtle concurrency issues are and how hard they are to miss, we will make a very small change that may look extremely inconspicuous; we will still use the `synchronized` modifier to synchronize access to the critical section in the above example, but we will feed each thread its own task instance:

```
class Task implements Runnable {
  static int sum = 0;
  synchronized public void run() { ++sum; }
```

```
}

public class Main {
  public static void main(String args[]) throws Exception {
    for (int i = 0; i < 10000; i++) {
      new Thread(new Task()).start();
    }
    Thread.sleep(1000); // ad-hoc wait till threads finish
    System.out.println("sum: " + Task.sum);
  }
}
```

The above program erroneously produces the following result:

```
## sum: 9998
```

Now that we are back to square one, we wonder what went wrong. It's easier in this context—thanks to the temporal proximity of the code change to the discovery of the concurrency issue—to point at the subtle code change as the smoking gun; however, in the real world, concurrency issues rarely manifest themselves right after the culprit code change starts running—it can take hours, days, months, or even years for such issues to manifest (if at all); and when they do, it may be a rare sighting like once in a blue moon. Sometimes, another trigger or catalyst causes the issue to surface; for example, change in CPU utilization on the same machine due to increased load or deploying the code on a different hardware configuration. If anything, concurrency issues teach us that "post hoc ergo propter hoc"[5] is a dangerous fallacy and should be avoided while debugging concurrency issues (and most bugs for that matter—one should always be skeptical about synchronicity[6]). To understand what went wrong here, we need to delve deeper into how the `synchronized` modifier modifies a method.

The `synchronized` keyword in Java can be used to declare a method—instance or static alike—as such; it can also be used to define a synchronized code block, which is the more general case; in fact, the former specifier (for methods) is mere syntactic sugar for a code block that subsumes the entire method. A synchronized block requires an object to lock on: in the case of an instance method, the lock object is the instance to which the method belongs; in the case of a static method, the lock object is the class object to which the method belongs. Since a class object is singleton by design, only a single thread at any point of time can execute a synchronized static method; while a synchronized instance method can be executed for different instances by concurrent threads, but it's never the case that two threads concurrently execute the same method of the same instance. When a thread enters the synchronized instance method, it blocks other threads from entering the same

[5]It means "after this, therefore because of this"; also known as the post hoc fallacy; one may attribute a causal connection between events based on their temporal order.

[6]Not to be confused with synchronization; synchronicity is the simultaneous occurrence of events that appear significantly related but have no discernible causal connection.

method on the same instance and other threads need to wait until the current one exits said method.

The following code snippet that illustrates the use of a `synchronized` code block is equivalent to the one above (where `synchronized` is used to declare an instance method as such):

```
class Task implements Runnable {
  static int sum = 0;

  public void run() {
    synchronized(this) {
      ++sum;
    }
  }
}

public class Main {
  public static void main(String args[]) throws Exception {
    Task task = new Task();
    for (int i = 0; i < 10000; i++) {
      new Thread(task).start();
    }
    Thread.sleep(1000); // ad-hoc wait till threads finish
    System.out.println("sum: " + Task.sum);
  }
}
```

Another example of ineffectual synchronization, that's not applicable in Java but can happen in C#, is demonstrated below:

```
using System;
using System.Threading;

class Program
{
  static int sum = 0;

  static void DoWork()
  {
    lock ((object) 0)
    {
      for (int i = 0; i < 10000; i++)
      {
        ++sum;
      }
    }
  }

  static void Main()
  {
    for (int i = 0; i < 4; i++)
    {
      new Thread(DoWork).Start();
```

```
        }

        Thread.Sleep(1000); // ad-hoc wait till threads finish
        Console.WriteLine("C# sum: " + sum);
    }
}
```

Despite locking to synchronize access to the `DoWork` method, the output that the program produces represents an incorrect sum:

```
## C# sum: 38540
```

Both Java and C# support autoboxing of primitive (native) types into their corresponding object wrapper types (e.g., from `int` to `Integer`); autoboxing creates a new object every time the primitive type is boxed; hence, in the example above, every time a thread enters `DoWork`, it locks on a new object and enters the code block even though other threads are executing the same block (because each acquired its own lock). In Java, attempting to lock on a primitive type produces a compilation error:

```
## Error:(5, 9) java: unexpected type
##     required: reference
##     found:    int
```

10.5.2 Synchronization vs. Volatility

We didn't have to declare `sum` as `volatile` anymore because `synchronized` introduces a stronger order guarantee that forces threads to enter the critical section in a synchronized (sequential) manner. It's important to note the difference between volatility and synchronization and when each of them is needed.

Declaring a field as `volatile` decreases the risk of running into some concurrency issues but doesn't entirely eliminate them. In Java, writes to a volatile field establish a "happens-before" relationship with subsequent reads of that same field; that's a guarantee for the order of instructions the processors execute, one that other languages—like C/C++—may not provide. It's key to realize that the behavior of the `volatile` modifier may vary in different languages and one must refer to each language's memory model as a reference to understand said differences. Even within the same language, said behavior can vary across versions; for example, since Java v1.5 (also known as Java 5) the `volatile` modifier guarantees both visibility—across processors—and order guarantees for volatile fields; while older versions only provided the visibility guarantee. The Java memory model specification can be found at https://docs.oracle.com/javase/specs/jls/se8/html/jls-17.html.

Declaring a field as `volatile` decreases is more efficient than serializing access to it using the `synchronized` keyword but can still be error-prone; minimizing the scope of the synchronized code block to the minimal required to

achieve thread safety can help reduce the performance penalty created by contention over shared resources. Each concurrency situation requires anticipating what could go wrong and performing a cost-benefit analysis, then coming up with a mitigation plan based on application and business requirements.

Going back to the use of synchronization in the previous example, we only had one task and we ended up synchronizing it; in doing so, we lost the benefit of parallelism; however, in practice, only smaller sections of the task are to be synchronized for thread safety (e.g., writing to a shared resource) while the rest of the task can be embarrassingly parallel (e.g., computations local to the current thread); here's an example:

```
class DeepThoughtTask implements Runnable {
  static int finalResult = 0;

  private int computeResult() {
    return 1; // assume this is a long-running task
  }

  public void run() {
    long partialResult = computeResult(); // lock-free
    synchronized (DeepThoughtTask.class) {
      finalResult += partialResult;
    }
  }
}

public class Main {
  public static void main(String args[]) throws Exception {
    DeepThoughtTask task = new DeepThoughtTask();
    for (int i = 0; i < 42; i++) {
      new Thread(task).start();
    }
    Thread.sleep(1000); // ad-hoc wait till threads finish
    System.out.println("answer: " + DeepThoughtTask.finalResult);
  }
}
```

which produces the following output:

```
## answer: 42
```

10.6 Starvation

So far we've been mainly using the `synchronized` modifier and its equivalent `synchronized` code block like that in the above example; however, we encourage software developers to eschew the use of said patterns. A common pitfall is locking using an object that can be used as a lock somewhere else (outside the enclosing

class); consider the following program for example where we add a new parallel task that hogs the CPU and uses the same lock as the `DeepThoughtTask` class:

```
class DeepThoughtTask implements Runnable {
  static int finalResult = 0;

  private int computeResult() {
    return 1; // assume this is a long-running task
  }

  public void run() {
    long partialResult = computeResult(); // lock-free
    synchronized (DeepThoughtTask.class) {
      finalResult += partialResult;
    }
  }
}

class ProcessorHog implements Runnable {
  public void run() {
    synchronized (DeepThoughtTask.class) {
      try {
        Thread.sleep(3000); // keep the CPU busy
      } catch(InterruptedException ex) {
        System.out.println(ex);
      }
    }
  }
}

public class Main {
  public static void main(String args[]) throws Exception {
    new Thread(new ProcessorHog()).start();
    Thread.sleep(1000); // simulates other work being done
    DeepThoughtTask task = new DeepThoughtTask();
    for (int i = 0; i < 42; i++) {
      new Thread(task).start();
    }
    Thread.sleep(1000); // ad-hoc wait till threads finish
    System.out.println("answer: " + DeepThoughtTask.finalResult);
  }
}
```

This time the program produces the following output:

```
## answer: 0
```

The answer is 0 because the 42 threads running the `DeepThoughtTask` were all blocked and were unable to enter the critical section that aggregates the final result. This class of concurrency issues is known as starvation, in which case a thread (or process) is blocked to gain access to a shared resource and thus unable to make progress; the most common cause for starvation is a greedy thread (or process)

hogging the shared resource and blocking others from accessing it for periods longer than normal.

10.7 Deadlocks

Here's another example for issues that can occur due to lock objects being visible outside the enclosing class where they belong:

```
class A implements Runnable {
  public void run() {
    synchronized (A.class) {
      System.out.println("A has A.class and now wants B.class");
      try {
        Thread.sleep(500); // simulates some work being done
      } catch (InterruptedException ex) {
        System.out.println(ex);
      }
      synchronized (B.class) {
        System.out.println("A got both locks and made progress");
      }
    }
  }
}

class B implements Runnable {
  public void run() {
    synchronized (B.class) {
      System.out.println("B has B.class and now wants A.class");
      synchronized (A.class) {
        System.out.println("B got both locks and made progress");
      }
    }
  }
}

public class Main {
  public static void main(String args[]) throws Exception {
    new Thread(new A()).start();
    new Thread(new B()).start();
    Thread.sleep(1000); // ad-hoc wait till threads finish
  }
}
```

The above program produces the following output and doesn't terminate or make any progress:

```
## A has A.class and now wants B.class
## B has B.class and now wants A.class
```

The issue in the above example is the infamous deadlock concurrency issue, in which case two or more threads are completely and permanently blocked waiting for each other. For a deadlock to occur, four conditions have to be met:

- **Mutual Exclusion:** competing threads are trying to gain access to shared resources that are unavailable; for example, a critical section that's being executed by another thread. Mutually exclusive resources can be utilized by a single processing unit at any point in time. Without mutual exclusion, deadlocks cannot occur because requests to access shared resources will be granted without contention.
- **Resource Holding:** also known as hold-and-wait or partial allocation; in which case, a processing unit (e.g., a thread or a process) is holding some resources while waiting for access to other resources it doesn't hold, which are being held by other processes. Without resource holding, deadlocks cannot occur because a processing unit would release the resources it's been holding for others to grab.
- **No Preemption:** Whatever is held by a processing unit cannot be taken away preemptively by another processing unit; a resource can be released only voluntarily by the processing unit that holds it. With preemption, a processing unit can force another to give up the resources it needs to make progress.
- **Circular Wait:** Two or more processing units have to form a circular chain, with each processing unit holding a resource that others in the chain want, for a deadlock to occur. This is also known as the cycle theorem, which states that "a cycle in the resource graph is necessary for deadlock to occur." Without circular wait processing unit can continue making progress.

Besides circumventing the above necessary conditions listed above, one may go about reducing the risk of deadlocks by reducing the number and scope of lock-protected regions in the code. Another approach is breaking the circular wait chain by enforcing lock acquisition order in such a way that no circular waiting occurs. A third approach is to never wait indefinitely for a lock to be acquired; giving up after a timeout and triggering a failure instead can help other threads make progress while altering monitoring systems to the concurrency issue that needs to be addressed. The choice of deadlock mitigation approaches depends on the application and business requirements.

When it comes to atomicity, critical sections are not the only answer; there are more options for commonly used operations like the one we've been trying to perform. Going back to our previous example that sums partial results—each computed in parallel—into a final one, we can obtain the correct final results with more efficient operations known as interlocked operations; said constructs in high-level languages (like Java and C#) provide methods that allow access and manipulation of variables that are shared among concurrent threads; modern processors allow those methods to be implemented using a single CPU instruction! For example, one can read, increment, then write an integer in one instruction instead of three; when an operation is performed using multiple instructions, multiple threads can see and operate on partially computed value of the shared variable as they take turn executing their own instructions. It's important to note that

interlocked operations are designed to be atomic regardless of the hardware support of single-instruction execution of each operation; when said hardware support is applicable, the runtime makes use of it in a lock-free (non-blocking) fashion and thus a performance gain is realized, which can be much needed when implementing certain kinds of locks that require high performance (e.g., spinlocks). Let's revisit one of the previous examples and make use of the `AtomicInteger` class:

```
import java.util.concurrent.atomic.AtomicInteger;

class DeepThoughtTask implements Runnable {
  static AtomicInteger finalResult = new AtomicInteger(0);

  private int computeResult() {
    return 1; // assume this is a long-running task
  }

  public void run() {
    // This is lock-free on modern CPUs
    finalResult.addAndGet(computeResult());
  }
}

public class Main {
  public static void main(String args[]) throws Exception {
    DeepThoughtTask task = new DeepThoughtTask();
    for (int i = 0; i < 42; i++) {
      new Thread(task).start();
    }
    Thread.sleep(1000); // ad-hoc wait till threads finish
    System.out.println("answer: " + DeepThoughtTask.finalResult);
  }
}
```

The above program consistently produces the correct output and achieves that goal without compromising performance, except that it has to sleep for an ad-hoc second, which seems arbitrary and causes an unnecessary delay after all 42 threads have finished doing their work, well before the second has elapsed. To remedy this issue, we need to coordinate those threads so that the main thread, which prints out the final result, sees the final value after all threads finished running. One can achieve that by requesting that each thread object we create joins the main thread after it's done with its fork:

```
import java.util.concurrent.atomic.AtomicInteger;

class DeepThoughtTask implements Runnable {
  static AtomicInteger finalResult = new AtomicInteger(0);

  private int computeResult() {
    return 1; // assume this is a long-running task
  }
```

```
    public void run() {
      // This is lock-free on modern CPUs
      finalResult.addAndGet(computeResult());
    }
  }

  public class Main {
    public static void main(String args[]) throws Exception {
      DeepThoughtTask task = new DeepThoughtTask();
      Thread[] threads = new Thread[42];
      for (int i = 0; i < threads.length; i++) {
        threads[i] = new Thread(task);
        threads[i].start(); // fork; non-blocking
      }
      for (int i = 0; i < threads.length; i++) {
        threads[i].join(); // wait; blocking
      }
      System.out.println("answer: " + DeepThoughtTask.finalResult);
    }
  }
```

10.8 The Producer-Consumer Problem

The running example we've been using so far follows a paradigm that's known as scatter-gather: We kicked off multiple parallel threads, each was performing an embarrassingly parallel task and reporting back the result to the main thread which has been waiting for all tasks to complete to gather the final result. Such paradigm requires minimal coordination between two classes of threads, the dispatcher and the dispatched, and doesn't require any sort of coordination among the dispatched threads since their tasks are embarrassingly parallel.

Another common paradigm in parallel computing is known as the producer-consumer or the bounded-buffer problem. In said scenario, we have a class of threads that produce data to put into a shared resource and another class of threads that consume data from said shared resource; the reason why it's called a bounded buffer has to do with constraints related to the buffer size: Consumers cannot remove data from an empty buffer (and they're blocked from making any progress until new data are produced); and producers cannot add new data into a full buffer (and they're blocked from making any progress until some data are consumed). Solutions to the producer-consumer problem are prone to deadlocks because they are susceptible to satisfying the necessary condition for deadlocks. Let's take a look at an example of a bad implementation; first, let's examine our shared resource, which is a bounded queue in this case that's protected with busy waiting:

```
class Shared {
  static final int MAX_BUFFER_SIZE = 3;
  static Queue<String> buffer = new ArrayDeque<>();
```

```
    private static volatile boolean shouldWait = true;

    static void waitUntilNotified() {
      while (shouldWait);
      shouldWait = true;
    }

    static void notifyWaitingThread() {
      shouldWait = false;
    }
  }
```

In the above class, we could have used a spin lock, which is akin to busy waiting. Spin locks are useful when we know that the wait is not going to last for long (in CPU time that means a few instructions, see Definition 2.4.1 for more details); they save threads that know they are going to acquire a look soon from the overhead of context switching,[7] which can get costly as the operating system restores the state of the thread to be executed into the CPU. Spin locks are more efficient when locking is extremely short-lived and many locks are required (e.g., a lock per object of a class that gets instantiated often); it also avoids blocking (by busy-waiting); however, if waiting for the lock takes longer than expected, spin locks should be avoided because of excessive spinning. Java doesn't have a built-in spin lock type; C#—on the other hand—provides an implementation.

The consumer class in our simple example consumes a message from the queue and prints it if there's one; otherwise it waits:

```
class Consumer implements Runnable {
  public void run() {
    while (true) {
      if (Shared.buffer.size() == 0) {
        Shared.waitUntilNotified();
      }

      consume();

      if (shouldNotifyProducers()) {
        Shared.notifyWaitingThread();
      }
    }
  }

  private void consume() {
    System.out.println("Consumed: " + Shared.buffer.remove());
  }

  private boolean shouldNotifyProducers() {
    return Shared.buffer.size() == Shared.MAX_BUFFER_SIZE - 1;
  }
```

[7] See https://en.wikipedia.org/wiki/Context_switch for more details.

```
}
```

When the size of the buffer reaches the maximum allowed, producers are blocked from adding data into the queue; so when a consumer removed a single element from the queue at that moment, it needs to notify the producers who have been waiting to start making progressing and producing again. Conversely, the reverse is true for the producer class:

```
class Producer implements Runnable {
  private static int i = 0;

  public void run() {
    while (true) {
      if (Shared.buffer.size() == Shared.MAX_BUFFER_SIZE) {
        Shared.waitUntilNotified();
      }

      Shared.buffer.add(produce());

      if (shouldNotifyConsumers()) {
        Shared.notifyWaitingThread();
      }
    }
  }

  private String produce() {
    return String.valueOf(i++);
  }

  private boolean shouldNotifyConsumers() {
    return Shared.buffer.size() == 1;
  }
}
```

Running the following program will result into a race condition:

```
public class Main {
  public static void main(String args[]) throws Exception {
    new Thread(new Producer(), "produce").start();
    new Thread(new Consumer(), "consume").start();
  }
}


## ... (truncated messages)
## Exception in thread "consume" java.util.NoSuchElementException
## ... (truncated stacktrace)
```

In the context of our producer-consumer problem, busy waiting results into a race condition, which is caused by the lack of locking when accessing the shared resource. In Java, it's idiomatic to use `wait` and `notify`, which require acquiring a lock on the object for which those methods are called:

```
class Shared {
  static final int MAX_BUFFER_SIZE = 3;
  static Queue<String> buffer = new ArrayDeque<>();
  private static final Object lock = new Object();

  static void waitUntilNotified() {
    try {
      synchronized (lock) {
        lock.wait();
      }
    } catch (InterruptedException ex) {
      System.out.println(ex);
    }
  }

  static void notifyWaitingThread() {
    synchronized (lock) {
      lock.notify();
    }
  }
}
```

Even though we used a lock in the above example, we still encountered a deadlock since only the `wait` and `notify` calls are in the critical sections while manipulating the buffer happens outside without concurrency control. The following program fixes the deadlock issue and works correctly:

```
import java.util.ArrayDeque;
import java.util.Queue;

class Shared {
  private static final int MAX_BUFFER_SIZE = 3;
  private static Queue<String> buffer = new ArrayDeque<>();
  private static final Object lock = new Object();

  static String remove() throws InterruptedException {
    final String message;

    synchronized (lock) {
      while (buffer.size() == 0) {
        lock.wait();
      }
      message = buffer.remove();
      lock.notify();
    }
    return message;
  }

  static void add(String message) throws InterruptedException {
    synchronized (lock) {
      while (buffer.size() == MAX_BUFFER_SIZE) {
        lock.wait();
      }
```

```
        buffer.add(message);
        lock.notify();
      }
    }
  }

  class Consumer implements Runnable {
    public void run() {
      while (true) {
        try {
          System.out.println("Consumed: " + Shared.remove());
        } catch (Exception ex) {
          System.out.println(ex);
        }
      }
    }
  }

  class Producer implements Runnable {
    private static int i = 0;

    public void run() {
      while (true) {
        try {
          Shared.add(String.valueOf(i++));
        } catch (Exception ex) {
          System.out.println(ex);
        }
      }
    }
  }

  public class Main {
    public static void main(String args[]) throws Exception {
      new Thread(new Producer(), "Producer").start();
      new Thread(new Consumer(), "Consumer").start();
    }
  }
```

10.9 Reader-Writer Locks

In the producer-consumer problem, we saw how both threads modify the shared resource; hence, we had the requirement of mutual exclusion when modifying the buffer. That's not always the case in parallel computing; there is a class of problem where the vast majority of the concurrent threads only require read access to the share resource while a few require write access. For example, a cache that gets updated infrequently while serving a consistent torrent of read requests. Allowing readers to read from the shared resource (the cache in this example) while writers are

modifying it is error-prone since writes may not be atomic and partial values may be erroneously returned as the answer to the readers' queries. Here's an example where we produce a dummy value to write to each element of the cached array so that its sum is either 0 (initial value) or its length (after any complete write operation):

```
import java.util.Arrays;
import java.util.HashMap;
import java.util.Map;

class Simulator {
  static void simulateWork() {
    try {
      Thread.sleep(3); // simulates work being done
    } catch (InterruptedException ex) {
      System.out.println(ex);
    }
  }
}

class Shared {
  static final Map<String, int[]> cache = new HashMap<>();
}

class Producer {
  protected int produce() {
    Simulator.simulateWork();
    return 1;
  }
}

class Writer extends Producer implements Runnable {
  public void run() {
    int[] referenceToValue = Shared.cache.get("key");

    for (int i = 0; i < referenceToValue.length; i++) {
      // Update value in place; sum will add up to length
      referenceToValue[i] = produce();
    }
  }
}

class Consumer {
  protected void consume(int[] value) {
    int sum = Arrays.stream(value).sum();

    if (sum != 0 && sum != value.length) {
      throw new IllegalStateException("Partial sum: " + sum);
    }

    Simulator.simulateWork();
  }
}
```

```
class Reader extends Consumer implements Runnable {
  public void run() {
    consume(Shared.cache.get("key"));
  }
}
```

To test the above logic, and to capture how fast each approach we implement takes to run, we create the following driver program:

```
public class Main {
  private static long runTrial() throws Exception {
    long startTime = System.nanoTime();
    final Thread[] threads = new Thread[1000];

    Shared.cache.put("key", new int[20]); // sum = 0 at this time
    threads[0] = new Thread(new Writer(), "Writer");
    threads[0].start();

    for (int i = 1; i < threads.length; i++) {
      threads[i] = new Thread(new Reader(), "Reader #" + i);
      threads[i].start();
    }

    for (int i = 1; i < threads.length; i++) {
      threads[i].join();
    }

    return System.nanoTime() - startTime;
  }

  public static void main(String args[]) throws Exception {
    long[] trials = new long[11];

    for (int i = 0; i < trials.length; i++) {
      trials[i] = runTrial();
    }

    Arrays.sort(trials);

    long median = trials[trials.length / 2];
    double average = Arrays.stream(trials).
      average().
      getAsDouble();

    System.out.printf(
      "Median time in nanoseconds: %1$,d\n",
      median);
    System.out.printf(
      "Average time in nanoseconds: %1$,.2f\n",
      average);
  }
}
```

Running the above program will result into the following error:

```
## Exception in thread "Reader #117"... (truncated)
## Median time in nanoseconds: 73,910,004
## Average time in nanoseconds: 96,375,563.45
```

Since the write operation is not atomic, readers saw partial results when querying the cache. In Java,[8] reads and writes for reference variables (e.g., updating a reference) and for primitive variables, except long and double, are atomic. In the example above, we opted for an in-place update to the cached array to demonstrate how partial updates can occur. In order to return the correct value, we may want to make the cache a mutually exclusive resource by making the following changes to the program:

```
class Writer extends Producer implements Runnable {
  public void run() {
    synchronized (Shared.cache) {
      int[] referenceToValue = Shared.cache.get("key");

      for (int i = 0; i < referenceToValue.length; i++) {
        // Update value in place; sum will add up to length
        referenceToValue[i] = produce();
      }
    }
  }
}

class Reader extends Consumer implements Runnable {
  public void run() {
    synchronized (Shared.cache) {
      consume(Shared.cache.get("key"));
    }
  }
}
```

On a 3.3 GHz Intel Core i7 machine, the above approach produces the following output for serving times:

```
## Median time in nanoseconds: 4,179,611,467
## Average time in nanoseconds: 4,174,615,845.73
```

Fixing the error using critical sections came at the cost of efficiency: The program now takes around 57x the time it used to take without locking! This slowdown is due to thread contention: a concurrency issue that happens when threads are waiting for a shared resource that's being exclusively held for longer than what's acceptable; since the waiting threads are blocked and can't make any progress, this issue affects the performance and throughput of systems that suffer from it.

[8] See https://docs.oracle.com/javase/tutorial/essential/concurrency/atomic.html for more details.

Of course there are many improvements we can make here to reduce contention; for example, following the best practice of reducing the time a thread spends inside a critical section:

```
class Reader extends Consumer implements Runnable {
  public void run() {
    int[] deepCopy;

    synchronized (Shared.cache) {
      int[] value = Shared.cache.get("key");

      deepCopy = Arrays.copyOf(value, value.length);
    }
    consume(deepCopy);
  }
}
```

Note the trade-off we made between CPU and memory; now each thread copies the cached value to get out of the critical section as soon as possible. Moreover, for demonstration purposes and to reflect what realistically happens inside critical sections, we will assume that the average workload inside a critical section is in the order of a couple of milliseconds. To put things into perspective, a thread that needs to hold on to a lock (that protects a shared resource) while sending a few packets from California to Boston will spend around 40 milliseconds at least inside that critical section.

The question that we should keep asking ourselves when designing big data systems is: Can we make it run faster? The answer—in this case—is a joyful yes; we modify the program to use a reader-writer lock, which allows multiple readers to get the cached value concurrently and only blocks when a writer enters a critical section to modify the cache:

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReadWriteLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;

class Shared {
  static final Map<String, int[]> cache = new HashMap<>();
  private static final ReadWriteLock lock =
    new ReentrantReadWriteLock();
  static final Lock readLock = lock.readLock();
  static final Lock writeLock = lock.writeLock();
}

class Writer extends Producer implements Runnable {
  public void run() {
    Shared.writeLock.lock();
    try {
      int[] referenceToValue = Shared.cache.get("key");

      for (int i = 0; i < referenceToValue.length; i++) {
        // Update value in place; sum will add up to length
```

```
          referenceToValue[i] = produce();
        }
      } finally {
        Shared.writeLock.unlock();
      }
    }
  }

  class Reader extends Consumer implements Runnable {
    public void run() {
      Shared.readLock.lock();
      try {
        consume(Shared.cache.get("key"));
      } finally {
        Shared.readLock.unlock();
      }
    }
  }
```

The above program outputs the following:

```
## Median time in nanoseconds: 97,496,003
## Average time in nanoseconds: 154,436,085.27
```

Note the huge improvement in throughput, which is close to the lock-less approach, while providing a correct solution! As we mentioned earlier, the efficiency advantage here is due to using a reader-writer lock, which allows multiple readers to read from a shared resource while writes can only happen sequentially (as they require mutual exclusion); a thread that's writing to the shared resource blocks all other threads from accessing it (either for reading or writing).

An important practice when using locks explicitly, like in the above example, is to always release the lock when done with it; so in the case of throwing an exception while holding a lock, the lock should be released; hence, we strongly recommend calling `unlock` from a `finally` block to ensure it gets executed.

10.10 Reentrant Locks

The class we used in the above example to implement a reader-writer lock is called `ReentrantReadWriteLock`; it's important to note that "reentrant" in the name indicates that the thread that acquires a lock (by locking it) owns said lock and can request it to reenter the locked state in another code path; this holds until the thread explicitly unlocks the object. This means that the lock object is aware of which thread owns it. This kind of locking is useful when one method acquires the lock and calls another that also acquires said lock; for example:

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
```

```
class ReentrantLockExample {
  private final Lock lock = new ReentrantLock();

  public void foo() {
    lock.lock();
    try {
      // ...
      bar();
    } finally {
      lock.unlock();
    }
  }

  public void bar() {
    lock.lock();
    try {
      // ...
    } finally {
      lock.unlock();
    }
  }
}

public class Main {
  public static void main(String args[]) throws Exception {
    new ReentrantLockExample().foo();
  }
}
```

Without lock reentry, the above example would have resulted in a deadlock when calling `foo`; for example, we may use a binary semaphore to implement a lock for mutual exclusion as in the following code snippet:

```
import java.util.concurrent.Semaphore;

class ReentrantLockExample {
  private final Semaphore lock = new Semaphore(1);

  public void foo() throws InterruptedException {
    lock.acquire();
    try {
      // ...
      bar();
    } finally {
      lock.release();
    }
  }

  public void bar() throws InterruptedException {
    lock.acquire();
    try {
      // ...
```

```
      } finally {
        lock.release();
      }
    }
  }

  public class Main {
    public static void main(String args[]) throws Exception {
      new ReentrantLockExample().foo();
    }
  }
```

Semaphores are commonly used to allow a maximum number of threads, called the semaphore count, to have access to a shared resource or to enter a critical section while other threads wait until there's room for them. In the above example, we used a binary semaphore that allows a maximum of one thread to acquire the lock, calling `foo` causes a deadlock as the lock gets acquired at the first statement in `foo`; then it calls `bar` while the lock is acquired, so the request to acquire the same lock inside `bar` blocks waiting for `foo` to release it—deadlock!

That doesn't mean that reentrant locks are safer than non-reentrant ones; they are different tools and each should be used properly to best achieve what the code is set to achieve. One may argue that non-reentrant locks are better for certain projects because they force developers to be explicit about a lock's state as a precondition for a method call. To see what we mean by that in action, let's craft an example that demonstrates the importance of said parallel programming guideline:

```
import java.util.concurrent.locks.ReentrantLock;

class NonReentrantLockExample {
  private final ReentrantLock lock = new ReentrantLock();

  public void foo() throws InterruptedException {
    assert !lock.isHeldByCurrentThread() :
      "lock is held by current thread";
    lock.lock();
    try {
      // ...
      threadUnsafeBar();
    } finally {
      lock.unlock();
    }
  }

  public void bar() throws InterruptedException {
    assert !lock.isHeldByCurrentThread() :
      "lock is held by current thread";
    lock.lock();
    try {
      threadUnsafeBar();
    } finally {
      lock.unlock();
```

```
    }
  }

  private void threadUnsafeBar() {
    assert lock.isHeldByCurrentThread() :
      "caller must hold the lock first";
    // ...
  }
}

public class Main {
  public static void main(String args[]) throws Exception {
    new NonReentrantLockExample().foo();
  }
}
```

In the above example, we assert a precondition for the two public methods `foo` and `bar`, which requires the lock to be used in a non-reentrant manner; this is a good practice because we force the application to crash with a clear assertion message if any of the concurrency preconditions we specified is violated; failing early and clearly is usually a good thing in system design; this helps us identify potential concurrency issues even when the methods are called from a single thread. The other precondition we assert is that the helper method `threadUnsafeBar` can only be called when the caller holds the lock; this separation of concerns is useful in many ways: sometimes an application may need to switch between single-threaded and multi-threaded modes of operations—such refactoring makes said switch easier; in addition, debugging and unit-testing the helper method can be performed in isolation when we're only concerned about business logic that's not related to concurrency; moreover, we're adopting the Design by Contract paradigm to ensure the correctness of our concurrency logic—it's explicit in the code and also enforced by the JVM (when the `-enableassertion` or `-ea` flag is set for the `java` command). The convention we followed in the above example requires public methods that need locking to acquire the lock before calling the thread-unsafe private helper methods (which assert that the caller already holds the lock). Those public methods cannot call other public methods that require locking (because the lock is non-reentrant). The non-entry goes hand-in-hand with fine-grained locking and limiting the scope of critical sections so that reduce the chances of concurrency issues.

Reentrant locking is especially useful with recursive methods; that's why sometimes reentrant locks are called recursive locks. For example, if a method `foo` were to call itself while it's holding a lock, a reentrant lock allows the flow to continue without a blocking wait. The same refactoring we applied in the previous example can be applied for recursive calls such that the method that holds the lock is not recursive while the helper method is.

10.10.1 Reentry of Intrinsic Locks

The locks created using the `synchronized` keyword in Java are reentrant; they are also known as intrinsic or monitor locks. Here's how reentrant synchronization works:

```
class ReentrantIntrinsicLockExample {
  private final Object intrinsicLock = new Object();

  public void foo() {
    synchronized (intrinsicLock) {
      // ...
      bar();
    }
  }

  public void bar() {
    synchronized (intrinsicLock) {
      // ...
    }
  }
}
```

Each object in Java comes with its own intrinsic lock that can be used as a monitor, which is a synchronization construct that supports both mutual exclusion, blocking wait, and signaling as we saw in earlier examples. The idea of making such locks intrinsic is rooted in the convention of introducing thread safety to an object by owing its lock; if a thread requires mutually exclusive access to an object, it should acquire its lock; we don't recommend following this convention unless the object associated with the lock is private and final so that only the enclosing class can gain access to its lock and to ensure that the declared reference keep referring to the same object (and, by association, to the same lock); otherwise the code may run into concurrency issues like deadlocks as we discussed earlier in Sect. 10.7.

10.11 Higher-Level Concurrency Constructs and Frameworks

The concurrency constructs we covered so far in this chapter are considered the building blocks of parallel programming. In order to create massively concurrent systems, like ones that process big data, system developers and architects usually use higher-level concurrency constructs, which are usually provided by libraries that went through scrutinized verification to ensure their correctness and optimality. Java, for example, provides the following thread-safe collections in the `java.util.concurrent` package:

- **BlockingQueue:** a thread-safe bounded queue that can be used to solve the producer-consumer problem.
- **ConcurrentMap:** a thread-safe map that provides atomic access when manipulating key-value pairs.
- **ConcurrentHashMap:** a thread-safe `HashMap`.
- **ConcurrentNavigableMap:** a thread-safe map that allows for approximate matching.
- **ConcurrentSkipListMap:** a thread-safe `TreeMap`.

10.11.1 Executors

So far in this chapter we've been demonstrating parallel programming in Java using threads that were created and managed explicitly; we've established an affinity between a task (a `Runnable` object) and a thread (a `Thread` object). Separation of concerns is a design practice to follow, which behooves us to decouple those two aspects. In this subsection, we explore various way to achieve such decoupling and scale in Java.

Thread Pools

Instead of creating a new thread every time we need to run a task in parallel, we can create a pool of threads from which we can get a thread to run a task and then put it back into the pool when we're done with it. An object pool is a generic construct that can be used for other types as well; it's useful when we need to manage resources that we know are going to be allocated; hence, we allocate them in advance and pool them together; pooling is commonly used for threads and sockets. The main purpose of pooling is to avoid repetitive allocation and garbage collection which can add a significant performance penalty for ultra-scale applications, especially when creating and freeing thread objects as they usually have a heavy memory footprint.

In Java's concurrency package, `java.util.concurrent`, thread pools are used to achieve decoupling of thread objects from task objects; they use worker threads which can be reused to run multiple tasks. Another feature of pooling is using a fixed-size pool to bound the maximum number of threads created (and hence, running in parallel). Take for example a web server that creates a new thread for each incoming request, a fixed thread pool can help mitigate CPU contention and running out of memory when traffic spikes. A fixed thread pool reuses a specified maximum number of threads to run tasks; if the fixed pool is exhausted (i.e., the maximum number of active threads is reached), new tasks are queued up—waiting for a thread to be available. A fixed thread pool maintains the same number of threads even if no task is running. Here's an example of a fixed thread pool in Java:

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class Main {
  public static void main(String args[]) throws Exception {
    final ExecutorService threadPool =
      Executors.newFixedThreadPool(3);
    final long startTime = System.nanoTime();

    for (int i = 0; i < 10; i++) {
      threadPool.submit(() ->doWork(startTime));
    }

    threadPool.shutdown();

    if (threadPool.awaitTermination(1, TimeUnit.SECONDS)) {
      System.out.println("Thread pool terminated gracefully.");
    } else {
      System.err.println("Thread pool timed out!");
    }
  }

  private static void doWork(final long startTime) {
    System.out.println(
      "Timestamp: " + (System.nanoTime() - startTime));
    try {
      Thread.sleep(100);
    } catch (InterruptedException ex) {
      System.out.println(ex);
    }
  }
}
```

The above program produces output that looks like the following:

```
## Timestamp: 62583914
## Timestamp: 62986875
## Timestamp: 63276047
## Timestamp: 163417883
## Timestamp: 163472024
## Timestamp: 163515169
## Timestamp: 263825100
## Timestamp: 264186091
## Timestamp: 264196284
## Timestamp: 367402589
## Thread pool terminated gracefully.
```

Notice how work is done in batches; each batch uses a maximum of 3 worker threads as specified in the call to create a new fixed thread pool.

Thread pools can be also be elastic: they scale with increased demand to run parallel tasks and shrink when said demand diminishes. An example of such thread

pool is called a cached thread pool; to demonstrate the difference, we replace the statement that creates the thread pool in the previous example with this one:

```
ExecutorService threadPool = Executors.newCachedThreadPool();
```

The output changes to indicate that the tasks were run using a higher degree of parallelism:

```
## Timestamp: 63849408
## Timestamp: 63868197
## Timestamp: 63849734
## Timestamp: 64142182
## Timestamp: 64228182
## Timestamp: 64338038
## Timestamp: 64485624
## Timestamp: 64553205
## Timestamp: 64620546
## Timestamp: 64723542
## Thread pool terminated gracefully.
```

A cached thread pool creates threads on demand and caches available ones for later use; the caching mechanism expires the stand-by threads after sixty seconds of not being used, which helps shrink the size of the thread pool when no work is being done. Creating a thread pool using the `ThreadPoolExecutor` class allows for customizing said behavior to meet the needs of the system. Another way to create an elastic thread pool is to use a work-stealing one, which allows worker threads with nothing left to do to steal work from those that are still busy with work. To demonstrate the difference, we replace the statement that creates the thread pool in the previous example with this one:

```
ExecutorService threadPool = Executors.newWorkStealingPool(3);
```

The output changes to indicate that the tasks were run using the specified degree of parallelism (a maximum of 3 threads in parallel):

```
## Timestamp: 82440622
## Timestamp: 82545730
## Timestamp: 82441045
## Timestamp: 185718331
## Timestamp: 185785889
## Timestamp: 185837149
## Timestamp: 288977003
## Timestamp: 288977003
## Timestamp: 288977003
## Timestamp: 391095535
## Thread pool terminated gracefully.
```

The difference between a work-stealing thread pool and a fixed one is elasticity: A work-stealing thread pool may increase or decrease the number of threads dynamically. There are other subtle differences as it doesn't guarantee the order of tasks to be executed and it may use multiple queues to reduce CPU contention.

By default, if the degree of parallelism is not specified, Java uses the number of processors available to the JVM as the thread pool's target degree of parallelism.

Scheduled Executor Services

A scheduled executor service extends the executor service we created in the past few examples to support running tasks after a specified time delay or periodically according to a given schedule. Here's an example where we create three schedulers and run them using a thread pool that always keeps 3 threads in the pool (even if they're idle):

```
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

public class Main {
  public static void main(String args[]) throws Exception {
    final ScheduledExecutorService threadPool =
      Executors.newScheduledThreadPool(3);
    final long startTime = System.nanoTime();

    for (int i = 0; i < 3; i++) {
      final int id = i;

      threadPool.scheduleAtFixedRate(() ->
        doWork(id, startTime), 0, 100, TimeUnit.MILLISECONDS);
    }
    threadPool.schedule(
      threadPool::shutdown, 300, TimeUnit.MILLISECONDS);

    if (threadPool.awaitTermination(1, TimeUnit.SECONDS)) {
      System.out.println("Thread pool terminated gracefully.");
    } else {
      System.err.println("Thread pool timed out!");
    }
  }

  private static void doWork(final int id, final long start) {
    System.out.println("ID: " + id +
      "\tTimestamp: " + (System.nanoTime() - start));
  }
}
```

The above program produces output that looks like the following:

```
## ID: 0 Timestamp: 71308285
## ID: 1 Timestamp: 74043046
## ID: 2 Timestamp: 74122412
## ID: 0 Timestamp: 171757979
## ID: 1 Timestamp: 171771924
## ID: 2 Timestamp: 173471667
```

```
## ID: 0 Timestamp: 272738086
## ID: 1 Timestamp: 272818754
## ID: 2 Timestamp: 273137315
## ID: 0 Timestamp: 370991647
## ID: 1 Timestamp: 370996823
## ID: 2 Timestamp: 373159930
## Thread pool terminated gracefully.
```

Notice how the schedulers ran in parallel and executed the task according to their respective schedules. Timers and schedulers are useful to schedule background jobs in a big data system that perform routine tasks like data crunching or health check. Java has a class called `java.util.Timer` which provides a similar interface for scheduling tasks; however, the `Timer` class uses a single-threaded implementation and is more sensitive to changes in the system's clock (which we discussed in detail in Sect. 2.4.2); moreover, the `ScheduledThreadPoolExecutor` provides more flexibility in thread management and handling exceptions thrown by the scheduled tasks.

Futures

So far we've used executors to run tasks that aren't result-bearing; to return results from other threads that run in parallel, we need some sort of a tie between the requestor that kicked off those tasks and their respective threads of execution that shall eventually return back results. Java provides an interface called `java.util.concurrent.Future<V>`, which represents a task's return value when it's done (in the future). A future object allows callers to check whether or not the value is ready, to block waiting for it, and to obtain it. Here's an example that shows how to return the results of tasks run in parallel:

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.stream.IntStream;

public class Main {
  public static void main(String args[]) throws Exception {
    final ExecutorService threadPool =
      Executors.newCachedThreadPool();
    final List<Future<Integer>> futures = new ArrayList<>();
    final int[] data = IntStream.range(0, 10000).toArray();
    int sum = 0;

    for (int i = 0, step = 1000; i < data.length; i += step) {
      final IntStream chunk = Arrays.stream(data, i, i + step);
```

```
        futures.add(threadPool.submit(chunk::sum));
      }

      for (final Future<Integer> f : futures) {
        sum += f.get();
      }

      System.out.println("Parallel Sum: " + sum);
      threadPool.shutdown();
    }
  }
```

The above example is trivial but it demonstrates the basic building blocks that can be used in complex applications; it produces the following output:

```
## Parallel Sum: 49995000
```

Streams

Java 8 introduced a new feature called streams, which allows data to be processed in parallel. Parallel streams in Java use the Fork/Join framework, which was introduced in Java 7 to make use of multiple processors and distribute work over worker threads in a thread pool (that's implemented by the `ForkJoinPool` class). Both paradigms of parallel programming in Java have been heavily criticized mainly because they are portrayed as general-purpose frameworks while many voices from the Java community called them out as special-purpose ones use them judiciously. That said, they are quite useful for short-lived non-blocking tasks like in the previous example, which can be simplified using parallel streams to be much more succinct:

```
import java.util.stream.IntStream;

public class Main {
  public static void main(String args[]) throws Exception {
    System.out.println("Parallel Sum: " +
      IntStream.range(0, 10000).parallel().sum());
  }
}
```

To learn more about executing streams in parallel, we recommend this tutorial: docs.oracle.com/javase/tutorial/collections/streams/parallelism.html

10.11.2 ParSeq

ParSeq is an open-source framework that LinkedIn created to simplify asynchronous Java code; it allows developers to compose dependency graphs of tasks that can

be planned to run in parallel or in sequence (pipelining and branching). it also provides powerful tooling for retry policies, timeouts, execution graph tracing and visualization, error propagation and recovery, etc. Before delving into the details of those features, let's take a look at a familiar example, rewritten using ParSeq this time:

```
import com.linkedin.parseq.Engine;
import com.linkedin.parseq.EngineBuilder;
import com.linkedin.parseq.Task;
import com.linkedin.parseq.trace.TraceUtil;

import java.util.Arrays;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.stream.IntStream;


public class Main {
  public static void main(String args[]) throws Exception {
    final int availableProcessors =
      Runtime.getRuntime().availableProcessors();
    final ScheduledExecutorService scheduler =
      Executors.newScheduledThreadPool(availableProcessors + 1);
    final Engine engine = new EngineBuilder().
      setTaskExecutor(scheduler).
      setTimerScheduler(scheduler).
      build();
    final int[] data = IntStream.range(0, 10000).toArray();
    final int chunkSize = data.length / 4;
    final Task<Integer> sum = Task.par(
      Task.callable(
        "a",
        Arrays.stream(data, 0 * chunkSize, 1 * chunkSize)::sum),
      Task.callable(
        "b",
        Arrays.stream(data, 1 * chunkSize, 2 * chunkSize)::sum),
      Task.callable(
        "c",
        Arrays.stream(data, 2 * chunkSize, 3 * chunkSize)::sum),
      Task.callable(
        "d",
        Arrays.stream(data, 3 * chunkSize, 4 * chunkSize)::sum)
    ).map("sum", (a, b, c, d) -> a + b + c + d);

    engine.run(sum);
    sum.await();
    System.out.println("Parallel Sum: " + sum.get());
    engine.shutdown();
    scheduler.shutdown();
  }
}
```

Adding the following line after the task's completion allows us to print its trace as JSON:

```
System.out.println(TraceUtil.getJsonTrace(sum));
```

The above line produces output that's similar to the following:

```
{
  "planId":6,
  "planClass":"com.linkedin.parseq.FusionTask",
  "traces":[{
    "id":0,
    "name":"a",
    "resultType":"SUCCESS",
    "hidden":false,
    "systemHidden":false,
    "startNanos":92149398460897,
    "pendingNanos":92149403324968,
    "endNanos":92149403370470
  },
  ... ## shortened for brevity ##
  {
    "id":1000,
    "name":"sum",
    "resultType":"SUCCESS",
    "hidden":false,
    "systemHidden":false,
    "startNanos":92149406549789,
    "pendingNanos":92149406700458,
    "endNanos":92149406710705
  }
  ],
  "relationships":[{
    "relationship":"PARENT_OF",
    "from":5,
    "to":1000
  },
  ... ## shortened for brevity ##
  {
    "relationship":"SUCCESSOR_OF",
    "from":1000,
    "to":4
  }]
}
```

Using the Trace Visualization Server from ParSeq, we can obtain a waterfall graph for the task's trace that helps us understand what happened:

From the graph in Fig. 10.1, we can see that there's some overhead that the first subtask that runs in parallel has to pay (which is common in many applications; for example, populating a cache or allocating objects). Such tracing gives developers insights into the system—especially when it grows to be more complex—and help drive performance improvements. We explore another way to achieve the same task in the previous example: Since our `sum` call is a CPU-intensive one, we want to

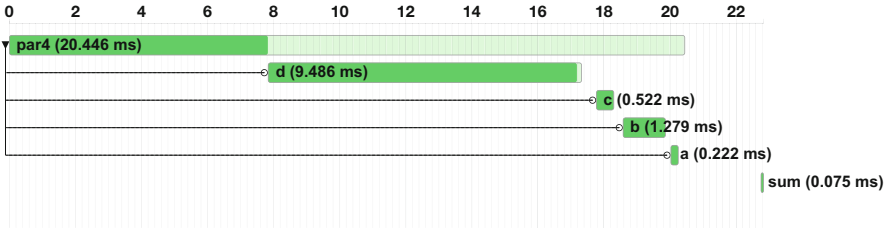


Fig. 10.1 Waterfall graph of the sum task using ParSeq

avoid running it on the thread ParSeq uses to run callable objects; we can explicitly choose the executor that runs our tasks by calling the `Task::blocking` method. In the example below, we choose to reuse the same thread pool from which ParSeq gets its dedicated thread, but we could have also created another thread pool if we wanted to; either way, we should be getting dedicated worker threads for our tasks from said thread pool. Here's the change we made in the way we construct our summation task:

```
final Task<Integer> sum = Task.par(
  Task.blocking(
    "a",
    Arrays.stream(data, 0 * chunkSize, 1 * chunkSize)::sum,
    scheduler),
  Task.blocking(
    "b",
    Arrays.stream(data, 1 * chunkSize, 2 * chunkSize)::sum,
    scheduler),
  Task.blocking(
    "c",
    Arrays.stream(data, 2 * chunkSize, 3 * chunkSize)::sum,
    scheduler),
  Task.blocking(
    "d",
    Arrays.stream(data, 3 * chunkSize, 4 * chunkSize)::sum,
    scheduler)
).map("sum", (a, b, c, d) -> a + b + c + d);
```

Figure 10.2 shows the trace of a more balanced execution plan.

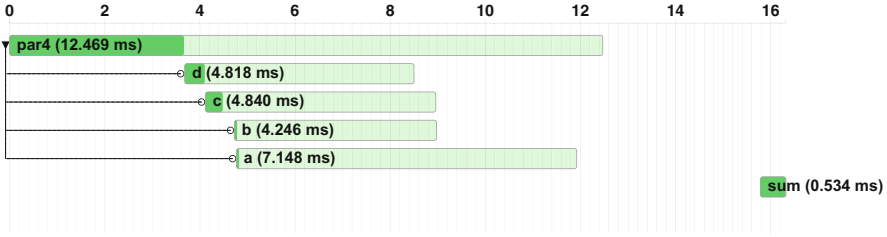


Fig. 10.2 Waterfall graph of the sum task using ParSeq

ParSeq really shines when used to perform asynchronous operations (e.g., I/O) in parallel, especially in the context of a large-scale system that requires special consideration for fault tolerance and other design requirements. Let's take the following code snippet as an example:

```
import com.linkedin.parseq.Engine;
import com.linkedin.parseq.EngineBuilder;
import com.linkedin.parseq.Task;
import com.linkedin.parseq.httpclient.HttpClient;
import com.linkedin.parseq.trace.TraceUtil;

import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

public class Main {
  public static void main(String args[]) throws Exception {
    final int availableProcessors =
      Runtime.getRuntime().availableProcessors();
    final ScheduledExecutorService scheduler =
      Executors.newScheduledThreadPool(availableProcessors + 1);
    final Engine engine = new EngineBuilder().
      setTaskExecutor(scheduler).
      setTimerScheduler(scheduler).
      build();
    final Task<Task<Void>> get = Task.par(
      getContentType("a://a"), // causes an exception
      getContentType("http://www.bing.com"),
      getContentType("http://www.facebook.com"),
      getContentType("http://www.google.com"),
      getContentType("http://www.yahoo.com"))
      .map((a, b, f, g, y) -> print(a, b, f, g, y));

    engine.run(Task.flatten(get));
    get.await();
    engine.shutdown();
    scheduler.shutdown();
    System.out.println(TraceUtil.getJsonTrace(get));
  }

  private static Task<String> getContentType(String url) {
    return HttpClient.get(url).task()
      .withTimeout(1, TimeUnit.SECONDS)
      .toTry()
      .map(t -> t.isFailed() ?
        "Failed to get " + url : t.get().getContentType());
  }

  private static Task<Void> print(String... strings) {
    return Task.action(() -> {
      for (final String s : strings) System.out.println(s);
    });
  }
}
```

There are a few things happening at the same time here (no pun intended); so let's break them down:

- The task at hand is a nested one; we created an action (which is wrapped in a `Task<Void>` object) as the return type of the `get` task. ParSeq allows for such use of tasks and provides the `flatten` method to deal with that.
- We used ParSeq's `HttpClient` class, which is a convenient wrapper for making HTTP requests and wrapping them into ParSeq tasks.
- We added a timeout of 1 s for each of the HTTP tasks.
- We also added an error handler for HTTP tasks that gracefully returns an error message instead of the content type and allows the parent task to continue as if nothing happened—a more sophisticated error handler is required for real-world applications.

The trace for the above example is shown in Fig. 10.3.

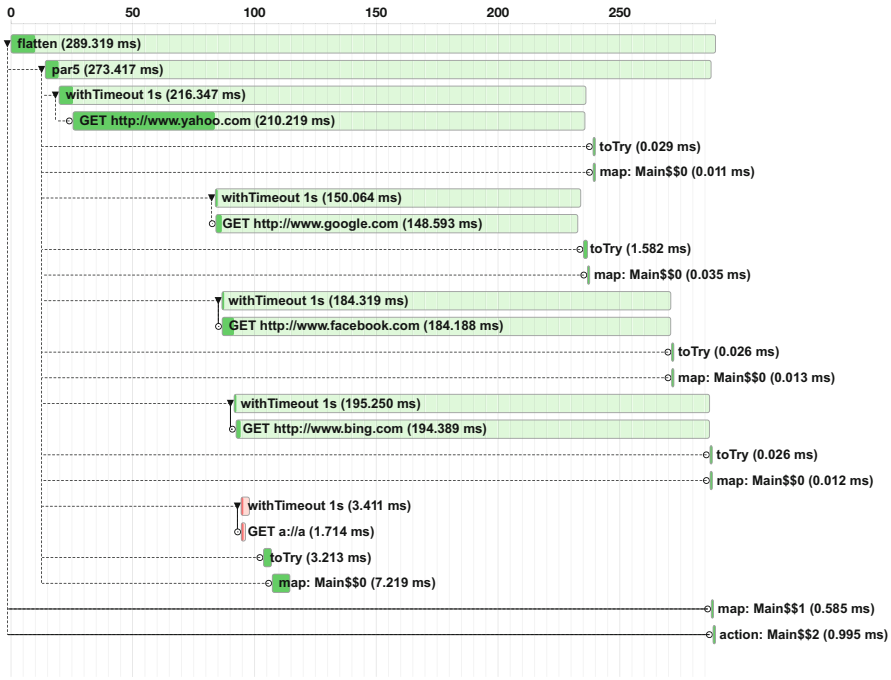


Fig. 10.3 Waterfall graph of ParSeq performing asynchronous HTTP operations

For further reading about ParSeq, check out its user's guide at https://github.com/linkedin/parseq/wiki.

10.11.3 Inter-Process Communication and Synchronization

In the overview section of this chapter we discussed key differences between threads and processes. So far, we have seen examples of synchronization and coordination of threads that belong to the same process, have access to its memory address space, and can intercommunicate using objects created in said address space (without having to use channels external to their parent process). That's the main advantage of parallelizing work using threads instead of processes: a relatively lower intercommunication cost. Processes, on the other hand, need to use external channels to communicate, which comes at a cost (usually I/O). Depending on the design choices developers make when creating a distributed system, some processes may communicate via channels that are available when they run on the same machine (e.g., files on a local disk storage); however, most big data systems are expected to be running a large number of processes distributed over many machines, which necessitates inter-process communication (IPC) remotely (over a network). In this subsection, we explore various techniques of IPC.

File Locks

Files on a local disk storage can be used for IPC and synchronization, thanks to operating systems that provide file locking support. In fact, most operating systems not only support mutually exclusive file operations (e.g., read and write), but also provide support for said operations at a finer granularity: a region of a file. Since file locks are native to the OS, many programming languages make use of that and programs that are written in different languages can intercommunicate using files with little to no interoperability concerns.[9]

In Java, we will use the `java.nio.channels.FileLock` class to demonstrate how to use it as a mutex that protects a critical section; this time around, only a single process can be inside said critical section:

```
import java.lang.management.ManagementFactory;
import java.nio.channels.FileChannel;
import java.nio.channels.FileLock;
import java.nio.file.FileSystems;
import java.nio.file.Path;

import static java.nio.file.StandardOpenOption.*;
```

[9] An infamous example of interoperability issues is discrepancy of endianness (the order scheme for bytes in words, a buffer, or a stream), which may differ based on the machine's hardware, the programming language, and/or a framework's implementation; for example, Java's `DataInputStream` is big-endian while its C# equivalent, `BinaryReader`, is little-endian. It's also worth mentioning that there's a performance overhead if processes need to translate endianness all the time while communicating.

```
public class Main {
  public static void main(String args[]) throws Exception {
    final Path lockPath =
      FileSystems.getDefault().getPath(".lock");
    final String processID =
      ManagementFactory.getRuntimeMXBean().getName();

    try (final FileLock _ =
      FileChannel.open(lockPath, WRITE, CREATE).lock()) {
      System.out.println(processID + " acquired the lock");
      Thread.sleep(1000); // simulates other work being done
      System.out.println(processID + " is releasing the lock");
    }
  }
}
```

The above program checks for the existence of a file called `.lock` in the current working directory; if it can't find it, it creates a writable file with that name. Then it opens a file channel to acquire a file lock on the entire file; it then proceeds to enter the critical section if it obtains the lock; otherwise, it waits until the file lock—which is held by another process—is released. The `FileLock` class is an `AutoCloser`, which means that it can be used within a try-with-resource statement; the `close` method releases the lock so that other processes may grab it. It's worth mentioning that a file lock in Java can also be released by invoking its `release` method, by closing the file channel from which it was created, or by terminating the process.

We then launch three concurrent processes that execute the above program by running the following command from the directory where the class file resides:

```
java Main& java Main& java Main&
## [1] 52261
## [2] 52262
## [3] 52263
## 52261@MACBOOK-PRO acquired the lock
## 52261@MACBOOK-PRO is releasing the lock
## 52263@MACBOOK-PRO acquired the lock
## 52263@MACBOOK-PRO is releasing the lock
## 52262@MACBOOK-PRO acquired the lock
## 52262@MACBOOK-PRO is releasing the lock
```

The output above shows how access to the critical section was serialized, thanks to the mutex we created using a file lock.

Distributed Locks

While file locks may be supported by some network filesystems, it's more apt to use a distributed lock manager (DLM) for big data systems. A DLM provides APIs for locking and synchronization of access to shared resources across a distributed system (or processes that can be running on different machines). At Microsoft,

we had to build an in-house DLM as part of a Platform-as-a-Service (PaaS) we built to power up Windows Live, Outlook.com, and OneDrive; At LinkedIn, we used Apache ZooKeeper,[10] an open-source high-performance coordination service for distributed applications that was originally created at Yahoo as part of Hadoop and has been widely used by many big companies like Netflix.[11] One of the many features that ZooKeeper provides is a synchronization service, which makes use of a shared hierarchical namespace to help with inter-processes coordination. ZooKeeper provides recipes that can be used to implement various locking techniques in a distributed fashion; it's even easier to do so using Apache Curator, which provides a high-level framework to simplify ZooKeeper; Curator's recipes can be found http://curator.apache.org/curator-recipes/ and they implements the distributed version of constructs we already covered in this chapter: locks, reentrant locks, reentrant read-write locks, semaphores, etc.

Using ZooKeeper v3.4.9 and Curator v2.9.1, we demonstrate below how to use ZooKeeper to rewrite the previous example:

```
import org.apache.curator.RetryPolicy;
import org.apache.curator.framework.CuratorFramework;
import org.apache.curator.framework.CuratorFrameworkFactory;
import org.apache.curator.framework.recipes.locks.*;
import org.apache.curator.retry.ExponentialBackoffRetry;

import java.lang.management.ManagementFactory;

public class Main {
  public static void main(String args[]) throws Exception {
    final RetryPolicy retryPolicy =
      new ExponentialBackoffRetry(1000, 3);
    final CuratorFramework client =
      CuratorFrameworkFactory.newClient(
        "127.0.0.1:2181",
        retryPolicy);
    final InterProcessMutex lock =
      new InterProcessMutex(client, "/path/to/lock");
    final String processID =
      ManagementFactory.getRuntimeMXBean().getName();

    try {
      client.start();
      lock.acquire();
      System.out.println(processID + " acquired the lock");
      Thread.sleep(10000); // simulates other work being done
      System.out.println(processID + " is releasing the lock");
    } finally {
      if (lock.isAcquiredInThisProcess()) {
        lock.release();
```

[10] https://github.com/linkedin/linkedin-zookeeper.

[11] https://github.com/Netflix/curator.

```
            }
            client.close();
        }
    }
}
```

Some notable differences in the above example: we're making a network call, so a client is required and it has its own retry policy for fault tolerance; calls to acquire and release the lock may throw exceptions mainly because they're making network calls; and a process needs to check before releasing a lock that it actually holds it to avoid an illegal state exception that's thrown by the Curator framework in such case.

Inter-Process Synchronization Caveats

Thread synchronization within the same process need not be concerned with failure modes that stem from relying on dependencies external to said process; inter-process synchronization is relatively more error-prone as developers need to worry about those external dependencies in addition to getting the synchronization code right. For example, a process that holds a lock managed by ZooKeeper can suffer from a network partitioning issue that isolates it from the rest of the cluster (including ZooKeeper); in similar cases, a ZooKeeper recipe may choose to mitigate starvation (of other processes waiting for the lock) by revoking the lease that the isolated process had on the lock to make it available for other processes to acquire. Curator's lock recipes use what's known as ephemeral znodes[12] to store locks. A process that owns an ephemeral znodes needs to keep telling ZooKeeper that it's alive (by sending a heartbeat message before its session times out); otherwise, ZooKeeper deletes its ephemeral znodes (and thus the lock stored in said znode is released). ZooKeeper doesn't notify an expired client that its session expired until the client reconnects to ZooKeeper (as it assumes the client is unreachable). So, a process that got disconnected from ZooKeeper may not get the message that the lock it used to own has been released and may think it still holds said lock. This class of race conditions is known as time-of-check-to-time-of-use (TOCTTOU) bugs.

Scenarios like the above call for utmost scrutiny when dealing with inter-process synchronization since the true state of a lock is stored somewhere external to the processes participating in said synchronization. They also call for balancing coarse-grain locking (which can improve performance, thanks to making fewer remote network calls) with fine-grain locking (to mitigate correctness issues like the above). As usual, business requirements should drive those design decisions; if a lock is required to ensure the correctness of a transaction, it will require more

[12] ZooKeeper facilitates inter-process coordination via a shared hierarchal namespace that mimics a file system; said namespace is composed of data registers known as znodes (akin to files and folders, or nodes, on a file system).

safety precautions than a lock that's merely required to avoid duplicate work to be done by multiple processes. For the sake of this discussion, we will assume that we want the former and delve deeper into how to reduce the chances of inter-process synchronization issues. Back to our previous scenario, one may choose to keep reacquiring the lock, for fear that ZooKeeper might have released it, before each transaction; the underlying assumption here is that a connection may only expire due to a networking issue—wrong! Other reasons may come into play, like when the JVM's garbage collector stops the world to reallocate objects in order to defragment free memory; such pauses in the JVM can happen at any time and for prolonged periods that may cause a ZooKeeper session to expire right after it acquired a lock; when the program resumes, it will be inside a critical section even though it doesn't hold the lock anymore. In such cases, two processes might be writing to a shared resource and possibly corrupting it.

Optimistic Concurrency Control

So what can we do about that issue? One solution is to make the service that manages the shared resource reject stale updates so that the order of update operations matches that of lock acquisitions. For example, we can use Optimistic Concurrency Control (OCC), a concurrency control method that's commonly used by transactional systems, to reject an update transaction that's based on stale data. When a process reads data from a shared resource, it also gets a version number, a random GUID, or a timestamp that identifies the most recent version of said data at that time.[13] And when a process requests an update to the shared data, it sends along the version number so that the service that manages the shared resource can validate the version number hasn't changed (or about to be changed by an in-progress request). If that check passes, the service updates the shared resource and increments the version number; otherwise, it rejects the update. In doing so, it prevents update requests from stepping over each other.

OCC is usually used in environments where competing processes seldom step over each other to avoid the overhead of locking, which is considered—in contrast—a pessimistic concurrency control method; locking is more efficient when conflicts are frequent as rejected transactions are a rarity and thus don't have to be re-performed. In this solution, we are combing both approaches: locks to mitigate conflicts and OCC to ensure correctness when a conflict occurs.

[13] As we discussed earlier in Sect. 2.4.2, such use of timestamps can be error-prone in a distributed system due to clock drift; however, they are easier to generate than a version counter that requires a consensus algorithm so that various machines agree it's correct. The trade-off between level of accuracy vs. performance gains shall be made based on the distributed system's requirements.

Linearizable Counters

Assuming we have a shared resource that's managed by a distributed system and we use a strictly increasing counter as its version, we will need to use a linearizable store to generate such numbers. Most distributed database management systems (DDBMS) support auto-incrementing a field on updates and they ensure consistency across different machines that host the DDBMS, thus making it easier for developers to rely on that feature when versioning a database record. In cases when a DDBMS is not a valid option, ZooKeeper, which we've already been using for distributed locking, can do the job. A change to a znode causes ZooKeeper to increment the znode's version number; we can create a znode for our counter and use Curator to commit a dummy transaction:

```
import org.apache.curator.RetryPolicy;
import org.apache.curator.framework.CuratorFramework;
import org.apache.curator.framework.CuratorFrameworkFactory;
import org.apache.curator.retry.ExponentialBackoffRetry;

public class Main {
  private static final String HOST = "127.0.0.1:2181";

  public static void main(String args[]) throws Exception {
    final RetryPolicy retryPolicy =
      new ExponentialBackoffRetry(1000, 3);
    try (final CuratorFramework client =
      CuratorFrameworkFactory.newClient(HOST, retryPolicy)) {
      client.start();

      final int version = client.inTransaction().
        setData().forPath("/path/to/counter").and().
        commit().iterator().next().
        getResultStat().getVersion();

      System.out.println("Version: " + version);
    }
  }
}
```

The above code assumes that the node already exists and it prints out the version number of the requested dummy transaction, which writes an empty byte array to the specified znode.

Lamport Timestamps

In a distributed system where many processes run concurrently or on a single processor where instructions may be executed out of order[14] (to improve perfor-

[14] https://en.wikipedia.org/wiki/Out-of-order_execution.

mance), it's sometimes important to establish an order of events that reflect causality (rather than what happened in reality). For example, the Java Language Specification defines happens-before relationships[15] guarantees in certain situations; like the one we discussed earlier in this chapter: writing to a `volatile` field happens-before subsequent reads of said field. In a distributed system, processes intercommunicate by sending and receiving messages; each of these operations is considered an event; those events follow a strict partial order, which helps with establishing happens-before relationships within the system. In turn, establishing said relationships enables us to implement concurrency constructs (e.g., mutex) for concurrent processes. And in order to establish such order of events correctly, we need to rely on a logical clock, like the one we introduced in Sect. 2.4.1, to avoid clock drift issues. Luckily, we have Leslie Lamport's algorithm[16] to maintain the happened-before order of events in a distributed system. Each process maintains a logical clock that ticks (increments its counter) before sending its value along with an outgoing message; when a process receives a message, it synchronizes its logical clock with the senders if the received counter value is greater than its own, then increments it. Here's an incomplete implementation in Python (for demonstration purposes only):

```
def send(self, message):
  self.counter += 1
  send(message, self.counter)

def receive(self):
  message, counter = receive()
  self.counter = max(self.counter, counter) + 1
```

Lamport's logical clock is closely related to the more generalized vector clock, which supports an arbitrary number of independent processes that run concurrently. In the heart of any big data system, you may always find a ticking logical clock.

10.12 Non-Blocking Parallel Computing

Locking, if not used correctly, can make code prone to serious concurrency issues like deadlocks, livelocks,[17] lock convoys,[18] etc. In addition, locking is expensive and may create a performance overhead in situations where performance is critical;

[15] https://docs.oracle.com/javase/specs/jls/se7/html/jls-17.html#jls-17.4.5.

[16] http://amturing.acm.org/p558-lamport.pdf.

[17] A livelock is akin to two people repeatedly trying to pass each other by stepping aside, only to recreate the same scenario over and over again.

[18] A lock convoy forms as a result of repeated underutilization of time slices, which may happen due to frequent context switching between threads of equal priority when some of them fail to acquire a lock.

that's why system designers sometimes need to find alternatives to locking when designing parallel computing systems.

One alternative is lock-free programming. The idea behind lock-free parallel programming is to provide thread-safe ways to access shared resources without the need for locks or synchronization primitives (e.g., mutual exclusion) that use locks. One approach to do so is to build thread-safe data structures (rather than generic algorithms) that don't use locks; instead, they use hardware-supported atomic operations, like the interlocked operations we discussed earlier in this chapter, and most notably: compare-and-swap. Some notable examples of lock-free data structures in Java belong to the `java.util.concurrent` package:

- `ConcurrentLinkedDeque`
- `ConcurrentLinkedQueue`
- `ConcurrentSkipListMap`
- `ConcurrentSkipListSet`

Lock-blocking algorithms should come with a warning; they may cause starvation and they are no silver bullet for contention either—use them judiciously.

In a system of threads where locks are used, a thread that holds a lock causes all other threads to block waiting for said lock to be released. If the current holder of the lock gets suspended for some reasons, all waiters are blocked from making progress—the system is stuck! In contrast, when threads compete to access a lock-free thread-safe shared data structure, at least one of them is guaranteed to make progress even in the case of suspension of failure, which in turn guarantees system-level progress at all times. Better yet, if a lock-free algorithm (or data structure) guarantees per-thread progress, it's called wait-free; all wait-free algorithms are lock-free but not the other way around. Java's `ConcurrentLinkedQueue` class, which is based on a paper (Michael and Scott, 1996) called "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," is wait-free.

10.13 Beyond the CPU

We briefly pointed out in Chap. 2 that a common way to scale computations is parallelization by distributing them over a large number of computers and/or using GPU-accelerated computing. Moreover, a typical laptop computer manufactured in 2017 has thousands of GPU cores made specifically to perform parallel computations efficiently; compared to a few CPU cores, each is tasked with processing sequential instructions. GPUs have been extremely popular among data scientists because of their incredible performance in data parallelism (e.g., executing the same instruction in parallel on elements of a massive vector). GPU manufacturers, like NVIDIA, produce GPUs made for general-purpose computing on graphics processing units (GPGPU) in addition to software development kits, like Compute Unified Device Architecture (CUDA), to make programming on GPUs a commodity. In addition, multiple GPUs can be combined to provide even more

parallelization to a single machine; in fact, a single NVIDIA DGX-1 box contains a whooping 40,960 CUDA cores and can perform up to 960 trillion floating-point calculations per second!

GPU-accelerated computing[19] is great for embarrassingly parallel tasks, especially vector and matrix processing; GPUs have been used for high-performance computing clusters, bioinformatics, scientific computing, databases, machine learning, etc. An open standard called OpenCL allows a cross-platform support for parallel computing on GPUs, CPUs, and field-programmable gate arrays (FPGAs); it's been actively supported by various chip makers.

Microsoft has been using FPGAs all over its data centers for its cloud offering, Azure, and for a real-time AI project called Brainwave.[20] The specialized hardware runs algorithms that are written onto the chips—making them highly efficient for parallel computing, especially machine learning tasks.

Google has been working on an application-specific integrated circuit (ASIC) that's dedicated for machine learning tasks; it's specifically made to work with Google's TensorFlow machine learning framework and hence called a Tensor Processing Unit (TPU). The key differences between a GPU and a TPU is that the latter produces higher IOPS per watt at a lower float-point precision and it doesn't include some graphics-specific hardware like that used for texture mapping; those differences make TPUs more efficient for Google's specialized parallel computations at scale; for example, Google reported that a single TPU can process over 100 million photos a day. The second generation of TPUs, which was announced in May 2017, is capable of performing 45 TFLOPS; the new TPUs can be combined into a single pod with 256 units—that's 11.5 PFLOPS[21]! To put that number in perspective, the fastest supercomputer at the time of writing—China's Sunway TaihuLight—recorded a peak performance of 125 PFLOPS and costed $273 million.

10.14 Notes

10.14.1 Python

Earlier in this chapter, we debated why Java is more apt than Python (or any other language at the time of writing) for demonstrating concurrency concepts. There are more reasons why Python (or more specifically, CPython, its reference and most popular implementation) is not a great candidate when it comes to multi-threading—namely, the global interpreter lock (GIL). An interpreter that uses a GIL, like CPython, only allows a single thread to execute at a time, even on a multi-core

[19] http://images.nvidia.com/content/tesla/pdf/Apps-Catalog-March-2016.pdf.

[20] https://www.microsoft.com/en-us/research/blog/microsoft-unveils-project-brainwave.

[21] A PFLOP is a quadrillion floating-point calculations per second.

or multiprocessor architectures, which are prevalent nowadays; the GIL is oblivious to that superb advantage. To put things in perspective, my desktop machine—which houses a 14-core Xeon processor—can run 28 threads in parallel[22]; yet a Python program with hundreds of threads can only execute one thread at a time on that processor!

So why use a GIL if it's bad for multi-threading? The reason such mutex is used in CPython is to provide thread safety; CPython's memory management is thread-unsafe; it also allows safe integration of C libraries and modules that are thread-unsafe. The GIL has been the center of many heated debates in the Python community[23]; we leave it to the enthusiastic reader to learn more about the GIL's specifics; that said, it's worth mentioning that an imperfect workaround to achieve better parallelism in Python is to use the `multiprocessing` library, which sidesteps the GIL by using subprocesses instead of threads.

10.14.2 Further Readings

There are many great references out there that further discuss concurrency in Java, we recommend (Goetz and Peierls, 2006) and Chapter 10 from (Bloch, 2008); to learn more about ZooKeeper and how to use it for coordination tasks and synchronization, we recommend reading (Junqueira and Reed, 2013).

References

Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '96, pages 267–275, New York, NY, USA, 1996. ACM. ISBN 0-89791-800-2. doi: 10.1145/248052.248106. URL http://doi.acm.org/10.1145/248052.248106.

B. Goetz and T. Peierls. *Java Concurrency in Practice*. Addison-Wesley, 2006. ISBN 9780321349606.

J. Bloch. *Effective Java: 2nd Edition*. 2008.

F. Junqueira and B. Reed. *ZooKeeper: Distributed Process Coordination*. O'Reilly Media, Incorporated, 2013. ISBN 9781449361303.

[22] Some processor architectures can be configured to run multiple threads per core, thanks to simultaneous multi-threading (SMT).

[23] https://wiki.python.org/moin/GlobalInterpreterLock.

Chapter 11
Essential Knowledge: Testing

We believe that testing is inevitable; the question is: When do you want it to happen? You can let users test your code for you in production, at a hefty cost; or you can catch bugs as early as possible in your code's lifecycle and save yourself the embarrassment and the money. Improper testing can lead a lot of things to go wrong—if not fatal. In the era of big data, and as machine learning is becoming more prominent, proper testing is more challenging than ever before. Take for example self-driving cars, it may take millions of miles driven to test such technology before releasing it. As the data start to dictate the behavior of a program, instead of solely testing the code, now we also have to test the data. So much has been written to show the cost of software bugs[1] and the value of software testing—that is, before releasing it to the unforgiving world—so we won't dwell on that. Almost every decent software development book out there has a chapter or two about the importance and the how-to of software testing. In this book, and throughout this chapter, we discuss various techniques, tips, and tidbits that we have been following at companies like LinkedIn, Microsoft, and Voicera—ones that work with big data to solve big problems.

A common question that gets asked frequently is: When to test? The answer should be obvious: always and as early as possible! Releasing code to production without testing it is akin to jumping off a plane without checking if your parachute works. Then thinking about what to test, similarly, the answer is everything that can be tested should be tested: code, configuration, localization, accessibility, etc.

Testing is a craft and an art; it takes knowledge and creativity to find out the weak spots in a system and break it. In our experience, great testing skills are hard to find, hone, and incorporate in the process of software development. Take for example the practice of filling gaps in testing with new changes introduced in the area under test. For example, it's a good rule of thumb to require bug fixes to have regression tests that cover the scenario that was missed at the first time around; that

[1] https://en.wikipedia.org/wiki/List_of_software_bugs.

© Springer Nature Switzerland AG 2018

G. Lebanon, M. El-Geish, *Computing with Data*,
https://doi.org/10.1007/978-3-319-98149-9_11

way, not only do you make sure that the fix works, but also you ensure that future changes don't introduce a regression to the intended behavior. Such practice should be enforced in code reviews; experienced code reviewers ask for tests to accompany code changes generally speaking, especially when it's a fix to a regression issue. Each test missed at the time of coding and only needed after an issue occurring in production should be treated as a lesson; keep note of recurring coding mistakes as such, especially if you notice that multiple people on your team are making them, then it might become ingrained in the team's culture and can become harder to rectify. When testing is not a priority for a team, the diagnosis could be because some team members don't believe in the importance of testing, think of it as a chore, and/or can't find the time to meet deadlines as well as testing code properly. Building a culture that promotes testing is one that cares about what the customer experiences when using the product.

Below, we explore testing strategies and methods that have been effective at various levels of testings.

11.1 Black-Box Testing

As the name suggests, the idea behind black-box testing is subjecting the code under test to scenarios that make no assumptions about how the code was written; the tester sees a black box that accepts inputs and produces outputs—one can't see what's inside said box. This is an extremely useful strategy as it eliminate any bias the tester may have formed about how to use the feature being tested. With enough number of users, one should expect any feature, no matter how small, to be subjected to a great variation in the way it's being used. Anchoring, a cognitive bias that causes humans to be heavily influenced by first pieces of information they learned, is a pitfall testers need to avoid when making decisions about testing code they know how it works. The effects of anchoring can create many blind spots for testers; it's harder to point out what's missing or hidden when there's so much at which to point. To draw an analogy, think about the process of code review: it's mainly focused on the code that's in the review request; a good reviewer finds issues with the code they see in the review request; a great reviewer checks for what's missing and finds issues in parts of the code that are hidden. We've always appreciated how a reviewer catches a subtle issue that's only revealed, thanks to deep knowledge and a keen eye for details.

A real-life example of past experience may help illustrate the power of anchoring and blind spots. We were testing an API that allows developers to set and get generic properties of files in a SQL Server database. For example, we can set the time at which a file was last checked out from a version control system. Because we already had these scenarios on mind, we were anchored to test the API—which is meant to be generic—in a limited set of scenarios. In fact, reading the above couple of lines may have anchored you to think the same way unless you were already cognizant of such bias. An effective way of fighting anchoring was black-

box testing and following best practices like testing extremes and bounds; testing the API with `DateTime.MaxValue` led us to find a subtle bug: that field's value in the .Net framework corresponds to 23:59:59.**9999999** UTC, December 31, 9999; however, the maximum value of SQL Server's `datetime` type is 23:59:59.**997** UTC, December 31, 9999. At first it may seem that it's sufficient to limit the range of values to what SQL Server supports, but we didn't stop there. After further investigation, another bug was uncovered: SQL Server's accuracy for the `datetime` type causes milliseconds to be rounded to the closest of 0, 3, or 7 milliseconds within the type's range. SQL Server's `datetime` supports an accuracy of 3.33 milliseconds while .Net's `DateTime` supports an accuracy of 100 nanoseconds! The rounding of milliseconds happens implicitly and without throwing an exception, which made this issue hard to discover. For most applications, especially the ones that we had in mind designing this API, the accuracy of 1 s would have been acceptable. Yet, the contract we exposed wasn't clear about such assumptions or expectations; so developers could have called the API to store any value they wished. For example, one may use `DateTime.MaxValue` as a sentinel value, which would have lost a couple of milliseconds after a round trip to the database, leading to erroneous equality checks. The fix was to only allow `DateTime` values that can be mapped—without loss—to .Net's `SqlDateTime` values.

Depending on who's testing a unit of code—its author or someone else—getting rid of bias when testing could be really hard. In a nutshell, black-box testing verifies what a feature should do and doesn't care about how it's being done. That's why black-box testing is also known as specification-based testing.

11.2 White-Box Testing

A white-box system is one whose internals are visible; so technically, it's a misnomer to call the box white—it's rather transparent. When testing a white-box system, one should come up with test cases that are more specific to how the system was built. Ignoring said details and choosing test cases that treats the system merely as a black box may lead to many tests exercising the same code path—they can be equivalent and should be treated as duplicates. That's akin to test-driving a car while throwing a veil on it; for the experienced tester, it helps to peer under the hood to know how to best test the car.

Effective white-box testing requires strong knowledge of the various system components, software libraries, algorithms used, and expected behaviors and limits of all of the above. For example, in unit testing, it's extremely useful to know how the code unit was written so that tests can cover the various branches in the code's control flow. It also makes a big difference in testing when a function call is remote or local to the same machine where the code is running, what protocols are used to make remote calls, which database engine is used for storage, etc. The idea behind white-box testing is that knowing about how the system works leads to more precise and comprehensive test cases; that said, that shouldn't limit test cases to those that exercise the code as is at the time of testing.

In our previous example where testing an API that used SQL Server to store dates, one can argue that white-box testing played a big role in identifying the issue; however, that requires knowledge of the various components in the white box beforehand in order to come up with those test cases that can reveal such issues. Here's an example in Python for a unit test that verifies the behavior of putting a value in a key-value store in a white-box fashion:

```
import unittest

class KeyValueStore(object):
  ''' A key-value store example '''
  def __init__(self):
    self._store = {}

  def put(self, key, value):
    ''' Puts a value in the store '''
    self._store[key] = value

  def get(self, key):
    ''' Gets a value from the store by its key '''
    return self._store.get(key)


class TestKeyValueStore(unittest.TestCase):
  ''' A unit test for KeyValueStore '''
  def test_put(self):
    ''' Tests put '''
    store = KeyValueStore()
    store.put('white-box', 'testing')
    self.assertEqual(store._store.get('white-box'), 'testing')


if __name__ == '__main__':
  unittest.main()
```

To inspect the inner workings of the `KeyValueStore`, the unit test digs into the implementation details of the class to verify the post-conditions of the put operation.

11.3 Gray-Box Testing

Gray-box testing is akin to peering a little into how the code works in order to add more test coverage of scenarios that may have been missed during black-box testing. Test case design, in the context of gray-box testing, is informed by partial knowledge of how the system works. Since gray-box testing is a combination both black- and white-box testing, it can combine the benefits of both if done correctly; the main advantages are increasing the specificity of tests, targeting the scenarios that matter most, and reducing test bias.

Here's an example of testing the `put` method of our key-value store in a scenario that requires some knowledge about how the store is implemented to verify the post-conditions of the method:

```
import unittest

class KeyValueStore(object):
  ''' A key-value store example '''
  def __init__(self):
    self._store = {}

  def put(self, key, value):
    ''' Puts a value in the store '''
    self._store[key] = value

  def get(self, key):
    ''' Gets a value from the store by its key '''
    return self._store.get(key)


class TestKeyValueStore(unittest.TestCase):
  ''' A unit test for KeyValueStore '''
  def test_put_key_already_exists(self):
    ''' Tests put when key already exists '''
    store = KeyValueStore()
    store.put('same key twice', 'first time')
    store.put('same key twice', 'second time')
    # This a black-box assertion
    self.assertEqual(store.get('same key twice'), 'second time')
    self.assertEqual(len(store._store), 1)  # a white-box one


if __name__ == '__main__':
  unittest.main()
```

11.4 Levels of Testing

At each stage of the code's lifecycle there's a certain level of testing that's required:

- **Unit Testing:** testing units of source code; said units can be as small as a single function.
- **Integration Testing:** testing multiple modules (or units) that are integrated together as a group.
- **System Testing:** testing the system end-to-end on the hardware on which it's meant to run; this also includes validation of feature requirements, performance (load, stress, endurance, etc.) testing, security testing, recovery testing, etc.

- **Acceptance Testing:** in the context of big data systems, acceptance testing can be viewed as piloting the system in an environment that mimics the production environment to test its operational readiness.
- **Real-User Testing:** before releasing software to the public (also known as general availability or GA), it should be tested by real users.

We will delve into each level in detail in the next few sections; most of the concepts we're about to discuss can be applied at any level of testing; even though we will introduce many of them in the unit testing section, you can generalize them to other levels of testing if applicable.

11.5 Unit Testing

Unit testing is an extremely effective first line of defense against egregious bugs. If you're following the practice of test-driven development (TDD), then keep at it as long as it's working well for you. TDD is great when you can afford it. Whether you write tests first or after the production code, we suggest frequent and fast iterations of testable units that are as small as possible; this allows for catching errors early before building on top of erroneous assumptions. What we mean by a testable unit is the smallest unit of code that can be tested while developing bigger pieces of code. For example, let's assume you're implementing a key-value store with an interface to put and get values; you may code and test the `put` and `get` methods of first as they constitute a scenario that requires both to be implemented, then you can move on to other scenarios like deletion and range iteration, etc.

```
class TestKeyValueStore(unittest.TestCase):
  ''' A unit test for KeyValueStore '''
  def test_put(self):
    ''' Tests put '''
    store = KeyValueStore()
    store.put('black-box', 'testing')
    self.assertEqual(store.get('black-box'), 'testing')
```

We typically use words like *building* a feature and *creating* software; engineers wear their construction hats to write code in order to ship features and—naturally—they take pride in what they create. This sense of feeling proud of what we create, even if we contributed only partially to the creation process, is deeply rooted in behavioral psychology. Take for example owners of IKEA furniture (the Swedish furniture retailer that sells furniture in pieces), which require labor to assemble; the process of assembling IKEA furniture is notoriously known for being frustrating and time consuming. However, after going through the hard labor, owners of IKEA furniture tend to assign the furniture they assembled a disproportionately higher value than what it's worth. The price of IKEA furniture is originally low because it saves the retailer labor, storage, and shipping costs, thanks to the compact boxes in which they sell furniture; in addition, the retailer reuses many of the components

when designing furniture so they can mass-produce them at lower costs. So why do the owners of IKEA furniture value their own pieces at a much higher value than what they are worth objectively? As Dan Ariely, Professor of Psychology and Behavioral Economics at Duke University and the author of Predictably Irrational, explains in his book (Ariely, 2010): We tend to overestimate the value of what we created (or even assembled); we fall in love with the fruit of our labor regardless of the quality or other objective measures. We are susceptible to a cognitive bias that's known as the IKEA effect.

Since unit testing is typically performed by the creator of the unit under test, it's crucial to set pride aside and strive to break the code you just built in order to test it properly. Generally speaking—and especially during testing—one must be cognizant of the IKEA effect and endeavor to rid one's self of it. If it helps, pretend that someone else wrote the code and your mission is to find what could be wrong with it.

Another cognitive bias to be cognizant of is confirmation bias, which is the tendency to search for evidence to support our preexisting beliefs and preconceived notions instead of seeking evidence of the inconvenient truth. In unit testing, it's far more convenient to write test cases that confirm what the code does instead of what it *should* do (as per the requirements' specifications). A series of confirmation bias experiments conducted in the 1960s by a British cognitive psychologist called Peter Cathcart Wason, who coined the term, showed that confirmation bias leads to overconfidence and poor decision making. Thus, it's harder—and far more important—to catch what's missing than test what already exists in terms of functionality and other considerations like security and such; for the latter can be spotted by going through the code and testing it at a superficial level while the former takes way more than that.

A big part of testing is creativity and paying a great deal of attention to detail; the power of observation is paramount. Ed Catmull, a computer scientist and president of Pixar and Walt Disney Animation Studios, describes in his book about creativity how artists draw: They see things differently; while most minds have a tendency to jump to conclusions, artists have more honed observational skills that allow them to see more (Catmull and Wallace, 2014). Our brains have been programmed to jump to conclusions due to our evolved pattern recognition and taking cognitive shortcuts, which helped save our ancestors when facing a life-threatening danger like a saber-toothed tiger. When examining code to review or write test case for, unless one has a keen eye for details, the brain may glance over details as it recognizes common patterns. That's why off-by-one bugs and other simple coding mistakes are common; we simply glance over them as we build a mental model of the business logic and the brain illusively fills out the details. One way we find useful to combat that illusion is to go about reading the code in a way that requires more cognitive effort. Ideally, good code requires low-level of cognitive effort to read through a large number of lines fast and yet grasp what it does (as we discuss in Chap. 15). Just like art students, who are learning how to draw, are instructed to recreate a drawing of an object that's flipped upside down so that their brains don't take shortcuts in seeing the details of said object; we recommend reading the code in bottom-up approach,

instead of the traditional top-down approach (following the call graph from the entry point). One can take that one step further for complex functions and read the code lines in the opposite order (going upward instead of downward). In doing so, one gets to spot details that otherwise could have been missed because they are so consumed by the flow of program and other constructs. That said, this approach is insufficient on its own to test or review code; we recommend adding it to the existing approaches you already take to perform such tasks.

11.5.1 Planning and Equivalence Class Partitioning

A test design (also known as a test plan) should be devised in advance so that one, and preferably one's team, has a good idea of what scenarios are going to be tested and how, more importantly, it helps revealing what test cases are missing and which ones are unnecessary or duplicated. A good tactic to avoid duplicate tests is equivalence class partitioning (ECP), using which one groups test cases that are exercising the same code path or scenario together into partitions and then only pick a representative test for each partition. Typically, the number of unit tests in any codebase exceeds that of integration, system, acceptance, performance, load, and stress testing; hence, the time taken to run unit tests increases as more features are added to the product being tested. If left unchecked, duplicate or unnecessary unit tests would lead to longer time to run them, which in turn would hurt the team's productivity and cause them to skip testing altogether. Typically, when unit testing takes a long time to run, developers end up either skipping writing new tests (which can be circumvented by code coverage checks) or defer running unit tests to the continuous integration machines in order to discover test failures (usually regressions in a related code modules they didn't think of testing before committing the code). Moreover, when the code under test changes, as it's expected of business rules and requirements to keep changing, a fewer number of tests will require changing.

11.5.2 Code Coverage

Code coverage (also known as test coverage) is measuring the percentage of code lines that are exercised when tests run. There are different ways of measuring code coverage depending on what it pertains to: function coverage, statement coverage, branch (as in control structures like `if` conditions) coverage, and predicate coverage (i.e., has each boolean sub-expression been exercised?), and parameter-value coverage. Code coverage checks can be implemented as a gate (precondition) for committing code such that new code changes cannot reduce the existing code coverage or cannot drop below a certain threshold. While exemptions can be made for special cases like code that cannot be exercised in unit testing or

auto-generated code, the rest of the code should get executed while running unit tests. More importantly, code coverage shouldn't be a goal per se, as some tests can achieve 100% code coverage but verify nothing—rendering them almost useless; instead, the goal of measuring code coverage should be to inspect what hasn't been covered in unit testing and why—finding test coverage holes is the real goal.

11.5.3 Coding for Testability

Thinking about testability while designing and coding helps a great deal when the time comes to write unit tests. Many times one may find it hard to test a function due to its reliance on external dependencies like a database server, ever-changing variables like using the current time, error paths that are hard to exercise like handling an out-of-memory exception, etc. TDD solves this problem by forcing developers to think about testability first as new code can only be written as minimally as possible to make a failed test case pass; if there's no corresponding test case, such code shouldn't be written. For those who don't or can't afford to follow the practices of TDD, shortening the cycle of coding then testing can make such testability limitations obvious earlier in the software development process.

11.5.4 Mocking

One way to exercise more of the code under test and to isolate concerns is mocking, replacing external dependencies with ones that are injected at test time. For example, instead of setting up a database for unit testing, one should introduce the data access layer to the system as an interface that can be implemented in various ways, one of which connects to a real database while another could be an in-memory replacement for said database during unit testing—a mock. In object-oriented programming (OOP), mock objects mimic the behaviors of other objects (ones that are typically used in production) to facilitate testing; following the dependency inversion principle in OOP helps as it requires code to depend on abstractions (like interfaces) instead of concretions. In other words, following the example we stated above, a database would be a plug-in, which can be easily replaced with a mock object. Here's an example of mocking in Java:

```
import org.testng.annotations.Test;
import org.testng.Assert;
import org.testng.collections.Lists;

import java.util.ArrayList;
import java.util.List;
```

```
interface DataLayer {
  void insertReminder(final Reminder reminder);

  Reminder getReminder(final String reminderID);

  void updateReminder(
    final String reminderID, final Reminder reminder);

  void deleteReminder(final String reminderID);
}

class Reminder {
  // Reminder fields go here
}

class Orchestrator {
  private final DataLayer dataLayer;

  public Orchestrator(final DataLayer dataLayer) {
    this.dataLayer = dataLayer;
  }

  public void addReminder(final Reminder reminder) {
    // Validation and pre-processing code goes here
    dataLayer.insertReminder(reminder);
    // ...
  }
}

public class OrchestratorTest {
  @Test
  public void testAddReminder() throws Exception {
    final Reminder reminderToAdd = new Reminder();
    final List<String> methodsCalled = new ArrayList<>();
    final Orchestrator orchestrator =
      new Orchestrator(new DataLayer() {
      @Override
      public void insertReminder(final Reminder reminder) {
        methodsCalled.add("dataLayer.insertReminder");
        Assert.assertSame(
          reminder,
          reminderToAdd,
          "reminder mismatch in dataLayer.insertReminder");
      }

      @Override
      public Reminder getReminder(String reminderID) {
        Assert.fail("dataLayer.getReminder is not expected");
        return null;
      }

      @Override
      public void updateReminder(
```

```
          String reminderID, Reminder reminder) {
          Assert.fail("dataLayer.updateReminder is not expected");
        }

        @Override
        public void deleteReminder(String reminderID) {
          Assert.fail("dataLayer.deleteReminder is not expected");
        }
      });

      orchestrator.addReminder(reminderToAdd);

      Assert.assertEquals(
        methodsCalled,
        Lists.newArrayList("dataLayer.insertReminder"),
        "expected methods not called");
    }
  }
```

EasyMock

The above approach can get extremely verbose very quickly, that's why there are many libraries, like EasyMock and Mockito, that make mocking easier and succinct. Using said libraries, one only needs to mock methods that are relevant to the test case. Here's an example of mocking in Java using EasyMock:

```
import org.easymock.EasyMock;
import org.testng.annotations.Test;

interface DataLayer {
  void insertReminder(final Reminder reminder);

  Reminder getReminder(final String reminderID);

  void updateReminder(
    final String reminderID, final Reminder reminder);

  void deleteReminder(final String reminderID);
}

class Reminder {
  // Reminder fields go here
}

class Orchestrator {
  private final DataLayer dataLayer;

  public Orchestrator(final DataLayer dataLayer) {
    this.dataLayer = dataLayer;
  }
```

```
  public void addReminder(final Reminder reminder) {
    // Validation and pre-processing code goes here
    dataLayer.insertReminder(reminder);
    // ...
  }
}

public class OrchestratorTest {
  @Test
  public void testAddReminder() throws Exception {
    final Reminder reminderToAdd = new Reminder();
    final DataLayer dataLayerMock =
      EasyMock.createStrictMock(DataLayer.class);
    final Orchestrator orchestrator =
      new Orchestrator(dataLayerMock);

    dataLayerMock.insertReminder(reminderToAdd); // expect
    EasyMock.replay(dataLayerMock); // switch to replay state

    orchestrator.addReminder(reminderToAdd);
    // Assert the above mock conditions
    EasyMock.verify(dataLayerMock);
  }
}
```

EasyMock enables developers to specify various expectations like how many times a method is called, expected parameters (or matching criteria of said parameters), expected return values, exceptions thrown, call order, and many more advanced features; it also allows partial mocking of a real object. To learn more about EasyMock, please refer to http://easymock.org/user-guide.html.

11.5.5 Test Hooks

Test hooks expose certain details in the code under test that allow test cases to inject certain behaviors or test some internal details that couldn't be tested otherwise. For example, if we want to record the timestamp at which a reminder is inserted in the database, we need to control what value is sent to the database at the time of insertion. The below example shows an incomplete way of verifying the behavior:

```
import org.easymock.EasyMock;
import org.testng.annotations.Test;


interface DataLayer {
  void insertReminder(
    final Reminder reminder, long creationTimestampInMillis);
}
```

```
class Reminder {
  // Reminder fields go here
}

class Orchestrator {
  private final DataLayer dataLayer;

  public Orchestrator(final DataLayer dataLayer) {
    this.dataLayer = dataLayer;
  }

  public void addReminder(final Reminder reminder) {
    // Validation and pre-processing code goes here
    dataLayer.insertReminder(
      reminder,
      System.currentTimeMillis());
    // ...
  }
}

public class OrchestratorTest {
  @Test
  public void testAddReminder() throws Exception {
    final Reminder reminderToAdd = new Reminder();
    final DataLayer dataLayerMock =
      EasyMock.createStrictMock(DataLayer.class);
    final Orchestrator orchestrator =
      new Orchestrator(dataLayerMock);

    dataLayerMock.insertReminder(
      EasyMock.same(reminderToAdd), EasyMock.anyLong());
    EasyMock.replay(dataLayerMock);

    orchestrator.addReminder(reminderToAdd);
    EasyMock.verify(dataLayerMock);
  }
}
```

The above example ignores the verification of the timestamp parameter—that's equivalent to not testing that new behavior. To properly test the use of the timestamp, we follow the dependency inversion principle again here and a test hook to set an instance of the `Clock` abstract class we use for generating the creation timestamp:

```
import com.google.common.annotations.VisibleForTesting;
import org.easymock.EasyMock;
import org.testng.annotations.Test;

import java.time.Clock;
import java.time.Instant;
import java.time.ZoneId;

interface DataLayer {
  void insertReminder(
```

```
      final Reminder reminder,
      final long creationTimestampInMillis);
  }

  class Reminder {
    // Reminder fields go here
  }

  class Orchestrator {
    private final DataLayer dataLayer;
    private Clock clock = Clock.systemUTC(); // for testability

    public Orchestrator(final DataLayer dataLayer) {
      this.dataLayer = dataLayer;
    }

    @VisibleForTesting
    void setClock(final Clock clock) {
      this.clock = clock;
    }

    public void addReminder(final Reminder reminder) {
      // Validation and pre-processing code goes here
      dataLayer.insertReminder(reminder, this.clock.millis());
      // ...
    }
  }

  public class OrchestratorTest {
    @Test
    public void testAddReminder() throws Exception {
      final Reminder reminderToAdd = new Reminder();
      final DataLayer dataLayerMock =
        EasyMock.createStrictMock(DataLayer.class);
      final Orchestrator orchestrator =
        new Orchestrator(dataLayerMock);
      final long creationTimestampInMillis =
        System.currentTimeMillis();

      orchestrator.setClock(
        Clock.fixed(Instant.now(),
        ZoneId.systemDefault()));

      dataLayerMock.insertReminder(
        reminderToAdd,
        creationTimestampInMillis);
      EasyMock.replay(dataLayerMock);

      orchestrator.addReminder(reminderToAdd);
      EasyMock.verify(dataLayerMock);
    }
  }
```

@VisibleForTestability Annotation

The `@VisibleForTestability` annotation, from Google's Guava library, indicates that the `setClock` method has been introduced to make the code more testable; in other cases, the annotation may be used to indicate that a program element is more widely visible than otherwise necessary, only for use in test code. It's recommended to limit access to program elements to the most restrictive level; in Java, a test class can belong to the same package as the class under test even though their respective code files reside in different folders. Hence, a package-private access level is recommended for nonpublic program elements under test—same-package test cases are able to access those elements. This approach is more stable and intentional than accessing private program elements via reflection.

Dealing with Randomness

The use of random number generators is fairly common in big data systems and in almost any nontrivial application; testing random behavior can be challenging. Here's an example of a Java unit test that verifies the behavior of a coin-flipping Bernoulli trial:

```
import org.testng.annotations.Test;

import java.util.Random;

import static org.testng.Assert.*;

class CoinFlipper {
  public enum Face {HEAD, TAIL}

  private final Random random = new Random();

  public Face flip() {
    return random.nextBoolean() ? Face.HEAD : Face.TAIL;
  }
}

public class CoinFlipperTest {
  @Test
  public void testFlip() {
    final CoinFlipper flipper = new CoinFlipper();
    final int trialsCount = 100;
    int headsCount = 0;

    for (int i = 0; i < trialsCount; i++) {
      if (flipper.flip() == CoinFlipper.Face.HEAD) {
        ++headsCount;
      }
    }
```

```
    assertTrue(Math.abs(headsCount - trialsCount / 2) <= 5);
  }
}
```

There are a few things to note about the above example; the most important of which is that the test case focused on testing the randomness of the `flip` method and, in doing so, lacked in testing its business logic. If you're using a well-tested random number generator, there's no need to retest it; you only need to make sure you're using it correctly. Another shortcoming to note is the absence of testing the randomness—or more accurately, the pseudo-randomness—of coin flips as a sequence of outcomes (rather than verifying the tally).

We should focus on testing the business logic of a coin flipper and how it uses the random number generator. To do so, we can set the seed of the random number generator so that each time the test case runs it's predictable instead of pseudo-random; here's an example:

```
import com.google.common.annotations.VisibleForTesting;
import org.testng.annotations.Test;

import java.util.Random;

import static org.testng.Assert.*;

class CoinFlipper {
  public enum Face {HEAD, TAIL}

  private Random random = new Random();

  public Face flip() {
    return random.nextBoolean() ? Face.HEAD : Face.TAIL;
  }

  @VisibleForTesting
  void setRandom(final Random random) {
    this.random = random;
  }
}

public class CoinFlipperTest {
  @Test
  public void testFlip() {
    final CoinFlipper flipper = new CoinFlipper();
    final Random random = new Random(42);

    flipper.setRandom(random);
    assertEquals(flipper.flip(), CoinFlipper.Face.HEAD);
    assertEquals(flipper.flip(), CoinFlipper.Face.TAIL);
    assertEquals(flipper.flip(), CoinFlipper.Face.HEAD);
  }
}
```

Another alternative would be to wrap the random number generator in another class that can be mocked and set the behavior of the mock so that it's predictable.

11.5.6 Test Case Anatomy

The most common composition of a test case consists of the following stages: setting up the preconditions, exercise of code under test, verification of results and post-conditions, and cleanup of shared state. In setup, one sets the stage for the preconditions requisite for the test scenario; for example, setting the state of an object to a certain value to guide the code through the path under test. In our key-value store test example, we can see the anatomy of the unit test as the following:

```
import unittest

class TestKeyValueStore(unittest.TestCase):
  ''' A unit test for KeyValueStore '''
  def test_put_key_already_exists(self):
    ''' Tests put when key already exists '''

    # Set up
    store = KeyValueStore()
    store.put('same key twice', 'first time')

    # Execute
    store.put('same key twice', 'second time')

    # Verify
    self.assertEqual(store.get('same key twice'), 'second time')
    self.assertEqual(len(store._store), 1)

    # No shared state between tests => no cleanup required


if __name__ == '__main__':
  unittest.main()
```

A clearer to indicate setup and cleanup is using the `setUp` and `tearDown` methods, respectively. Here's an example that makes use of the `setUp` method to create a new instance of the key-value store in a clean state for each test case:

```
class TestKeyValueStore(unittest.TestCase):
  ''' A unit test for KeyValueStore '''
  def setUp(self):
    ''' Runs before each test method '''
    self.store = KeyValueStore()

  def test_init(self):
    ''' Tests __init__ '''
    self.assertEqual(self.store.get('key'), None)
```

```
    self.assertEqual(len(self.store._store), 0)

  def test_put_with_value(self):
    ''' Tests put '''
    self.store.put('key', 'value')
    self.assertEqual(self.store.get('key'), 'value')
    self.assertEqual(len(self.store._store), 1)

  def test_put_with_none_value(self):
    ''' Tests put with none value '''
    self.store.put('key', None)
    self.assertEqual(self.store.get('key'), None)
    self.assertEqual(len(self.store._store), 1)
```

Almost every single unit test framework supports the functionality outlined in the above example. Setup code should be purposeful; although it may be tempting to group all setup steps for all scenarios together, it's better—for debugging and separation of concerns—to only set up the minimum required preconditions for each test separately.

Another tempting shortcut to take is testing multiple scenarios in the same test method; it's harder to track what failed when a failure occurs. When using a test harness—as one should—to track the history and meta-data of tests and their runs, it makes sense to keep each test's scope as immutable as possible.

If we had to choose a few attributes that make unit testing effective, we'd have to go with choosing the minimum number of tests and lines of code that verify all testable scenarios with the intention of breaking them. Other properties that also matter include the time taken to run unit tests, which should be really fast; in addition to the ability to run them anywhere—as they don't have any external dependencies—so that developers can run them on their laptops while commuting to work if they wanted to. Moreover, test cases should be placed as close as possible to the code being tested; some languages are opinionated about that (e.g., Go) while others allow developers to place tests wherever they want (e.g., C#). We believe that tests double as a contract of what the desired behavior should be; so the next time you wonder how to comment on the expected behavior of a unit of code, think about writing a test case to enforce said behavior instead. Finally, test code should be treated with the same care and attention to details as production code.

11.5.7 Smoke Testing

Also known as sanity checks or build-verification testing, smoke testing is there to find egregious bugs as early as possible. The etymology of the term is attributed to tests performed by plumbers to find leaks in pipes, a practice that goes back to the nineteenth century. Smoke is pushed, using a bit of pressure, through a system of connected pipes so that plumbers can look for smoke where leaks and cracks could be. As opposed to finding leaks or cracks later after liquids start running through the

pipes, testing with smoke causes no damage to the building where the pipes run—early discovery of such issues leads to containing the damages and a lower cost of fixing. Similar, in software testing, it's strongly encouraged to pick a set of key test scenarios to use as a smoke test for new changes; typically, that's done with every build and are run on a build machine as part of continuous integration. Smoke tests need to be run very frequently, which in turn require them to be fast in order to avoid being a time sink.

11.5.8 Happy-Path Testing

Also known as positive test cases, happy-path test cases verify the behavior of code when everything work as expected—no errors, no exceptions. Typically, this kind of testing is the easiest to perform and requires little effort to design. Simply covering the various code paths that don't lead to an exception or an error should suffice to test the positive test cases as long as the results and invariants are verified.

11.5.9 Data-Driven Testing

Data-driven testing (DDT) facilitates running multiple test scenarios that share a common structure, using a single test case. Various aspects of test setup like the inputs and their respective expected outputs are loaded as data and run through a test case. Such data may be loaded from a table (e.g., array) in memory, a test resource (e.g., a JSON file), etc. Here's an example of table-driven testing in Java using TestNG:

```
import org.testng.annotations.DataProvider;
import org.testng.annotations.Test;

import static org.testng.Assert.*;

class Adder {
  public int add(int x, int y) {
    return x + y;
  }
}

public class AdderTest {
  @DataProvider(name = "testAdd")
  private static Object[][] provideTestData() {
    return new Integer[][]{
      new Integer[]{0, 0, 0},
      new Integer[]{2, 3, 5},
      new Integer[]{-3, 5, 2},
      new Integer[]{-5, -3, -8},
```

```
        new Integer[]{-5, 3, -2},
        new Integer[]{0, Integer.MAX_VALUE, Integer.MAX_VALUE},
        new Integer[]{0, Integer.MIN_VALUE, Integer.MIN_VALUE},
        // ...
      };
    }

    @Test(dataProvider = "testAdd")
    public void testAdd(int x, int y, int expected) {
      final int z = new Adder().add(x, y);
      assertEquals(z, expected, "failed for " + x + " and " + y);
    }
  }
```

At Voicera, we use DDT extensively as they are commonly used in Go, our main programming language; we open-sourced a test library that makes DDT even easier for large tests as it can load the data from a JSON file and provides some other features that make testing in Go more consistent. The source code for that library can be found at https://github.com/voicera/tester.

11.5.10 Fuzzing

Also known as fuzz testing is a technique used to automate test cases so that arbitrary inputs are provided to the code under test. The goal of fuzzing is to explore code paths and corner cases that other tests missed. When the fuzz tests run, they verify that the code under test doesn't violate any of the invariants specified; for example, the program shouldn't crash, fail any assertions, or leak memory. For more details on fuzzing, the following article is a good read: https://www.ibm.com/developerworks/library/j-fuzztest/index.html.

11.6 Integration Testing

When multiple modules are integrated together, said integration need to be tested for fear that their interactions may cause issues. Since each module is verified in isolation, integration tests focus on the glue code that puts them all together. Integration testing also verifies the behavior of the integrated modules as a group.

11.7 System Testing

System testing covers many aspects of verifying that the system meets its specifications; it covers areas like usability, performance, load, stress, comparability, accessibility, etc. We cover below three of the above areas.

11.7.1 Performance Testing

To measure performance is to measure how long it takes the system to perform a transaction or an operation; a very common example is to measure how long it takes to load a webpage. System designers should set performance goals that take into account factors like external dependencies. When measuring the time it takes to load a webpage, location matters: Users served by nearby servers typically observe better performance (when communicating with said servers) than users who are farther away. Another factor to consider is caching (whether server-side, client-side, or on a content delivery network). That's why measuring performance properly takes into consideration location; performance should be measured at multiple locations that resemble where users reside and at various locations in relation to the servers: first-mile (close to the server), mid-mile (somewhere in between), and last-mile (close to the client) locations. Based on the business requirements of an application, the performance of first-time requests (without caching) may be important to verify; otherwise, performance testing should focus on typical use cases during which caching is utilized. In fact, most performance testing requirements come from business requirements; for example, what's acceptable for a news webpage is not for a trading system. That's why the NYSE offers co-location services to companies that host their trading systems in its data centers to minimize network latency—every microsecond counts. Another requirement to consider is what percentage of requests need to meet the ideal performance requirements; outliers are a fact of life and need to be considered in performance testing. Depending on the nature of the application under test, this number can vary from 90%, 95%, 99%, or higher. Each scenario and each class of operations may have different performance requirements—not all operations are created equal.

Instrumentation and Profiling In order to measure how long an operation takes, one may go about this as a task external to the system; for example, writing code that loads a webpage and measuring how long that operation takes, performing the operation under test to meet the specifications of the test, which may include warming up the cache then running the operation hundreds of times to collect statistics like the 95th percentile, etc. Assuming that the test shows that performance issues exist, what's next? The answer is debugging. A common practice used in performance testing is instrumenting the code under test so that it can be easily debugged if need be; instrumentation injects code statements, without changing the source code, that allow profiling tools to show a breakdown of how much time is

spent in each code unit (e.g., methods) and how many times each was called. This kind of debugging helps developers find "hot spots," or places in the code where performance deteriorates.

11.7.2 Load Testing

Load testing borrows its name from mechanical engineering; physical load testing checks the ability of machines (e.g., engines) to withstand various levels of load which it's built to handle (and beyond that in some cases). In software testing, load testing subjects the system under test to various levels of load as well to check it can take them and still operate correctly and within the required performance SLA. Load testing is usually related to concurrency, reliability, and throughput.

11.7.3 Stress Testing

Like load testing, stress testing borrows its name from mechanical engineering as well; unlike load testing, it seeks to test the system at its breaking point, which is typically at an abnormally high load level. Sometimes the goal of stress testing is finding that breaking point and/or how errors cascade from one area in the system to the other; finding where how the system cracks from stress can help fix its bottlenecks—an exercise that can be performed after load testing as well. The line between load and stress testing is usually blurred as one can turn into the other depending on the test scenario.

11.8 Acceptance Testing

Acceptance testing is more concerned with the user than any other kind of testing; it's also known as user acceptance testing (UAT) and end-user testing. In agile software development, and particularly extreme programming, UAT is used to verify the completion of user story according to the specified requirements. Acceptance testing doesn't concern itself with system-wide issues like crashes or unit-specific issues like the wrong exception type getting thrown; instead, it focuses on verifying real-world end-to-end scenarios.

11.9 Real-User Testing

What better way to test software than releasing it and getting real feedback from real users, right? You may think that real-user testing as we just defined it goes against everything we've been warning about earlier in this chapter. You thought right: The cost of finding issues after releasing software is much higher than finding it earlier in the software's lifecycle. That said, there remains a class of issues that is hard to test and is easier to find, thanks to real-world usage. Trade-offs in software are a fact of life; some products require much faster speed of iteration and can tolerate some minor issues to creep into production to be discovered by a test set of users; in which case, it's acceptable—and sometimes encouraged—to release the new changes to a small set of users to get their feedback either quantitatively (using A/B tests and metric tracking) and/or qualitatively (by soliciting their feedback). In A/B testing, a set of users get the new version of the software (treatment group) while the rest stay at the current stable version (control). When the treatment group is the organization that built the software, such experiment is called "dogfooding": eating your own dog food before serving it. At companies like Microsoft, LinkedIn, and Facebook, employees use alpha and beta versions of software they produce in real-world usage to test it (in staging and/or production environments). At Voicera, new features ship as experiments that can be configured to specify control and treatment groups. Facebook rolls out new features first to users in New Zealand, which share similar demographics to users in the US and the UK but still fairly isolated from the rest of the world; it's also a strongly connected clique of a few millions of users. At LinkedIn, code changes and deployments to production are frequent to increase the speed of iteration. Thanks to an A/B test framework, known internally as LiX,[2] hundreds of new experiments are ramped daily.

11.9.1 Canary Deployments

Mining coal used to be an extremely dangerous job; historically, deaths due to inhaling carbon monoxide, among other toxic gases commonly found in many coal mines, were hard to prevent because of how hard it was to discover. Carbon monoxide is odorless, colorless, and deadly; because it's heavier than air, it would find its way to miners who were deep underground—too deep to get out in time. In the early twentieth century, a Scottish physiologist by the name of John Scott Haldane proposed the idea of using canaries in coal mines as sentinel animals that help humans find early warning signs of risks to their health. Coal miners used to bring along canaries in cages into coal mines; if a toxic gas (like carbon monoxide) was at a concentration high enough to harm the miners, the birds would become

[2] https://engineering.linkedin.com/blog/2015/11/monitoring-the-pulse-of-linkedin.

sick long enough before the miners due to their significantly smaller size; in doing so, the canaries warn the miners to the imminent danger and give them a chance to react.

A canary deployment is similar to the canary in a coal mine: New changes are deployed to a very small number of machines in production (typically one in each unique configuration) and observed carefully before releasing the changes to the rest of the machines in production. Because the new changes are deployed to a smaller group of machines, bugs are contained to said group and the percentage of operations they perform; it's a fair cost to pay for discovering issues early on in production. The premise for real-user testing is pragmatism. It's virtually impossible to find all functional and performance issues before shipping to production because the vast majority of organizations cannot afford to reproduce every single production scenario in test environments perfectly. There will always remain a class of issues that's prohibitively expensive to find in preproduction environments; for example, configuration bugs where a system is configured to use certain configurations that are unique to the production scenario (like a connection string). That said, one should always strive to maximize the quality of software prior to releasing it; issues covered in production should be extremely hard to find otherwise, and even then, one should conduct a postmortem to figure out how to mitigate such issues in the future.

Using a canary deployment allows engineers to inspect the canary machines carefully for early signs of failure. Thus, it's important to figure out in advance what warning signs (metrics) to look for when observing a canary. For coal miners, they were signs of sickness or distress; for big data systems, there are KPIs and system metrics to watch for, like performance, throughput, memory and CPU usage, etc.

Even though one cannot emphasize enough that testing in preproduction environments should be very thorough to flush out any egregious issues before shipping a canary into the coal mine, mistakes will be made and bugs will make their way to the canary machines in production. In certain situations, such risk cannot be tolerated; for those scenarios, one approach to go about that is to fork real production data (e.g., web requests) to two streams: one that goes to machines that run the stable version of software and the other to canary machines, which are not allowed to make persistent changes or respond back; we will still be monitoring the same metrics we watch for in a normal canary deployment. In doing so, we can test production behaviors and scenarios while minimizing the risk of the new changes being destructive.

Other types of staged deployments can be implemented as well using the same line of thinking; instead of a 2-stage deployment (canary then all machines), one may pick another strategy that makes more sense for the application being deployed. For example, an application may be deployed on canary machines first, then on 20% of the machines, and if the metrics still look good, it gets deployed on the rest. Automated deployments should take care of each step and rolling back when the health criteria are not met.

11.10 Notes

If you don't test your code, your customers will test it for you; that's why so many books were written about the subject of software testing, of which we highly recommend: Chapters 9 and 15 of (Martin, 2009), Chapter 22 of (McConnell, 2009), and (Kaner et al., 2011).

References

D. Ariely. *Predictably Irrational, Revised and Expanded Edition: The Hidden Forces That Shape Our Decisions*. Business & economics. HarperCollins, 2010. ISBN 9780061353246.

E. Catmull and A. Wallace. *Creativity, Inc.: Overcoming the Unseen Forces That Stand in the Way of True Inspiration*. Random House Publishing Group, 2014. ISBN 9780679644507.

R.C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Robert C. Martin series. Prentice Hall, 2009. ISBN 9780132350884.

S. McConnell. *Code Complete*. DV-Professional. Microsoft Press, 2009. ISBN 9780735636972.

C. Kaner, J. Bach, and B. Pettichord. *Lessons Learned in Software Testing: A Context-Driven Approach*. Wiley, 2011. ISBN 9781118080559.

Chapter 12
A Few More Things About Programming

In this chapter, we explore a few more things about programming; specifically, tools used to write, store (version control), build, debug, and document code. In addition, we explore miscellaneous topics like exceptions.

12.1 Notebooks

Live code in the browser is probably one of the most widely used techniques for developing software for data science these days. Notebooks, like Jupyter, allow creation and sharing of live code, documentation, visualization, etc. Many services support Jupyter for various reasons and use cases: Google's Colaboratory allows developers to share Jupyter notebooks and collaborate just like they do using Google docs and sheets, while Amazon's SageMaker supports Jupyter to facilitate building, training, and deploying machine learning models at scale.

Jupyter supports over 40 programming languages, including Python, R, Julia, Scala, and even C++; it also supports frameworks and libraries like Apache Spark, pandas, scikit-learn, `ggplot2`, and TensorFlow. To get started with Jupyter, it's as simple as running the following commands (Fig. 12.1):

```
pip install jupyter
jupyter notebook
```

12.2 Version Control

Version control systems manage different versions of the software and enable switching back and forth between different versions. This is very useful since most development processes are composed of many small incremental steps rather than

© Springer Nature Switzerland AG 2018

G. Lebanon, M. El-Geish, *Computing with Data*,
https://doi.org/10.1007/978-3-319-98149-9_12

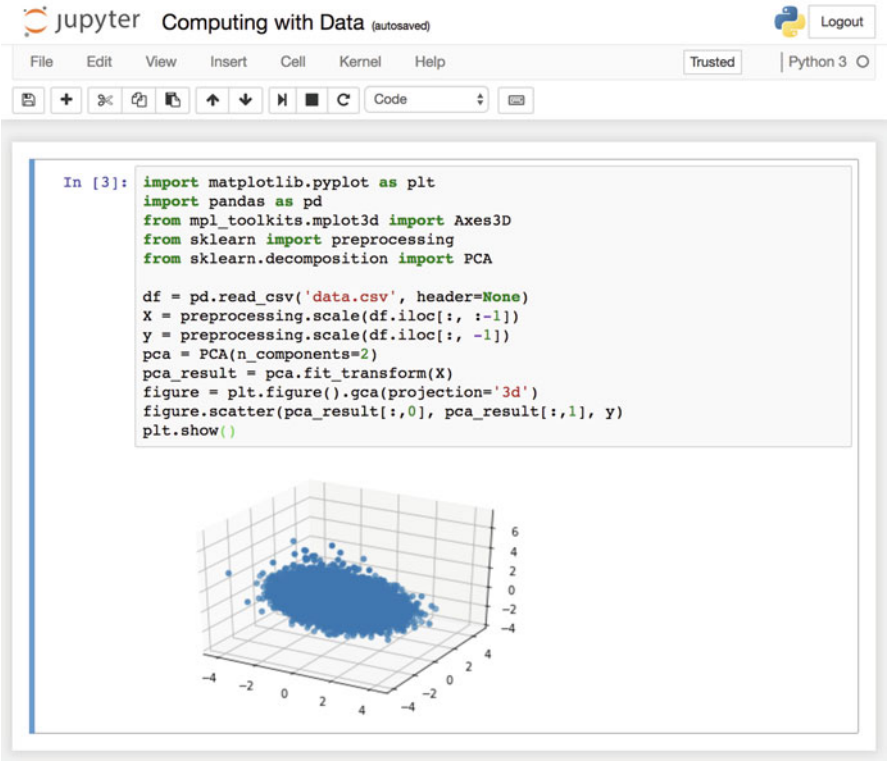


Fig. 12.1 An example Jupyter notebook with live Python code and a 3D plot

one single monolithic and immediate contribution. For example, if a problem is discovered in one of the recent version the version control system can revert back to an earlier version without the problem right away. After the revert step the problem can be studied and fixed with less urgency.

Conceptually, each version can be thought of as a node in a graph with outgoing edges to another node that represents the previous version. Version control systems typically store only the differences between subsequent nodes in order to reduce storage space and avoid unnecessary redundancy. Each version or node is annotated with the author, the date and time the version was committed to the system, and an explanation of the changes.

Version control is useful also for a single developer, but it is extremely useful when several developers simultaneously collaborate on a codebase. Developers working on different projects can work on a part of the system by branching off from the current version, creating a version dedicated to their project that does not impact other versions. When the project work is done the separate branch can be merged back into the main branch that represents the current version of the stable system. Merging one branch into another is done automatically by the version

control system, unless there are conflicts that it cannot resolve (in which case the developer needs to resolve the merge conflicts manually).

There are two types of version control systems: centralized and distributed. Centralized version control systems such as CVS, Subversion, and Perforce have a single server that contains all the repository information, and several clients that check out files from the centralized repository. In order to access the version control system on the centralized repository, such systems need to communicate over the network.

In distributed version control systems (such as Git or Mercurial), clients do not simply check out the latest snapshot of the files. Instead, the clients mirror the entire version control repository including the complete version history. Distributed version control systems have several advantages over centralized version control system: (a) each client stores a complete clone of the repository, which is useful for backup purposes, (b) much of the version control functionality can be done without network connectivity, and (c) many operations are much faster since they can be carried out locally.

In some cases, distributed version control system do have a single unique server that is designated as the primary server and that can be trusted to have the latest version. New clients typically copy the repository from that designated server, but as described above all clients hold a copy of the entire version history.

We start below with a description of Git, a popular distributed version control system. It is also open source, which makes it attractive for low cost companies or academic researchers and forms the base for GitHub, a popular version control system hosted in the cloud that has some additional functionality. We conclude with a description of Subversion, an older centralized version control system that is still in wide use.

12.2.1 Git

Git was created by Linus Torvalds (the inventor of Linux) in 2005 and has since then increased in popularity to the point where it is today perhaps the most popular version control system. Many major companies such as Google, Microsoft, Twitter, and LinkedIn use Git, as well as many small companies and start-ups

We start below by covering local Git functionality, where the repository exists on a single machine. Afterwards, we cover the case where multiple clients (or a client and a server) have copies of the same repository.

Basic Functionality of Local Git

A Git repository is a collection of files in a directory that contains a `.git` subdirectory. The files represent the current working version and the `.git` subdirectory

contains additional information that allows tracking different versions and switching between them.

After installing Git (if it is not already installed on your computer by default), the developer needs to enter his or her name and email address. These details will be recorded as part of the version log messages and are very useful when collaborating with other developers. The name and email address can be entered in the shell as follows.

```
git config --global user.name "John Smith"
git config --global user.email "john@example.com"
```

Git occasionally uses an editor to prompt the developer to enter a message describing the new version. To inform Git about your favorite editor set the `GIT_EDITOR` shell environment variable. For example, in bash or zshell we can set the `GIT_EDITOR` to the Emacs editor using the following shell command `export GIT_EDITOR="emacs"`. It is useful to add such a line in your shell initialization file (`.zshrc` for zshell and `.bashrc` in bash).

Typically, a Git repository is created by cloning an existing repository from a different computer. But to start an empty repository from scratch use the following command sequence.

```
# change to parent directory where repository should lie
cd dir_name
# create a new directory representing repository
git init repository_name
# enter directory representng new repository
cd repository_name
```

The new directory will have a `.git` subdirectory that stores the version control information.

To add files to the repository, create a file in the repository directory and then type `git add file_name`. At that point the file is added to a logical version of the repository called "the staging area." The added files that are currently in the staging area are not yet added to the repository. To update the repository with the new files we need to type `git commit`. Using the sequence `git add X` and `git commit` we can add multiple files and directories to the repository. See below for a short example.

```
cd dir_name
git init repository_name
cd repository_name
touch new_file  # create an empty file
git add new_file  # add new file to staging area
git commit -m "demo commit: adding an empty file new_file"

## [master (root-commit) 42a72e0] demo
## 1 file changed, 0 insertions(+), 0 deletions(-)
## create mode 100644 a
```

The message flag `-m` followed by the comment describing the version may be omitted, in which case Git will open up an editor and prompt you to enter the commit message in it (see above on how to inform Git on which editor to use). Git will execute the commit with the edited message upon saving and exiting the editor.

Files in the Git directory that are part of the index or repository are called "tracked files." Other files are considered "untracked files" and unless explicitly added to the repository using `git add` will not be added to it. The command `git status` shows the list of tracked and untracked files and the current status of the staging area.

```
git status

## On branch master
## nothing to commit, working directory clean
```

The command `git commit` can also update the repository with new versions of existing files. Specifically, after editing the repository files we can update the repository (in the `.git` subdirectory) with the new version by `git commit file_name`. See below for a short example.

```
cd dir_name
git init repository_name
cd repository_name
touch new_file  # create an empty file
git add new_file  # add new file to staging area
# commit addition
git commit -m "demo commit: adding an empty file new_file"
echo 123456 >>! new_file  # add some text to empty file
# commit new version of new_file
git commit -m "added 123456 to file" new_file

## [master 9110e49] added 123456 to file
## 1 file changed, 1 insertion(+)
```

The command `git commit -a` commit changes to all files that are tracked by the repository and have changed since the last commit.

```
git commit -a -m "updating multiple files"
```

The command `git reset -hard` resets the working directory to the last previous commit. This is useful if you made some changes that you regret but did not commit them yet.

Branches and Merges

We considered above a linear development process where each commit updates the previous commit. Using the analogy of a directed graph where nodes are commits and outgoing edges connect commits to previous commits we have a linear graph

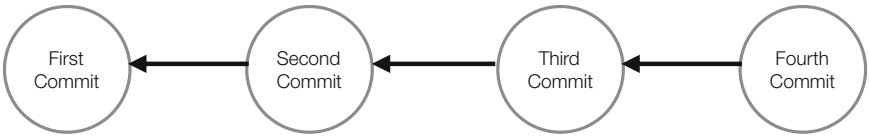


Fig. 12.2 Linear development pattern in a version control system. Each node corresponds to a commit and outgoing edges point from a commit to its preceding commit

leading from the most recent commit back in time to previous commits until we get to the first commit. Figure 12.2 visualizes this development pattern.

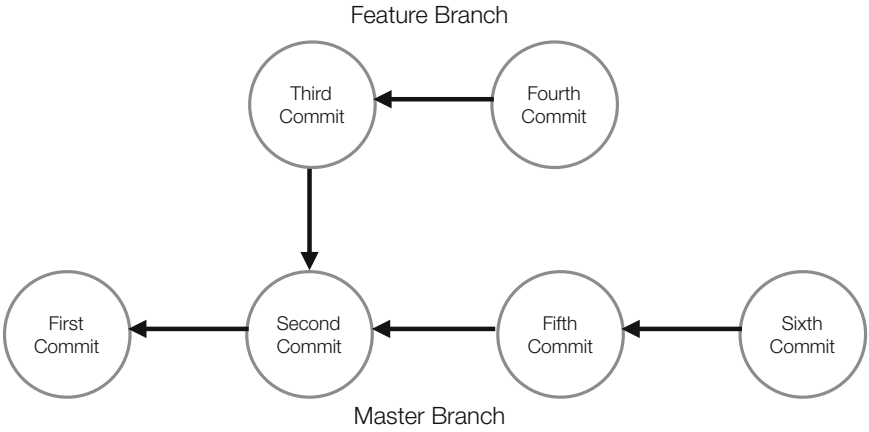


Fig. 12.3 Branch development pattern in a version control system. The bottom row represents the master (main) branch consisting, for example, of minor bug fixes. After the second commit, development of the new feature commenced with two subsequent commits. Then, a minor bug was discovered and a fix was deployed to the master branch rather than the feature branch since the feature work was not completed. That commit was followed by another minor bug fix resulting in the sixth commit in the master branch. At that point we have two branches that hold different versions of the system

In some cases it is desirable to branch out from the main line of development (usually called master) and create a new line of commits in parallel, for example when developing a new feature in a system. For example, the master branch can represent the stable system and subsequent commits in master represent minor bug fixes. The secondary branch may represent work on a new feature that is added to the system and its line of commits represent work on that project. When the new feature is complete the branch can be merged back into the master branch. Figure 12.3 illustrates such a development pattern involving two branches. Keeping isolated lines of development out of the main branch prevents incorporating unstable code to the main branch.

The command `git branch branch_name` creates a branch without switching to it. The command `git branch` lists all branches. To switch to a different branch type `git checkout branch_name`. Any subsequent commits will update the new branch. The initial branch is called `master`.

```
git branch new_branch  # create a new branch
git branch  # list all branches

## * master
##   new_branch
git checkout new_branch  # move to new_branch
git branch  # list all branches

##   master
## * new_branch
```

Referring to specific commits in Git is usually done via refs, which are references to commits. The ref `HEAD` refers to the most recent commit on the current branch (it updates automatically). Branch names are also refs that update automatically to refer to the most recent commit in the corresponding branch. If the current checked out branch is `master`, then the two refs `HEAD` and `master` refer to the same commit.

Appending a branch name (including `master`) with `~k` creates a ref to the commit that is `k` commits before the commit at the top of the branch. Appending to the branch name `^` corresponds to `~1` and `^^` corresponds to `~2` (for example, `master^^` is equivalent to `master~2`).

Commits can also be identified by refs generated automatically using a secure hash algorithm (SHA-1).[1] To see the SHA-1 codename that correspond to one of the refs above type `git rev-parse ref_name` where `ref_name` represents the ref. The 40 character SHA-1 refs can also be replaced by shortened SHA-1 versions.

```
git rev-parse master~2 # display SHA-1 codename
git rev-parse --short master~2 # display shortened SHA-1
    codename
```

Another possibility to refer to commits is using tags. We can create a user specific tag to annotate the current commit using the command `git tag tag_name`.

To replace the current version with an alternative version, run `git checkout ref_name`. This can replace the current version with an older or a newer version on the same branch or on a different branch depending on the specified ref. The example below adds text to a new file, adds some more text, and then reverts to the previous version.

[1] SHA-1 algorithm is using hashing to assign codenames to commits rather than sequential numbers due to the distributed nature of Git (when you commit a change, Git doesn't know what other commits occurred on other clients. The codenames are displayed as 40 character hexadecimal strings.

```
git init demo_repos
cd demo_repos
echo 123 > demo_file
git add demo_file
git commit -a -m "adding demo_file containing 123"
echo 456 >>! demo_file
git commit -a -m "adding 456 to demo_file"
cat demo_file

## 123
## 456
git checkout master~1  # change version to previous version
cat demo_file

## 123
```

We can also revert a bad commit using the command `git revert ref_name`.

At some point we may want to merge one branch into another. For example, in the situation described in Fig. 12.3 we may wish to merge the feature branch into the master branch after the feature development work is finished. The command `git merge branch_name` merges `branch_name` into the current branch.

```
git init demo_repos
cd demo_repos
echo 123 > master_file
git add master_file
git commit -a -m "added master file with 123"

## [master (root-commit) cb133d5] added master file with 123
##  1 file changed, 1 insertion(+)
##  create mode 100644 master_file
git branch feature1
git checkout feature1

## Switched to branch 'feature1'
echo abc > feature_file
git add feature_file
git commit -a -m "added feature_file with abc"

## [feature1 5c82dfc] added feature_file with abc
## 1 file changed, 1 insertion(+)
#3 create mode 100644 feature_file
git checkout master

## Switched to branch 'master'
git merge feature1  # merge feature1 branch into master

## Updating cb133d5..5c82dfc
## Fast-forward
## feature_file | 1 +
## 1 file changed, 1 insertion(+)
## create mode 100644 feature_file
```

```
ls  # both files appear in master after the merge
## feature_file master_file
```

A merge creates a new commit that has two outgoing edges into the two merged commits (see Fig. 12.4). After the merge, we can keep the merged branch alive and add more commits to it (possibly to be merged again later), or we may delete the branch that was merged and continue development from the current branch (for example, `master`) and consider creating a new branch in the future if needed. The command `git branch -d branch_name` deletes a branch (use `-D` to delete a branch with unmerged changes).

Rebasing is a variation of merging that converts the parallel branches pattern in Fig. 12.3 to a linear pattern with the branch that is rebased becoming a linear continuation of the current branch. To rebase a branch into the current branch use the command `git rebase branch_name`. Figure 12.5 illustrates this process. Merge and rebase have different pros and cons and sometimes different organizations use one alternative over the other.

```
git checkout feature1
git rebase master  # rebase master into feature1 branch
```

Conflicts

When a merge is executed changes in the merged branch are incorporated into the current branch. If there are no conflicts (as in the example above where the work in the feature branch affects files that are different from the files that are modified in the master branch) the merge proceeds automatically. But in some cases there

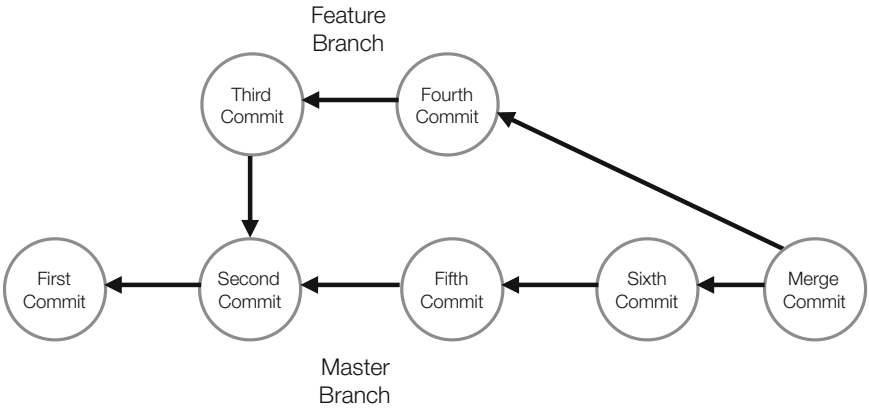


Fig. 12.4 A feature development branch merged into the master branch

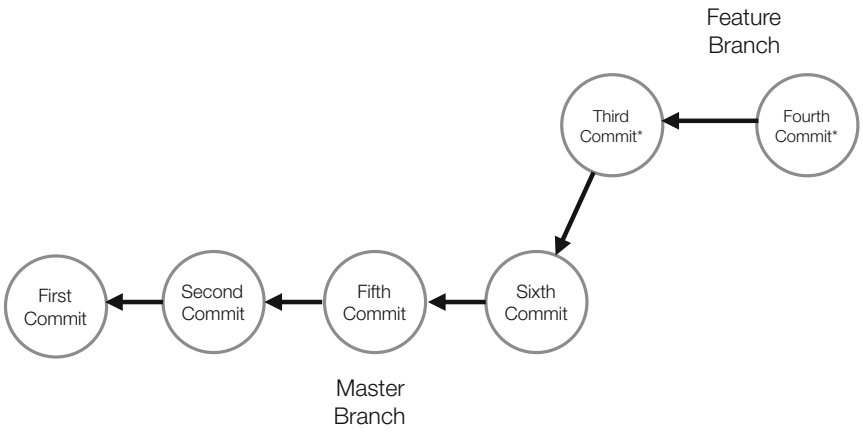


Fig. 12.5 The master branch rebased into the feature development branch

are conflicts that Git cannot resolve, for example if both commits modify the same line in the same file in different ways. In this case, conflicted merge modifies the conflicted files by inserting markers where there are conflicts.

Below is a portion of a file that underwent a merge with a conflict. Git edits the file and adds markers showing the conflict. It takes some experience to read the markers below, but basically content following «««« marker corresponds to one version and content following ======= corresponds to the other version. The marker »»»» specifies the end of the conflict region.

```
this part is in agreement between the two branches
  - no conflict exists

<<<<<<< HEAD
This is an edit in the incoming branch that conflicts
  with the existing branch

=======
...
This is the edit in the current branch that conflicts
   with the incoming branch
...
>>>>>> current-branch
...
this part is in agreement between the two branches
   - no conflict exists
...
```

There are two basic ways of helping Git in resolving merge conflict. The first is to edit the files with the markers above, remove the markers, and make sure

the files contain the appropriately merged version. After the conflicts are resolved manually by editing, we need to perform a `git add file_name` and `git commit` commit to inform Git that we resolved the conflict. The second way to resolve conflicts is to use a graphic merge tool by running the command `git mergetool`. This command starts a graphic merge tool such as `opendiff` or `kdiff3`. The precise tool depends on the operating system and `git mergetool` needs to be configured appropriately first (see `git help mergetool` for help on configuring the mergetool).

Remote Git

In many cases developers need to work with Git repositories on other servers. In this case, the first stage is to clone the repository on a local computer using the `git clone` command where `repository_name` has a pattern such as `ssh://username@address.com/path/projectName.git`. For example the following command creates a cloned repository of the `projectName.git` repository in the local `projectName` directory. The repository that was cloned is referred to as `origin` and the cloned repository is referred to as `remote`.

```
git clone ssh://username@address.com/path/projectName.git
```

At this point the developer can move into the newly created directory and start editing files, adding files, and committing as outlined in the previous section. One major difference is that the commit will be stored in the local repository (`remote`) index rather than the repository that was cloned (`origin`). To propagate the commits to `origin` we need to execute the `git push` command.

The `git pull` command contacts `origin` and checks for recent updates. If there are updates on `origin`, they are downloaded to the local directory and merged with the working version (if conflicts occur they need to be resolved manually as described above).

Additional Features

We describe below a few commonly used additional features of Git.

We saw above how to add files to a Git repository. Git provides additional file system functionality such as `git mv file1 file2` for moving or renaming a file and `git rm file` for removing a file. As in `git add` these commands also require a subsequent `git commit` command to update the repository with the change.

The command `git log` prints the repository history: the commits annotated with their date, author, files that were changed, and commit message. Note below the SHA-1 codes displaying the most recent commits and the commit messages.

```
git log | head
```

The command `git diff ref1 ref2` displays the difference between two references. For example, the command below prints the diff between the commit on top of `master` and the one before it.

```
git diff master~1 master | head
```

The format of the diff printout takes some getting used to. In general lines that start with a minus sign (−) represent deleted lines and lines that start with a plus sign (+) represent newly added lines.

There are various tools that visualize the Git repository history as a directed graph, such as `gtk+` (Linux Gnome) and `gitx` (Mac OS).

An optional file `.gitignore` in the main Git directory contains filenames or filename patterns (for example, `*.log`) that should be ignored by the Git repository. This prevents Git from printing a long list of untracked files when executing the `git status` command that need not be tracked or considered for addition to the repository.

12.2.2 GitHub

GitHub is a Git service hosted in the cloud that provides Git functionality as well as a few additional features. Many open source software projects use GitHub to host their source code and development efforts. Some companies (for example, Netflix) also use GitHub as their source code and code development repository.

It is easy to start a free account on GitHub (visit http://www.github.com) and start a Git repository. Afterwards you can clone your GitHub repository and create a local repository. At that point you can issue the standard Git commands to commit changes locally, and the push and pull so that the local repository (remote) synchronizes with the original repository (origin). Alternatively, GitHub offers a web interface through http://www.github.com where you can execute many GitHub commands instead of using the command line.

GitHub offers two additional features beyond standard Git: fork and pull-requests.

The fork operation is similar to Git clone, but is different in that it creates a copy without assuming or requiring permission to modify the original repository.

GitHub hosts many open source projects whose code is meant to be disseminated widely, but only a small team of developers are meant to modify the code. GitHub's fork operation addresses this issue. It is similar to clone, but rather than downloading the entire repository it creates a clone on GitHub's servers that is a separate repository. The user can create a local version, make changes to it, and do commits, but any pushes will modify the hosted copy and not the original repository. This facilitates open source collaboration process where a few people have permission

to modify the main source code project, but everyone have access to download the latest version (or a development branch) of the software as well as create their own version (fork) and modify it.

Pull requests allow people without permission to modify the original (forked) repository submit requests to people with permission to modify the original repository. After committing a change to the forked repository, a pull request sends a request to one or more people with commit access to the original repository. These people are notified of the request and they can view the commit and its diff. They can then approve the request and merge the commit into the main repository, or can leave comments outlining objections to the new version or requests for modifications. If a request for modifications is made, the person who made the original commit can make modifications after which the new commit is sent again for review.

Pull requests can also be used with the original repository, rather than a forked one. This encourages having developers submit their code for review by other developers who can find problems or make suggestions for improvement. Such code review process is standard in professional software development industry.

Executing a fork or a pull request can be made through the web interface at http://www.github.com or through a command line tool `gh` using `gh fork` and `gh pull-request`. The website http://www.github.com contains tutorials explaining how to use fork and pull requests in detail.

12.2.3 Subversion

Subversion is an older version control system that is a successor to the older version control system Concurrent Versions System (CVS). It is still widely used today in many open source and industry projects. In contrast to Git, it is centralized, rather than distributed. The working copy that is stored on the client does not hold the entire repository information. Instead, it stores the recent commit and the edits made on top of it.

To check out a working copy of a repository, run `svn checkout repo` where `repo` specifies the address of the repository, for example: http://server.com/svn/project1 or svn+ssh://username@server.com/svn/project1.

To update the local version with changes that were made to the repository we use the command `svn update` (analogous to `git pull`). This performs a merge if needed, including manual resolution of merge conflicts if they exist (similar to the Git merge process).

Since the working copy in Subversion is not a repository (as in the case of Git), the commit command `svn commit -m "commit message"` updates the repository. It is analogous to `git commit` followed by `git push`. It is necessary to run `svn update` before `svn commit` if there are new updates on the remote repository.

Several other Subversion commands are similar to Git: `svn add` to add files, `svn cp` to copy files, and `svn mv` to move files. Also, Subversion has its own diff

(`svn diff`), log (`svn log`), status (`svn status`), and revert (`svn revert`) commands. These commands are similar to the Git counterparts described in Sect. 12.2.1.

In contrast to Git's SHA-1 commit codenames, Subversion uses sequential numbers to denote versions. This is not a problem since Subversion is centralized and can thus enforce a clear numeric structure on the commits of different contributors. Switching to an earlier version in Subversion can be done by a reverse merge: for example, if the repository is at version 10 the command `svn merge -r 10:9` will bring back version 9 to the working copy. A subsequent `svn commit` will commit this change to the repository.

Subversion's implements branch by copying files into a new directory representing the new branch. For example, the command below copies the repository trunk (representing stable version) into a new directory that can host feature work.

```
svn copy trunk path-to-branch/branch_name
```

Making changes to the files in the new directory and committing them will keep track of the branch versions without affecting other branches. The copied directory is linked to the original directory to preserve its history and save storage space. Specifically, the branches store history up the time of copy as well as more recent changes. The branch version can be merged back into the original branch by changing directory to the original branch followed by a merge command such as the command below

```
svn merge path-to-branch/branch_name
```

This updates the working copy. Executing `svn commit` will update the repository.

Git has several advantages over Subversion, including: (a) not requiring network access for many operations, (b) faster commands (due to no need for network latency in some cases), (c) considerably smaller repository files, and (d) more lightweight implementation of branches. As a result Git has been gaining considerable popularity recently.

12.3 Build Tools

Complex software requires a complex build process: (a) compiling files that were changed since the last compilation, (b) linking different files together and possibly with external libraries, and (c) running tests to diagnose potential problems with the code. Build tools enable automating this process so that less manual effort is needed and fewer mistakes are made. It is an essential part of almost all large scale software development processes.

At the heart of build tools is declaring a list of tasks that need to be executed, specifying the dependency between tasks and files and between different tasks,

and providing implementation details for executing the tasks. These details are expressed in one or more build files that are written in a specific format depending on the build tool that is used. The build tool reads the build file or files and executes the appropriate tasks in order.

We start below with Make, one of the earlier build tools used primarily for building C and C++ projects. We follow with a description of Ant, a more sophisticated build tool for Java, and conclude with Gradle, a state-of-the-art build tool for a variety of programming languages.

12.3.1 Make

Make is an open source build tool that is part of the GNU Linux operating system. Make was very popular in the late 1990s and early 2000s and is primarily used to build C and C++ programs, though its generality makes it useful for other programming languages and also tasks that are not related to building or compilation.

The make program executes a sequence of instruction specified in a text file called `makefile` or `Makefile`. Specifically, running the command `make target_name` in the shell prompt executes the sequence of instructions (specified in the `makefile`) needed to build the target `target_name`.

The `makefile` file is a text file containing a list of instructions for building one or more targets. Each target has the following format:

```
# comment
target: prereq_1 ... prereq_k
  command_1
  ...
  command_m
```

Above, `target` is a label describing the task and `prereq_1` to `prereq_k` specify files that the target depends on. In other words, if the prerequisite files change since the previous make execution, the make program needs to rebuild the target. The commands `command_1` to `command_m` are prefixed by a tab and describe the process that builds the specified target. In general there may be zero or more prerequisite files and zero or more commands in each target. Text following # are interpreted as comments.

The above syntax for building a target is typically repeated multiple times in the makefile, usually with more general targets appearing at the top of the file and including more specific targets as prerequisites. Running `make` without a target argument builds the first target.

Prerequisite in the makefile can be specified using wildcards (for example, `*.cpp` expands to all files ending with `.cpp` in the current directory). Make can also use variables, denoted by uppercase labels, in the specification of the commands. There are several core variables including: `$`—the filename representing

the current target, and `$<`—the filename of the first prerequisite. Custom variables can also be defined in the makefile (for example, `CC=gcc`) and referred to afterwards using the `$` symbol (for example, `$(CC)`).

Make also has pattern rules that can significantly simplify the makefile structure. A pattern rule is similar to a normal rule except that the stem of the file (filename without suffix) is represented by the `%` character. For example, the rule below can be thought of as multiple rules matching each .o file with a target whose prerequisite file is the corresponding .cpp file and whose command compiles (without linking due to the `-c flag`) the .cpp file into the object file whose name is specified by the target.

```
%.o: %.cpp
gcc -c -o $@ $<
```

As our first example, let's consider a "hello world" program in C++ that includes three files. The first file represents the main executable function and is called `hello.cpp`.

```
#include <hello.h>

int main() {
  helloWorld();
  return 0;
}
```

The second file contains the implementation of the `helloWorld` function.

```
#include <iostream>
#include <hello.h>

using namespace std;

void helloWorld() {
    cout << "Hello makefiles!" << endl;
}
```

The third file contains the .h header file of the `helloWorld` function.

```
void helloWorld();
```

The normal compilation command is as follows. It instructs the compiler to compile `hellomake.c` and `hellofunc.c`, name the executable file `hello`, and look at the current directory for any header files included in the specified source files.

```
gcc -o hello hellomake.cpp hellofunc.cpp -I.
```

Running this command whenever we want to build the project has the disadvantage that it recompiles all files whenever executed, even if we modify only one of the files. This is not a major problem in this simple example, but for large projects with many files and a lengthy compilation process it can be a major issue.

A simple makefile that resolves the difficulty above for this case is as follows.

```
CPPC=gcc  # C++ compiler
IDIR=.  # directory storing .h header files
CPPFLAGS=-I$(IDIR)  # flags for the C++ compiler
DEPS = hello.h  # general dependency to add to any compilation
OBJ = hello.o hellofunc.o  # object files

# primary project target, linking .o object files
# into the final executable
hello: $(OBJ)
     $(CPPC) -o hello hello.o hellofunc.o $(CPPFLAGS)

# all .o files depend on corresponding .c files and header file
# compile, without linking, source files into object files
%.o: %.cpp $(DEPS)
$(CC) -c -o $@ $< $(CPPFLAGS)

# cleanup target, removing intermediate files including
# object files (*.o), editor auto-save files (*~), and core dump
clean:
rm -f *.o *~ core
```

The first rule above specifies that the `hello` target depends on the two object files (compiled but not linked) and specifies the compilation command using the custom variables `CPPC` (compilation command) and `CPPFLAGS` (flags).

The second rule specifies that each .o object file depends on the corresponding .cpp file and the include file. The command compiles the .cpp file (using the variable `$<`) into the .o file (using the variable `$@`) (the `-o` flag specifies compilation without linking).

The third rule cleans up the directory removing intermediate files, leaving only the original source files and the final executable.

When we run `make hello`, the make program checks if any of the .o prerequisites have changed and require rebuilding. This is done by checking the second target that depends on the corresponding .cpp file and the header file. If the .cpp and the .h files did not change nothing is compiled. If only one of the .cpp files changed the second rule is applied for that file followed by the first rule. Generalizing this simple makefile to a larger more complex program can automate the build process without running unnecessary compilation stages.

There are two main principles for writing good makefiles: (a) separating compilation steps as much as possible, and using variables to separate logic from names and directory choices. We saw above how the first principle above can lead to significantly reduced compilation time for large projects. The second principle is useful in that revising the program (for example, by deciding to move the header files to a different directory or by adding additional .cpp files) requires a simple modification to the definitions of the custom variables rather than rewriting the build commands.

12.3.2 Ant

Ant is a build tool that is similar to Make, but is implemented using the Java language and is best suited for building Java projects. Ant is an open source project hosted by the Apache Software Foundation.

One difference between Ant and Make is that Ant uses an XML file (by default build.xml) to describe the build process and its dependencies, while Make uses the Makefile format described above.

A second difference is that Ant is more portable in that the same XML build file can run appropriately on multiple operating systems, for example Linux and Windows. Make is based on shell commands that would typically be different on different operating systems.

A third difference is that Ant provides a nice integration with Java unit tests. As a result, Ant conveniently integrates unit tests into the build process and is particularly convenient for test-driven development process.

We describe below several important aspects of the Ant XML file format, followed by an example build file and some final remarks.

The XML build file starts with a line describing the XML format and follows with a project element that includes inside of it one or more target elements. For example, the skeleton file below has an XML format specification line followed by an outer project element containing several inner target elements.

```
<project name="hello" default="build">
  <target name="build">
    ...
  </target>
  <target name="test" description"perform unit tests">
    ...
  </target>
  <target name="deploy" depends="build,test">
    ...
  </target>
  <target name="clean">
    ...
  </target>
</project>
```

XML elements, in general, follow one of the following formats:

```
<tagType attr1=val1... attrK=valK>
  ...
</tagType>


<tagType attr1=val1... attrK=valK/>
```

The first option is used to host additional nested elements inside the outer element denoted above using the `...` notation. The second option is used for elements that do not have nested elements inside of them. In the XML example above the project

element is the outer element and has the attribute `name` and `default` that get the values `"hello"` and `"build"` respectively. The project name attribute is used to refer to the project and the default attribute refers to the default target tasks.

The target element has a `name` attribute and optionally a `depends` attribute hosting the list of targets that it depends on (for example, in the build file above the `deploy` target depends on `build` target and `test` target) and a `description` attribute that documents the target using a short readable comment.

When the command `ant target_name` is executed in the shell, Ant looks for a build.xml file in the current directory and then matches `target_name` with one of the target elements in the build file. Ant then executes the dependencies of that target followed by the instructions inside the target element. If we execute the command `ant` without a target name, Ant executes the target marked as default in the project element (for example, `build` target in the example above).

The instructions for executing the targets are usually given using XML elements such as `<delete>` (deletes files), `<javac>` (Java compiler), and `<mkdir>` (creating a directory). These XML elements and others map to executable commands that are platform independent (for example, the `<delete>` element will map to the right shell command for removing files based on the operating system).

Below is a simple Ant file for building and cleaning a simple hello world Java application that contains a single file. Note that presence of the two targets: `build` and `clean`, the first of which is the default target. The first target creates a directory for the class file and runs the Java compiler on the specified directory. The `javac` element describes the location of the source files and several additional optional attributes that are passed to the Java compiler. The `delete` element has a `fileset` element that describes the set of files to act one. The `fileset` tag has a directory attribute and an optional nested `include` element that filters the files to act on using for file name patterns (for example, `**/*.java` refers to all files with a `.java` file extension that lie in the current directory or one of its subdirectories).

```
<?xml version="1.0"?>
<project name="hello" default="build">
  <target name="build" description="build project">
    <mkdir dir="classes"/>
    <javac destdir="classes">
      <src path="src"/>
    </javac>
  </target>
  <target name="clean" description="clean class files">
    <delete>
      <fileset dir="classes">
        <include name="*.class"/>
      </fileset>
    </delete>
  </target>
</project>
```

Putting a Hello World Java application (for example, the file below) in the subdirectory `src/` and running `ant` will build the class file using the Java compiler and store it in the `classes/` subdirectory.

```
public class Hello {
   public static void main(String[] args) {
      System.out.println("Hello, World!");
   }
}
```

Ant can define variables (for example, for storing directory paths) using the XML element `property`. Predefined properties include for example `basedir` (base diretory), `ant.file` (location of the build file), and `ant.java.version` (version of JDK used by Ant). Additional properties can be defined by specifying a corresponding XML element (see example below).

```
<property name="ProjectURL" value="www.computingwithdata.com">
```

To refer to properties in the XML file we enclose the property name with the following structure `${...}`. Properties can also be specified in an external file, in which case they are listed as a list of `property_name=property_val` (the name of the external file needs to be specified in the build file using the following element: `property file="filename"/>`). This allows using switching between different property files without modifying the XML build file.

12.3.3 Gradle

Gradle is a recent build tool that has several advantages over Ant. It is suited for a large number of languages (in addition to Java) and its build files are more readable and concise than Ant's XML file or Makefiles. Specifically, Gradle build files are specified using Groovy—a dynamic programming language similar to Python that is dynamically compiled to run on the Java Virtual Machine. Gradle is gaining in popularity in recent years and is integrated with most popular IDEs (for example, Eclipse or IntelliJ).

Since Gradle is generic, it has multiple plugins or extensions corresponding to different programming languages. We describe below how to use Gradle with the Java plugin for building Java projects. Additional plugins are available for building projects in C++, Scala, and several other languages.

By default, Gradle's Java plugin assumes the following directory structure, borrowed from Maven, another popular Java build tool.

. : The base directory of the project containing the following subdirectory and the build file `build.gradle`.

`src/main/java` : Contains the Java production source files.

`src/main/resources` : Contains additional project files (for example additional JAR files)

`src/test/src` : Contains the Java test source files
`src/test/resources` : Contains additional test files (for example additional JAR files)
`build` : Contains the output compilation files

In contrast to Ant and Make, Gradle can proceed with a nearly empty build file assuming we use the directory structure above. As a simple example consider copying the hello world Java file described in Sect. 12.3 to `src/main/java` and having a nearly empty build file `build.gradle` instructing gradle to use the Java plugin as follows.

```
apply plugin: 'java'
```

When we execute the command `gradle build`, Gradle will compile the source file and put the compiled class file in `build/classes/main`. The command `gradle clean` will remove compiled files in the `build` directory and `gradle test` will compile the source code and run tests specifying in the `src/test` directory. Gradle defines automatically these targets or tasks (build, clean, check), as well as other targets such as assemble, test, and jar. The implementations are already defined for these tasks as well as the dependencies (for example, the build target depends on the assembly target as well as the check target that in turns depends on the test target).

Gradle can adapt to unconventional directory structures by assigning values to the appropriate variables. For example, the following build.gradle file instructs gradle to use a custom definition of the source file directory.

```
apply plugin: 'java'
sourceSets.main.java.srcDirs = ['java_src']
```

Alternatively, we can use define a custom directory structure in the following way in the build file.

```
sourceSets {
  main {
    java {
      srcDir 'java_src'
    }
    resources {
      srcDir 'resources_src'
    }
  }
}
```

Gradle can be used as in the examples above, where the definition of the directory structure, target implementations, and dependencies are prespecified. In this case, writing the build file is very easy and fast and it provides significant functionality. Gradle can also be used when customizing some or many of these conventions (directory structure, target implementations, dependencies), in which case it offers significant flexibility at the cost of a need to carefully author the build file. In any

case, Gradle's build files are more readable than Makefiles or Ant's XML build files and are thus well suited for large complex projects.

12.4 Exceptions

Code can fail due to a variety of reasons. Examples include inadmissable input, incorrect logic, and undefined values such as division by zero. It is important to not let the code fail inappropriately and rather handle it in a carefully designed manner.

Traditionally, handling potential failures was done by checking input carefully before entering a function and making sure the output value is properly assigned. In the calling environment, the value returned from the function is checked again and handled appropriately if the returned value indicates a failure.

Below is an example in C++ where a function attempts to insert an object into a list in a specific position. The code first check if the list is long enough and if not returns with a specific code (-1). Otherwise, the function calls `insert_res` and returns the return value from that function.

```
int insertInPosition(List L, int pos, Object o) {
  if (pos >= L.size())
    return(-1);  // return -1 value for a fail
  int insert_res = L.insert(pos, o);
  return insert_res;
}
```

The process described above requires (a) writing code that will diagnose whether there is a problem and (b) handle the problem, and so at multiple key places such as function entry points and function return. Doing so eventually leads to writing duplicated code that is hard to maintain.

Exception handling is a more effective way of handling code failures that at this point is supported in many major programming languages, including C++, Java, Python, and R. We describe below exception handling in Python. Exception handling in other languages is similar and once the concept is clear it is easy to learn how to use it in any of the languages above.

Exceptions are objects that represent a problematic situation that needs to be logged or handled. Usually, exception classes may be implemented in the language definition or standard library, or may be defined and implemented by the programmer. In either case, exception classes inherit from the base exception class (for example, in Python it is `BaseException`) or one of its descendants.

When the appropriate situation is discovered, the exception is activated. This is called throwing an exception. At that point program execution is suspended and is transferred to the try block containing the code that lead to the thrown exception.

We describe the process below in the case of the Python programming languages. The process is similar (though some differences remain) in other programming languages.

The code below computes a mathematical expression, and it requires that the argument be a positive integer. We thus check the argument and throw an exception if needed.

```
import math

def foo(x):
  if not isinstance(x, int):
    raise TypeError("arguments to foo must be integers")
  if x <= 0:
    raise ValueError("arguments to foo must be positive")
  return x + math.log(x)

foo(-3)  # exception triggered and program halted
foo(3)  # never executed
```

In the example above, we did not have specific code to handle the thrown exception, and thus the program is halted immediately and never reaches the last line. Also, if we have a function that calls another function that raises an exception without exception handling code the program halts and no code afterwards is executed.

When the program is halted we get a printout of the traceback object that records the program path that lead to the exception (for example, which function called the function that triggered the exception). This information is very useful in debugging and in understanding the behavior of complex code. For example, the traceback below shows the path `bar(3)` followed by `foo(-x)`.

```
def bar(x):
  foo(-x)

bar(3)

## ---------------------------------------------
## ValueError   Traceback (most recent call last)
## <ipython-input-17-2f0fa4d9f487> in <module>()
## ----> 1 bar(3)
##
## <ipython-input-15-a80ffbcd5b6f> in bar(x)
##      1 def bar(x):
## ----> 2  foo(-x)
##      3

## <ipython-input-12-1165b706a01c> in foo(x)
##      3  raise TypeError("arguments to foo must be integers")
##      4    if x <= 0:
## ----> 5 raise ValueError("arguments to foo must be positive")
##      6    return x + math.log(x)
##      7
##
## ValueError: arguments to foo must be positive
```

12.4.1 Handling Exceptions

In order to trigger a more specific handling of the exception (rather than simply terminate the program), we include code that may trigger exceptions in a `try-except` clause. The code following the `try` keyword is executed, and if an exception is triggered execution passes immediately to the code following the `except` keyword that handles the exception. After exception handling code is finished, the program continues to execute the code following the `except` block (rather than terminate).

```
import math

def foo(x):
  if not isinstance(x, int):
    raise TypeError("arguments to foo must be integers")
  if x <= 0:
    raise ValueError("arguments to foo must be positive")
  return x + math.log(x)

try:
  foo(-3)  # exception triggered and program halted
  foo(3)  # never executed
except:
  print("exception in sequence of foo() calls")
print("program resumes execution and is not terminated")
## exception in sequence of foo() calls
## program resumes execution and is not terminated
```

The syntax above catches and handles all exceptions in the same way. We can write custom handling code for different exceptions by including multiple `except` blocks followed by the exception class that they handle. If there are multiple `except` blocks and an exception occurs, the program executes the exception handling code and then continues after the last `except` block. An optional block `else:` is called if no exception has been handled (`else:`) and an optional block `finally:` is always called and can be used to host cleanup code (e.g., closing a file or database connection).

```
import math

def foo(x):
  if not isinstance(x, int):
    raise TypeError("arguments to foo must be integers")
  if x <= 0:
    raise ValueError("arguments to foo must be positive")
  return x + math.log(x)

try:
  foo(-3)  # exception triggered and program halted
  foo(3)  # never executed
except TypeError:  # handle TypeError exceptions
```

```
    print("Incorrect Type in sequence of foo() calls")
  except ValueError:  # handle ValueError exceptions
    print("Non-positive value in sequence of foo() calls")
  else:
    print("no exceptions handled")
  finally:
    print("always handled")
  print("program resumes execution and is not terminated")
  ## Non-positive value in sequence of foo() calls
  ## always handled
  ## program resumes execution and is not terminated
```

If we want to write `except` blocks that handles exceptions, but also terminate the program with printing the traceback object to the console (as if the exception has never been caught) we can include the `raise` keyword (with no arguments) at the end of the appropriate `except` block. In this case, the code in the `except` block up to the `raise` keyword is executed followed by program termination and printing of the traceback object.

If there are multiple `except` blocks and an exception is raised, only the first match will execute. An exception can match an `except` block also if it specifies an exception class that it inherits from. This is useful for handling some exceptions in a particular way, and all other exceptions in a default way: include the specific exceptions in separate `except` blocks first followed by an `except Exception:` block that will handle all remaining exceptions (since all exception classes inherit from the base Exception class). Since the first matching `except` block is used, multiple `except` blocks are ordered from more specific exceptions to less specific exceptions.

If we want access to the exception object in the `except` block, we can modify the syntax `except TypeError:` to `except TypeError as e:` that exposes the exception object as `e` in the block.

```
import math

def foo(x):
  if not isinstance(x, int):
    raise TypeError("arguments to foo must be integers")
  if x <= 0:
    raise ValueError(f"arguments to foo must be positive: {x}")
  return x + math.log(x)

try:
  foo(-3)  # exception triggered and program halted
  foo(3)  # never executed
except TypeError:  # handle TypeError exceptions
  print("Incorrect Type in sequence of foo() calls")
except ValueError as e:  # handle ValueError exceptions
  print("Non-positive value in sequence of foo() calls:")
  print(e.args)
print("program resumes execution and is not terminated")
## Non-positive value in sequence of foo() calls:
```

```
## ('arguments to foo must be positive: -3',)
## program resumes execution and is not terminated
```

12.4.2 Custom Exceptions

In some cases it is useful to define new exception classes that represent specific situations. To do so we simply specify that it inherits from the base `Exception` class. The class name usually communicates what went wrong.

```
import math

# Custom exception: either type or value error
class InvalidTypeOrValueError(Exception):
  def __init__(self, a_val, a_type):  # custom constructor
    # calls super class constructor (Python 3 syntax)
    super().__init__("incorrect type and value. " +
      "val: " + str(a_val) + " " + "type: " + str(a_type))
    self.val = a_val
    self.type = a_type

def foo(x):
  if not isinstance(x, int) or x<=0:
    raise InvalidTypeOrValueError(x,type(x))
  return x + math.log(x)

try:
  foo(-3)
  foo(3)
except InvalidTypeOrValueError as e:
  print("Incorrect type or value in sequence of foo() calls: ")
  print(e.args)

## Incorrect type or value in sequence of foo() calls:
## ("incorrect type and value. val: -3 type: <class 'int'>",)
```

12.5 Documentation Tools

It is useful to write self-documenting code in the sense that the source code files contain significant documentation about the code implementation as well as how to use it. Many programming languages have a built-in documentation format that if the programmer adheres to can generate separate documentation files based on the in-code comments. Below we describe first Docstrings—a built-in documentation format used in Python, and later describe Javadoc—a similar format that is used in Java.

12.5.1 Docstrings

Python docstrings appear as a string literal in the first statement following the definition of functions, methods, classes, and modules. Below is an example of a docstring documentation of the Point class from Chap. 6.

```
class Point(object):
  "A point in a two dimensional space"

  def __init__(self, x=0, y=0):
    """
    Initializes the point object to the origin by default,
    or otherwise the initializes to the passed x,y arguments.
    """
    self.x = x
    self.y = y
  def __del__(self):
    "Prints a message when an object is destroyed."
    print "destructing a point object"
  def displayPoint(self):
    "Prints the object by displaying its x and y coordinates."
    print "x: %f, y: %f" % (self.x, self.y)

help(Point)  # shows class doctring
help(Point.displayPoint)  # shows method docstring
```

Docstrings can be accessed by the __doc__ attribute on objects inside the program or by typing `help(X)` in the Python prompt where `X` is the name of a class, method, function, or module.

The docstring for a class should summarize its behavior and list the public methods. The docstring of a function or method should describe its effect, the arguments, the return values, and any restriction on its usage. The docstring for a module should list the classes and functions that are exported by the module with brief high-level descriptions.

Another way to generate documentation from docstrings is using the pydoc program that can be called from the OS prompt as follows: `pydoc X` where is a built-in or programmer defined module, function, or class (in the latter case of built-in a path should be provided in `X` including a slash character).

For example, typing at the OS prompt `pydoc /Point.py` produces the following documentation for the Point class above.

```
FILE
    /Users/glevanon/tmp.py

CLASSES
    __builtin__.object
        Point

    class Point(__builtin__.object)
     |  Represents a point in a two dimensional space
```

```
 |
 |  Methods defined here:
 |
 |  __del__(self)
 |      Prints a message when an object is destroyed
 |
(truncated for brevity)
```

Adding the `-w` flag to the pydoc command produces the generated documentation as an HTML file. A more sophisticated and recent alternative to pydoc is the Sphinx program that can include math formulas, ipyhton interactive sessions, inheritance diagrams, and figures generated using matplotlib. Since it is considerably more complex than pydoc we do not discuss it in this book. More details about Sphinx are available at http://sphinx-doc.org.

12.6 Program Diagnostics

12.6.1 Debugging

Core R features some debugging and profiling functionality. Additional debugging and profiling functionality is available through third party packages, as described at the end of this section.

The `print` and `cat` functions display values of relevant variables at appropriate positions and can be used while debugging code. Specifically, the `print` function displays the value of a variable, while the `cat` function displays the concatenation of values of several variables passed as parameters.

The `browser` function suspends the execution of the R code, providing an interactive prompt through which the programmer can evaluate variables or execute new R code while debugging. Other options include stepping through line by line (by typing `n`), continuing execution (by typing `c`), or halting execution (by typing `Q`). The interactive session below demonstrates debugging with the `browser` function.

```
foo2 = function(i) {
  a = i + 1;
  b = a + 1;
  browser();
  a = b + 1;
  return(b)
}
> foo2(3)
Called from: foo2(3)
Browse[1]> a  # display the value of a
[1] 4
Browse[1]> b  # display the value of b
[1] 5
Browse[1]> n  # executes next command
debug: a = b + 1
```

```
Browse[2]> c  # continue execution
[1] 5
```

The function `debug` binds the `browser` function to a user-specified function, invoking a debug session at each call site of said function. The `undebug` cancels the binding.

```
foo3 = function(i) {
  a = i + 1;
  b = a + 1;
  a = b + 1;
  return(b)
}
> debug(foo3)
> foo3(1)
debugging in: foo3(1)
debug: {
    a = i + 1
    b = a + 1
    a = b + 1
    return(b)
}
Browse[2]> n  # execute first command
debug: a = i + 1
Browse[2]> n  # execute next command
debug: b = a + 1
Browse[2]> a  # display value of a
[1] 2
Browse[2]> p=2*a+2  # define a new variable
Browse[2]> p
[1] 6
Browse[2]> n  # executes next command
debug: a = b + 1
Browse[2]> b
[1] 3
Browse[2]> c  # continue execution
exiting from: foo3(1)
[1] 3
> undebug(foo3)
```

The `trace` function accepts the name of a function as a parameter and prints the function call with the passed arguments each time the function executes. If an error occurs, the variable `.Traceback`, a character array containing the currently active function calls, appears in the workspace.

Additional debugging functionality is available from contributed packages. In particular, the `debug` package (Bravington, 2003) provides capabilities similar to those of standard graphical debuggers, including a graphical window showing the current command, setting conditional breakpoints, and the opportunity to continue debugging after an error. Another package is `edtdbg`, which integrates R debugging with external text editors such as vim.

Profiling code refers to diagnosing which parts of the code are responsible for heavy computational efforts and memory usage. The function `Rprof(X)` starts a

profiling session and saves the information to the file `X`. A call `Rprof(NULL)` indicates the end of the profiling session and `summaryRprof` shows the profiling summary.

```
# start profiling
Rprof("diagonsisFile.out")
A = runif(1000)  # generate a vector of random numbers in [0,1]
B = runif(1000)  # generate another random vector
C = eigen(outer(A,B))
Rprof(NULL)  # end profiling
summaryRprof("diagonsisFile.out")
## $by.self
##                      self.time self.pct total.time total.pct
## "eigen"                  10.56    93.78      11.26    100.00
## "sys.function"            0.46     4.09       0.46      4.09
## (truncated for brevity)
## "nrow"                    0.02     0.18       0.02      0.18
##
## $by.total
##                       total.time total.pct self.time self.pct
## "eigen"                    11.26    100.00     10.56    93.78
## "sys.function"              0.46      4.09      0.46     4.09
## "formals"                   0.46      4.09      0.00     0.00
## "match.arg"                 0.46      4.09      0.00     0.00
## (truncated for brevity)
## "nrow"                      0.02      0.18      0.02     0.18
##
## $sample.interval
## [1] 0.02
##
## $sampling.time
## [1] 11.26
##
```

Total time is time spent in the function and its respective call subgraph; self time is the time spent only in the function itself. In the example below, the function `eigen` is responsible for nearly 100% of the total time, but most of the execution time occurs within other functions called from within `eigen` (see the second report above). The methods that are called from within `eigen` are clearly the computational bottleneck, as the outer product and the generation of random samples are not responsible for any substantial total time.

See http://cran.r-project.org/doc/manuals/R-exts.pdf for more information on `Rprof` and a discussion on profiling memory usage. The packages `proftools` and `profr` provide additional profiling functionality including analyzing the call graphs.

Reference

M. Bravington. Debugging without (too many) tears. *The R Journal*, 3(3):29–32, 2003.

Chapter 13
Essential Knowledge: Data Stores

This chapter is all about data in the various shapes, forms, and formats they take. We explore different ways to format and store data: JSON, databases, SQL, NoSQL, and memory mapping. We also discuss concepts like atomicity, consistency, isolation, and durability of data transactions.

13.1 Data Persistence and Serialization

Data persistence is the process of saving information for later usage, potentially after the program terminates its execution. Data serialization is a process that converts information to a sequence of bytes for transmission to a file or to across a computer network. We already saw one form of persistence earlier in this book: saving data in text files and reading data from text files. Using such save and load functionality, a program can make its information persist and available after the original program terminates.

Below, we introduce JSON, a widely used method for data serialization based on text format. Many programming languages, including C++, Java, Python, and R, support saving data in JSON format and reading from it. The following two sections introduce two binary data serialization methods: Pickle (for Python) and Java Object Serialization.

13.1.1 JSON

JSON (JavaScript Object Notation, usually pronounced Jason) is a data serialization format based on human-readable text. JSON's origin is in encoding information for transmission between internet browsers and servers. Recently, JSON became

© Springer Nature Switzerland AG 2018

G. Lebanon, M. El-Geish, *Computing with Data*,
https://doi.org/10.1007/978-3-319-98149-9_13

popular also in transmitting and saving data in data analysis application. Its advantage over binary formats and other text-formats (like XML or tab separated format) is the ease with which humans can read and comprehend JSON data.

JSON encodes the following objects:

- strings (using double quotation marks, with backslash as an escape character)
- numbers (using decimal or exponential notation), (c) boolean (true or false)
- arrays (comma separated list surrounded by square brackets), and
- key-value maps (comma separated list of key:value pairs surrounded by curly braces).

Several JSON requirements and properties are listed below.

- JSON uses the value null to denote an empty or unknown value.
- The keys in key-value maps must be have string values.
- JSON ignores whitespace (space, tab, line feed, carriage return) around or between its elements. Indentation is often used to enhance the readability of JSON data.
- JSON elements may be nested, for example a key-value map may have as one of its values an array holding additional key-value maps.

For example, consider the example below of a JSON format representing the name, date of birth, address, and phone number of a person. Note that the outer object is a map structure. The `address` key of that map is a new map object and the `phoneNumbers` key is an array of maps.

```
{
  "firstName": "Jane",
  "lastName": "Doe",
  "yearOfBirth": 1975,
  "address": {
    "streetAddress": "100 Boren Avenue North",
    "city": "Seattle",
    "state": "WA",
    "postalCode": "98109"
  },
  "phoneNumbers": [
    { "type": "home", "number": "(404) 555-2314" },
    { "type": "office",  "number": "(404) 555-1321" }
  ]
}
```

Many languages provide support for parsing JSON, for example Python's `json` package provides the function `loads` for parsing JSON format into a Python object and `dumps` for converting a Python object into a JSON formatted string. Note that JSON is very similar to legitimate Python code, making Python particularly easy for interacting with Python code.

```
import json

# creating a Python object (map) holding a person's address
personRecord = {
  "firstName": "Jane",
  "lastName": "Doe",
  "yearOfBirth": 1975,
  "address": {
    "streetAddress": "100 Main Street",
    "city": "Los Angeles",
    "state": "CA",
    "postalCode": "90021"
  },
  "phoneNumbers": [
    { "type": "home", "number": "(444) 555-1234" },
    { "type": "office",  "number": "(444) 555-1235" }
  ]
}

# convert the Python object into a JSON string
personRecordJSON = json.dumps(personRecord)
personRecordReconstructed = json.loads(personRecordJSON)

# print the firstName field of the reconstructed Python object
print(personRecordReconstructed['firstName'])
```

13.1.2 Pickle and Shelves in Python

Python's pickle module is an alternative to JSON serialization that converts Python objects to serialized binary form. Binary serialization is more efficient than text serialization in terms of space, but JSON's text serialization has the advantage of being compatible across multiple programming languages and being human readable.

```
import pickle
patients = {
  "age" : [25.2, 35.4, 52.1],
  "height" : [68.1, 62.5, 60.5],
  "weight" : [170.2, 160.7, 185.5]
}
# binary serialization
serialized_patients = pickle.dumps(patients)
# deserialization
patients_reconstructed = pickle.loads(serialized_patients)
```

Pandas offer convenient functionality to serialize and deserialize dataframes using pickle. Specifically, the pandas functions to_pickle and read_pickle convert a dataframe to pickle serialization and vice versa.

```
import pandas as pd
patients = {
  "age" : [25.2, 35.4, 52.1],
  "height" : [68.1, 62.5, 60.5],
  "weight" : [170.2, 160.7, 185.5]
}
patients_DF = pd.DataFrame(patients)
# serialize and save to binary disk
patients_DF.to_pickle('patients.txt')
# read binary serialization to Python object
patients_reconstructed=pd.read_pickle('patients.txt')
print(patients_reconstructed)
```

The Python module cpickle implements the same algorithm as pickle but in C rather than in Python, and thus may run faster. Switching between pickle and cpickle can be conveniently done by inserting or removing the line `import cpickle as pickle` at the beginning of the Python program.

Shelves is a module that provides persistent dictionary objects whose values can be any Python object that can be pickled. The programmer can conveniently organize multiple pickled objects for easy future retrieval using the dictionary keys. The data is stored on disk using the `anydbm` database module.

The code below stores data as a shelves object.

```
import shelve
patients_DF = {
  "age" : [25.2, 35.4, 52.1],
  "height" : [68.1, 62.5, 60.5],
  "weight" : [170.2, 160.7, 185.5]
}
records = shelve.open('patients')
records['patients'] = patients_DF
records.close()
```

The code below reads the saved data from the shelves object.

```
import shelve
records = shelve.open('patients')
patients_reconstructed = records['patients']
del records['patients']  # delete value from disk
records.close()
```

13.1.3 Java Object Serialization

Java's serialization framework provides a convenient mechanism to serialize arbitrary objects into a binary sequence. To serialize an object using this framework, the object in question must implement the serializable interface. Many Java classes that hold data (for example, lists of strings or hash tables mapping strings to doubles)

implement that interface. New classes defined by the programmer in many cases implement the interface without any additional work (though it is important to denote this by adding `implements Serializable` after the class name). In other cases, or in cases where a customized serialization is needed, the serialization and deserialization methods need to be implemented for the object in question.

Below is an example of serializing and deserializing a list of records. The first code segment below is an implementation of a class that holds the records and the second code segment serializes a list of records and saves it in a binary file. The third file deserializes the file back into a Java object.

```
import java.io.Serializable;

public class PatientInfo implements Serializable {
  private String name;
  private int age;
  public PatientInfo(String a_name, int an_age) {
    this.name = a_name;
    this.age = an_age;
  }
  public int getAge() {
    return age;
  }
  public void setAge(int an_age) {
    this.age = an_age;
  }
  public String getName() {
    return name;
  }
  public void setName(String a_name) {
    this.name = a_name;
  }
}


import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.util.ArrayList;
import java.util.List;

public class PatientPersist {
  public static void main(String[] args) {
    String filename = "patients";
    PatientInfo patient1 = new PatientInfo("John Smith", 33);
    PatientInfo patient2 = new PatientInfo("Jane Doe", 30);
    List PatientList = new ArrayList();
    PatientList.add(patient1);
    PatientList.add(patient2);
    FileOutputStream fos = null;
    ObjectOutputStream out = null;
```

```
      try {
        fos = new FileOutputStream(filename);
        out = new ObjectOutputStream(fos);
        out.writeObject(PatientList);
        out.close();
      } catch (IOException ex) {
        ex.printStackTrace();
      }
    }
  }
```

```
import java.io.FileInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.util.ArrayList;
import java.util.List;

public class GetPatientInfo {
  public static void main(String[] args) {
    String filename = "patients";
    List PatientsInfo = null;
    FileInputStream fis = null;
    ObjectInputStream in = null;
    try {
      fis = new FileInputStream(filename);
      in = new ObjectInputStream(fis);
      PatientsInfo = (ArrayList) in.readObject();
      in.close();
    } catch (IOException ex) {
      ex.printStackTrace();
    } catch (ClassNotFoundException ex) {
      ex.printStackTrace();
    }
    System.out.println(PatientsInfo.size() + " patients found");
  }
}
```

Compiling the three files above and executing the programs above produce the following output.

```
java PatientPersist
java GetPatientInfo

## 2 patients found
```

13.2 Hierarchical Data Format

Hierarchical Data Format (HDF) is a file format designed to store groups of large multidimensional numeric arrays. It was originally developed at the National Center

for Supercomputing Applications and it is currently in its fifth generation, named HDF5. HDF has a permissible open-source license (BSD) and many programming languages provide API for accessing data stored in HDF format including C++, Java, R, Python, and R.

HDF files are self-describing in the sense that they hold meta-data describing the structure and content of the file. A single file can contain complex objects built from multidimensional arrays such as color images, astronomical measurements, and time series. Accessing specific content within HDF content (for example, a particular portion of a multidimensional array) is typically faster than a similar access operation in relational (SQL) database, but NoSQL databases may provide additional speedup.

13.2.1 Accessing HDF from Python Using PyTables

Python offers a particularly convenient mechanism for accessing data in HDF files using the PyTables module. Specifically, PyTables contains functions for accessing HDF data using NumPy's API for accessing dataframes in memory. As a result, a programmer can work with large data stored on disk, in much the same way as working with dataframes in memory (however, a performance penalty naturally applies to operating out of memory). PyTables brings data from disk to memory lazily (data is moved from disk to memory only when it is needed), and operates intermediate operations on the data in-memory, which provides a significant speedup. At the end of the program, the HDF files are updated and the relevant memory content is cleared.

The example below shows an elementary interaction with HDF files using PyTables.

```
from tables import *
# creating an object representing table rows
class Person(IsDescription):
  name = StringCol(16)  # 16 character string
  age = Int32Col()
# opening a HDF5 file
h5file = open_file("people.h5", mode = "w", title = "personnel")
# defininining a group in the HDF4 file
group = h5file.create_group("/", "records", "names and ages")
# defining a table in the group
table = h5file.create_table(group, "Personnel1",
                            Person, "Personnel Table 1")
Person = table.row
for ind in xrange(10):
  Person['name'] = 'John Doe ' + str(ind)
  Person['age'] = 20 + ind
  Person.append()
for ind in xrange(10):
  Person['name'] = 'Jane Smith ' + str(ind)
```

```
    Person['age'] = 20 + ind
    Person.append()
table.flush()  # update disk

# getting a pointer to table
table = h5file.root.records.Personnel1
# running a query against file
arr = [x['name'] for x in table.iterrows() if
  x['age'] > 25 and x['age'] < 28]
print(arr)

## ['John Doe 6', 'John Doe 7', 'Jane Smith 6', 'Jane Smith 7']
```

Find more about PyTables at http://pytables.github.io/usersguide/.

13.3 The Relational Database Model

A relational database management system (RDBMS) is a computer program that simplifies the process of storing and retrieving data, in particular when the data is large and there are several users storing and retrieving data. Relational databases typically provide the following capabilities.

1. Relational databases allow users to create a new database and define its schema, which is a specification of the type of data that can reside in the database.
2. Relational databases let users query the database using a query language called structured query language (SQL). The RDBMS processes the query and returns to the user the corresponding portion of the data (or sometimes a computation based on the data).
3. Relational databases are good at storing large amounts of data (typically on the hard disk) and are relatively efficient in executing queries over large datasets.
4. Relational databases enable multiple users to store and access the data, ensuring that the results of the operations are identical to a corresponding sequence of requests that are performed independently.

The last property above is sometimes referred to as atomicity. Each individual request (deletion, addition, modification, or querying) is executed in its entirety before other requests are processed. The scheduler manages such concurrent requests, and places locks that prevent additional requests from being processed until the previous requests are completed. Exceptions may be made where requests are performed in parallel if it is guaranteed that the end result remains the same.

We describe below the relational model that is used in RDBMS, and then describe the SQL query language. We conclude this section with a description of how programs can interact with relational databases.

13.3.1 The Relational Model

The relational model that is used in RDBMS is based on two dimensional tables, called relations. Each relation has rows, which are called tuples or records, and columns, which describe properties of tuples. The columns are usually annotated by descriptive names known as attributes. The values in specific row-column combinations are known as fields. The values in the different columns are constrained to be within a certain domain (for example, string, integer, or floating point). The name of the relation, the attributes, and the column domain constraints form the schema of the relation.

For example, consider the `retail-1` relation in Fig. 13.1 that contains a list of items described by their name, ID, and retail price. The columns in this case may be constrained to lie in the string, integer, and floating-point domains, respectively.

Definition 13.3.1 A set of columns form a unique key if their values uniquely identify the record. In other words, at most a single record can have any specific value of the unique key.

Definition 13.3.2 A primary key is similar to a unique key, except that it may not have Null values and any relation can have at most a single primary key.

Since the relation columns are referred to by the attributes (column names), and the rows are referred to by the primary key the ordering of the rows and columns is immaterial.

For example, in the `retail-1` relation above, the column corresponding to the ID attribute may be assigned as the primary key. There can be at most one tuple associated with any ID value.

Keys may span multiple columns, in which case the precise combination of column may appear only once. For example, tuples in the `retail-2` relation (see Fig. 13.2) correspond to clothes of different sizes that may have the same ID. In this case, the key should be the column combination (ID, size).

A relational database may contain more than a single relation. One relation can refer to a tuple in a target relation based using the primary key of the target relation.

Definition 13.3.3 A foreign key in a relation is a set of columns whose values correspond to a primary key in another relation.

name	ID	retail price
T Shirt	2314	15.95
Jeans	1492	39.95
Dress	9824	50.00

Fig. 13.1 Relation `retail-1` describing items for sale in a store. The columns are restricted to lie in the string, integer, and floating-point domains respectively

name	ID	size	retail price
T Shirt	2314	Large	15.95
T Shirt	2314	Small	13.95
Jeans	1492	Medium	39.95
Jeans	1492	Small	34.95
Dress	9824	Medium	50.00
Dress	9824	Small	50.00

Fig. 13.2 Relation `retail-2` is similar to `retail-1`, but it admits clothes of different sizes with the same ID. In this case the column combination (ID, size) is the primary key. There can be at most a single tuple with the same (ID, size) values

vendor name	phone
AB Importers	404-555-1234
Dresses Inc.	404-555-3297

Fig. 13.3 Relation `vendor-1` contains phone numbers of different vendors. The primary key is the vendor name

name	ID	size	vendor name	vendor price
T Shirt	2314	Large	AB importers	11.95
T Shirt	2314	Small	AB importers	10.95
Jeans	1492	Medium	AB importers	33.95
Jeans	1492	Small	AB importers	31.95
Dress	9824	Medium	Dresses Inc.	35.00
Dress	9824	Small	Dresses Inc.	35.00

Fig. 13.4 Relation `vendor-2` containing vendor name and wholesale prices of different items. The primary key (ID, size), which is also a foreign key linking the tuples with the tuples of relation `retail-2` in Fig. 13.2

For example, `vendor-1` relation in Fig. 13.3 describes vendor phone numbers and may coexist with the `retail-2` relation in Fig. 13.2. The primary key is the vendor name attribute.

The `vendor-2` relation in Fig. 13.4 describes wholesale and vendor information of different items and may coexist with both `retail-2` and `vendor-1` relations. The primary key of `vendor-2` is (name, ID). The relation `vendor-2` also has a foreign key vendor name linking it to the relation `vendor-1` in which vendor name is a primary key.

13.3.2 ACID

ACID stands for Atomicity, Consistency, Isolation, Durability. These are four properties that guarantee that database transactions are processed reliably.

Atomicity: Atomicity requires that each transaction succeeds in its entirety or fails (in which case the database is unchanged). In other words, it is impossible for a transaction to be executed in part due to errors, crashes, or power outages.

Consistency: Consistency requires that each transaction maintains the database in a valid or consistent state. This includes (a) ensuring that relation keys remain unique and non-null (entity integrity), (b) fields are in the prescribed domain (domain integrity), and (c) foreign keys correctly refer to primary keys in other tables (referential integrity).

Isolation: Isolation requires that the concurrent execution of transactions results in the same system state that would be obtained if the transactions were executed sequentially.

Durability: Durability requires that the effect of executed transactions persist, even in the event of an error, a power outage, or a crash.

13.3.3 SQL Language

Structured Query Language (SQL[1]) is a programming language specifically designed for managing relational databases. SQL statements can create relations, insert new data into an existing relation, modify existing data, and retrieve data. The precise specification of SQL is not always adhered to by the various commercial relational databases, making SQL code not completely portable across different database systems. Nevertheless, most relational databases support basic SQL functionality and many simple SQL programs work identically across different database systems. The examples below are executed on PostgreSQL (see Sect. 13.3.4), but they should execute similarly on other standard database programs.

SQL code is separated into statements, each of which is separated into clauses. SQL ignores whitespace and SQL keywords are not case sensitive (uppercase is typically used). SQL statements are terminated by semicolons.

Data Definition

We start by describing below basic SQL statements for creating and managing tables. The statement `CREATE TABLE TABLE_NAME (Z);` creates a new table named `TABLE_NAME` having attributes defined by `Z`—a comma separated list of `ATTRIBUTE_NAME DATA_TYPE CONSTRAINT` objects.

For example, the following statement creates a table named `LOCATIONS` with an integer `ID` and three attributes, of type `CHAR(20)`, called `CITY`, `STATE`, and `COUNTRY`.

[1] SQL is often pronounced SEQUEL.

data type	description
CHAR(n)	fixed length character array of length n
VARCHAR(n)	variable length character array with length less than or equal to n
INTEGER	Integer
FLOAT	floating-point value
DATE	date
TIME	time of day
TIMESTAMP	time of day and date

Fig. 13.5 Common data types in SQL

constraint	description
NOT NULL	attribute cannot store a Null value
UNIQUE	attribute cannot have duplicate values in multiple rows
PRIMARY KEY	attribute stores a primary key
FOREIGN KEY	attribute stores a foreign key
DEAFULT X	assigns default value to attribute

Fig. 13.6 Common constraints in SQL

```
CREATE TABLE LOCATIONS(ID INTEGER PRIMARY KEY,
   CITY CHAR(20),
   STATE CHAR(20),
   COUNTRY CHAR (20) NOT NULL);
```

The PRIMARY KEY constraint above states that the ID attribute is a primary key of the table. The NOT NULL constraint specifies that Null values that represent missing values in SQL are not allowed for the COUNTRY attribute. Common data types in SQL are listed in Fig. 13.5 and constraints are listed in Fig. 13.6.

An existing table may be modified using the ALTER TABLE X statement where X encodes the alteration. For example, the SQL code below adds a new attribute to the existing table LOCATIONS.

```
ALTER TABLE LOCATIONS ADD COORDINATES INT;
```

The SQL statement DROP TABLE X deletes the table X from the database.

Data Manipulation

Data manipulation statements add or remove rows from a specific table. The statement INSERT INTO TABLE_NAME VALUES W inserts a record W into the table TABLE_NAME. For example, the following statements create and alter the LOCATIONS table and then populate it with three records.

```
CREATE TABLE LOCATIONS(ID INTEGER PRIMARY KEY,
   CITY CHAR(20),
```

```
   STATE CHAR(20),
   COUNTRY CHAR (20) NOT NULL);
ALTER TABLE LOCATIONS ADD COORDINATES INT;
INSERT INTO LOCATIONS VALUES (13, 'Los Angeles',
                              'CA','USA',32143324);
INSERT INTO LOCATIONS VALUES (21, 'Chicago',
                              'IL','USA',12324323);
INSERT INTO LOCATIONS VALUES (54, 'San Juan',
                              NULL,'Puerto Rico',
                              87430123);
```

The `NULL` keyword in the last statement indicates a missing value.

After the above sequence of commands the table has the following values.

```
 id | city        | state |   country    | coordinates
----+-------------------+-------------------------
 13 | Los Angeles | CA    | USA          |    32143324
 21 | Chicago     | IL    | USA          |    12324323
 54 | San Juan    |       | Puerto Rico  |    87430123
```

The statement `UPDATE TABLE_NAME SET X=Y WHERE Z=W` updates the table `TABLE_NAME` as follows: the record whose `Z` attribute equals `W` is modified so that its `X` attribute equals `Y`.

For example, the following SQL code modified the above table so that `'San Juan'` is modified to `'San-Juan'`. Note that we use a primary key attribute for the attribute defined by the `WHERE` clause (primary or unique key attributes are a convenient way to access a specific record in the table).

```
UPDATE LOCATIONS SET CITY='San-Juan' WHERE ID=54;
```

After the command above is executed the table `LOCATIONS` will have the following values.

```
 id | city        | state |   country    | coordinates
----+-------------------+-------------------------
 13 | Los Angeles | CA    | USA          |    32143324
 21 | Chicago     | IL    | USA          |    12324323
 54 | San-Juan    |       | Puerto Rico  |    87430123
```

The statement `DELETE FROM TABLE_NANE WHERE X=Y` removes the record from the table `TABLE_NAME` whose `X` attribute equals `Y`. For example, the statement below removes the third entry.

```
DELETE FROM LOCATIONS WHERE ID=54;
```

```
 id | city        | state |   country    | coordinates
----+-------------------+-------------------------
 13 | Los Angeles | CA    | USA          |    32143324
 21 | Chicago     | IL    | USA          |    12324323
```

Basic Queries

Perhaps the most frequently used SQL statement is `SELECT`, which queries the database and returns a list of records and attributes. The statement `SELECT X FROM Y WHERE Z` returns attributes specified by `X` from table `Y` that satisfy the constraint specified by `Z`.

For example, consider the table `LOCATIONS` above. The following query retrieves the CITY attribute for all records.

```
SELECT CITY FROM LOCATIONS

         city
----------------------
 Los Angeles
 Chicago
(2 rows)
```

Multiple attributes can be retrieved by specifying a tuple, as below.

```
SELECT CITY,STATE FROM LOCATIONS;

         city          |        state
-----------------------+----------------------
 Los Angeles           | CA
 Chicago               | IL
(2 rows)
```

The wildcard `*` can be used to refer to all attributes.

```
SELECT * FROM LOCATIONS;
 id |   city       |    state   | country  | coordinates
----+--------------------+----------------------+-----
 13 | Los Angeles | CA         | USA      |    32143324
 21 | Chicago     | IL         | USA      |    12324323
(2 rows)
```

The `WHERE` clause can specify constraints on attribute values using boolean algebra. For example, the following query retrieves all coordinates, whose state value equals `'CA'` and country value equals `'USA'`.

```
SELECT COORDINATES FROM LOCATIONS
   WHERE STATE='CA' AND COUNTRY='USA';
 coordinates
-------------
    32143324
(1 row)
```

The examples above summarize the basic usage of the SELECT statement. Some additional functionalities are listed below.

1. Appending AS W to an attribute name in the select clause renames the retrieved column. For example, `SELECT CITY AS TOWN from LOCATIONS;` returns the `CITY` attribute for all records under the new column name `TOWN`.
2. Boolean algebra in the `WHERE` clause includes multiple logical conditions and attributes, for example

```
SELECT * FROM LOCATIONS
   WHERE (COORDINATES>2000000 OR COORDINATES>
   3000000)
      AND COUNTRY='USA';
```

3. The WHERE clause supports flexible template matching of string values using the `LIKE` keyword. This is useful in cases where strings may appear in the database in more than a single canonical form.
4. the statement `X IS NULL` or `X is NOT NULL` can be used in the WHERE clause to restrict the retrieved records to only missing or only non-missing values of `X`.
5. Adding a clause `ORDER BY T` at the end of a SELECT statement orders the retrieved records by the value of the attribute `T`. If `T` is two attribute names separated by a comma the first attribute name is used to sort with the second attribute name used to break ties.
6. The statement `SELECT DISTINCT X FROM Y WHERE Z` is similar to `SELECT X FROM Y WHERE Z`, except that duplicates are eliminated.
7. The statement `SELECT X FROM Y WHERE Z GROUP BY Q;` is similar to `SELECT X FROM Y WHERE Z;`, but the retrieved records will have an ordering that groups records with identical values of `Q` together.

Queries Involving Multiple Tables

We can use the `SELECT` statement to query from multiple tables simultaneously. To illustrate this concept we consider the LOCATIONS table above and create an additional table ORDERS, which stores retail orders.

```
CREATE TABLE ORDERS(ORDER_ID INTEGER PRIMARY KEY,
   CITY CHAR(20),
   PRODUCT CHAR(20),
   QUANTITY INTEGER);
INSERT INTO ORDERS VALUES (5,'Chicago','CAMERAS',23);
INSERT INTO ORDERS VALUES (8,'Chicago','BATTERIES',53);

 order_id |         city  |       product    | quantity
----------+---------------+------------------+----------
        5 | Chicago       | CAMERAS          |       23
        8 | Chicago       | BATTERIES        |       53
(2 rows)
```

Note that both the ORDERS table and the LOCATIONS table have a CITY column, which holds names of cities.

To query from multiple tables we use a SELECT statement with the WHERE clause having multiple table names separated by commas. The attribute names in the SELECT statement can correspond to attribute names in any of the tables, but in case some tables have overlapping names the attributes may be disambiguated by prefixing the attribute name by the table name followed by a period. For example, `ORDERS.CITY` refers to the CITY attribute in the table `ORDERS`.

For example, the following query retrieves all combinations of STATE, PRODUCT, QUANTITY from the tables LOCATIONS and ORDERS whose CITY attributes agree. Note that the `WHERE` clause below has disambiguated CITY attribute names.

```
SELECT STATE,PRODUCT,QUANTITY
   FROM LOCATIONS, ORDERS
      WHERE  LOCATIONS.CITY=ORDERS.CITY;

        state          |       product         | quantity
-----------------------+-----------------------+----------
 IL                    | CAMERAS               |       23
 IL                    | BATTERIES             |       53
(2 rows)
```

The keywords UNION, INTERSECT, and EXCEPT can be used to execute the set operation of union, intersection, set-minus (see TAOD Volume 1, Appendix A) between results returned from separate SELECT statements. For example, the statement below retrieves all cities in Illinois from LOCATIONS that have more than 40 orders in the table ORDERS. Note that since there are two separate SELECT clauses we did not need to disambiguate the attribute name CITY.

```
(SELECT CITY from LOCATIONS WHERE STATE='IL')
   INTERSECT
(SELECT CITY FROM ORDERS WHERE QUANTITY>40);

         city
----------------------
Chicago
(1 row)
```

Subqueries

Since SELECT statements return a list of attribute values they can be used in a functional form as a nested component of an outer SELECT statement. The nesting can occur in different parts of the outer SELECT clause, including the WHERE clause and the FROM clause.

For example, the SQL code below returns all states from the ORDERS table corresponding to cities having fewer than 50 orders (in the table ORDERS). The keyword `IN` before the nested SELECT clause matches a value to a set of possible values.

```
SELECT STATE
FROM LOCATIONS
WHERE CITY IN (
  SELECT CITY FROM ORDERS WHERE QUANTITY<50);

        state
----------------------
 IL
(1 row)
```

Joins

Section 13.3.3 showed how to construct SQL queries that apply to multiple tables. Join statements in SQL are convenient short hand that simplifies such complex queries.

The statement `T1 CROSS JOIN T2` creates a new table having attributes that correspond to attributes from table `T1` and attributes from the table `T2`. A new record is created for every possible combination of rows from the two tables. Thus, if `T1` has 5 columns and 2 rows (as does the table LOCATIONS) and `T2` has 4 columns and 4 rows, the table `T1 CROSS JOIN T2` will have 9 columns and $2 \cdot 2 = 4$ rows. The SQL code below displays the `ID`, and `ORDERE_ID` columns of the resulting table.

```
SELECT ID, ORDER_ID FROM (LOCATIONS CROSS JOIN ORDERS);

 id | order_id
----+----------
 13 |        5
 21 |        5
 13 |        8
 21 |        8
(4 rows)
```

The statement `T1 JOIN T2 ON X=Y` creates a new table that combines all rows of `T1` with all rows of `T2` for whose value of `X` (on `T1`) agrees with the value of `Y` (on `T2`). `JOIN` is sometimes called `INNER JOIN` to differentiate it from `CROSS JOIN`.

For example, the following code matches records in LOCATIONS with ORDERS that have identical CITY attribute. Note that this gives the same result as the SELECT query in Sect. 13.3.3.

```
SELECT STATE,PRODUCT,QUANTITY
   FROM (LOCATIONS JOIN ORDERS
      ON LOCATIONS.CITY=ORDERS.CITY);

        state          |       product         | quantity
-----------------------+-----------------------+----------
 IL                    | CAMERAS               |       23
 IL                    | BATTERIES             |       53
(2 rows)
```

Three additional types of joins are described below.

T1 LEFT OUTER JOIN T2 ON X: First, an inner join is performed on X. Then, for each row of `T1` that does not satisfy the join condition with any row of T2, a new row is added with null values in columns of T2. The resulting table has at least one row for each row of T1.

T1 RIGHT OUTER JOIN T2 ON X: First, an inner join is performed on X. Then, for each row of `T2` that does not satisfy the join condition with any row of T1, a new row is added with null values in columns of T1. The resulting table has at least one row for each row of T2.

T1 FULL OUTER JOIN T2 ON X: First, an inner join is performed on X. Then, for each row of `T1` that does not satisfy the join condition with any row of T2, a new row is added with null values in columns of T2. Then, for each row of `T2` that does not satisfy the join condition with any row of T1, a new row is added with null values in columns of T1. The resulting table has at least one row for each row of T2.

For example, the SQL code below shows the difference between left outer join and right outer join on the tables `LOCATIONS` and `ORDERS`. Since `LOCATIONS` has two cities (Los Angeles and Chicago), while `ORDERS` has a single city (Chicago) the left outer join will have an additional row where CA is not matched on any of the rows of `ORDERS`.

```
SELECT STATE,PRODUCT,QUANTITY
  FROM (LOCATIONS LEFT OUTER JOIN ORDERS
     ON LOCATIONS.CITY=ORDERS.CITY);

        state          |       product         | quantity
-----------------------+-----------------------+----------
 IL                    | CAMERAS               |       23
 IL                    | BATTERIES             |       53
 CA                    |                       |
(3 rows)

SELECT STATE,PRODUCT,QUANTITY
   FROM (LOCATIONS RIGHT OUTER JOIN ORDERS
```

```
       ON LOCATIONS.CITY=ORDERS.CITY);

         state          |       product         | quantity
------------------------+-----------------------+----------
 IL                     | CAMERAS               |       23
 IL                     | BATTERIES             |       53
(2 rows)
```

13.3.4 PostgreSQL, MySQL, and Other Database Solutions

The most popular commercial relational database programs are made by Oracle, SAP, and Microsoft. The two most popular open source relational database systems are PostgreSQL and MySQL. Commercial database programs as well as PostgreSQL and MySQL can scale up to hold multiple terabytes of data. Scaling up to very large dataset sizes typically requires partitioning the rows of the table into multiple disks, a process known as sharding. Most standard database programs can handle heavy workloads with many users concurrently query the database.

PostgreSQL, often abbreviated as "Postgres," is a free and open source relational database management system (RDBMS) with a commercial friendly license. It is easy to install on many operating systems including Windows, Linux, and Mac OS. It is fully ACID compliant (see Sect. 13.3.2). Postgres is also extensible and has many extensions developed by third parties including support for spatial and geographic data (for example, geographic coordinates or geographic regions). PostgreSQL is used by many large corporations and is also available on the Amazon Web Services (AWS) cloud computing platform.

MySQL is a free open source popular relational database program that is currently owned by Oracle. Its GNU General Public License is less appealing for commercial use than PostgreSQL, though MySQL has several paid versions available as well. Like PostgreSQL, it is scalable to large dataset sizes and it powers many high profile websites. MySQL is generally ACID compliant, though it has a less rigorous approach to robustness and data integrity.

Both PostgreSQL and MySQL typically run as a separate process and a client application program accesses that database using interprocess communication (sometimes using network protocol). In other words, the application that queries or updates the database runs as a separate process and must interact with the database process using a service call. SQLite is another open source ACID compliant database that takes a different approach. In contrast to PostgreSQL and MySQL, SQLite is typically an integral part of the client process. Because of its small size and simple accesses, SQLite is used by internet browsers (Google Chrome, Mozilla Firefox), mobile operating systems (Google Android, Apple iOS), and applications (Skype, Adobe Reader).

13.3.5 Working with Databases: Shells and Programmatic APIs

The sections above demonstrate relational databases and SQL through an interactive prompt. Many relational databases provide an interactive shell where the user can type database creation, alternation, or query commands and see the results that they produce. For example, the examples above were typed into the PostgreSQL interactive shell.

In real-world applications, databases are often queried and updated from computer programs written in languages such as C++, Java, Python, and R. Each of these languages (and many other languages as well) has libraries that support querying and updating relational databases. Below, we illustrate that process by calling SQLite from a Python program. Calling PostgreSQL or MySQL from Python is similar but requires an extra initial step of authenticating and establishing a connection with the database process or database server (if the database resides on a networked machine that is separated from the machine running the client program). Calling databases from other languages require some adaptation but is mostly similar.

The Python library `sqlite3` provides convenient access to an in-memory SQLite database. After establishing a connection object, the next step is to create a string representing the database command (for example, table creation, table update, or query). The connection object then sends the string for execution by the database and returns the results back to the Python program. The following code illustrates this process by first creating a table, then adding two rows, and then sending a query and displaying the results. Note that in Python we can use triple quotes to create multi-line strings.

```
import sqlite3
import pandas.io.sql
# create a connection
con=sqlite3.connect(':memory')
# create a table
createStmt = """
  CREATE TABLE LOCATIONS(ID INTEGER PRIMARY KEY,
       CITY CHAR(20),
     STATE CHAR(20),
     COUNTRY CHAR (20) NOT NULL);"""
con.execute(createStmt)
con.commit()

# add two rows to the table
data = [(13, 'Los Angeles', 'CA', 'USA'),
        (21, 'Chicago', 'IL', 'USA')]
addStmt = "INSERT INTO LOCATIONS VALUES (?, ?, ?, ?)"
con.executemany(addStmt, data)
```

```
con.commit()

# query the databse
data = con.execute('select * from LOCATIONS')
print(data.fetchall())

## [(13, u'Los Angeles', u'CA', u'USA'),
##  (21, u'Chicago', u'IL', u'USA')]

print(data.description)

## (('ID', None, None, None, None, None, None),
##  ('CITY', None, None, None, None, None, None),
## ('STATE', None, None, None, None, None, None),
## ('COUNTRY', None, None, None, None, None, None))
```

Above, the SQL query returned to Python an object `data` that holds both the table (via `data.fetchall()`) and the column names (via `data.description`). The Python Pandas package contains a convenient way for creating a Python dataframe from that object.

```
DF = pandas.io.sql.read_frame(
  'select * from LOCATIONS', con)
print(DF)

##    ID         CITY STATE COUNTRY
## 0  13  Los Angeles    CA     USA
## 1  21      Chicago    IL     USA
```

13.4 NoSQL Databases

With the increased demand of the massive scale of big data, thanks to Web 2.0, which caused an explosion of user-generated content on the Internet, relational databases had a hard time coping. NoSQL is a different approach for storage and retrieval of data that doesn't rely on relational data; originally, NoSQL meant non-SQL, but that evolved to mean not only SQL to indicate that SQL can be used with the non-relational data for querying. NoSQL storage systems typically follow a simpler design, compared to their relational counterparts, which helps with scaling out (horizontal scaling) by adding more machines to a cluster.[2] The key distinction and the most important criterion when choosing between relational and NoSQL storage systems is the shape of the data: relational systems work best for tabular

[2]More on how to design scalable architectures in Chap. 14.

data with obvious relationships (foreign keys) while NoSQL systems work best for data whose shape morphs and evolves. That doesn't exclude NoSQL from being the proper choice in scenarios where data are tabular but the scale is just too prohibitive to use a relational database. The most common data structures that NoSQL systems support are key-value records and documents (a set of fields with a key). In NoSQL, since the data is not necessarily tabular, the values stored can have a very flexible schema (or no schema at all); it's typically the responsibility of the client to interpret the value (akin to loosely typed programming).

Earlier in this chapter, while introducing the ACID properties in Sect. 13.3.2, we talked about the importance of consistency in database management systems. Because of such strict constraints and guarantees that relational databases provide, things tend to slow down when dealing with a massive data set at high throughput. NoSQL takes a different approach in order to be able to cope with big data: relaxation of consistency in favor of performance and other desirable system properties; instead, they may be inconsistent for some periods of time and they guarantee that consistency will be restored eventually; this is known as "eventual consistency." The reasoning behind such approach is the requirement of scaling out, which means that data need to be distributed over multiple machines, usually with multiple degrees of redundancy; in turn, that requires changes to said data to propagate, and while that's happening, the data might be inconsistent.

SQL makes it easy to work with relational databases because most of them, if not all, support the standard/ANSI SQL syntax. On the other hand, there's no such standard for NoSQL databases; instead, each system provides a set of APIs that are considered primitive compared to the powerful features of SQL. For example, there's no built-in join operation; developers have to implement those operations themselves on the client-side. In Chap. 14, we discuss an example of performing joins using RocksDB (a key-value store). Some databases, like VoltDB and Google's Spanner, were designed to provide the massive scale of NoSQL while supporting the relational data model, SQL as a method of interfacing, and maintaining the ACID properties.

For a more in-depth look into NoSQL databases, we recommend reading Chap. 14 first to know more about their architectures then referring to the Amazon DynamoDB example in Sect. 14.15.2.

13.5 Memory Mapping

Memory mapping is a technique to associate (map) a segment of virtual memory in a way that allows applications to see the mapping as if it were main memory. This can be useful for mapping a file on disk, a shared memory object, or a device. The main benefit of doing so is improving performance because an application doesn't have to go through system calls, which are significantly slower; instead, it simply accesses the mapped memory as if it's local to the application. The vast majority of modern operating systems support memory mapping; in POSIX-

compliant operating systems, like Linux and Mac OS X, the `mmap` function can be used to achieve that; Microsoft Windows also support memory mapping using the `CreateFileMapping` function. Memory mapping is also widely supported in most programming languages like C++ (via the Boost library, among others) and Java (via the `FileChannel`, which we use in examples in Sect. 10.11.3), Python (via the `mmap` module), and R (via the `bigmemory` library).

13.6 Notes

To learn more about SQL, we recommend reading (Forta, 2013) for a quick tour; for a more of an in-depth read about database systems, we recommend (Elmasri and Navathe, 2011).

References

B. Forta. *Sams Teach Yourself SQL in 10 Minutes*. Always Learning. Sams, 2013. ISBN 9780672336072.

R. Elmasri and S. Navathe. *Fundamentals of Database Systems*. Addison-Wesley, 2011. ISBN 9780136086208.

Chapter 14
Thoughts on System Design for Big Data

In the context of computing with data, what exactly is a system? Generally speaking, a system is an aggregation of computing components (and the links between them) that collectively provide a solution to a problem. System design covers choices that system designers make regarding such components: hardware (e.g., servers, networks, sensors, etc.); software (e.g., operating systems, cluster managers, applications, etc.); data (e.g., collection, retention, processing, etc.); and other components that vary based on the nature of each solution. There's no free lunch in system design and no silver bullet; instead, there are patterns that can jumpstart a solution; and for the most part, there will always be tradeoffs. Skilled system designers learn how to deal with novel problems and ambiguity; one of the skills they practice is decomposing a complex problem into more manageable subproblems that look analogous to ones that can be solved using known patterns, then connect those components together to solve the complex problem. In this chapter, we put on our designer hats and explore various aspects of system design in practice by creating a hypothetical big-data solution: a productivity bot.

14.1 Where to Start?

System design is a means to an end: solving a problem. Therefore, it makes sense to start with codifying the problem, nailing down the requirements, and clarifying ambiguities. Thus far in this book, we have been focusing on programming and coding; they are useful tools, when wielded aptly, that enable us to execute. The challenge is planning ahead: deciding which tools to use and how. One can use a screwdriver as a crowbar, but that may damage the screwdriver and fail to achieve the goal. "Coding skill is just one small part of writing correct programs. The majority of the task is [about] problem definition, algorithm design, and data structure selection." Those wise words in *Programming Pearls* (Bentley, 2016),

© Springer Nature Switzerland AG 2018

G. Lebanon, M. El-Geish, *Computing with Data*,
https://doi.org/10.1007/978-3-319-98149-9_14

which we highly recommend, were first written by Jon Bentley years ago and still apply today. We claim that system design subsumes algorithm design and is an overarching principle that needs to be skillfully practiced for big-data system to thrive. To solve big problems, other principles and disciplines, like project management, need to be implemented in unison—design, in comparison, is the work of maestros.

There are many approaches to managing the software development process, the choice of which may impact the choices you make in the context of system design. That said, in the spirit of focusing on system design per se, we marginalize the development process in the context of this chapter and simply assume an agile development method for our project. For the enthusiastic reader, we recommend the following books about software project management and development processes: (Hunt and Thomas, 1999), (Brooks, 1995), (Cohn, 2005).

Based on what approach you take, you may seek answers to as few—or as many—questions to get started with designing your system. Chapter 15 delves into the details of how to strike a balance between the two extremes. Using our productivity bot as an example, we can start with obvious questions like:

- **What are the functional requirements for the system?** We want a bot that can manage reminders for our users. For the minimum viable product (MVP), the bot and our users can only interact using email; for example, a user can email bot@computingwithdata.com to add a reminder using syntax like "remind me today at 4 pm to buy flowers" and the bot would send an email to the user at 4 pm with the reminder.
- **How do we identify users?** By email address. In the future, we may want to allow users to link multiple email addresses to a single account.
- **What input and output behaviors are required?** For the MVP, we will only consider the email's subject line to specify a reminder.

If you were to hack a quick proof of concept for this project, you may consider the above sufficient to start hacking away. System designers, who like to measure twice and cut once, would follow up with many questions like:

- **How many users do we need to support?** For the MVP, thousands; later on, we want to support hundreds of millions.
- **What are the availability and latency requirements?** We define the following service-level agreement (SLA) metrics: the system should be available 99.9% of the time; we define availability as the system's ability to process transactions while meeting the latency requirement of a ten-second maximum delay.
- **What are the transactions specified in the SLA?** Processing of incoming emails means that valid reminders are parsed and persisted while failures are handled and communicated back to users. On the other hand, processing of outgoing emails means that the emails carrying the timely reminders get accepted by the users' email servers.

After collecting enough answers to key questions, system designers would ask themselves questions that pertain to design choices they have to make, review the

decisions with other stakeholders, and possibly ask for change in requirements to meet certain constraints, like deadlines, and/or to maximize the return on investment (ROI) of certain features. In the following sections, we explore such design choices and how to think about them.

14.2 The Big Picture

Although it sounds like a clicheé, that's how most system designers start thinking about a problem: starting with a blank whiteboard, they would draw boxes and arrows that describe the big picture of various system components and how they interact with each other. Then, depending on the software development processes they follow, designs are formalized—possibly using the Unified Modeling Language (UML)—and documented. System architects may choose to design the coarse-grained components and leave the fine-grained details to developers to figure out; ideally, those two roles should be conflated to ensure that designer don't live in an ivory tower and design castles in the sky.

Let's take a look at one possible architecture diagram for our productivity bot system:

Looking at Fig. 14.1, we can follow the flow of data and operations that address the project's requirements: an email arrives from the Internet into one of our email

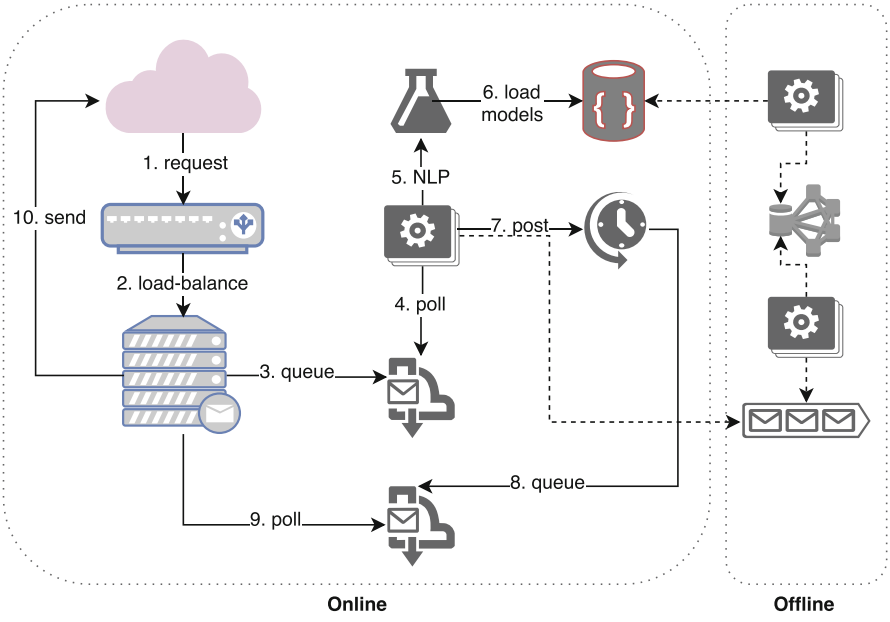


Fig. 14.1 The system architecture diagram for a possible solution to our problem

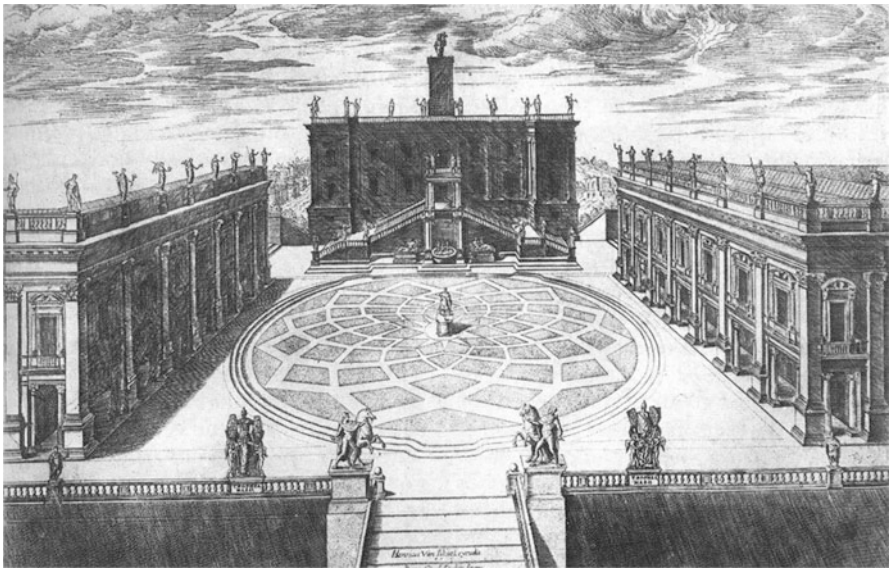


Fig. 14.2 Michelangelo's design of Capitol Hill in Rome, Italy. This work is in the public domain in its country of origin

servers, which in turn queues it up for another service to pick up when ready; in turn, that service communicates with other services to resolve and transform entities from the email in order to post them to a scheduler, which queues an outgoing email—at the desired time—for the email server in order to send it to the user. All of that happens in the online realm, where real-time systems[1] live; on the other hand, offline systems[2] aggregate data requisite for various supporting services like analytics and model training.

We just summed up the high-level design of the system in a few lines, which may take thousands of lines of code to implement; and if we considered the underlying infrastructure on which it stands, we're talking about millions of lines of code to run our system! However, the most important job for the architecture diagram to achieve is communicating the high-level ideas and concepts behind the design; the goal is similar to that of an architecture sketch, which shows how a building would look like from afar while leaving the details for other detailed plans like blueprint schematics (Fig. 14.2).

[1]Real-time systems are required to process transactions timely; they must meet certain service level agreements (SLAs) of availability and performance; we're talking about a couple of seconds of latency at the worst case.

[2]Offline systems handle big batches of data, as in ETL (Extract, Transform, Load) jobs; typically, offline systems are not connected to the Internet.

The missing details will be clarified later on after the designers approve the big picture. There's no strict rule for how granular an architecture diagram should be; we suggest to follow Antoine de Saint-Exupéry's quote: "A designer knows he has achieved perfection not when there is nothing left to add, but when there is nothing left to take away." That statement also sums up a key principle that designers should follow when thinking about systems: simplicity, which Gordon Bell in one of his quotes: "The cheapest, fastest, and most reliable components of a computer system are those that aren't there" (Bentley, 2016).

Many readers may wonder: If simplicity is so desirable, why did we create many subcomponents that can be merged together in monolithic solution? The short answer is: scalability; one of the main requirements is the ability to scale to server—orders of magnitude—more users. Another subtle requirement eluded to the potential of extending the interaction model, which is currently restricted to email, to include other modalities like instant messaging. Hence, building the solution as a substrate that can be quickly reshaped—while maintaining the cost of doing so in check—is akin to killing two birds with the same stone. In our case, keeping those single-purpose boxes (microservices) small and generic actually promotes simple design and separation of concerns[3] (SoC). In the following sections, we explore various aspects of system design, infrastructure, and frameworks that we can use to build our solution.

If your goal is to only process a few requests a minute, your system need not to worry about scale; depending on the criticality of your system, you may choose to simply install your program on a single machine and call it a day; that's not the case with big data though. Processing data at scale requires considerations of performance, throughput, and correctness. Big data are processed using systems that are distributed; some are what we call "embarrassingly parallel," which means that they require little to no effort to distribute the processing over multiple machines; others require coordination and careful planning to avoid the pitfalls of concurrency.

14.3 Load Balancing

Following the lifecycle of data in our system, we can see that requests from the Internet hit a load balancer as the first point of contact. The load balancer distributes the incoming requests across multiple machines, we will call them nodes, based on an algorithm that the system designer picks. Common objectives of load balancing algorithms include maximizing throughput and/or performance. A simple algorithm for load balancing is round robin: The load balancer picks the next node from a rotation list to handle a new request; if the list is exhausted, it goes around to the first one, and so on. The underlying assumption here is the nodes are homogeneous and the requests are equally weighted, but that's hardly the case. A custom distribution

[3] https://en.wikipedia.org/wiki/Separation_of_concerns.

list can specify how much load each node should handle compared to others; that's usually correlated to the computing resources available for each. For example, if the processing of a request is CPU-intensive and we have three SKUs[4] of nodes that vary in CPU performance, we can assign each class a relative weight, such that a higher number means more traffic routed to the corresponding SKU.

Understanding the bottleneck of each system is a major factor to consider when picking a load-balancing algorithm; a memory-bound system may require a different policy than a CPU-bound one. Here are a few examples of algorithms that are commonly used for load balancing:

- **Weighted Round Robin:** a variant of the round-robin algorithm, which distributes requests based on their number and normalized weight (weight divided by mean request size).
- **Weighted Total Size:** new requests are routed to nodes in a manner that balances the distribution of total size of requests handled by each node to keep said distribution as uniform as possible; in other words, the node selected to receive the next request is the one with the smallest tally of payload size in bytes.
- **Least Response Time**: maximizes performance by selecting the node with the fastest response time.
- **Sticky Session**: requests from the same client go to the same server as long as the client's session is active; this approach maximizes locality of reference for session data stored or cached on a server.

The above algorithm depends on statistics collected during a sampling time window, which can be configured to suit each solution's need. The key point here is that said statistics can be observed from the load balancer, which only needs very little about each node (e.g., know how to connect to each node). A more sophisticated load balancer would know more about each node to keep track of its readiness to receive a new request; using probes, it can tell whether a node is alive, what ports are responding timely, how much of the node's resources are free, etc. The more a load balancer knows about the nodes it manages, the better it can distribute the incoming workload.

In many system configurations, a load balancer is a dedicated hardware device (appliance) that does more than just load balancing. For example, BIG-IP appliances[5] can offload SSL processing to relieve the nodes from spending CPU cycles on decryption and encryption; in addition, they also provide security features like a firewall and protection against distributed denial-of-service (DDoS) attacks. The name that better describes this class of appliances is not load balancers but rather Application Delivery Controllers (ADCs).

An alternative to appliances is software load balancers (SLBs), which can run on nodes alongside other services (or on dedicated servers); they tend to be much

[4]Stock Keeping Unit (SKU) represents a class of items, like machines in a data center, and all of its respective properties that are used for inventory tracking.

[5]https://www.f5.com/pdf/products/big-ip-platforms-datasheet.pdf.

cheaper than appliances since they can run on commodity servers. One well-known example of an SLB is the NGINX Plus web server's built-in load balancer; its software-based ADC provides capabilities like application-aware probes, DoS protection, and content caching. There are many ADCs out there, software- and hardware-based, that provide a plethora of features in order to ensure that our systems are highly available. The following companies are known for providing such solutions: Barracuda, Cisco, Citrix, F5, NGINX, and Amazon; each solution has its own pros and cons—there will always be tradeoffs.

In our system, the nodes setting behind the load balancer run Simple Mail Transfer Protocol (SMTP) services, which are responsible of handing inbound and outbound emails. They don't need to share any state among themselves or know about each other; each can think independently as if it's the only SMTP server in the system—just like how it runs on a developer's machines. It's the responsibility of the load balancer, and other services like a cluster manager, to make sure that a sufficient number of SMTP servers are up and running. When the SMTP traffic spikes, the cluster manager service can change the role of nodes in the cluster to serve SMTP traffic instead of lower priority work they were assigned before.[6] Also, if a node goes down for any reason, another can be brought in to replace the broken one without suffering a loss of disk-persisted data (because there isn't any). In order to scale horizontally and achieve service elasticity, by dynamically changing the nodes on the load balancer's rotation list, the email servers must be stateless (none of the messages is stored on disk); in addition, there shouldn't be any bottlenecks that can get overloaded because of the increase of SMTP nodes routing messages downstream. In order to understand the second prerequisite in detail, we explore a couple of options to how work can be distributed between different layers of a system.

14.4 Partitioning

The SMTP nodes in our system can scale out as far as the supporting infrastructure can withstand; otherwise, the entire system topples to the floor. To draw an analogy, imagine that the incoming traffic is a bunch of shipping containers that need to be transferred; one may stack them on top of a single trailer, which will collapse from the overwhelming load; or, one may replicate the process of shipping a single container on a single trailer multiple times instead. The system design we explore here is similar: we want to design a stencil for our online system that can be easily cloned; each clone is self-contained and handles a partition of the incoming traffic; this way, the problem becomes—almost—embarrassingly parallel once we figure out a good scheme for partitioning. Now, scaling horizontally means adding new clones of the online system, which usually requires some housekeeping due to

[6] Google uses Borg (https://research.google.com/pubs/pub43438.html) for that.

change in the number of partitions. One possible solution, using partitioning, looks like the diagram in Fig. 14.3. In addition, in our case, we still need to figure out how to scale out the offline system along with the online one; although we can include an offline cluster inside a partition, each partition won't be self-contained as operations performed on offline data usually require aggregation across all partitions. Hence, it's much more streamlined to keep the offline cluster intact to avoid inter-cluster data transfers, which can get expensive. We will discuss how to solve this problem later in this chapter.

In order to partition online traffic, we need another layer of routing that determines to which cluster a request will be assigned. Note the difference between the roles of each routing layer: the first layer picks a specific cluster, which is mapped to the request using some function; once routed to that cluster, the second layer picks—based on a load-balancing algorithm—an SMTP server inside the cluster to handle the request. Figure 14.3 depicts this setup, which is agnostic to whether the first layer is implemented using a hardware- or a software-based load balancer (or partitioner); either way, the load balancer needs to inspect the request to find a key that's going to be used for partitioning.

One option is to map the domain name of the user's email address to a cluster: Every time the system sees a new domain name, it maps it in a round-robin fashion to a cluster and stores that mapping for future lookups. That way, we end up with clear mapping from email address to cluster index. However, the load and number of requests getting routed to each cluster will be heavily unbalanced since the number of users is not evenly distributed across said domains; instead, we expect to see much more outlook.com users than abc.xyz users.

In our case, a better choice for a key, let's call it k, is the uniformly distributed output of a hash function that takes the user's email address as input. Uniformity is extremely desirable because we'd like to achieve a balanced distribution of user assignments to the n clusters as we determine the index of the cluster to which a user is assigned to be: $k \mod n$. There are subtle issues that developers need to worry about when choosing a hash function that's well suited for partitioning; for example:

- **Negative k**: It's possible for hash functions, like Java's `hashCode()`, to return negative hash codes. When k is negative, the index $k \mod n$ is a nonnegative value. However, when implementing the mod operator in languages like C++ or Java, it can be easily confused with the remainder operator, `%`, which can return a negative index when calculating $k\%n$. For example: `-17 % 5 = -2`. In Python, on the other hand, `%` is the modulus operator and its return value cannot be negative; for example, `-17 % 5 = 3`. To prevent this issue, implement the mod operator yourself when coding in languages that don't support it out of the box.
- **Poor Bit Dispersion**: Many languages, like Java and C#, provide a default implementation for an object's hash code getter function, which tends to be fast as it converts the internal memory address of the object into an integer. The downside of that manifests in poor bit dispersion or distribution of the hash code over the range of all possible values.

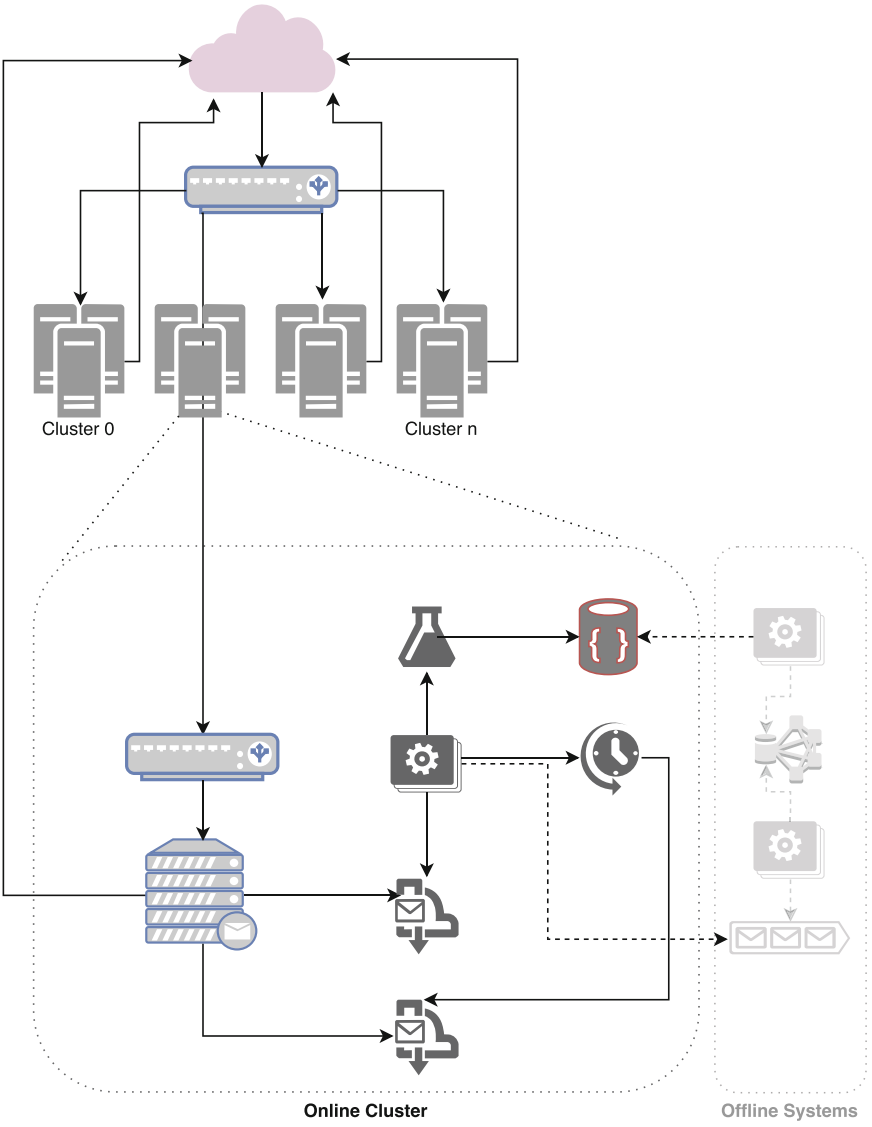


Fig. 14.3 Partitioning incoming traffic and routing each subset to an online cluster that serves our bot system

- **Lack of Interoperability:** Different languages provide different default hashing functions; for the same string, Java and Python return different hash codes. Changing the language of the routing layer can have a significant impact on where the users' data are stored.

- **Inconsistency**: Some hash functions, like Java's `hashCode()`, are not guaranteed to return the same output for the same input when invoked on different machines; even worse, the output—for the same input—may differ from one execution of the program to the next. To prevent these issues, a more reliable hashing function like MurmurHash.[7] That said, the hash code of certain types, like `string` in Java, tends to be stable across all of the above variations; for a given string s, the hash code in Java is:

$$\sum_{i=0}^{n-1} s[i] \times 31^{n-1-i}$$

Now that we have picked a key and a hash function, let's walk through a couple of examples: Assuming that we have five clusters ($n = 5$), a request from `"mohammed@outlook.com"`—using the above string hash function in Java—gives us a hash code ($k = -1543695718$), which maps to the third[8] cluster ($-1543695718 \mod 5 = 2$); a subsequent request from `"steve@me.com"` maps to the first cluster ($-1711373000 \mod 5 = 0$), and so on.

To illustrate how this partitioning scheme allows for a balanced cluster assignment, we used 67802 email addresses from the Enron dataset[9] to simulate arbitrary email addresses (keys) and we calculated how they would be assigned across our 5 clusters using the Python script below:

```
N = 5
counts = [0 for i in range(N)]

def route(email):
  return hash(email) % N

for email in open('enron.txt'):
  counts[route(email)] += 1

print(counts)
## [13597, 13429, 13608, 13660, 13508]
```

This solution may work in cases where the data doesn't grow and all servers stay up to fill the entire hash space. System failures are inevitable and nodes will go down; with more email addresses being added to the system, which is a desirable effect, we face a problem when we need to add more nodes to handle the extra load: reassignment of users due to changing N causes so much data movement, which can be costly (in both I/O operations and maintenance time). This has to be done every time a new node is added to—or removed from—the system! We leave the exercise

[7] https://en.wikipedia.org/wiki/MurmurHash.

[8] Recall that we're using a zero-based index.

[9] https://en.wikipedia.org/wiki/Enron_Corpus.

of counting how many swaps it takes to increment N in the previous example to eager readers.

One solution to this problem is consistent (aka ring) hashing, which we discuss in detail below.

14.5 Consistent Hashing

Consistent hashing is a load-balancing technique that's been used in many distributed systems like Amazon's Dynamo storage system, Apache Cassandra, LinkedIn's Voldemort key-value store, Akamai's CDN (Content Delivery Network), etc. Its history goes back to 1997, when David Karger et al. at MIT introduced the term "consistent hashing" in an academic paper titled, "Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web" (Karger et al., 1997), which demonstrated a key application of consistent hashing to load-balance requests among a changing set of web servers. Large web applications rely on caching to relive the application servers from requests that can be satisfied from the cache (e.g., static files) and they benefit handsomely from consistent hashing, which only requires K/N keys to be reassigned when the set of nodes changes (where K is the number of keys). In the case of web caches, this is a necessary requirement; otherwise, the application servers would be bombarded with requests to populate the cache nodes whenever N changes (almost all keys have to be remapped, which defies the purpose of having caches altogether).

So how does consistent hashing drastically reduce the number of key reassignments? The hash ring—hence the name—maps keys to points on the ring, which subsumes a large number of points (P), using a hash function (h_k); and also maps nodes to points on the ring using another hash function (h_n). To find the node that is assigned to key k, we find the point $h_k(k)$ on the ring and then walk clockwise along the ring to find the next point that maps to a node. In doing so, each node handles a range of points; hence, distribution of the hash values over the range of points P is important to balance the load among the nodes equally. In addition, we can add a node to the ring multiple times using a factor (R) such that a node's identifier is composed of its name (or IP address) in addition to a value r between 1 and R; when resolving the node's name, we remove r from the identifier.[10]

To add a new node to the system, we map the node N to a point $p_N = h_n(N)$, which is now responsible for handling requests whose keys map to any point in the node's range. Adding this new node only impacts the key assignments of its next neighbor in the ring (clockwise). In the case of a web cache, those keys that used to be mapped to the neighbor need to be invalidated since they are now served from the new node N. Assuming an equal distribution of load, only K/N keys to

[10] r doesn't have to be uniform across nodes; it can be used as a weight assigned to a node or a SKU depending on the machine's capacity (or other factors).

be reassigned when N changes; the same scenario applies when a node is removed from the system. In practice, each node is mapped to r pseudo-randomly distributed points such that the removal (or addition) of said node impacts a fraction of the keys assigned to its neighbors.

14.6 Scatter-Gather

Let's assume that we picked a different key scheme, which is the uniformly distributed output of a hash function that takes the user's reminder time—instead of her email address—as input. To query for a user's reminders, we broadcast (scatter) the query to all servers, each of which may contain fragments of said user's reminders, and then reconstruct (gather) the response from those fragments, which are usually indexed by the key we're querying for (in this case, the user email's address) to speed up the query. Services that execute the fan-out query and assembly of the response are called dispatchers or brokers. Brokers may apply further processing of the results (e.g., sorting) if need be. The scatter-gather strategy of storing and retrieving data is also known as query-on-demand or fan-out-read-model. LinkedIn feed uses this strategy to store and retrieve LinkedIn members' activities to show on the homepage and elsewhere.[11] Google, and other search services, use a similar strategy to construct the search results for a user's query. For example, Twitter uses a modified version of Apache Lucene to index tweets in memory; its search service, called blender, scatters the search query across index shards hosted on various machines and gathers the results for ranking, de-duplication, etc., and then serves it back as the query's result.

14.7 Pre-Materialization

Another strategy for storing and querying data is known as pre-materialization or fan-out-on-write model; as the name suggests, writes are fanned out to various buckets to allow read-heavy to query for data faster (than using the scatter-gather model). Twitter uses this strategy to store tweets and query for users' time-lines: When a user tweets, a fan-out service iterates over the user's followers and appends a reference to the tweet into the follower's time-line's storage bucket (in a Redis cluster with a replication factor of 3); the tweet is also stored in a MySQL database in the back-end. This strategy is sometimes called the inbox model since it's similar to how emails are sent and stored across various mail transfer agents. When a user visits Twitter to view her time-line of tweets posted by other users she follows, the

[11] https://engineering.linkedin.com/blog/2016/03/followfeed–linkedin-s-feed-made-faster-and-smarter.

time-line service requests the user's time-line from the three hash rings in parallel, each of which needs to find and relay the request to a single server (since the time-line is not scattered); the time-line service returns the result that came back the fastest (since they are replicas of each other, any of them would suffice).

Pre-materialization requires much more storage space than scatter-gather, but it works well for products like Twitter where the vast majority of users read tweets rather than post them.[12] If a user's time-line is lost for some reason, it gets reconstructed using scatter-gather materialization from the back-end services that store all tweets; and then cached in the Redis cluster. Caching the time-line for read optimization allows Twitter to serve millions of users their respective time-lines rather efficiently. Pre-materialization is unfit for other services, like search, where the results are dynamically gathered; that's why Twitter uses scatter-gather for its search service instead.

14.8 Blackboard

Another example of data processing at scale is the blackboard (or shared-space) model, where work is put into a shared space (the blackboard) and workers keep monitoring it, do some work, and share back their respective partial results. This model was first introduced in speech recognition projects; for example, multiple threads process segments of the audio to be recognized and share the words they recognized on the blackboard; then a consumer process reads the words produced on the blackboard for further processing. At Voicera, we use the same model for many tasks; for example in speech recognition and indexing. We split an audio file that needs to be transcribed into shards, each of which is processed in parallel; when all shards are transcribed, another process stitches the results and performs other tasks like auto-punctuation and auto-capitalization. Another example we have at Voicera is a speech-indexing service that enables users to search within audio; one key difference here is indexing is an incremental process and doesn't require all shards to be completed for downstream tasks to start working on the available data. Similarly, the indexing service shards an audio file and processes shards in parallel; whenever it makes progress, it posts the output to the shared space for workers (that get notified via a message queue) to perform other tasks (for example, extract phrases the user may be interested in). Downstream tasks may decide to wait for a certain percentage of work completion before proceeding with their own processing; for example, to produce a word cloud for words that represent the recording of a meeting at a glance, the word-cloud process may choose to wait for the entire recording to be processed.

[12] According to a tech talk in 2013, the write rate was about 4K requests-per-second (RPS) while the read rate was about 300K RPS.

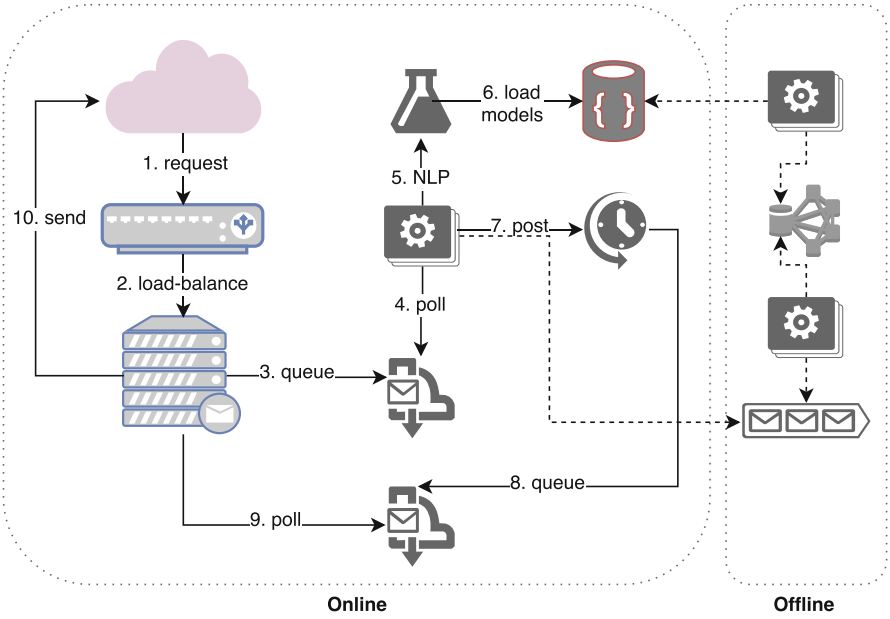


Fig. 14.4 The system architecture diagram for a possible solution to our problem

14.9 Pipelines

Just like function decomposition in a program, a system may be broken down into stages of a pipeline; a stage's output may be used as input to the next akin to a physical pipeline. This model is also known as a workflow or pipes and filters, where filters refer to the processing components and pipes are the connections between them. Inside our example productivity bot's system architecture from earlier in this chapter, we used the pipes-and-filters model to process data at scale.

Looking again at Fig. 14.4, we can follow the flow of data and tasks: email servers queue up incoming email messages for another service to pick them up when ready; later on, results are queued up to become outgoing email messages for the email servers to send them to users. Instead of processing the entire flow in a monolithic process, message queues allow system designers to break work into smaller pieces, each is handled by a microservice, which can be scaled independently from other microservices. Microservices, or filters, may evolve over time in ways that require different scale requirements; some may become CPU-bound and have to run on compute-optimized hardware or horizontal scaling; others may become memory-bound; etc. At Voicera, we use Kubernetes to scale microservices that consume messages from message queues hosted by AWS (Amazon Web Services) based on the number of messages in their input queues; when the load increases, Kubernetes automatically increases the number of instances of a microservice and/or up their resource allocation (CPU, memory, etc.). Increasing the number of instances of a

filter allows for parallel processing in order to increase the system's availability and throughput by distributing the load and eliminating bottlenecks in the pipeline; it's important to cascade capacity increases, proportionally, downstream so that bottlenecks aren't just shifted elsewhere. This pattern is known as the Competing Consumers pattern.

Another feature of pipes and filters that's supported by the Simple Queue Service (SQS) on AWS is fault tolerance: When a filter (consumer) reads a message, the message doesn't go away forever, but rather we can think of the read operation as a tentative pop. Consumers have to delete a message explicitly to remove it from the queue; otherwise, a message is only hidden for a configurable period of time (called the visibility timeout) when a consumer receives it. This feature allows consumers to read a message in an attempt to process it; if the processing fails, consumers don't delete the message and it becomes available again for consumption (by any consumer) after the visibility timeout. So what happens for messages that never get deleted by consumers (for example, ones that always fail to be processed)? The queue retains messages for a configurable period of time known as the message retention period, after which SQS automatically deletes the message. By setting those two timeout values, we can control how often to retry a failed message-processing operation and for how long before giving up on the message. As we mentioned before in previous discussions about fault tolerance, retrying may be ineffective in some failure modes; in such cases, the proper remediation is to alert a human and delete the message from the queue. Intermittent failures should also be monitored; humans should be alerted if the message consumer failed to process a message after a configurable threshold of attempts (which may vary based on the task at hand, the message's content, the failure mode, etc.); such decisions are an important part of keeping the pipeline operational as expected.

Notice how in the case of SQS, the state of the queue is global for all consumers: a message is either there or not at any point of time; the message queue is a shared resource and all consumers see the same view of the queue all the time. This model is different from a publisher-subscriber queue systems, like Kafka, which was created by LinkedIn and has been widely used for data processing at scale at many big data companies. Kafka allows publishers to pick a key for each message they publish into a topic (akin to a message queue) so that the message is directed to a partition that maps to said key; if a key is not specified, Kafka uses a round-robin algorithm to balance the load. A subscriber subscribes to consume messages from a partition of a topic; Kafka keeps track of how much progress a subscriber has made (its read offset, which Kafka advances automatically with every message it reads).

Let's draw analogies to demonstrate the differences between the two models exemplified by SQS and Kafka. SQS adopts a model that's analogous to a bank where customers stand in a single line waiting for any of many tellers to receive them. On the other hand, Kafka's model is analogous to a registration table at an event where an individual is assigned to one of many lines based on the first letter of her name; each line is assigned an organizer to receive guests; the line moves when an organizer processes the registration of a guest, who has to wait in her assigned line even if other organizers are not as busy as the one assigned to her line. Picking

a model that serves your system best is key; for example, one system may require partitioning by a specific key scheme (like domain name) to keep data that belong to the same key close to each other (locality of reference) and speed up queries for data in the same partition.

Most message queue systems are made to process data at scale; they are distributed systems that guarantee a certain SLA of high availability, which is usually achieved by using multiple machines for redundancy. That's why many message queue systems elect to err on the side of resending messages (rather than not discarding messages) in case the distributed system that hosts the queue is in an inconsistent or a failure state. This behavior is called At-Least-Once Delivery. For those cases where messages might be resent, filters/consumers should be idempotent: producing the same results when executed multiple times and doing so without producing any side effects different from those of a single execution; an example of a non-idempotent operation would be inserting the same record into a table that will cause a unique constraint violation.

The following comparison explains the differences between different message delivery guarantees:

- **At Least Once:** Retry policies cause some duplicate messages to guarantee that at least one of them was successfully enqueued. This usually happens when a failure interrupts one of the nodes in the distributed message queue system; the node may fail to acknowledge to the producer of a message that it was delivered and stored into the queue; so the producer, which requires unanimous acknowledgment from all nodes that host the queue, retries to resend the same message.
- **At Most Once:** This one is similar to the above except that the producer doesn't retry for fear of duplicate messages.
- **Exactly Once:** This is what most developers want when using a message queue; regardless of failure conditions, a message is delivered and that's done only once.

Apache Kafka, since version 0.11, supports exactly once message delivery. SQS introduced first-in-first-out queues in 2017 to guarantee message order and exactly once delivery as well. In conclusion, we find that the pipeline model—in any of its various flavors—is extremely common and is strikingly useful for improving systems' availability, resiliency, throughput, and scalability.

14.10 Redundancy, Recovery, and High Availability

Let's take a failure scenario in our system as an example: a cluster may go offline due to a networking issue; what happens to the system now? The requests that used to be served by this partition will be rejected; the system now suffers from a partial outage and it's considered "down" or unavailable. Availability (or uptime) is proportion of time a system is up and running; it's a metric that systems engineers must track and

monitor[13] since it's a strong requirement for any system. For long-running systems, availability is usually measured annually as a percentage of uptime; the number of 9's in said percentage is a common measure of availability:

Percentage	Maximum Annual Downtime
90% (one nine)	36.5 days
99% (two nines)	3.65 days
99.9% (three nines)	8.76 h
99.99% (four nines)	52.56 min
99.999% (five nines)	5.26 min
99.9999% (six nines)	31.5 s
99.99999% (seven nines)	3.15 s

For systems that require high availability, the number of 9's required can go up to eight or even higher; that requirement is known as a Service Level Agreement (SLA). An SLA may specify a multitude of requirements including what availability means; a system can be up but completely—or partially—unavailable. Take our bot system for an example, when the requests are partitioned to be served by independent clusters, a failed cluster can cause the system to be partially unavailable. To achieve high availability in said failure scenario, we need to plan for failure; one of the most important lessons senior engineers learn about distributed systems is that things will eventually fail and there's no such thing as a continuous happy path. One approach we can take is redundant hash rings, each uses an independent hashing function to assign nodes to ranges on the ring (and consequently, keys to nodes); by overlaying those redundant rings on top of our physical nodes, we can achieve some degree of redundancy. There are multiple configurations we can employ to achieve high availability for our bot system:

- **Active/Active:** the incoming requests are routed to all redundant rings simultaneously, and the fastest response is relayed to the client. We need to configure system monitors and probes to know when the redundancy factor falls below a desired threshold; we don't want to wait until all rings fail to start recovery.
- **Active/Passive:** the incoming requests are routed to the active ring; we have non-active ring(s) waiting for when the active one fails to swap roles. We need to minimize the time it takes to detect a failure and the time it takes to swap roles. Also, we need to keep an eye on the redundancy factor for fear it falls below a safe threshold.

So far we have assumed that the redundant rings are utilizing the same set of nodes, that's not always the case; said rings can span nonidentical set of

[13]There are many Software-as-a-Service (SaaS) solutions that monitor uptime from different locations around the world; for example, uptime.com (30 locations) and honeybadger.io (5 locations).

nodes. Moreover, said nodes can be housed in separate data centers to ensure high availability in case of a severe failure like a natural disaster. Other approaches that increase the availability of nodes inside a cluster (like having spare nodes that can be re-purposed to take over failed ones' responsibilities) can be used in combination with the above. Redundancy can also be introduced at different layers and components of the systems: networking devices, network interface controllers, storage disks like Redundant Array of Independent Disks (RAID) and just a bunch of disks (JBOD), etc. Many factors impact where and how to introduce redundancy in a system like cost, bottlenecks (CPU, I/O, etc.), and availability SLA.

As we mentioned earlier, the time it takes a system to detect and recover from failures is crucial in determining its availability. Now that we have a practical way to scale out system partitions and replace nodes that failed while minimizing the number of partition keys reassignments, we can start thinking about minimizing the time it takes to failover; in other words, we will try to minimize Mean Time To Repair (MTTR), which represents the mean time taken to repair a failed system component (a node or a cluster in our case). Failover should kick in as soon as a failure is detected,[14] which involves redirecting requests, tasks, and/or data over to a redundant system component to cover for the one that failed. This could be done automatically after detecting a failure using a monitoring system, in which case it's called a failover; if the system is too complex to failover automatically or if failover fails, a human intervenes to perform the system recovery actions, in which case it's called a switchover. Said methods of recovery should be exercised regularly and frequently; our confidence in remediation plans should stem from proven executions of said plans and not just from their mere existence.

An infamous example of failing to recover from failure took place at GitLab.com, a Git repository manager and an issue tracking system, in early 2017: Live production data directories were deleted by accident by a system admin; when recovery was attempted, system engineers discovered that backups were ineffectual. GitLab published live notes during the incident, which included some interesting takeaways:

- Logical Volume Management (LVM) were only taken once every 24 h; this was too infrequent given how frequent the data GitLab stores change.
- Engineers were unable to figure out where the backups were stored for a long time.
- Backups were created incorrectly; they produced files that were only a few bytes in size.
- Disk snapshots in Azure (Microsoft's cloud computing platform) were enabled for the Network File System (NFS) server, but not for the database servers.
- Backups to the Simple Storage Service (S3) in AWS (Amazon's cloud computing platform) were not functioning properly and the S3 bucket that's supposed to host the backup files was empty.

[14] Mean Time To Detection (MTTD) is another metric that should be tracked.

- The replication procedure was error-prone and relied on badly documented ad-hoc shell scripts.

As the incident's notes summed it up, "... in other words, out of 5 backup/replication techniques deployed none are working reliably or set up in the first place."

14.10.1 Choas Engineering

On the other hand, Netflix created the concept of choas engineering, which is akin to red team exercises[15] in security (and intelligence) for distributed systems. Netflix created a system called Choas Monkey[16] that goes wild and breaks things to test how their distributed systems would react. The idea is to identify failures and fix their underlying causes regularly, in an effort to be prepared for any failure mode; in addition to exercising recovery plans. The description of the project on GitHub sums up the rationale behind it: "Chaos Monkey randomly terminates virtual machine instances and containers that run inside of your production environment. Exposing engineers to failures more frequently incentivizes them to build resilient services." One can think of each run of the Choas Monkey as an experiment: Would the system withstand this new failure mode and stay available?

Thinking of what can go wrong when designing software is the first step of building big data systems; you can't protect against what you don't know can happen—unknown unknowns are your worst enemy. What chaos engineering fosters is a culture of due diligence and strengthening the muscles of fault tolerance. Other companies like Amazon, Facebook, Google, LinkedIn, and Microsoft employ similar techniques to validate system resiliency in the face of failures in productions. At Microsoft, we created a system that wreaks havoc in a cluster—during business hours like Chaos Monkey—by turning off machines, wiping out hard disks, slowing down the network, and performing other actions to test system resiliency and exercise recovery plans; we called it Havoc Hippo.[17] We allowed it to be configured to produce various levels of havoc depending on the severity of damage desired: none, low, medium, and high. It was capable of causing damage to a single machine or an entire cluster. Havoc Hippo integrated well with the monitoring and repair systems we created for data centers that hosted Windows Live, Outlook.com, and OneDrive.

Failovers between data centers were routine at LinkedIn to ensure that high availability measures we had in place will be up to the task when the time comes to failover due to an actual failure. In the early 2000s, Amazon started a program called GameDay[18] to inject failures into critical infrastructure and test system resiliency and discover unknown dependencies and failure modes; they restarted an entire

[15] A red team exercise is when an organization employs an independent group to penetrate said organization by any means necessary, in order to identify security holes.

[16] Choas Monkey is open source and can be found at https://github.com/Netflix/chaosmonkey.

[17] Our Platform-as-a-Service (PaaS) product had an aquatic theme.

[18] http://queue.acm.org/detail.cfm?id=2371297.

data center and then discovered that a good percentage of machines stayed offline because they ran out of DHCP (Dynamic Host Configuration Protocol) leases—unpredictable failure. In 2014, Facebook shut down an entire data center to test resiliency; it's no small feat to be able to withstand such failure but it went smoothly for the most part. Even though those exercises might sound extreme, we argue that they're absolutely necessary: Natural disasters, like Hurricane Sandy or the tsunami in Fukushima, can cause entire data centers to be disconnected, and the thought of that should drive companies in the big data business to think carefully about high availability. More frequent than natural disasters is human error; in 2017, an S3 outage[19] took the Internet by storm bringing down so many services to their knees; and it was all caused by a simple typo! As the incident's summary describes it: "The Amazon Simple Storage Service (S3) team was debugging an issue causing the S3 billing system to progress more slowly than expected. At 9:37AM PST, an authorized S3 team member using an established playbook executed a command which was intended to remove a small number of servers for one of the S3 subsystems that is used by the S3 billing process. Unfortunately, one of the inputs to the command was entered incorrectly and a larger set of servers was removed than intended... Other AWS services in the US-EAST-1 Region that rely on S3 for storage, including the S3 console, Amazon Elastic Compute Cloud (EC2) new instance launches, Amazon Elastic Block Store (EBS) volumes (when data was needed from a S3 snapshot), and AWS Lambda were also impacted while the S3 APIs were unavailable." Unlike most Internet services, like Quora and Trello, that depend on S3—and other AWS services—one of the services that weren't affected by the outage was Netflix, which is so used to such failures as a matter of course, it was just another Tuesday. In fact, in 2015, Netflix added Chaos Kong (think Chaos Monkey on steroids) to the simian army[20]; Chaos Kong doesn't just kill a service or a server, it simulates bringing down an entire AWS region (like US-EAST-1) when it runs. It's a rare event but it has happened before in September 2015 as well, Amazon's DynamoDB suffered from an outage in the same region and the ripple effect caused an outage for another 20 AWS services that depended on DynamoDB, and in turn that caused many Internet sites to be unavailable for about 8 h. The Netflix systems were resilient enough to fail over gracefully despite the outage; they had fixed issues their engineers identified during day-to-day operations when Chaos Kong ran and tested their systems' resiliency. Because Netflix engineers knew that an AWS region failure is expected, thanks to Chaos Kong, they had to plan for it and harden their systems to withstand it; this way, when it happens for other reasons, Netflix systems can continue to serve their users without an outage, and that's the essence of high availability.

Principles of Chaos Engineering

Netflix coined the principles of chaos engineering, which one can find at principlesofchaos.org; they describe a chaos experiment as the following steps:

[19] https://aws.amazon.com/message/41926/.

[20] https://medium.com/netflix-techblog/chaos-engineering-upgraded-878d341f15fa.

- Define a steady state of the system under test: How does it perform under optimal conditions? What are the values of the metrics being tracked at said conditions? In what configurations does it run (number of nodes, their roles, etc.)?
- Hypothesize that said steady state will remain the same during the experiment; systems in the control group won't be affected and the ones in the treatment (experiment) group should also stay the same.
- Introduce chaos that reflects plausible real-world failures like crashes, data loss, faulty hardware, slow networks, etc.; make sure the experiment is well contained and doesn't cause a failure that can be business-ending.
- Look for differences in the observed steady state between the control and the treatment groups; try to find evidence that disproves the hypothesis.

After the experiment is conducted, identify and address the weaknesses that cause the treatment group to fail; then regularly reiterate. A good experiment should uncover a new failure mode—embrace such failures and be grateful they occurred in a controlled setup during business hours. When first adopting the principles of chaos engineering, the frequency of running chaos experiments is important, there's a sweet spot that lies between never (which uncovers nothing) and all time (which may keep the engineers busy fighting fires); once the system is mature enough to handle routine failures, the knob can be turned to inflect more chaos onto the system and chaos experiments should be run continuously and automatically. Building the habit of running chaos experiments requires not only software changes, but also cultural ones: Organizations that follow the principles of chaos engineering should foster such culture of embracing failures and learning from them. Engineers must see that catching said failures early in the lifecycle of a system is crucial; otherwise they run into the risk of failures and weaknesses that cascade and become systematic and extremely costly to mitigate because the system wasn't designed to tolerate them.

Recovery should be a second-nature reaction in failure scenarios; practicing service recovery methods makes the process feel almost effortless. It's important to remain calm and collected when running through recovery procedures, that's why we recommend performing a rollback instead of fixing a bug and deploying the fix, which is also called forward fixing. Let's compare the two methods, fixing forward and rolling back, but first, let's establish a set of tenets that pertain to service recovery:

- **Do It Right the First Time:** Build high quality well-tested software that strives not to fail; unexpected system failure should be an exception. Albeit the inevitability of unknown failures, you must strive to prevent and tolerate them; quality is of the essence.
- **Do It Right the Second Time:** Embrace Murphy's law: Anything that can go wrong, will go wrong. Make sure you have a recovery plan that works and that's proven to work every time.
- **Restore the Service First:** The highest priority in a recovery situation is to restore the service; maintaining availability and quality of service is more important than shipping a new feature, understanding the root cause of an issue, or fixing it.

- **Minimize Risk:** Devise a recovery plan that gets the job done while minimizing the risk of failure or introducing another issue.
- **Be Ready to Recover from Recovery:** A recovery plan must not cause any irreversible damage; If recovery fails, you shall be able to go back to the failure state without making it any worse.
- **Keep It Simple:** Pick a recovery plan that involves as few moving parts as possible.
- **Use a Big Hammer:** Target as many failure modes as possible with the same recovery plan; minimize the number of recovery plans and reuse them. Custom-tailoring a recovery plan for each failure mode might sound optimal, but it actually produces a complex recovery system that has less test coverage in the real world, which in turns makes it more susceptible to failure.

Now let's examine the two recovery strategies mentioned above, fixing forward and rolling back, and how they adhere to the aforementioned tenets.

14.10.2 Fixing Forward

Sometimes, it's the only viable option; Consider a three-tier architecture where data were migrated to a new schema, and new business-critical data started to flow into new columns. On the other hand, fixing forward can be costly because of the following:

- It requires a moratorium which means hindering team productivity and progress of the business.
- It introduces code churn that increases system entropy and technical debt.
- There's a risk of breaking something else because of the new changes.
- Quick fixes are—by definition—rushed out, so they don't usually follow the same scrutinized process as regular changes and they may have less test coverage. In addition, they are more prone to human error due to the high-stakes situation.
- You should fix forward if it's the only feasible recovery action—if that's the case, you need to reevaluate your system architecture to enable other recovery options in the future.

14.10.3 Rolling Back

Restoring the system to the last known good state should be your default recovery plan when the following conditions are met:

- The failure is caused by a change that's internal to the system; beware of synchronicity and the post hoc ergo propter hoc fallacy: for example, a deployment that changed the system significantly could seem to be the culprit while—in fact—the failure was caused by a networking issue that had nothing to do with said deployment. Rolling back is more susceptible to that fallacy because it can be easily automated while fixing forward is manual and intentional.

- The failure is caused by a change that can be rolled back; new code should have an off switch so that it can be killed by a quick change like an experiment control knob that can be turned down or a configuration change. If such mechanism is not available, a deployment of the last known good software version is required; backward and forward compatibility between various system components is a prerequisite in that case.
- The last known good state of the system is known and verified; some failures are dormant and that makes picking a last known good state more challenging.
- The last known good state subsumes the various components of the system: binaries, static resources, configuration files, etc.
- The last known good state had sufficient mileage in production and is battle-tested to prove its goodness.
- The rollback process is well maintained and regularly exercised.

It's worth noting that Google blogged back in 2011 about how it mitigated a Gmail bug by rolling back the breaking change; the Gmail team wrote: "We released a storage software update that introduced the unexpected bug, which caused 0.02% of Gmail users to temporarily lose access to their email. When we discovered the problem, we immediately stopped the deployment of the new software and reverted to the old version." In summary, rolling back changes works best for systems that satisfy the aforementioned prerequisites and it adheres to the tenets we established earlier.

14.11 Fault Tolerance

Mediocre system designers create subsystems that fail and, in turn, cause system-level failures; good system designers create high-quality subsystems in an attempt to reduce system-level failures; great system designers create high-quality subsystems and plan for failures to happen at any point of time. The latter faction understands that failures are a fact of life in large-scale systems: hardware fails, software has bugs, and human operators make mistakes—system designers should plan for known failure modes and anticipate unknown ones.

14.11.1 Retry Policies

In anticipation of intermittent failures and downtime between failure and repair, system designers should mitigate such issues by making a system fault-tolerant. For example, in Sect. 10.11.3 we used a client API that allows for retrying an operation in case of failure, which is common practice in distributed systems. When done right, a retry loop can improve availability and quality of service remarkably; the key here is to understand each failure mode to determine whether retrying would help or not (or worse, exacerbate the issue). Moreover, the frequency and number

of retries are important and may differ from one scenario to another; such factors constitute a retry policy.

Some failure modes are long-lasting or even permanent; for instance, receiving a 400 (Bad Request) HTTP status code from a web server indicates that the client sent a malformed request; so trying to resend the same request will be, almost always, in vain. On the other hand, receiving a 408 (Request Timeout) HTTP status code indicates that the client may resend the same request using a new connection—it's basically an invitation to retry. A 429 (Too Many Requests) status code indicates that the server is rate-limiting the client, so retrying to send failed requests right away may exacerbate the issue; for example, the server may choose to blacklist the client for fear of abuse or denial-of-service (DOS) attacks; instead, the client needs to slow down and retry after a relatively long delay.

A simple retry policy would be to do so once right after a failure; for example, that might be useful when retrying to update a data record after an optimistic concurrency control check has failed.[21] But why stop at a single attempt? What if the failure is still taking place when the second attempt is made? A more practical retry policy would be to retry multiple times (where the number of retries depends on the failure mode and the fault-tolerance mechanism) to give the failed operation higher odds of success. The number of retries—let be one or more—depends mainly on the trade-off between incurring the cost of a retry and the increased probability of success, thanks to retrying N times.

Ideally, one should add a time delay between retries in order to give the failed system a chance to recover; the number of attempts and durations of delays between them determine the total amount of time during which retries take place. Delays between attempts can range from nanoseconds to days depending on the urgency of the operation being retried. One may choose to retry using a constant delay, which implies that attempts are uniformly distributed over the range of time attempts are made; or time delay between attempts can grow in an exponential manner to reduce the frequency of retries as time passes (and, in turn, their total cost). Another approach is to retry until a deadline is met (let it be elapsed duration or a specific time). Retry policies can be custom-tailored to suit specific failures and a combination of the aforementioned (either on the client-side, server-side, or both) may be used to allow for flexibility.

Retry policies are useful not only for fault tolerance but also for polling and other tasks that require checking for some conditions to become true. To demonstrate retry policies, we implement one that combines exponential delay with a chance to reset the delay interval back to its smallest value, at any time, based on the result of a Bernoulli trial. We use a similar technique at Voicera to poll messages from queues that Amazon Web Services (AWS) hosts; AWS charges for each read request even if it returns empty-handed (no messages), so we implemented a cost-effective way to reduce cost of polling: adding an exponential backoff delay between requests. For example, a poller requests to read messages from a queue and if none were found,

[21] See Sect. 10.11.3 for details about optimistic concurrency control.

it backs off for 1 second; after the delay, it checks again but this time it waits for 2 seconds if no messages were found; then 4 seconds, then 8, etc. The amount of delay gets reset back to its initial value (of 1 second in our example) after a message is received. The downside of this approach is risking a great delay between the time a message arrives in the queue and the time it gets picked up; to mitigate that, we relax the conditions of exponential delay growth:

1. We cap the delay to a limit that allows us to meet the SLA for the time it takes to process a message; for example, we would cap the delay to 32 seconds if the SLA is 1 minute (and processing time is safely less than the remainder 28 seconds); when the maximum value for a delay is reached, gets reset back to its initial value (of 1 second in our example).
2. We conduct a Bernoulli trial that determines whether or not to reset the delay interval to its initial value; for example, if we let the probability of success in our Bernoulli trial to be 0.05, we expect the delay interval to be reset 5% of the time. Such stochasticity serves two purposes: a mitigation for an unnecessarily overestimated maximum delay and a jiggering technique that prevents concurrent clients from hitting a service at the same schedule (which causes a spike in requests).

Here's an example of a probabilistic exponential backoff poller; this one starts with an initial delay of 2 milliseconds and gives up when the total delay exceeds 60 milliseconds; it also allows resetting the delay interval with a probability of 0.42:

```
import java.util.Random;
import java.util.concurrent.TimeUnit;

class ProbabilisticExponentialBackoffPoller {
  public Object[] poll(
      final TimeUnit timeUnit,
      final int initialDelay,
      final int maxTotalDelay,
      final double delayResetProbability)
        throws InterruptedException {
    final Random random = new Random();
    Object[] messages = readMessages();
    int currentDelay = initialDelay;
    int totalDelay = 0;

    while (messages.length == 0 && totalDelay <= maxTotalDelay)
     {
        System.out.printf(
        "Sleeping for %d %s\n", currentDelay, timeUnit);
      timeUnit.sleep(currentDelay);
      totalDelay += currentDelay;
      currentDelay *= 2;

      if (delayResetProbability > random.nextDouble()) {
        System.out.println("Resetting delay interval");
        currentDelay = initialDelay;
```

```
        }

        messages = readMessages();
      }
      return messages;
    }

    private Object[] readMessages() {
      return new Object[0];
    }
  }

  public class Main {
    public static void main(String args[]) throws Exception {
      final Object[] messages =
        new ProbabilisticExponentialBackoffPoller().poll(
        TimeUnit.MILLISECONDS, 2, 60, 0.42);
    }
  }
```

The above program produces output that looks like the following:

```
## Sleeping for 2 MILLISECONDS
## Sleeping for 4 MILLISECONDS
## Resetting delay interval
## Sleeping for 2 MILLISECONDS
## Sleeping for 4 MILLISECONDS
## Sleeping for 8 MILLISECONDS
## Sleeping for 16 MILLISECONDS
## Sleeping for 32 MILLISECONDS
```

The code above shows a retry policy for a poller, but it can be easily used for fault tolerance by replacing the condition that inspects the messages' length with an error check. Retry policies can be combined into various recipes to meet the needs of the operations and tasks at hand.

14.11.2 Circuit Breakers

While retry policies can be effective in the face of transient faults, sometimes services take longer than expected to recover from failures. In order to contain longer failures and to prevent them from cascading into a system-wide failure due to a ripple effect, system designers use a design pattern called circuit breakers. Named after physical circuit breakers that interrupt the flow of electricity after a fault is detected in a circuit, a circuit breaker in software does something similar: It isolates failed components so that other components don't get affected by said failure. For example, at LinkedIn where we worked on the news feed's relevance, we put in place circuit breakers for the various services that provide content for the feed's blending service so that if any of these services failed to meet its SLA, it gets cut out (and a human may get notified to intervene). That way, if a content-providing service is suffering from a performance issue or a bug, that issue doesn't cascade

to the feed's blending service and it can continue without it. Between the time the failure was detected and until the circuit breaker is reset, the feed's blending service cuts off the offending content-providing service. Resetting the circuit breaker can be automated by polling the failed service using a retry policy to check for its health, and/or a human may override the circuit breaker after confirming that everything is working as expected.

For further reading about circuit breakers, we recommend the following article: docs.microsoft.com/en-us/azure/architecture/patterns/circuit-breaker.

14.12 Offline, Near-Line, and Online Data Processing

The safeguards and mitigations of failures we discussed so far are mainly concerned with online services, which need to be highly available to server live traffic. Classifying data into those three categories (online, near-line, and offline) helps system designers make different plans for each scenario. At LinkedIn, we followed that classification and we will use some LinkedIn features here to exemplify what each class represents. Online data are characterized by how immediate changes are reflected; for example, when a LinkedIn member changes her profile data, said changes must be reflected right away (within seconds) in all online systems. Near-line data have more relaxed constraints as changes can be reflected within minutes or hours; for example, statistics of how many times a member's profile was viewed and by whom. Offline data have even more relaxed constraints as changes can be reflected later (within hours, days, or weeks); for example, aggregated training data used to train machine learning models for recommendation systems.

Regards of class, system designers should strive to have data always available; that said, issues that affect online data are of higher priority and should be triaged accordingly.

14.13 Hot, Warm, and Cold Data Storage

Classifying data by their "temperature" is another angle for looking at designs of big data systems; it might also go hand-in-hand with the above classification. The idea is to create three tiers of data storage based on how often data need to be accessed and their rate of change. For example, caches are a good example of hot data storage systems; hot data are usually kept in memory and/or stored on fast media like flash storage or solid-state drives (SSD). Warm storage examples include data stored on slower media like network-attached storage or rotating-platter disks (which are considered warm storage nowadays); for example, data that get extracted, transformed, and loaded (ETL'd) once a day. Cold data are accessed infrequently and thus can be stored on slower devices like tape drives, which are commonly used for backups.

Facebook—in an engineering blog post—described how the company stores cold data in data centers that selectively power on servers and drives as needed, which allow them to consume less than one-sixth of the power required for other Facebook data centers (ones that serve hot and warm data). Facebook uses rotating-platter hard drives but also showed how optical storage, like Blu-ray discs, can be used for cold storage at scale. When reading about data storage challenges and solutions that big-data companies have, things start to sound familiar as patterns emerge; At Microsoft, we faced similar challenges when dealing with data storage at scale for hundreds of millions of email users at Outlook.com—any organization that works on big data projects need to ready itself for such challenges. At scale, the subtlest and tiniest details matter as they add up. For example, magnetic hard drives may degrade due to bits losing their magnetic orientation (also known as bit rot) when unchanged for prolonged periods; rewriting the same data mitigates such failures (and helps validate the integrity of the data). Another example is picking the file system's allocation unit (block) size; for big data systems that store small files—like text messages—small block sizes help avoid wasting storage space; for systems that store big files—like photos and videos—big block sizes help avoid fragmentation, which can cause I/O latency due to increased seek time and rotational latency of the read/write heads in rotating-platter drives. At scale, natural disasters and unpredictable external events, like Thailand's devastating floods in 2011, can affect big data systems worldwide! The floods disrupted supply chains for hard drives, which in turn triggered a serious global shortage lasting throughout 2012. Putting all of the above in perspective, designers of big-data systems need to plan for when and where the bits meet the metal.

14.14 The Cloud

Storing and processing big data usually happen in data centers that have enough capacity to accommodate the storage and compute requirements needed. Traditionally, companies used to build or lease dedicated clusters of machines in a data center to perform big data tasks at scale; that's a high barrier to entry for companies that can't afford the upfront capital expenditures (capex), not to mention other challenges like operational expenditures (opex), inefficiencies caused by overestimating the requisite compute and storage resources (capacity planning), delays when expansions are desired, maintaining hardware, physical security, networking, power, etc.—a logistical nightmare! Nowadays, with the prevalence of cloud-based solutions, provisioning a cluster of computers takes a few minutes; startups can build solutions that crunch a colossal amount of data without owning a single server. Companies typically prefer to pay for cloud-base infrastructure and solutions until they become cost-prohibitive (in which case, it makes more sense to build their own); even then they may choose to use bite the bullet and keep paying for said services because of convenience and business focus. There are multiple

kinds of cloud-base solutions to which we have been referring throughout the book, here's a brief comparison between them.

14.14.1 Infrastructure-as-a-Service (IaaS)

IaaS provides infrastructure like compute and storage resources in the cloud (e.g., Amazon's Elastic Compute Cloud a.k.a EC2); IaaS providers build their own data centers and allow others to use their resources, via a self-serve model, for a fee. Said data centers—and most of the time, machines too—are shared among multiple tenants, who typically only get access to virtual machines they rent usually on demand by the hour (or the minute in the case of Google's Compute Engine). IaaS providers may allow tenants' direct access to resources, reserve resources for years, or bid on resources that may become available based on supply and demand. With a few clicks, tenants can increase or decrease the compute, I/O, and storage capacity of their infrastructure in the cloud; IaaS provides flexibility and elasticity for infrastructure provisioning and decommissioning via easy-to-use tools and APIs. Tenants also have access to dashboards that monitor how their infrastructure is doing. A key advantage of IaaS is high availability as providers guarantee SLAs that allow tenants to focus more on the software rather than worrying about the availability and reliability of infrastructure. The prevalence of IaaS is akin to specialization in trade: both parties benefit by focusing on—and getting better at—what they do best. Examples of IaaS providers include Amazon, exoscale, Google, Microsoft, Navisite, and Softlayer.

14.14.2 Platform-as-a-Service (PaaS)

PaaS can be thought of as a layer of abstraction on top of IaaS; it provides tools for software developers to create and deploy applications. In other words, PaaS manages servers for developers with minimal input and configuration to setup machines, operating systems, networking, load balancing, software frameworks, monitoring, etc. Typically, PaaS provides minimal control over the underlying infrastructure of an application; developers happily surrender control to PaaS systems, which promise optimality and convenience. While IaaS systems provide dashboard to monitor infrastructure resources, PaaS systems provide monitoring and dashboard for applications they host. Focusing on developing application-level logic and leaving the rest to PaaS systems is their raison d'etre; application developers don't have to worry about managing infrastructure, operating systems and their updates, manual installation of popular frameworks and database management systems, etc.

Heroku, a PaaS system that was acquired by Salesforce, is a commonly used PaaS system, especially for web applications; it provides a plethora of integrations and software frameworks as add-ons; it also offers auto-scaling of web applications so

that they can maintain a desired 95th percentile (p95) response time. While Heroku's auto-scaling feature is reactive, an AI-powered PaaS system called Magalix provides anticipatory auto-scaling that's learns from historical traffic and resource usage patterns. Other cloud solutions like Amazon and Google provide a mix of IaaS and PaaS. For example, Google App Engine allows developers to choose between auto-pilot style or high-control infrastructure that hosts their applications; it also provides running containers[22] and automatic provisioning of infrastructure that's triggered by an application's requirement for compute resources. Amazon EC2 Container Service (ECS) provides a layer on top of EC2 for managing and running containers; it also allows developers' direct access to the underlying EC2 instances used to build ECS clusters. Docker Platform offers container management and execution called Docker Swarm, in addition to supporting Kubernetes.[23]

14.14.3 Functions-as-a-Service (FaaS)

Some of the PaaS examples we listed above offer what's known as serverless architecture; a more illustrative example of said architecture is Amazon's Lambda, which allows developers not only to run code in the cloud without the need to provision or manager servers or containers, but also without having to pay when their code is not running; simply, all what developers have to do is to upload their code to Lambda and it runs in the cloud. Lambda is commonly used as functions in the cloud that get called by configurable triggers (e.g., an SQS message), which make them apt building blocks for pipelines; Amazon Simple Workflow Service is simply a platform to build pipelines using Lambda functions. IronFunctions is another example of FaaS platforms; it supports AWS Lambda in addition to public, private, and hybrid clouds.

14.15 Other Notable Cloud Services

Throughout this chapter, we discuss some popular cloud services like the ones Amazon—the leading cloud services provider—offers through AWS; here, we will list a few examples of notable cloud services, what they are used for, and related frameworks and solutions that can be found elsewhere. We highly recommend reading AWS Whitepapers at https://aws.amazon.com/whitepapers/ to glean a deeper understanding of AWS.

[22] A container is a self-contained executable package that runs, in isolation, on its own virtual operating system.

[23] Google's system for managing and running containerized applications, which can be deployed on many IaaS clouds as well (just like DC/OS).

14.15.1 Amazon Athena

Amazon Athena is yet another serverless Amazon service; it allows developers to ingest petabytes of data from Amazon Simple Storage Service (S3) for interactive querying using ANSI (standard) SQL. Since Amazon Athena is—from the user's perspective—serverless, there's no need to provision machines or worry about managing data warehouses. Data stored in Amazon S3 can be analyzed immediately as Amazon Athena works directly with data stored in Amazon S3 without having to load them first into another service; users only need to point Athena to where data are stored in Amazon S3, define the schema, and get back query results in a few seconds. Amazon Athena eliminates the complexity of ETL jobs that typically take hours to run in order to prepare big data for analysis; it supports various data formats that are common in big data systems like Avro, CSV, JSON, ORC, and Parquet. As with most AWS serverless services, users of Athena pay for usage time (only for the queries that they run). Amazon Athena's schemas can be auto-generated by crawling the target S3 bucket using AWS Glue, which we will discuss later on in this chapter.

At Voicera, we use Amazon Athena to run queries against JSON files stored in S3 to gain insights about various machine learning systems. Getting started Amazon Athena took exactly 2 steps: selecting a data source (an S3 bucket) and defining a schema of a table that represents the JSON data using an easy-to-follow wizard. The entire configuration process that started from scratch to running SQL queries against an S3 bucket took around 3 min; most query results came back in less than a minute. There are a few caveats though (at the time of writing):

- Support for running SQL queries programmatically from other applications is limited to connections that use a Java Database Connectivity (JDBC) driver.
- JSON records had to be on a single line—which is usually recommended anyways to save space—as Amazon Athena doesn't support multi-line JSON records.

Presto

Amazon Athena is built on Presto: an open-source distributed SQL query engine that Facebook started in 2012 and released as an open-source project for Apache Hadoop in 2013. Presto is optimized for low-latency, ad-hoc querying, and analysis of data via distributed queries that workers in a cluster process in parallel; the key difference—compare to Apache Hive or MapReduce—is that Presto does not store intermediate results to disk to speed up query processing. Netflix used Presto to analyze 10 petabytes of data stored using the Parquet file format in the Amazon S3,[24] which is used to host the storage layer of Netflix's datawarehouse. The separation of storage and compute allows Netflix to share data used for

[24] https://medium.com/netflix-techblog/using-presto-in-our-big-data-platform-on-aws-938035909fd4.

analytics among multiple compute clusters that can be stateless and transient. In October 2014, Netflix announced its own Presto connector to the Amazon S3 file system, which predates Amazon Athena (announced in November 2016). Netflix's evaluation of Presto shows that it's 10 to 100 times faster when processing queries that take one or two map-reduce phases in Hadoop.

14.15.2 Amazon DynamoDB

Amazon DynamoDB is a highly scalable NoSQL database that supports both document and key-value data structures. A key-value stores typically supports indexing by a single key, which is the primary key. Indexing, compared to scanning for a value, allows faster access to data by specific criteria. In addition to supporting primary-key indexing, Amazon DynamoDB supports one or more secondary indexes on a table so that developers can query efficiently for data using attributes other than the primary key. To achieve an average service-side latencies that are typically single-digit milliseconds, DynamoDB stores data on solid state disks (SSDs) as hot storage; it also automatically replicates data across multiple clusters in an AWS region, providing built-in high availability. NoSQL databases, like DynamoDB, sacrifice some of the features that relational databases provide, like transactions and complex query constructs, in order to achieve better performance and scalability. To make working with Amazon DynamoDB even more efficient, Amazon offers DynamoDB Accelerator (DAX) which is highly available in-memory cache that can reduce the response time, even at high throughput, from milliseconds to microseconds! Companies like Amazon (of course), Airbnb, Netflix, and Lyft use DynamoDB at scale.

One of DynamoDB's useful built-in features is the ability to specify a retention policy for data based on a time-to-live (TTL) after which DynamoDB automatically deletes expired items; this feature helps reduce storage utilization and, in turn, costs of hosting said items. At Voicera, we use DynamoDB for multiple scenarios where the TTL feature comes in handy to clean up transient data after they become irrelevant.

Here's an example to illustrate how DynamoDB works: we create a table called `reminder` to store user-requested reminders for our previous example of a reminder bot. When creating a DynamoDB table, there's no need to create a schema like tables in relational databases; DynamoDB partition documents that belong to the table and it just needs to know how to refer to document by its hash key (also known as partition key). We covered partitioning earlier in this chapter and we saw how it can be used to scale processing of big data over multiple nodes in a distributed system; DynamoDB does the same thing with documents it stores. The recommendations for picking a hash key in DynamoDB are similar to those we discussed earlier in Sect. 14.4: DynamoDB recommends choosing an attribute whose range of values allows for evenly distributed access patterns. DynamoDB supports a limited set of types for a hash key; it can be either a string, a number,

or binary. Binary keys can be basically a binary serialized version of any other type but serialization and deserialization cost CPU cycles. For our application we use a number for our hash key; it represents the `timestamp` (Unix epoch) of the reminder. In addition to the hash key, we define a sort key, which is part of the table's primary key. Once a partition is located using a hash key, a sort key allows for efficient searching within said partition. In our example, we choose to sort by `userID`. For the rest of the table's settings, like read and write capacity, we accept the defaults, which suffice for the illustrative example but may not work for a production scenario. Once the table is created, other settings can be changed; for example, we can now manage TTL by specifying an attribute that represents when a document/item expires. To that end, we add a new attribute, `ttl`, which is a number data type containing the Unix time of when a reminder expires. Assuming we want to keep reminders for a day (86,400 s) after the reminder time specified in the `timestamp` key, we calculate the value of `ttl` as such. When an item expires according to the value of its `ttl` field, DynamoDB will delete it in the background.

Before running the example below, we made sure the `boto3` library is installed on the machine where the script is running and configured with the proper credentials and AWS region where the reminder table is hosted. Now we can execute the following code to create reminders:

```
import boto3
dynamodb = boto3.resource('dynamodb')
dynamodb.Table('reminder').put_item(
    Item={
        'timestamp': 1511643670,
        'userID': 'geish@voicera.ai',
        'ttl': 1511730070,
        'text': 'write a DynamoDB example',
    }
)
```

Notice how we didn't mention the field `text` earlier since we only needed to create a key schema (and not a schema for the entire record), yet the item got with the fields we specified in the example above and their types were inferred. To get that item we just created, we simply can query for it:

```
from pprint import pprint

import boto3

dynamodb = boto3.resource('dynamodb')
response = dynamodb.Table('reminder').get_item(
     Key={
     'timestamp': 1511643670,
     'userID': 'geish@voicera.ai',
     }
)
pprint(response['Item'])
## {u'text': u'write a DynamoDB example',
##  u'timestamp': Decimal('1511643670'),
```

```
##  u'ttl': Decimal('1511730070'),
##  u'userID': u'geish@voicera.ai'}
```

DyanmoDB also supports batch operations to speed things up when performing multiple operations and to reduce the number of responses sent to the service, for example:

```
import boto3

dynamodb = boto3.resource('dynamodb')
with dynamodb.Table('reminder').batch_writer() as batch:
     batch.put_item(
          Item={
              'timestamp': 1511647270,
              'userID': 'geish@voicera.ai',
              'ttl': 1511733670,
              'text': 'batch write example',
          }
    )
    batch.put_item(
          Item={
              'timestamp': 1511647270,
              'userID': '@voicera.ai',
              'ttl': 1511733670,
              'text': 'batch write example',
          }
    )
    batch.put_item(
          Item={
              'timestamp': 1511650870,
              'userID': 'geish@voicera.ai',
              'ttl': 1511737270,
              'text': 'another item to write',
          }
    )
    batch.delete_item(
          Key={
              'timestamp': 1511643670,
              'userID': 'geish@voicera.ai',
          }
    )
```

Because we specified the `timestamp` field as the partition key, querying for reminders at a specific timestamp is efficient using a key condition:

```
from pprint import pprint

import boto3
from boto3.dynamodb.conditions import Key

dynamodb = boto3.resource('dynamodb')
response =
dynamodb.Table('reminder').query(
```

```
    KeyConditionExpression=Key('timestamp'
).eq(1511647270)
)
pprint(response['Items'])
## [{u'text': u'batch write example',
##   u'timestamp': Decimal('1511647270'),
##   u'ttl': Decimal('1511733670'),
##   u'userID': u'geish@voicera.ai'},
##  {u'text': u'another user',
##   u'timestamp': Decimal('1511647270'),
##   u'ttl': Decimal('1511733670'),
##   u'userID': u'user@example.com'}]
```

To get all reminders that belong to a specific user, we scan the table using the following attribute condition:

```
from pprint import pprint

import boto3
from boto3.dynamodb.conditions import Attr

dynamodb = boto3.resource('dynamodb')
response = dynamodb.Table('reminder').scan(
    FilterExpression=Attr('userID').eq('geish@voicera.ai')
)
pprint(response['Items'])
## [{u'text': u'batch write example',
##   u'timestamp': Decimal('1511647270'),
##   u'ttl': Decimal('1511733670'),
##   u'userID': u'geish@voicera.ai'},
##  {u'text': u'another item to write',
##   u'timestamp': Decimal('1511650870'),
##   u'ttl': Decimal('1511737270'),
##   u'userID': u'geish@voicera.ai'}]
```

When it comes to choosing between relational and NoSQL databases like DynamoDB, besides considering scale as a criterion, the flexibility of schema (or lack thereof) is another key factor. For example, we just saw how we can add new fields by simply adding an item that contains said fields. More importantly, the data are not necessarily tabular, which opens up the door for flexibility when storing a record. For example, we can add a check-list reminder that includes an array which other reminders don't specify at all. Another example would be adding a complex object like what actions to take when the reminder is triggered; each action may contain different types of records with information about callback URLs and other specifications. Typically, developers would add a type field that helps parse the record according to the value specified in the type field (akin to a factor method given a string that represent the type of the object being constructed). In other words, NoSQL databases are a great fit for unstructured data, like in the following example:

```
from pprint import pprint

import boto3
from boto3.dynamodb.conditions import Key

dynamodb = boto3.resource('dynamodb')
table = dynamodb.Table('reminder')
with table.batch_writer() as batch:
    batch.put_item(
        Item={
            'timestamp': 1511652070,
            'userID': 'checklist@example.com',
            'ttl': 1511738470,
            'checklist': ['milk', 'eggs', 'bread']
        }
    )
    batch.put_item(
        Item={
            'timestamp': 1511652070,
            'userID': 'complex@example.com',
            'ttl': 1511738470,
            'actions': [{
                 'callback': 'example.com/callback',
                 'payload': {'text': 'flash office lights'},
                 'method': 'POST'
            }]
        }
    )

response = table.query(
    KeyConditionExpression=Key('timestamp').eq(1511652070)
)
pprint(response['Items'])
## [{u'checklist': [u'milk', u'eggs', u'bread'],
##  u'timestamp': Decimal('1511652070'),
##  u'ttl': Decimal('1511738470'),
##  u'userID': u'checklist@example.com'},
## {u'actions': [{u'callback': u'example.com/callback',
##  u'method': u'POST',
##  u'payload': {u'text': u'flash office lights'}}],
##  u'timestamp': Decimal('1511652070'),
##  u'ttl': Decimal('1511738470'),
##  u'userID': u'complex@example.com'}]
```

in Sect. 13.3.2, we talked about database consistency, which simply means each operation leaves the database in a consistent state. Since DynamoDB is built to handle big data, it's desirable to relax consistency constraints in favor of performance and high availability. A DynamoDB table may be inconsistent for some period of time, usually within 1 s or less, and they guarantee that consistency will be restored eventually; this is known as "eventual consistency." DynamoDB is distributed over multiple AWS availability zones for failure isolation and tolerance, so changes to a table need propagate across availability zones to ensure consistency.

DynamoDB supports both modes when reading data: eventually consistent and strongly consistent reads; developers can choose between them by specifying the value for the `ConsistentRead` flag for read operations like `get_item`, `query`, and `scan`. By default, read operations use the eventually consistent read mode, which means that some results might not reflect the most recent changes performed by write operations.

14.15.3 Amazon Elasticsearch Service (ES)

Elasticsearch is a search engine based on Apache Lucene; both are commonly used for full-text indexing, search, and recommendation systems. Their APIs are very flexible and can be used for a multitude of applications as they manipulated documents containing fields of text. Elasticsearch provides a distributed full-text search engine that can also act as a NoSQL data store; it's commonly used in combination with Logstach (a log collection and parsing engine) and Kibana (an analytics, dashboards, and visualization platform) to form what's known as the Elastic Stack (previously known as the ELK stack). At Voicera, we used Kibana, Elasticsearch, and Filebeat (which ships logs from various machines to an Elasticsearch service) to collect, aggregate, index, analyze, and visualize logs. We can follow and analyze specific data records as they flow across multiple microservices in a near-line fashion, thanks to indexes built by Elasticsearch. As with most AWS solutions, setting up a managed Amazon ES cluster took only a few minutes.

Elasticsearch's distributed design allows for scalability as logs grow in size and data become big; that said, provisioning and managing Elasticsearch at scale can be taxing—the cloud to the rescue. Amazon Elasticsearch Service (ES) takes care of the latter for developers with a few commends; it also supports the entire Elastic Stack natively. Amazon ES allows developers control access to an ES domain via easy-to-implement policies; in addition, an ES domain can be accessed securely from an Amazon Virtual Private Cloud (VPC) or from a public endpoint.

14.15.4 Amazon Elastic Map Reduce (EMR)

Basically a managed Apache Hadoop in the cloud, which competes with companies like Cloudera. Amazon EMR makes use of Amazon EC2 and Amazon S3 to build, manage, monitor, tune, and scale Hadoop for users of EMR. In minutes from uploading data into S3 buckets, EMR can be up and running to crunch said data and produce the results into other S3 buckets. EMR can be run on demand, so that developers only pay for the compute time when they need to crunch data, or it can be used in a long-lasting setup. Since data are loaded from S3 buckets, the same source of data can be reused in multiple EMR clusters, which can provide even more parallelization and separation of concerns in an efficient manner as each cluster can

be optimized for a specific use case. EMR, since it's Hadoop in the cloud, works with the variety of frameworks that work with Hadoop like Hive, Pig, and Spark.

14.15.5 Amazon Glue

To extract, transform, and load (ETL) big data is no small feat; yet, Amazon Glue can make it easy for developers to do so with a few clicks. As the name may suggest, glue software or middleware helps connect different pieces of a system together. One of the main use cases of Amazon Glue is schema discovery: Developers can point Amazon Glue to a data source, like S3, and it crawls it to find table definitions and schemas. Schemas are essential for indexing and transformation of data; and cataloging big data schemas at scale can get hectic; that's where Amazon Glue shines to automate that process. Once schemas are cataloged, developers can select data sources and targets, and then Amazon Glue auto-generates Python code to perform the transformation from the source to the target schema. The code can then be modified if need be then run as a recurring ETL job within Amazon Glue, which supports creating a pipeline of chained ETL jobs; and as you may have guessed, it takes care of scaling requisite infrastructure, fault tolerance, retry policies, etc.

14.15.6 Amazon Kinesis

Amazon Kinesis enables applications to process streams of data (similar to what Apache Samaza or Apache Flink provide for Apache Kafka). Since streaming data are typically massive and pass into systems at a very high throughput, capturing and processing such data in motion are no small feat. For example, processing billions of users' events daily in a near-line fashion; At LinkedIn, we did just that for the news feed using Apache Samza, which was open-sourced by LinkedIn. The same solution can be done using Amazon Kinesis; moreover, many stream-processing applications can be designed similarly. The challenge for our scenario was that relevance of content was insufficient to produce and rank the updates that show up on the feed—we needed to ensure freshness as well. We wanted to keep track of what users viewed on the news feed and use that data as input to our ranking algorithms. The news feed on the homepage of LinkedIn has hundreds of millions of users; at that scale, we had to think carefully about the tracking data. On the client-side, we used lightweight events that track what the user viewed; the payload had to be tiny to preserve data usage on cellular networks. Client-side events end up in a Kafka topic (stream) via a proxy server. On the other hand, server-side tracking involved events that are richer and contained more data about the feeds served to clients. Server-side events ended up in another Kafka topic. The task at hand was to join both streams to enrich the lightweight client-side events with data from matching server-side events, and then writing the matched events into a third Kafka topic (which contains the enriched events); for that, we used a stream-stream join task that buffered events from both streams in a cache and produced

the results of the join. At that time, Apache Samza supported RocksDB as a built-in key-value store, which was optimal for our cache/buffer, except that it didn't support automatic retention policies even though RocksDB supported it via its time-to-live (TTL) feature. We decided to implement our own TTL mechanism the deleted stale events using Samza's built-in timer (the `window` method). The client-side stream processor looked up incoming client-side tracking events, by ID, in a RocksDB that stores server-side tracking events; when matched, it would join the two events in a newly created event and write it to the output stream; when mismatched, it would buffer it up in a RocksDB that stores client-side tracking events. Conversely, the server-side stream processor looked up incoming server-side tracking events, by ID, in the RocksDB that stores client-side tracking events; when matched, it would join the two events in a newly created event and write it to the output stream; when mismatched, it would buffer it up in the RocksDB that stores server-side tracking events. The anatomy of the events and the lack of TTL support back and then necessitated many custom-made changes to the stream-stream join task; notably, the addition of a new method to bulk-get values from key-value stores in Samza for better performance and throughput. One of the key challenges that face stream-processing tasks is the ability to match the throughput of the streams they process without creating impedance or delays; in other words, to keep drinking from the firehose. We included details about that project in the footnotes[25] for the eager readers.

14.15.7 Amazon Redshift

Amazon Redshift is a data warehouse service in the cloud; it supports standard SQL to run queries efficiently against petabyte-scale structured data. Redshift is mainly used for analytics and business intelligence. At Voicera, we use Redshift as a data source that powers up our analytics dashboards. Redshift takes care of management of resources, scale, backups, encryption (at rest and on the wire), query tuning, etc. It uses columnar storage on high-performance local disks and a massively parallel query-processing architecture to deliver low-latency and high-throughput query results at relatively low costs.

For querying exabytes of data, Amazon offers Redshift Spectrum, which supports querying unstructured data stored in Amazon Simple Storage Service (S3) data lakes without having to load or transform the data. Redshift Spectrum supports common file formats like Avro, CSV, Grok, ORC, Parquet, RCFile, RegexSerDe, SequenceFile, TextFile, and TSV; they can be queried using standard SQL queries, which can also include results from data sources in Redshift (in addition to those from S3 data lakes via Redshift Spectrum). The flexibility of file format and storage location (S3) choices makes Redshift Spectrum an extremely useful data warehouse.

[25] https://www.slideshare.net/MohamedElGeish/harvesting-the-power-of-samza-in-linkedins-feed.

Both Amazon Redshift and Amazon Athena compete with BigQuery: a serverless data warehouse that Google offers through their public cloud solution. BigQuery supports loading petabytes of data from Google Cloud Storage (a service that's similar to Amazon's S3), Google Cloud Datastore, or data that developers streamed into BigQuery from sources like Spark and Hadoop.

14.15.8 Amazon Relational Database Service (RDS)

RDS is basically a relational database management system (RDMBS) in the cloud; it provides easy-to-use controls to create, operate, scale, and monitor relational databases like Amazon Aurora, MariaDB, Microsoft SQL Server, MySQL, Oracle, and PostgreSQL. Database management at scale is one of the hardest problems that face big data systems because they are—by design—stateful. RDS takes care of provisioning, availability, replication, backups, elasticity, patching, disaster recovery, etc. and offers a pay-as-you-use cost model. At Voicera, we're using RDS to manage multiple Aurora databases so that we can benefit from the speed improvements it promises and focus on the application-level logic instead of babysitting the infrastructure that hosts our databases.

14.15.9 Amazon Simple Storage Service (S3)

Amazon S3 provides simple-to-use object storage at a massive scale. Storage is organized as buckets; inside each bucket, each object is identified using a key. Although S3 may look like a file system, it doesn't support basic file system operations like counting the number of objects in a folder. S3 is widely used to store data, host websites, and a lot more. With multiple options for replication and transfer speeds, S3 can be customized to be hot or warm storage. For cold storage, AWS offers a whole other product called Amazon Glacier, which provides significant savings compared to other storage options (at the cost of slower access time, from a few minutes to several hours). S3 can be configured to move old data automatically to Glacier via lifecycle policies. To list a few of its features, S3 provides comprehensive security features like access control, encryption (using your own key or an AWS-provided one), auditing, etc. Its integration with Amazon Athena makes it a unique storage solution that provides in-place querying and analytics capabilities; similarly, its integrations with Amazon Glue makes it a great candidate for ETL jobs. Other useful features include the ability to version objects, trigger workflows (by calling lambda functions, sending SQS messages, or sending SNS notifications) for various events like object creation, etc. S3 features are numerous and they provide developers with so much flexibility. For example, it can be used to host a static website with very little setup. Another neat feature it provides is the ability to redirect a request to access an object to a web URL specified by the `x-amz-website-redirect-location` property, which opens up the

door for many applications on top of S3. It's safe to predict that the vast majority of AWS customers, notably Netflix and Airbnb, use S3.

14.16 Information Security

The vast majority of big data systems process data that are valuable assets and need to be protected. The sensitivity of data may vary from logs of what links anonymous users clicked to medical records that are extremely detailed and confidential. At Microsoft, data are classified into one of three classes:

- **High Business Impact (HBI):** Access to said data class is tightly controlled as security incidents (like unauthorized access, loss, or wrongful disclosure) lead to severe material impact (e.g., credit card theft); in some cases, such incidents can be business-ending! Extreme security measures are usually taken to protect this class of data (e.g., physical cages to protect the servers where the data reside).
- **Medium Business Impact (MBI):** for this class of data, security incidents mean damage to the company's brand and reputation; once can't put a price tag on that, but such incidents usually lead to users abandoning the company's services. Examples of MBI data include email addresses, phone numbers, etc.
- **Low Business Impact (LBI):** data that basically don't carry much value or whose security incidents have minute ramifications; for example, data that are publicly available or very easy to deduce (like a user's gender).

Understanding the class of data helps system engineers—and everyone else involved in building and operating software—take appropriate measures to protect the confidentiality, integrity, and availability of data. The aforementioned three attributes of information security are known as the CIA triad; a forth attribute, non-repudiation, is usually mentioned alongside the CIA triad to constitute the main goals of information security. Those goals are what usually get documented in security requirements, threat models, design documents, etc.; what everyone really seeks is a lack of security incidents. This applies to security anywhere, not just in the digital world. Had we lived in a world where everyone is trustworthy, we wouldn't have needed the security controls we have today.

The main objective of security controls is to prevent as many security incidents as possible from occurring; working hard to secure a system may go unnoticed because nothing happens: no incidents, no failures, and everything works as expected. It might be a thankless job, but it's absolutely crucial for everyone involved in the making of a system to get right. Awareness of security threats, mitigations, and methods to reduce the chances of incidents is the first step along the way of securing big data systems. Since security issues can happen virtually in any component or layer of a system, security becomes everyone's responsibility.

There are countless infamous security failures that affected well-established organizations including government agencies and software powerhouses; we know about them because they made the news. Conventional wisdom suggests that con-

siderably more security incidents occurred but weren't reported; or worse, weren't undetected. Here are a few examples of the infamous cybersecurity incidents that made the news:

- **Yahoo:** In 2017, the company disclosed that a 2013 security breach compromised all 3 billion email users. Initially, in mid 2016, the number of affected users was estimated to be 500 million and was revised to 1 billion by the end of 2016. Yahoo's forensics indicated that attackers were able to access accounts by forging cookies (no password required).
- **Equifax:** In 2017, the security breach affected 143 million accounts in the USA (almost half of the US population); the disclosed data included social security numbers, names, birth dates, and addresses—basically, enough data to steal the identity of any account holder for financial gain. The credit firm's stock price went around 31% after the security incident was disclosed.
- **Target:** Ranked as one of the worst ever at the time it was discovered, the breach that hit Target in 2013 allowed attackers to steal 40 million customers' credit and debit card information. It was later disclosed that a group of 70 million customers had their personal information compromised as well. Attackers were able to gain access to Target's network using credentials stolen from a vendor that monitors energy consumption and temperatures at various stores; then the attackers managed to break into Target's point-of-sale systems and install malware on the card readers to steal customers' card information.
- **Sony PlayStation Network:** In 2011, attackers stole records of 77 million network members; in addition, the gaming network was down for 23 days, vexing gamers all over the globe. The attack resulted in the largest data breach in history at the time.
- **Code Spaces:** was a company that provided software management solutions (akin to GitHub); in June 2014, after seven years of operating, it ceased to exist due to a security breach of its cloud infrastructure control panel. In mere hours, attackers obliterated customers' data after the company declined to pay a ransom; when Code Spaces finally regained control of its cloud infrastructure, it was completely wiped out of existence.

Security threats come in different shapes and from various sources. Below are a few examples of where threats may originate:

- **Accidental Discovery:** one of my favorite security stories is the one behind Twitter's follow bug in 2010. A Turkish user discovered that when he tweeted "accept pwnz," a user whose handle is "pwnz" was forced to follow him! He was trying to show his admiration for the metal band called "accept"; instead, he stumbled upon a way to make any user follow him.
- **Curious Researchers:** Security researchers, students, and academics who study security and try to find and analyze vulnerabilities.
- **Script Kiddies:** They may not know how it works or how to create their own, but they are equipped with an arsenal of scripts and tools that can cause serious damage to your systems. Nowadays, anyone has access to tools like Vault 7,

which WikiLeaks claimed it obtained from the Central Intelligence Agency (CIA).

- **Botnets:** networks of devices that run bots used in cyberattacks. Typically, the devices are compromised by the attackers or on their behalf as botnets can be leased and controlled using command-and-control software. One of the biggest botnets is called BredoLab, which runs 30 million bots.
- **Motivated Attackers and Criminals:** government agencies seeking leverage; hacktivists promoting a cause, criminals seeking financial gains, etc.

Security usually entails boring work with a side of inconvenience. For example, at Voicera, we follow the principle of least privilege; when a new AWS principal[26] is provisioned, we allow it the least privileges required for it to function properly—nothing more, nothing less. From the point of view of a software developer who has to follow this policy, it's more work—that may be boring—to figure out exactly which resources and permissions are needed for the new principal; and more work means inconvenience. Had security been neglected, developers would have simply used the root account for all sorts of operations (since it has access to everything). Of course, that's insecure because it puts all the eggs in one basket: if a micro-service that uses AWS resources gets compromised, the attacker automatically gets root access. On the other hand, only allowing the least privileges to said microservice minimizes the surface area of what gets compromised. In addition, creating a separate identity for each principal makes it easier to audit who did what, which is useful not only for debugging, but also for non-repudiation.

14.16.1 Non-Repudiation

Non-repudiation is—simply put—the ability to disprove others' false claims. For example, for a bank to prove that a disputed withdrawal from an ATM actually occurred, it needs to keep meticulous records of client transactions: timestamps, amounts, locations, cards used, number of times a PIN is entered, video footage at the ATM, etc. Online service providers keep records of requests made, IP addresses from which they originated, users' digital signatures, etc. for the same reasons. Auditing operations that users and system components perform is a key mechanism in ensuring non-repudiation. One particular example comes to mind when looking for a good demonstration of the importance of record-keeping and data logs: Tesla. In 2013, Tesla's CEO, Elon Musk, used the carefully detailed data logs Tesla collected to refute claims made by The New York Times regarding the performance of Tesla's Model S cars during a test drive. Repudiation and tampering are common threats to businesses and must be mitigated to ensure information

[26] A principal in computer security is an entity that may obtain access to resources.

security. Prerequisites for non-repudiation in most systems include ensuring the confidentiality, integrity, and availability of data.

14.16.2 Confidentiality

Confidentiality in the context of information security means that systems must only allow authorized access to secrets. In other words, confidentiality controls provide access to sensitive data for the authorized principals and prevent access to said data by anyone else. Privacy is typically related to confidentiality since private data are secrets that require guarding. Techniques used for enforcing confidentiality include cryptography, identity and access management (IAM), etc. to fight threats like spoofing, man-in-the-middle attacks, cryptanalysis, wrongful information disclosure, and privilege escalation.

Cryptography techniques like encryption key exchange protocols allow communication over secure channels (e.g., Transport Layer Security) to ensure privacy and integrity. Another useful cryptography technique is hashing, which is typically used to uniquely identify sensitive data. Let's take storing passwords as an example that utilizes hashing to ensure confidentiality of data: Passwords need not to be known to a system for it to authenticate users; instead, systems should store a password's hash (that's generated by a one-way function) in order to compare it to the hash of what a user claims is her password at login. In order to reduce hash collisions, collision-resistant hashing functions are typically used in this scenario. Passwords are usually "salted" with some random input that's stored with the user's record so that an attacker who manages to obtain said records has a close-to-zero chance looking up the hash of salted passwords in a table of precomputed hashes for commonly used passwords (a rainbow table).

IAM is the discipline of declaring and enforcing rules that specify who can do what, when, and why in a system. This includes concepts like principals, users, roles, resources, policies, access control lists, etc. IAM systems are also responsible for authentication (verification of identity) and authorization (what can a principal access and do).

14.16.3 Integrity

Data Integrity ensures that data at rest and in motion stay the same and only change intentionally. Tampering is one of the main threats to data integrity, which can be caused by an attacker trying to change data to specific values or simply just tampering with the intent of data corruption. In order to protect against such threats, cryptography is used to ensure data integrity. One example is encrypting data at rest using an encryption key that's also required to read the data; AWS provides a key

management system that can be used to issue and manage keys, which can be later used with other AWS services like S3 and SQS.

Transferring data on the wire is risky business that needs to be secured; MITM attacks are a common threat when encryption is absent and plain-text data are in motion. There are many ways cryptography can help here depending on the scenario. For example, if data are moving on the wire between two ends that belong to the same company, a single key can be used to encrypt the data at one end and decrypt the received payload at the other—similar to the storage scenario. Most of the time though, the sender and the recipient are two separate entities, in which case each has their own keys. To ensure integrity (and confidentiality), data are encrypted using the recipient's public key so that no one else can decrypt the data as only the recipient should have access to the matching private key that's required to decrypt the data; this technique is known as asymmetric key cryptography. In order to use public-key encryption, keys must be exchanged beforehand in a secure manner or vouched for by a trusted authority (e.g., certificate authorities) to certify their authenticity; the latter is part of what's known as the public key infrastructure (PKI). The problem with PKI is that it's not perfect; certificate authorities can be compromised and that is a huge security gap as it can lead to phony certificates being trusted by almost all computers out there. Lack of a perfect solution in this realm is called the public key authentication problem.

Just like signing a piece of paper, digital signatures are used to certify the integrity and authenticity of data. Data are encrypted using the sender's private key, which make the data available to anyone who has the sender's matching public key; decrypting the data successfully verifies that the sender is, most likely, who they claimed to be (the proprietor of said private key). In addition to mitigating impersonation of the sender's identity, digital signatures—just like ink signatures—can be used for non-repudiation as well; the sender cannot repudiate having sent the message, which was encrypted using their private key. Signing data doesn't guarantee confidentiality since the sender's public key is used to decrypt the message, but it can be used in combination with other cryptography techniques, like public-key encryption, to ensure confidentiality as well.

Digital signatures are still prone to the public key authentication problem; PGP (Pretty Good Privacy) infrastructure provides a decentralized system known as the "web of trust" to certify the authenticity of public key ownership, which is more accessible for individuals to belong to rather than a certificate authority.

14.16.4 Availability

Availability from a security perspective is not that different from what we've been discussing from a system design perspective; systems need to stay "up" and perform their tasks while maintaining a high quality of service. Attackers can cause serious damage to systems by tampering attempts or orchestrating denial-of-service attacks that are typically distributed to evade security measures. Attacks that affect

availability aren't limited to the cyberworld; physical attacks happen, though less often, and need to be guarded against. The techniques used to maintain and defend availability of systems against attacks include: throttling so that systems don't crash when bombarded; blacklisting clients and/or IP addresses that are suspected to be attacking the system; circuit breakers; etc. And to ensure business continuity in the wake of an attack, recovery from backups might be the only way to restore lost data—proving the importance of exercising data recovery regularly.

14.16.5 The STRIDE Threat Model

To make it easier to remember major security threats lurking out there, and as part of the exercise of thread modeling, one should remember the acronym STRIDE that stands for security threats listed in Fig. 14.5.

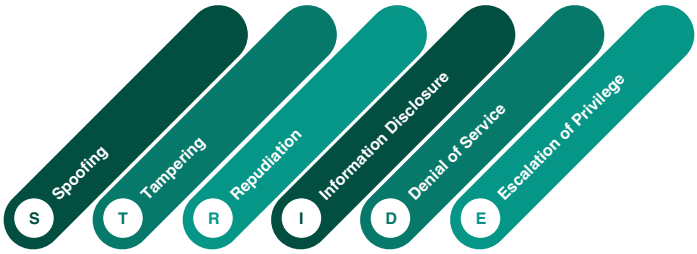


Fig. 14.5 Information security threats that make up the acronym STRIDE

Threat models help system designers answer questions that pertain to information security; for example:

- What assets require protection?
- How to classify said assets in terms of business impact?
- What security threats can affect those assets?
- How can an attacker exploit a weakness in the system to bring about such threats?
- How can we protect our assets from such attacks?
- etc.

More than any aspect of system design, security thread models need to be documented, well understood, and well reviewed. Using a model like STRIDE helps system designers group threats into those six categories and communicate them with the rest of the team. STRIDE covers the following:

- **Spoofing:** when an attacker successfully masquerades as another principal; hence fooling a security system and leading it to think that the attacker's identity is rather a trusted one; for example, pretending to be you and withdrawing funds from your bank account.

- **Tampering:** modification of data (at rest or in motion) in order to change them in a specific way (e.g., reducing the amount owed for a purchase or a transaction) or to corrupt the data.
- **Repudiation:** when attackers can get away with their false claims; for example, denying charges for purchases.
- **Information Disclosure:** exposure of information—accidentally or otherwise—to unauthorized principals who are not supposed to obtain it otherwise. This includes public disclosure of private information; for example, attackers gaining access to data transfer on the wire.
- **Denial of Service:** preventing legitimate users from accessing a service; for example, a DoS attack is successful when attackers deny users from visiting a website by bombarding the webserver with requests.
- **Escalation of Privilege:** when an unauthorized user successfully obtains permission to perform an operation that requires higher privilege.

14.17 Notes

There's a lot more to say about security and system design; while nothing tops learning about such topics through first-hand experiences, the next best thing is reading about real-life cases and learning from others' experiences—successes and, more importantly, mistakes. We highly recommend the following books to learn more about system design and security: (Martin, 2017), (Nygard, 2007), and (Howard and LeBlanc, 2003).

References

J. Bentley. *Programming Pearls*. Pearson Education, 2016. ISBN 9780134498034.

Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. ISBN 0-201-61622-X.

F.P. Brooks. *The Mythical Man-Month, Anniversary Edition: Essays On Software Engineering*. Pearson Education, 1995. ISBN 9780132119160.

M. Cohn. *Agile Estimating and Planning*. Pearson Education, 2005. ISBN 9780132703109.

David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 654–663, New York, NY, USA, 1997. ACM. ISBN 0-89791-888-6. doi: 10.1145/258533.258660. URL http://doi.acm.org/10.1145/258533.258660.

R.C. Martin. *Clean Architecture*. Robert C. Martin. Pearson Education, 2017. ISBN 9780134494166.

M.T. Nygard. *Release It!: Design and Deploy Production-ready Software*. Pragmatic Bookshelf Series. Pragmatic Bookshelf, 2007. ISBN 9780978739218.

M. Howard and D. LeBlanc. *Writing Secure Code*. Best Practices Series. Microsoft Press, 2003. ISBN 9780735617223.

Chapter 15
Thoughts on Software Craftsmanship

Why should we care about code craftsmanship? The long answer is in this chapter. First, let's see what a French philosopher, by the name of Guillaume Ferrero, discovered: the Principle of Least Effort. He wrote about it in 1894—long before a single line of computer code, as we know it today, was written. Nevertheless, we can't help but think that, had he been alive to witness how most code is written, he would have demonstrated coding as the epitome of the least effort principle: "[Coding] stops as soon as minimally acceptable results are found."

Sounds familiar? Well, we've all been there; once we find acceptable results, there's no immediate need to improve the current solution; we just want to get the task done, get that dose of instant gratification, and move on to the shiny new project. Putting out fires and plugging holes are good examples of tasks that we don't want to revisit once we found a good enough fix; and that notion is a perilous inclination, especially in the face of deadlines. It's also hard to resist the temptation of instant gratification; it takes discipline, self-control, and foresight to seek delayed gratification instead. A landmark psychological study, the Stanford marshmallow experiment, found a correlation between seeking delayed gratification and being more successful in life. I believe the same concept can be applied to code: We can stop at acceptable results and have a marshmallow to celebrate checking off a box that says we're done; or we can strive for more and have two marshmallows to celebrate the better results of going the extra mile.

But is completing a task as tempting as enjoying a sweet treat? You bet! It may even be more tempting than you think; thanks to dopamine: a neurotransmitter connected to certain feelings like reward (completing tasks) and pleasure (enjoying sweets). Usually, code requires multiple iterations to reach acceptable results; with each disappointing attempt, the level of dopamine that gets fired is reduced; when our code works—according to our acceptance criteria (test cases)—we get a pleasant surprise: increased levels of dopamine (Schultz et al., 1997). But why are we inclined to seek a new task after finding minimally accepted results for the one at hand? According to Hilary Scarlett in Neuroscience for Organizational

© Springer Nature Switzerland AG 2018

G. Lebanon, M. El-Geish, *Computing with Data*,
https://doi.org/10.1007/978-3-319-98149-9_15

Change (Scarlett, 2016), "[dopamine] levels rise when we experience something new"; so it's no surprise that our brains seek more dopamine by making us want to move on to that shiny new task. Yet, experienced coders understand that building a successful and growing business is more about making sure existing solutions scale sustainably rather than creating new ones.

We should care for the code we inherited, and the code we are writing for others to inherit someday; we should exert more effort to keep on improving all code to reach a higher level of craftsmanship than what's minimally acceptable. Michael Feathers, author of *Working Effectively with Legacy Code* (Feathers, 2004), eloquently stated: "Clean code always looks like it was written by someone who cares. There is nothing obvious that you can do to make it better." In order to justify that extra effort though, we need to have a good reason; otherwise, our actions will fade away because we weren't inspired to fight the uphill battle against our natural tendency to be lazy (by sweeping the messy code under the rug). One good reason is that craftsmanship is vital for software to really thrive; we explain why in later sections of this chapter.

So how do we define software craftsmanship? This is a hard question to answer; one may borrow an analogy of hardware craftsmanship because it's tangible; a good example is that drawn by Walter Isaacson in his book about Steve Jobs (Isaacson, 2011): "Jobs had always indulged his obsession that the unseen parts of a product should be crafted as beautifully as its façade, just as his father had taught him when they were building a fence. This too he took to extremes when he found himself unfettered at NeXT. He made sure that the screws inside the machine had expensive plating. He even insisted that the matte black finish be coated onto the inside of the cube's case, even though only repairmen would see it." Similarly, software developers should pay attention to details that repairmen—other developers—will see when they read up the source code; reading code is much more frequent than writing it. Code is not only written for machines to execute, but also represents a means of communicating with the next person who will read your code (or your future self).

Besides code, big data represent a set of craftsmanship challenges that are shaped by the four V's of big data: Volume, Velocity, Variety, and Veracity. It's challenging to keep track and make sense of the colossal number of data points collected and processed at a mind-boggling throughput; the various data sets, formats, and sources we handle; and the high bar of quality that enables us to trust said data.

In this chapter, we explore ways to elevate the state of computing with data from minimally acceptable result to balancing craftsmanship and pragmatism.

15.1 Guiding Principles of Crafting Big Data Systems

"As to the methods there may be a million and then some, but principles are few. The man who grasps principles can successfully select his own methods. The man who tries methods, ignoring principles, is sure to have trouble."—Ralph Waldo Emerson

Fear not, we also explore methods that we believe are helpful in the pursuit of software craftsmanship; that said, your mileage may vary; you should think of what works best for you and come up with your own methodologies.

15.1.1 Sustainable Rapid Growth

While a product is starting, shipping features might trump all other priorities; time-to-market (TTM), feature richness, and business requirements may force development teams to take loans (tech debt) to get things rolling. Such approach is unsustainable—debt must be controlled. The bigger a product gets, the more scrutinized processes have to be; otherwise, failures become more impactful, more noticeable, and more probable. In terms of the spectrum of software quality: at one end of the spectrum, we have never-ending fires to fight in the production environment (flying low), and at the other end, we have years of formal software verification methods before shipping (flying high). For the vast majority of software products out there, there's a sweet spot in between that enables us to iterate fast, release often, keep innovating, and delight users; and doing it all while maintaining a high-quality bar—and more importantly—our sanity. To reach that nirvana, there are a few requirements:

- **Fostering a culture of self-awareness, balance, and support:** being aware and critical of tradeoffs we have to make; understanding when and why such tradeoffs are necessary; and having the support of upper management when the time comes to pay off tech debt.
- **Automation:** automating quality processes, as much as possible, to reduce friction; asking hundreds of developers to spend numerous hours on reformatting code to adhere to standards is much harder than asking machines to do it.
- **Understanding the rationale:** this is the most important prerequisite; enforcing rules without explaining the rationale behind them leads to frustration and resentment. Sometimes, the best explanation is showing the ramifications of failing to follow said rules; Icarus might have been more careful had he seen a failed attempt of flying too high.

15.1.2 Balancing Rush Delivery and Craftsmanship

The master craftsman, Daedalus, and his son Icarus tried to escape from exile in Crete by taking flight using wings made from feathers, and wax. Daedalus warned his son of complacency and hubris: to fly "neither too low nor too high, so the sea's dampness would not clog his wings or the sun's heat melt them. Icarus ignored his

father's instructions not to fly too close to the sun, whereupon the wax in his wings melted and he fell into the sea."[1]

What can we learn from the talented craftsman in this Greek myth? One takeaway is the importance of balance. In software development, in order to achieve sustainable progress, we need to strike a balance between striving to create the perfect product (flying too high), and shipping the product upon achieving the minimally accepted results.[2] There's no standard formula of what percentage of time should be spent on improving the quality of existing code vs. adding new features; many factors—like the age of the product—influence such decisions. Good software development balances craftsmanship and pragmatism.

For all intents and purposes, code is never perfect; that's actually desirable; you can keep on testing every single scenario, and squeeze every single bit of performance out of your code, and it will still be less than perfect. As a professional coder, you don't have infinite time; and—in most cases—you need to ship a solution as soon as you can to solve a problem in the real world. That's why you need to balance between getting things done, and getting them done as close as possible to perfect. Yes, we need to be honest with ourselves about the state of our code; we should be our worst critics, but we should also learn when to ship it to start the next stage of its lifecycle. And most importantly, we need to be aware of such compromises and their consequences.

So your coding task is done and the feature works (or at least we think it does); how fast should you ship it? For a platform where contributors add said feature, they may expect to see it in production upon the next scheduled release; for a user-facing product, the product manager may push for an expedited release to shorten the TTM. While innovation speed is an important objective, software quality is also key; so how do we confidently ship high-quality changes? We can start with the usual: testing at various levels (unit, end-to-end, integration, stress, performance, compatibility, etc.) and taking safety measures (e.g., circuit breakers, big red switches, Andon cords, etc.) to isolate potential failures. Such measures require thorough monitoring, and effective repair systems to detect, and remedy failures in production. We encourage readers interested in this topic to read (Nygard, 2007) to learn more about designing and deploying fault-tolerant systems. Investing in such measures—justifiably—introduces delays, and that's just the tip of the iceberg; faster cadence means that things may break more often—adding more stress on operations teams and causing burnouts; and it also means that we have to invest in better logging and debuggability in the code (more than the usual). Managing the lifecycle of software changes is crucial to its survival; and managing tech debt is key for it to thrive. Software has to be cynical—especially towards new changes—to operate reliably, and following said processes may come at the cost of delayed general availability (TTM) of new changes—a worthwhile investment to make.

[1] https://en.wikipedia.org/wiki/Icarus.

[2] The principle of least effort: Efforts to reach a goal stop as soon as a minimally acceptable result is achieved.

In the midst of the battle that is software development, one tends to forget about the war that is making sure the software you create thrives in the real world. To win the war, you may have to sacrifice myopic gains in favor of long-term ones. Cutting corners during development can have costly ramifications in production like outages and degraded quality of service; you'd cost your company orders of magnitude more than what it would have taken you to do it right the first time. For example, if you decide to depend on a manual process to ship your software because it would have taken more time to automate it, you will eventually make a mistake—according to Murphy's law[3]—and ship faulty software to production; the damage could be anything from a subtle UI bug to a business-ending one; even the former can cause enough damage to your brand that makes the cost of automating said process look like pocket change in comparison. You should try to do the following if you need to justify the upfront cost of investing in software quality:

- Keep track of losses: outages, moratoriums, etc.
- Put a dollar value on each of them.
- Conduct postmortems to trace them back to development choices.
- Estimate the opportunity-cost of choosing a reasonable alternative over the choice that caused the loss.
- Determine whether or not it was worth it.

When we immediately see the consequences of our choices, we find it easier to establish a causal nexus: hitting a piece of furniture's sharp corner causes pain; similarly, pinpointing a bad code change that failed stable pre-commit tests is a no-brainer—it's the code you're trying to add. On the other hand, debugging an issue caused by a bug in a year old code change is a totally different story; you most likely forgot about the choices you had made a year ago, and your thought process is plagued by synchronicity and coincidences. The poor choices you make in the early stages of software development will hunt you down, ruthlessly.

15.1.3 Frequent Reassessment of Design Decisions

Design decisions that we make today determine the destiny of the software product we're building, and some of them can be too costly to change. The challenge is: what seemed like the best option today might not be the optimal solution in the future; in fact, it shouldn't be because there's almost always a cost associated with planning for future growth (new features, scale, etc.)—that's why simplicity is key: "The best approach is to create code only for the features you are implementing while you search for enough knowledge to reveal the simplest design."[4]

[3] Murphy's Law: Everything that can go wrong will go wrong.

[4] http://www.extremeprogramming.org/rules/simple.html.

Again, there's a balance to strike here: planning for the future is righteous, but we need to be aware when we're hitting diminishing returns—it's all about the return on investment (ROI). As a product rapidly grows, aspects like scalability and capacity planning become greater-than-ever determinants of its success. "Grow fast or die slow" is the title of a study of rapid-growth software companies that McKinsey released in 2014,[5] which—as the title suggest—found that "[if] a health-care company grew at 20% annually, its managers and investors would be happy. If a software company grows at that rate, it has a 92% chance of ceasing to exist within a few years." Rapid growth at extraordinary rates requires planning for economics of scale and efficiency—aspects that small companies can afford to neglect at the beginning in favor of convenience and faster iterations.

At every growth milestone, it's of vital importance to understand and question the rationale behind the status quo; and the design decisions that led to it: Why is so-and-so the way it is today? Do we believe the rationale still holds? How can we improve on it to cope with the product's rapid growth? etc. **Takeaway:** Software that was built to serve thousands of users may not work as expected when asked to serve millions of users; revisit design decisions regularly.

15.1.4 The Incremental Cost-Effectiveness Ratio

We talked about evaluating the ROI when deciding whether or not to invest in features that aren't immediately required, and implementing the simplest thing that does the job. In some cases, we actually know that we want to invest in a complex system to solve problems we will most probably face in the near future. In such projects, I'd like to borrow a metric from the health-care industry: The Incremental Cost-Effectiveness Ratio (ICER), "a statistic used in cost-effectiveness analysis to summarize the cost-effectiveness of a health care intervention. It is defined by the difference in cost between two possible interventions, divided by the difference in their effect."[6]

$$\text{ICER} = \frac{C_1 - C_0}{E_1 - E_0}$$

C_1 and E_1 represent the cost and gain of investing in the forever (strategic) solution, respectively; conversely, C_0 and E_0 represent the cost and gain of implementing a stopgap (tactical) solution instead, respectively. The opportunity cost here is a matter of utmost importance; we must be cognizant of the tradeoff between realizing short-, and long-term gains. That said, we may be able to have our cake, and eat it too if we can accrue E_1 as we build the strategic solution in

[5] http://www.mckinsey.com/industries/high-tech/our-insights/grow-fast-or-die-slow.

[6] https://en.wikipedia.org/wiki/Incremental_cost-effectiveness_ratio.

a bottom-up fashion. Ideally, each project milestone will contribute $\frac{E_1}{n}$ where n is the number of milestones required to build the strategic solution; in which case, the cost of building the strategic solution per milestone should be $\frac{C_1}{n}$, which shall be used to estimate ICER_m for each milestone m. The latter may be compared to a specified threshold used to determine how much cost we allocate for the strategic solution vs. other tactical features in a given milestone based on short-term business needs. Assuming that ICER is constant over time, the goal here is to accrue uniform incremental gain as we invest in building the strategic solution. **Takeaway:** to justify building big systems for the future, analyze the cost-effectiveness of immediate value added at each milestone.

15.1.5 Repairing Broken Windows Frequently

Even though tech debt may be a necessary evil to add new features faster, it's still evil—and it must be controlled. Death by a thousand papercuts is a plague that hits many big data systems, but you can prevent it by detecting and scrutinizing symptoms like:

- Letting a code review comment slip between the cracks because the author had a bigger fish to fry or a deadline to meet.
- Accepting "this scenario is never going to happen" as a rebuttal to a potential design defect you pointed out in a review.
- Code smells; e.g., adding more lines to legacy code functions that already have too many lines to follow without taking notes.
- etc.

Such symptoms are known as broken windows, and they must be repaired before things get out of hand like what happened in two separate events:

- In 1969, Philip Zimbardo, a Stanford professor, abandoned a car in a wealthy neighborhood. He left the hood up and removed the license plates. It sat there untouched for a week. Then, he deliberately smashed a window. Soon after, the car was completely destroyed by well-dressed "vandals."
- In 1993, New York City's mayor, Rudy Giuliani, started the Zero Tolerance program. Police started cracking down on small infractions: broken windows, graffiti, etc. As a result, rates of both petty and serious crime fell significantly.

The aforementioned events were set in motion to test the Broken Windows Theory, which was covered in the March 1982 edition of The Atlantic Monthly[7]: "Consider a building with a few broken windows. If the windows are not repaired, the tendency is for vandals to break a few more windows. Eventually, they may

[7] www.theatlantic.com/magazine/archive/1982/03/broken-windows/304465/.

even break into the building, and if it's unoccupied, perhaps become squatters or light fires inside."

So—you may ask—what does a theory in criminology have to do with software? It's a metaphor from The Pragmatic Programmer (Hunt and Thomas, 1999) (a good read that we recommend for software development best practices); known issues, bad design, and poor code hygiene are a few of software's many broken windows. Code churn—that doesn't actively fix broken windows—will break more windows. It's analogous to the Second Law of Thermodynamics: "The state of entropy of the entire universe, as a closed isolated system, will always increase over time." This means that every time one adds a new feature or fixes a bug, the amount of disorder in the system increases, unless one does something about it. Lehman's second law of software evolution (Lehman, 1980) captures the importance of such balance: "As an evolving program is continually changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain or reduce it."

Consider an example of library development, where changes had been made to publicize a few classes that were supposed to be private. Did you hear the sound of shattered glass? Good. Now think about the tendency for taking more shortcuts like that, how bad could it be? The harm was already done, and it's already broken. This is the critical point at which the building gets broken into and fires start. A 2014 paper titled, "Impacts of Design Pattern Decay on System Quality" (Dale and Izurieta, 2014), found out that "[as] systems evolve, the coupling between pattern and non-pattern classes tends to increase and the intended design patterns can become obscured by code that violates the pattern's intended purpose."

Take the same concept and apply it to code smells: spaghetti code, cyclomatic complexity,[8] nasty dependencies, long methods, large classes, sprinkled string literals, etc. Find them and understand why they exist in your codebase. Think of the technical debt that your team has accumulated along the years, and have a plan to pay it off. Do you feel overwhelmed? Don't panic; if NYPD in the 90s can do it, so can you. Here's a recipe that you can follow after committing a change in a module (which makes it an evolving code construct to which Lehman's law applies):

- Triage debt in terms of priority, severity, scope, and risk.
- Pick the debt items you want to fix, as you see fit.
- If needed, add covering tests for the existing code to avoid regressions.
- Make the changes—following your team's change submission process.

It's also worth noting that you should:

- Figure out the right balance between business-related changes and paying off the technical debt (remember Icarus); sometimes, doing the latter allows you to make reach the former goal much faster (e.g., refactoring legacy code may help you add new features faster).

[8] Cyclomatic complexity—simply put—is the number of independent code paths in a program.

- Create different change lists—when paying off tech debt—in case you need to undo any of them. It's also easier for your team to review them separately.
- Payoff tech debt while the module is still fresh in your mind; context switching is costly.

Takeaway: Repair broken windows, and stop new ones from getting broken.

15.1.6 System Design Priorities

Creating big data systems requires focus and prioritization of system properties:

Security

Security can't be emphasized enough; it must be prioritized above all aspects of a big data system. Protecting data assets is job number one for all (and not just system or security engineers). Depending on the kind of business you're in, a single vulnerability could be business-ending. Ensuring a culture of cyber security and hygiene is crucial to protect big data projects and allowing them to thrive in a safe environment. Cyber security is the responsibility of the organization to protect its assets and enforce following best security practices in building its systems; cyber hygiene is the responsibility of individuals to complement the organization's efforts by performing their duties with security in mind and taking strong security measures to ascertain that the data they work with remain protected; that includes practices like using multi-factor authentication for all accounts they use (online security) and using a privacy filter when working outside the office (physical security).

Correctness

A system that is skillfully architectured, highly performant, and extremely secure is useless if it's incorrect. Ensuring the correctness of distributed concurrent systems is hard, especially when they scale to thousands of machines! Few systems require a formal language, like TLA+, to verify the correctness of the system in advance; for the vast majority of most systems, design by contract is sufficient to ensure correctness.

Quality

According to Murphy's law, anything that can go wrong will go wrong; maintaining a high bar of quality along features' lifecycle (specifying requirements, designing, coding, testing, shipping, maintenance, etc.) is key to reduce the number of

things that will go wrong. For example, bugs infest code that was not tested for the scenarios that trigger them. 100% code coverage of newly added code is a necessity; would you be comfortable shipping code that has never run before? That said, 100% coverage doesn't mean testing was thorough; exercising code without proper verification is inadequate; the raison d'etre of code coverage is not to feel complacent about what was covered, but rather to investigate what was missed.

Stability

Failures are inevitable; embrace this maxim and design the system features to anticipate and mitigate failures. Failures happen due to various reasons, and they come in many forms; to list a few: hardware fails, software has bugs, and people make mistakes. Stability is the real test for consistent correctness in spite of inevitable failures. Understanding failure modes, system dependency graphs, failure domains, mean time between failures (MTBF), etc. is key to coming up with a mitigation plan. Big data systems depend on many externalities—have a plan for when they fail; understand how the system is going to react. Cynicism and skepticism are essential when designing reliable systems, but one should also know when to stop layering safety nets.

Usability

A hard-to-use system will just drive potential users away regardless of how powerful it is; all user interfaces should be self-explanatory and easy to use: configuration, tools, GUI, APIs, metrics, logs, etc. Think as a user of the system first before designing a feature.

Excellence

A service level agreement (SLA) is not a goal; it's a worst-case scenario. One should strive for maximal availability, throughput, and performance. SLAs must be agreed upon—even with users in the same organization—ahead of time, and they must be regularly reviewed.

Operability

Deploying a feature to production is not the end of its lifecycle but rather its emergence. Designing for operability is crucial to keep the system running. Just like how a new engine suffers from impurities during the break-in process and requires closer inspection in the beginning, adding new features to the system destabilizes it and require scrutiny; thorough logging, instrumentation, and monitoring of new

code (which should be relaxed a bit after the new features stabilize) is a must. Investing in automated remediation and forensics (e.g., troubleshooting guides and processes to investigate production issues) saves precious time when the system is down and needs to be restored, which minimizes mean time to repair (MTTR); thus, to pay a one-time development cost that reduces an ongoing one: operational expenditure (OPEX) is truly beneficial.

Simplicity

Einstein said it best: "Everything should be made as simple as possible, but no simpler." That said, assuming simplicity doesn't always contradict designing for reusability; systems can be generic and simple at the same time. Unix was developed using 9 KB of memory; it had to be elegantly simple, just like how da Vinci described it: "Simplicity is the ultimate sophistication."

Efficiency

High throughput and performance are necessary for the success of a big data system; efficiency is a favorable goal and should always be a major factor in design and coding choices; achieving said goal might tempt developers to optimize prematurely—"the root of [much] evil"[9]; starting with the simplest solution that meets reasonable performance goals is a success; further performance improvements should be pursued only when need be. Simplicity is one of the rules of extreme programming, as mentioned above, and it's summarized in the following statement: "The best approach is to create code only for the features you are implementing while you search for enough knowledge to reveal the simplest design."[10] Simplicity and performance go hand in hand, as Gordon Bell once said: "The cheapest, fastest and most reliable components of a computer system are those that aren't there."

Learning

Humans represent the most important factor in the success of a system (or any piece of software for that matter); they build it, use it, maintain it, and keep improving it. Investing in humans is essential. Most engineers get overwhelmed by day-to-day tasks and ignore personal growth, which is unfortunate. Learning a new technology, understanding what goes under the hood in a framework, or knowing that a library

[9] The full version of the quote by Donald Knuth is "We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil."

[10] http://www.extremeprogramming.org/rules/simple.html.

that does exactly what you've been trying to do exists can save precious time when most needed. That said, knowing won't eliminate mistakes, but it allows us to only make new ones.

Most developers love abstraction; the higher the level, the better. When ASP.NET added abstractions like the login control and membership providers that meant that a login page can be built in a few clicks; no more fiddling with hand-made SQL tables, hashing, salting, stored procedures, SQL connections, SQL injection prevention, server- and client-side input validation, printing out your own HTML, password reset workflows, etc. The result is a huge productivity boost; developers are able to add more features than required, and deliver the project earlier than expected. Most developers use such frameworks focus solely on how to glue things together; only a small percentage of them would smash the building blocks to see what's inside. In all fairness, some framework are too complex to grok, but you can tell what a framework is doing at a high-level if you have accomplished the same objective the hard way; learning can save you so much time.

Peeling the onion and understanding what's under the hood is key in the following situations:

- **Debugging:** Knowing the insides of abstractions before using them is extremely beneficial, not only while debugging, but also while development—to avoid the need to debug all together. For example, calling Java's `Iterator::remove()` without checking the behavior of the class that implements it can be dangerous as its behavior may be unspecified if the implementing collection is modified.
- **Poor Documentation:** In this case, you have no other option but to take a peek under the hood. Luckily, documentation systems nowadays recognize the need for implementation remarks like JavaDoc's `@implNote` and C#'s `<remarks>` tag. An example that jumps out here is documenting the time complexity of an operation.

15.2 Coding Style

Coding to solve a problem is like making a door: one can nail a few parts together and call it a day; some take the time to learn how to use the tools of the craft and take pride in creating something beautiful, durable, intuitive, unobtrusive, simple, consistent, extensible yet minimalistic, and—above all—useful. Many of those qualities overlap with Dieter Rams' principles for good design (which he demonstrated in a 2009 documentary about industrial design called Objectified) and are applicable for coding style and software design as well.

Coding is communication and professional programmers are good communicators: they express their desires to machines in terms of instructions the machines can follow unambiguously; more importantly, they communicate to readers of their code (including their future selves) what they really wanted to do. In fact, technical

communication is such an integral part of the job that companies like LinkedIn dedicated a whole interview out of the handful each candidate has to go through in order to gauge that skill. Writing code to achieve a task may be easy; machines will understand what you mean as long as it's correct, but writing code—that achieves the same task—for humans to understand is not as easy. Unfortunately, we've seen many instances in the industry where the latter factor was brushed off as "details that don't matter"; attention to details can tell a lot about code hygiene and longevity; and a lot more about the coder's craftsmanship. Donald Knuth sums it best: "Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do."

Coding follows a production process; you take an idea and you ship it as software. One of the ways to improve said process is the 5S methodology, which stems from the following Japanese words:

- **Seiri (Sort):** When code is sorted, its purpose is clear to the reader; one must remove dead code and vestigial features that cause software bloat.
- **Seiton (Set in Order):** When the code is neatly arranged, readers can follow it easily and reduce cognitive load; such reduction is essential to increase learning capacity and understanding.
- **Seiso (Shine):** When the code is clean, it doesn't smell; code smells indicate bad design (e.g., contrived complexity), which can be fatal for software products. One of the best books on software craftsmanship is called *Clean Code* (Martin, 2009) (by Robert Cecil Martin, aka Uncle Bob).
- **Seiketsu (Standardize):** When coding is standardized, it's easier to follow and automate best practices.
- **Shutsuke (Sustain):** When coding is a discipline, a team becomes more productive; code reviews become discussions about improving the proposed solution that aren't muddied by bickering.

Each of us has witnessed at least once an engineering team fighting a civil war over the use of tabs vs. spaces; it's funny because it's—sadly—true. Teams that don't fight said wars over coding conventions either don't care about them or understand that the primary goal of having coding conventions is standardization (and not picking the best style). Everyone should care about coding conventions and style; James Coplien wrote in *Clean Code*'s foreword that "consistent indentation style was one of the most statistically significant indicators of low bug density"; inconsistencies lead to attention deficit—a byproduct of increased cognitive load when reading through the code. Daniel Kahneman, author of *Thinking, Fast and Slow* (Kahneman, 2011), describes how we have two modes of thinking: system 1, for thoughts we form automatically; and system 2, for more effortful thoughts. Failing to find the next code statement where your brain expects it to be causes a shift from the effortless system to the effortful one. Even after moving to system 2, bad style distracts you from focusing on the logic where you should have paid all of your attention to find those bugs. You see—as Kahneman pointed out—phrases

like "pay attention" and "attention deficit" are not mere coincidences; attention is a budget that has to be efficiently allocated and wisely spent.

Coding style is subjective and depends heavily on what the code author is used to; hence, conflict ensues when someone tries to change your coding style because that means more work for you and a conscious effort to reorient your (cognitive) muscle memory—recall the principle of least effort. There's a simple solution for that: automate code formatting and style rules enforcement as much as you can. It gets better: you can use a tool like Naturalize[11] that uses machine learning automatically understand coding conventions in your source tree and help make the coding style consistent; it can even autosuggest accurate method and class names[12]! The goal is to make your codebase read like it's written by the same author to reduce cognitive load while writing and navigating through code. In this section, we explore key aspects of coding style that help achieve this goal.

15.2.1 Naming

Precise and accurate communication in code starts with impeccable choice of names. Think carefully about names and spend time debating—in your mind and in code reviews—which ones best describe the entities to which they refer. It's time well spent; we spend disproportionately more time reading coding (than writing it). It's like choosing what goes on a traffic sign; you want to make sure that it's impeccable because lives depend on it. Names should be self-explanatory; one quick look at a name should tell you what it refers to, its raison d'etre, how to interact with it, etc. Another requirement is exactness: names shouldn't carry more meaning than what they actually have. For example, one may refer to "a number of things of the same kind that belong or are used together" as a set,[13] but this has the side-effect of tricking you into thinking that a variable (say, `studentSet`) is of the type set, which indicates that it contains unique elements. The variable name `students` is a more exact name to refer to a group of students.

In *Thinking, Fast and Slow*, Daniel Kahneman talks about two systems of thinking: system 1 and system 2; system 1 is associated with fast effortless thinking, and one of the things it does best is understanding language. We use language while naming code entities, which should be easy to understand by system 1; for starters, they should be pronounceable. One of the examples that the author of chapter two of *Clean Code*, Tim Ottinger, mentioned of bad names is a variable called genymdhms; he used to pronounce it as it reads: gen-yah-mudda-hims, until a teammate "corrected" him, and told him that it's actually pronounced "gen why emm dee aich emm ess"; Can you imagine how many brain cycles were wasted on

[11] http://groups.inf.ed.ac.uk/naturalize/.

[12] http://groups.inf.ed.ac.uk/cup/naturalize/.

[13] https://www.merriam-webster.com/dictionary/set.

calling this variable by its horrendous name and linking the name back to what it means? A one-time cost of choosing a good name would have saved the team a recurring cost which they kept on paying in the long run. A better name would have been descriptive of the thing it's trying to capture even if it became (a bit) longer. Ottinger tried to highlight the importance of choosing a good length for a name in the following statements:

- "In [terms of searchability], longer names trump shorter names, and any searchable name trumps a constant in code."
- "Shorter names are generally better than longer ones, so long as they are clear. Add no more context to a name than is necessary."

At first glance, the two statements may sound opposing; However, a deeper read would reveal that they actually complement each other; once can sum this concept up as a single statement: Don't be afraid of using long names; be expressive, but not verbose.

Names are not set in stone; code entities should be renamed when they change and poor names should be replaced. That said, there are cases where renaming comes with caveats:

- **Tangled code reviews:** merging a change in logic (e.g., a bug fix) with an improvement like renaming makes it harder to review the more important change; for example, renaming a public Java class requires renaming the file in which it resides (let alone the fact that it breaks backward-compatibility of the public contract); a bug fix in that class is now lost in a myriad of green/yellow (new changes).
- **Subtle bugs:** A few years ago, I stumbled upon a private member field that was poorly named, so I changed its name for the better—or so I thought—only to discover that I broke backward-compatibility of client-server communication. It wasn't clear that my change was the culprit at first, since a suite of extensive unit, and E2E tests couldn't find the issue before it was committed; compatibility tests—which weren't as frequent—caught the bug, and saved the day. This sort of issues is extremely costly because it halts the assembly line, so to speak; finding the root cause was also an effort that required a lot of time, and debugging (deploying a client, and a server of different versions to reproduce the issue can be an annoying exercise due to product idiosyncrasies). It turned out that a legacy binary serializer[14] crashed as it tried to deserialize a payload that included instances of the class I changed; in .NET, "[during serialization], the public and private fields of the object and the name of the class, including the assembly containing the class, are converted to a stream of bytes, which is then written to a data stream." We mainly used data contracts,[15] which provide a more flexible alternative, and allowed us to explicitly name your data contracts (so

[14] https://msdn.microsoft.com/en-us/library/72hyey7b(v=vs.110).aspx.

[15] http://msdn.microsoft.com/en-us/library/ms733127(v=vs.110).aspx.

that underlying types can be renamed independently), but we had to support that legacy behavior. The lesson learned here is that you better make sure that renaming—as innocent as it may look—doesn't break your code.

15.2.2 Functions

No matter what programming language you use, there's almost always a construct that allows you to encapsulate a set of instructions, like a subroutine or a function, to reuse them and/or to label them. Functional decomposition of a program is a great way to simplify the way the author communicates intent and flow of instructions; albeit a complex program would still work if the entire thing was written as a single slab of code, it sure is a hellish exercise for a human to read it. Take a quick trip down memory lane and think of answers to the following questions: How many times have you stared shockingly at a function that's a few hundred lines long? How long was the longest function that you've ever seen? Ask your fellow colleagues the same questions and the answers might surprise you; a bigger surprise is what you'd hear as inconsistent answers to the following question: What's a good range for the length of a function?

There are many studies out there that looked into function length as a metric of software quality and they offered different findings: For example, research on measuring software design quality (Card and Glass, 1990) found that long (65+ lines) functions are cheaper to develop per LOC (line of code); another claimed that short functions had 23% more errors per LOC than longer ones (Selby and Basili, 1991). Uncle Bob, author of *Clean Code*, and Steve McConnel, author of *Code Complete*, recommend that functions should be shorter than 20 and 200 lines respectively; that's an order of magnitude difference between both recommendations! Each author offered a good rationale and subjectively opposing viewpoints. After reading §7.4 of *Code Complete 2* to contrast it with Uncle Bob's succinct section on keeping functions small, we tend to agree more with the latter—as we share similar tastes when it comes to function guidelines.

More important than the length of a function is its reason of existence. The Single Responsibility Principle (SRP) that Uncle Bob introduced in an article[16] about the SOLID principles of Object-Oriented Design (OOD) is key here. McConnell, in chapter 7 of *Code Complete 2*, defines a routine as "an individual method or procedure invocable for a single purpose." He also shows an example of a low-quality routine and calls out the fact that it does more than one thing. Determining that one thing for a function to do can get tricky sometimes and it's often easier to keep a bunch of local variables in scope for when you need them later on inside the same function (instead of decomposing it to multiple functions). Functional decomposition is just like breaking your ideas into paragraphs when writing an

[16] http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod.

essay: each paragraph has a single idea and each new idea calls for a new paragraph; you can take the same advice from Strunk and White—authors of *The Elements of Style*—by simply substituting function for paragraph: "Make the [function] the unit of composition: one [function] to each topic."

Breaking the Single Responsibility Principle (SRP) can be tempting even for shrewd API designers, we can find an example of what Uncle Bob calls command-query conflation in Java's Set::add method:

```
boolean add(E e)
```

The `add` method adds the specified element to the set and returns true if the element to add was not found in said set; conversely, it leaves the set unchanged and returns false if the element to add already exists in said set. That sounds like a convenient way to kill two birds with one stone, but things can get confusing. To adhere to the SRP, we split the single method call into the following to make it more readable:

```
if (set.contains(element)) {
  return false;
}
set.add(element);
return true;
```

We find the command-query paradigm extremely pragmatic in certain situations, especially when it comes to atomicity in concurrent programming. Take, for example, the following atomic operations:

- **test-and-set:** writes to a memory location, and returns the old value.[17]
- **compare-and-swap:** compares the value x at a given memory location to a given value y, and sets the former to a given new value z only if x is equal to y; the return value indicates whether or not the swap occurred.[18]

A concrete example of the compare-and-swap atomic operation is .NET's Interlocked.CompareExchange,[19] which violates SRP's directives—yet it's a must have method for pragmatic concurrent programming. You can also find the same pattern used in database APIs where a single method writes a (new) value into a record and returns the old value to the caller; another more prominent example is a function that inserts a record into a table, and returns the record's auto-generated primary key. The challenge, just like with almost everything else in software design, is finding the right tradeoffs between following the best practices and breaking them to be more pragmatic.

[17] https://en.wikipedia.org/wiki/Test-and-set.

[18] https://en.wikipedia.org/wiki/Compare-and-swap.

[19] https://msdn.microsoft.com/en-us/library/801kt583(v=vs.110).aspx.

15.2.3 Comments

Do you find it frustrating when you pull a door handle only to find out there's a sign that reads "Push"? Well, you're not alone; Don Norman—after whom such doors are named and the author of *The Design of Everyday Things*[20]—couldn't agree more. It's frustrating because the design bug was masked by a comment! Instead of taking the time to think about what makes a good design, a comment—slapped on as an afterthought—gave the designers a false sense of coherence when they assessed their design. Design should be intuitive: a handle means pull; a flat panel (or a bar) gives you no other option but to push. Code should be the same way; you should only read code to know what said code does; comments are to be added only as a last resort.

Comments are spells of magic; they charm the reader into thinking they must be truer than code; after all, they are the sacred instructions to the human commanding the machine. There's a catch though, the machine—by design—ignores those comments and thus it can't verify their correctness. A change in requirements warrants a code change, but not necessarily a comment change (even though it should). Code represents the ultimate truth of what a program does; comments that reiterate that are "alternative facts."[21] Here's an example of a conflict between what the code does and what the comment says as they got out of sync:

```
if (isFoo || isBar) {
  doSomething();
} else { // !isFoo
  doSomethingElse();
}
```

You can see how easy it is to miss that comment when adding `isBar` to the `if` condition, especially when the `if` block is long (which is another code smell). Test cases won't fail because of that; modern code review tools may choose to hide that `else` block since it's not in the context of the change being made; and I don't know of any static analysis tools that would catch this discrepancy, which will cause future readers to do a double-take in order to wrap their heads around what's going on there; that's why such comments are costly. Code is the source of truth, and comments are lies until proven otherwise. That said, there are cases where comments are required. For example, public API documentation (like javadoc) that gets published for others to read in order to understand how to interface with a software library. Another example of good use of comments is explanation of intent and rationale.

Before resorting to writing a comment, take the time to explore other options like renaming a variable or decomposing a function into to multiple ones to convey the message of said comment and eliminate the need for it. Another option is to use a

[20] https://en.wikipedia.org/wiki/The_Design_of_Everyday_Things.

[21] https://en.wikipedia.org/wiki/Alternative_facts.

strongly typed construct that replaces a comment like an annotation,[22] which is a superior alternative to good comments. Let it be a comment to warn about thread-safety, a `TODO`, or a testability hook clarification; an annotation has the following advantages:

- provides a declarable intention that can be checked at build-time.
- has precise targets; for example: CONSTRUCTOR, FIELD, METHOD, PACKAGE, PARAMETER, TYPE, etc.
- could be reflectively accessed at runtime.
- shows up in generated documentation.

`@VisibleForTesting` and `@Expiration`[23] are two examples of Java annotations that are meant to replace comments.

The takeaway about comments is to think twice before writing a comment in hopes of eliminating it and if that fails, think thrice before writing it to word it impeccably.

15.2.4 Formatting

I've seen—time and again—the correlation between poor coding style and the number of bugs in a program; it goes back to my days in college when friends of mine would ask for help debugging their code; the first thing that I would do, before even reading the code, was to reformat it to fix inconsistent indentations. It used to irritate my friends because they liked it the way it was, but they eventually gave in after I had explained why: I couldn't read it otherwise! It's like inconsistent formatting is kryptonite; it weakens one's cognitive abilities. I couldn't understand why that was the case until I learned about *Thinking, Fast and Slow*, which we discussed earlier in this chapter, and attention deficit. Paying attention to inconsistent formatting distracted me from following the program's logic and debug the real issues. In fact, reformatting code to adhere to best practices can help you reveal issues right away; here's an infamous example—the dangling `else`:

```
if (foo)
  if (bar)
    doSomething();
else
  doSomethingElse();
```

The dangling `else` is ambiguous, it's hard to tell to which `if` statement it corresponds without braces in place. Even worse, the indentation can be misleading

[22] Attributes in Java, or attributes in .NET's, are metadata that can be added to code constructs to tag them.

[23] https://github.com/elgeish/ExpAnn.

as it may conflict with what the compiler decides; for example, most C compilers associate an else statement with the closest if, in which case the example above is extremely infuriating to readers who would have assumed the opposite behavior, thanks to the hint provided by an incorrect indentation (or lack thereof). That's why early programmers figured out that automatic code formatting is a great way to avoid such pitfalls; In 1967, Bill Gosper created a Lisp called GRINDEF (grind function for pretty-printing) to format Lisp programs. Moreover, some languages, like Python and Haskell, have strict rules about indentations and use them for code structuring; for example, Python hits two birds with the same stone: instructing the compiler to define a block using indentation and using the same rules to be more human-readable. The above dangling `else` example looks unambiguously clear in Python:

```
if foo:
  if bar:
    doSomething()
else:
  doSomethingElse()
```

Other languages, like Go, are opinionated[24]; they don't spell out a style guide to follow but rather provide tooling to automate code formatting and encourage developers to run said tools automatically in IDEs that support file-save hooks and/or before committing the code to a version control system. For example, the `gofmt` tool is the idiomatic way to format Go code, which makes it easier for Go developers to read others' Go code and contribute to it without having to worry about style differences. Another good example from Go is mandatory braces for control structures (e.g., `if` and `for` statements) even for single-line blocks. Opinionated languages, like Go, seek to follow the Principle of Least Surprise[25] to reduce confusions caused by idiosyncratic and variable coding styles; towards that end, they elevate conventions to become requirements.

Much of coding style shows in how you format your code. In fact, Uncle Bob goes to an extreme and emphasizes code formatting over getting the code to work because style survives while code keeps on changing; the discipline of following style rules consistently is something that will stick with you for a long time. More importantly, your team must agree upon the same set of rules so that the coding style across a project is coherent; the best way to do so is to automate the application and validation of said coding style rules. Code—as the name suggests—is already cryptic; having various styles in the same body of work would make it even harder to comprehend. Ideally, anyone who can read the language in which a piece of code is written should be able to understand what it's doing without the need to suffer because of style discrepancies. Team rules should make it easier for team members to work together on the same project; company rules should make it easier

[24] https://golang.org/doc/effective_go.html#formatting.

[25] https://en.wikipedia.org/wiki/Principle_of_least_astonishment.

for employees to move between teams seamlessly; language conventions (like Go's) should make it easier for new hires to write code when joining a new company and for everyone to contribute to open source solutions.

Code organization guidelines are extremely important; they can make reading a file feel like a walk in the park or like running around in circles trying to untangle a sticky web of call graphs.[26] Vertical ordering and conceptual affinity of functions and other coding structures dictate what goes where in a source file. Those two aspects have roots in psychology: Because of the cognitive effort it takes to read code, it's easy to get frustrated when you have to jump between pages (or files) of code to understand the relationship between various constructs. The psychologist Mihaly Csikszentmihalyi, author of *Flow: The Psychology of Optimal Experience*, highlighted that sense of frustration in his Flow Theory[27]: there's a sweet spot for difficulty and required skills to perform tasks that are neither too hard (which can cause frustration) nor too easy (which can cause boredom)—to find that sweet spot is to find flow or to be in the zone. Effortless concentration while reading code is what you want—to comprehend it without getting frustrated or distracted. Keeping a consistent coding style and enunciating the flow of code across various constructs help code readers get into the zone and stay there. The rule of thumb here is simple: Keep things that belong together close to each other in a way that follows the call graph top-to-bottom (from caller to callee). In order to appreciate the subtlety and benefits of such writing style, let's take a look at an example of what **not** to do:

```
private void regenerateVariables() {
  // ...
}

private void regenerateInitComponents() {
  // ...
}

private void regenerateEventHandlers() {
  // ...
}

public void regenerate(boolean force) {
  if (canGenerate && (!isCodeUpToDate || force)) {
    isCodeUpToDate = true;
    regenerateVariables();
    regenerateInitComponents();
    regenerateEventHandlers();
    // ...
  }
}
```

[26] A call graph is one that describes the flow of control between subroutines in a program.

[27] https://en.wikipedia.org/wiki/Flow_(psychology).

The example above is an actual code snippet I wrote more than a decade ago for my graduation project.[28] Notice the discrepancy between the flow of reading the code and the call graph flow. In the actual source file, the `regenerate` method starts at line 195; it's confusing to read a bunch of private methods first before getting to the enter point that calls them in turn—it's like reading a story backwards. Inserting the `regenerate` method first then the methods it calls in order would have made for a much more readable story. On a side note, criticizing one's code is a great learning opportunity and allows us to be honest with ourselves about the state of our code; to reiterate, we should be our worst critics.

15.2.5 API Design

Abstraction and encapsulation go hand in hand; they help accelerate the pace of innovation by allowing software to evolve without being stuck in the pitfall of version incompatibility. Exposing a public API should be a premeditated decision because once it's out in the wild, there's no taking it back without ramifications. That said, there are also ramifications when changing implementations under the hood; this kind of freedom is not absolute. The concept of respecting contracts is not limited to code; it also covers any contract that governs how others may be interacting with your system including input (like configuration schema), output (like format of logs), and behavior; the latter is subtle, so here are a few examples:

- Performance characteristics changes: e.g., changing the time complexity from O(1) to O(n).
- Internal changes that weakens security: e.g., replacing .NET SecureString[29] instances in memory with plain strings.
- Semantic changes: e.g., enabling RegEx matching for a string value that used to be used for exact matching.
- Source of truth changes: e.g., making a remote request to get the returned data instead of returning a locally cached value.
- Changing possible exceptions a method could throw: e.g., C#, unlike Java, doesn't have the concept of checked exceptions (they are not part of the method's contract).
- etc.

Another aspect of API design is data structure design, which subsumes decisions like picking the right language construct to represent data and behaviors. There's a strong distinction here between data structures (that don't have behaviors and are created to hold data) and objects (that are mainly created to expose behaviors) that shouldn't come as a surprise to the vast majority of developers; yet, it's a distinction

[28] You can view the rest of the code at https://github.com/elgeish/auc-plethora.

[29] https://msdn.microsoft.com/en-us/library/system.security.securestring.aspx.

that tends to be neglected in favor of laziness. Separating business logic from data records means creating more types, which is favorable for reasons like separation of concerns and following the Single Responsibility Principle. Most systems require the use of both paradigms; the challenge is to know when to use an object vs. a data structure; choose wisely.

15.2.6 Error Handling

Unless your program is overly simple, you can't expect all executions paths to be happy.[30] The world that we live in is imperfect: humans make mistakes, networks are unreliable, storage gets corrupted, etc.; programs have to be prepared for such failures. Apt error handling is crucial for designing and implementing high-quality software; in some cases, it's the difference between keeping a business running and costing millions of dollars in irreparable damages. Here's an example from *Release It!: Design and Deploy Production-ready Software* (Nygard, 2007) that demonstrates a subtle failure in error handling that grounded an airline for a few hours and costed the company heftily:

```
try {
  conn = connectionPool.getConnection();
  stmt = conn.createStatement();
  // ...
} finally {
  if (stmt != null) {
    stmt.close();
  }
  if (conn != null) {
    conn.close();
  }
}
```

The `stmt.close()` call inside the `finally` block throws a `SQLException` under certain conditions, which causes *conn.close()* to be skipped—that led to a resource leak that crashed the application after exhausting the pool of available connections. This example shows that simply wrapping your error-prone code inside an error handler is insufficient; one must think of Murphy's Law (anything that can go wrong, will go wrong) and prepare for all those things that can go wrong. Key craftsmanship skills in error handling are honed by learning and experience; you don't know what you don't know so you can't prepare for unknown unknowns. The more mission-critical your program is, the more defensive you should get about error anticipation and handling; simply, you can't assume that all errors are going to be handled correctly. This kind of advice disagrees with what you may read

[30] A happy path is one that executes end-to-end and achieves its purpose without running into error conditions.

in *Clean Code* about error handling; Michael Feathers, the author of the chapter on error handling, suggest to write error handling code first (e.g., try-catch-finally statements), then to add the rest of the code assuming nothing can go wrong in the latter blocks of code. In chapter 3, Uncle Bob recommended to extract the bodies of try-catch blocks into their own functions; such that said functions represent the logic that assumes errors are handled somewhere else. We have reservations about such recommendations as developers may call those error-oblivious functions accidentally; humans make mistakes and there's no way to perform some automated check, like static analysis, for that kind of convention. In addition, assuming error-free code zones can be perilous as we demonstrated in the earlier example. The takeaway here is to think twice before assuming that you've handled all possible errors in your code.

Another good practice we suggest is to define a normal flow of code—avoiding throwing exceptions when possible. Here's an example:

```
try {
  sum += getX();
} catch (final NotFoundException ex) {
  sum += getDefaultX();
  logger.log(ex)
}
```

You can see this paradigm in hash tables that throw a `KeyNotFoundException` when trying to get the value of a key that doesn't exist, but they also provide other methods that return a default value if said key cannot be found. The anti-pattern that we strongly discourage is using exceptions to control the flow:

```
public Config load(final String path) {
  try {
    if (!isFileTypeSupported(path)) {
      throw new SomeException(UNSUPPORTED_FILE_TYPE);
    }
    return parseFile(path); // also throws SomeException
  } catch (final SomeException ex) {
    logger.log(ex);
    return DEFAULT_CONFIG;
  }
}
```

Exception throwing is expensive and it shouldn't be used to change the flow of control for foreseeable errors. In the example above, the error handling logic can be refactored to handle both error cases:

```
public Config load(final String path) {
  if (!isFileTypeSupported(path)) {
    return handleConfigLoadingError(USUPPORTED_FILE_TYPE);
  }

  try {
    return parseFile(path);
```

```
    } catch (final SomeException ex) {
      return handleConfigLoadingException(ex);
    }
  }
```

Note the added benefits of such refactoring: separation of concerns between the two code paragraphs—each handles a separate code path—which allows them to change error handling behavior independently if need be. In addition, such separation opens the door for another refactoring opportunity to enhance readability:

```
public Config load(final String path) {
  if (!isFileTypeSupported(path)) {
    return handleConfigLoadingError(USUPPORTED_FILE_TYPE);
  }

  return parseFile(path);
}

private Config parseFile(final String path) {
  try {
    // parsing logic
  } catch (final SomeException ex) {
    return handleConfigLoadingException(ex);
  }
}
```

The above code snippet reads like a newspaper story that gives the high-level view of events first and then unfolds to delve into the details. Inside the `parseFile` method, we increased code collocation so that readers can get a better picture of how parsing errors are handled as close as possible to the source of said errors. One assumption we made here is that all parsing errors are to be handled the same way; if that isn't the case, then exception bubbling may be required.

Another good practice we strongly recommend is changing error handling behavior dynamically based on when and where code is running. Different application lifecycle stages require different error handling behaviors; the same applies to different environments: developer machines (dev environment), integration, production, etc. During early stages of newly added features, the error handling behavior should be to fail fast and could be as verbose as you want it to be; for example, you can set the program to crash and take a memory dump to inspect later. At Microsoft, one of the developers who used to work on Visual Studio Team Foundation Server followed a strict practice of attaching a debugger to his server process in a visible window all the time while using the client, the Visual Studio IDE, to catch exceptions and log them as he's using the product.

For some applications, crashing the process when encountering certain errors is a much better option than continuing to operate in an unknown failure state. Assume that the previous code snippet belongs to a rocket's guidance software; failing to load a specific configuration should cause its launch to be aborted (rather than loading a default configuration and hoping for the best). A program in an unknown failure state, due to a transient bug, has better chances of recovering from said

state thanks to external remediation than what internal error handling can do. The operating assumptions here are error triangulation and understanding, without which remediation is a shot in the dark. For example, a common remediation is restarting the failed process, which can be effective in cases when the failure conditions are not repeated shortly; otherwise, we risk running into a vicious cycle of failure and vain remediation. Not all errors are created equally; therefore, different error classes should be handled differently.

15.2.7 Logging

Silent code is dangerous; it doesn't speak out when something goes awry. Logs are breadcrumbs that tell a story of what happened after the fact; contrast that to debugging where you have to be actively paying attention to hunt down a bug. Just like how error handling requirements vary depending on when and where the code is running, logging follows suit. In early stages of newly added features and/or in dev environments, logging can be as verbose as you want it to be. It's important to note that logging in production is not just for warnings and errors; you can adjust error levels as you see fit for your application; most debugging libraries support leveled logging. That said, verbose logging shouldn't penalize the performance of a program; especially when serializing an object to a string just for the sake of logging it, here's an example of what **not** to do:

```
public void foo(final Bar bar) {
  logger.debug(bar.toString());
  // ...
}
```

Even though the log line is not going to be written, the call `bar.toString()` will be executed to pass its return value as a parameter, which can be excessively wasteful. Instead, we can change the code to only create said object when debug logging is enabled:

```
public void foo(final Bar bar) {
  if (logger.isDebugEnabled()) {
    logger.debug(bar.toJson());
  }
  // ...
}
```

To reap the benefits of logging, it's essential to think about it as part of the program's UI; otherwise, it can be worthless. Simple things can go a long way like indicating where a log line starts using a preamble of the timestamp when it was written and some identifier like name of the module that created it. Additional fields that give some context can be very useful depending on the nature of the program; for example, the current thread's ID for multi-threaded applications, the request ID for web application, or the invocation context ID for pipeline of microservices. The

body of the log line matters the most as it communicates a message to the reader. The goal of logging said details is enabling readers to take a quick glance at them to understand what's going on. Another goal is to identify log lines and make them easily searchable; it should be easy to find relevant log lines that pertain to the issue being investigated. Log line fields can also be crunched to generate statistics, provide insights, or even power the product itself like Google did by mining the logs of users' actions on the site to evolve its ranking algorithms (Levy, 2011).

Structuring logs in a format that makes it easier for machines to parse, like JSON, enables a myriad of tools to perform better data extraction, transformation, and loading (ETL). Schemas of structured logs should be versioned as logs are to be treated like any other data source. A good example of a framework that indexes structured logs is Elasticsearch, which works best with log fields represented as JSON objects. Once indexed, logs can be searched and visualized (using Kibana, a plugin for Elasticsearch). To process unstructured logs, one can use a log ingestion tool like Logstash and its parser, Grok, which uses pattern matching to produce structured data in order to feed them into Elasticsearch. The three tools combined (Elasticsearch, Logstash, and Kibana) are called the Elastic Stack[31] (aka ELK).

Once log events are structured and made to follow a strictly typed schema, they can also be used for telemetry; in a sense, we can view log events as a special case of tracking events. Generally speaking, we can direct tracking events to a variety of destinations depending on how we want to make sense of them; for example, at Microsoft, we used to build various consumers of the same stream of events emitted by an application to log them into files, populate an online analytical processing (OLAP) cube with code performance measurements, and reset a watchdog timer[32]—all using the same source of events. At LinkedIn, tracking events are used to measure key performance indicators (KPIs) of products, draw insights for employees (via internal tools) and LinkedIn members (by powering up product features), train machine learning models, and debug issues. LinkedIn's tracking events are strongly typed using Avro/Pegasus schemas[33] and they run through Kafka streams, which handle a whopping 1.5 trillion events a day,[34] according to LinkedIn's estimate in 2017.

Instrumenting code can be useful in finding correctness and performance issues; for example, you can fire events that report the time elapsed executing the instrumented code blocks, then generate interesting statistics to identify performance-critical hotspots. Another example is identifying dead code (or features that are rarely used) to eliminate. Along the same lines of thinking as a frequentist, one

[31] https://www.elastic.co/webinars/introduction-elk-stack.

[32] A watchdog timer is a mechanism to detect failures: An application resets its watchdog timers periodically to prevent them from timing out and alerting a monitoring system that something is wrong with the application being watched; see https://en.wikipedia.org/wiki/Watchdog_timer for more details.

[33] https://github.com/linkedin/rest.li/wiki/DATA-Data-Schema-and-Templates.

[34] https://engineering.linkedin.com/data/distributed-data-systems.

can refine assumptions in code based on collected instrumentation statistics; for example, code can start with arbitrary values for things that are hard to predict, like specifying timeouts and polling intervals for novel tasks, in which case an educated guess is picked and then refined by observing instrumentation data. Insights drawn from tracking events can influence the program's behavior; let's consider the following example:

```
def answer():
  if isThisCheapCheckSatisified():
    return 0
  if isThisExpensiveCheckSatisified():
    return 1
  if isThisMoreExpensiveCheckSatisified():
    return 2
  return 3
```

Conventional wisdom suggests that, other things being equal, we should order the checks above by the cost of performing them; this way, we start with cheapest ones first hoping that we don't have to perform the more expensive ones. The underlying assumption here is that such conditions follow a uniform distribution—which is almost never the case. In fact, CPUs count on the fact that most code branches follow a skewed distribution—when looked at through a sliding time window—to optimize conditional jump instructions in a pipelined architecture; a branch predictor[35] can save precious time by guessing which code path to take; some branch prediction techniques yield success rates that exceed 90%, according to benchmarks performed by SPEC (the Standard Performance Evaluation Corporation). The code above can be rewritten, in light of new evidence collected from logs, to order the checks by the probability of being satisfied—regardless of the cost of evaluation.

Distributed systems can benefit immensely from logging across various components of a system: In addition to the conventional uses of logs for each component, distributed logs can be grouped by unique identifiers to tell a story that's otherwise fragmented and lost. Tracing allows developers to follow a certain transaction across the system and pinpoint what went wrong in a complex web of interactions among a plethora of machines. Distributed systems can get so complex that tracing actual remote procedure calls becomes the most effective way to figure out how components interact. For example, Amazon Web Services (AWS) offer instrumented web clients so that developers can see details about interactions with downstream AWS services and other HTTP web APIs; AWS X-Ray collects instrumentation data and generates detailed service-call graphs, which can be used to identify correctness and performance issues.

Error handling and logging go hand in hand; error logs should be easy to understand and identify to improve search and debugging. Moreover, a runbook or troubleshooting guide (TSG) can reference a specific error and explain what to do when said error are logged. Assigning unique IDs to errors is one way to

[35] https://en.wikipedia.org/wiki/Branch_predictor.

go about that; for example, Visual Studio Team Foundation Server displays errors that have been assigned unique IDs in the form `TFnnnnn` and lists them in the documentation[36]; for example:

```
TF10107: Source control cannot compare binary files.
```

Logging fits right in the top system design priorities we discussed earlier in this chapter as it indirectly improves the correctness of the system by helping programmers find issues to fix. That said, it can be disastrous if misused, for example, in violation of the most important priority: security. Logging user-specific data (more formally known as personally identifiable information or PII) can be a major vulnerability—especially when logs are not secured. The rule of thumb is to scrub PII and HBI (high business impact) data before logging them; and to treat logs just like any data source: keep them secure and well protected with proper defenses and access control lists (ACLs) in place.

15.2.8 Tests

Chapter 11 discusses testing methodologies in great detail. Here, we reiterate the importance of craftsmanship in testing; the main idea is to treat test code just like production code; in fact, sometimes test code requires a higher level of craftsmanship. Just like with production code, following consistent patterns in test code reduces cognitive load; most tests follow the same pattern: set up, invoke the unit under test, assert, and then clean up (if need be); said pattern encourages test code reuse and consistency. By using test utilities, you can spend more time thinking about test strategies and less time typing boilerplate code.

15.3 Big Data Craftsmanship

Data is king. The first—and arguably the most important—step in creating a big data solution is understanding the data and making sure it fits the proposed solution. For example, machine learning allows data to become the program by learning from known data a solution that can be used with new data. That means that there's a strong correlation between the quality of your data and the quality and effectiveness of your solution.

Similar to code, data needs to be handled with care; otherwise chaos ensues and systems start to rot. The same principles that apply to code craftsmanship apply to data as well: readability, discoverability, maintainability, debuggability, etc.

[36] https://msdn.microsoft.com/en-us/library/aa337645.aspx.

That said, big data represents an additional set of craftsmanship challenges that are caused by the four V's of big data: Volume, Velocity, Variety, and Veracity.

15.3.1 Metadata

In the early days of a small company, data may seem manageable; datasets are unique and small, you know who owns what, and the sources of data are obvious. As the company grows, especially when said growth is exponential, the number of datasets collected can get out of hand. At that point, metadata—data that describe other data—becomes essential to stay sane while dealing with big data. Using said metadata, tools and systems can improve the way we use big data and help us establish a high standard of big data craftsmanship.

15.3.2 Discoverability

One of the first questions one may ask when faced with a new big-data problem is: How can I find that data? The answer can be anywhere between a simple lookup and a painful quest. A craftsman understands the value of making data discoverable; besides the obvious time saving while looking for datasets, such ease encourages data reuse. On the other hand, when data is hard to discover, one may find it easier to spend precious time on reacquiring a dataset than to look for it in vain.

Simple methods of data organization can go a long way to make data more discoverable; for example, ensuring that your Hadoop Distributed File System (HDFS) has an easy-to-follow cluster and directory structures, each dataset has metadata about how it was collected and who owns it, etc. Sometimes, you may want to build or install custom tools to make metadata searchable; for example, a web console for your Kafka deployment that lists all topics and their metadata.

15.3.3 Versioning

Data, just like software, evolve over time; hence, change tracking is crucial to avoid data bugs. Imagine adding a new field to any of your datasets; are your systems resilient enough to continue functioning correctly despite the change? Versioned strongly typed schemas act as data contracts between data producers and consumers, which can help prevent incompatibility issues. In that sense, a consumer would ask for a certain version of the data, and the data-serving system guarantees that the consumer gets back the requested data in the format it expects per the specified data contract. A simple, yet effective, way to version data is Semantic Versioning 2.0.0,

which is a common versioning scheme for software; in summary, a dataset is given a major.minor.patch version number, which is incremented as follows:

- Major is incremented when making an incompatible change.
- Minor is incremented when adding fields in a backward-compatible fashion.
- Patch is incremented when making a backward-compatible bug fix.

It's obviously a good practice to maintain backward compatibility when making changes. Versioning of data doesn't always require keeping multiple versions of the physical data for each major version; in many big data systems, the concept of a logical view can provide a layer of abstraction on top of how the physical data look like. In the realm of relational databases, one can create views to project different layouts of the underlying tables; this way, changes in the underlying tables are decoupled from how the consumers see said tables, even when the changes are backward-incompatible. For example, one may choose to encode a string field, say using base64 encoding, in the underlying table; a view that returns the field before the change can be altered to return the decoded string and thus maintain backward compatibility from the consumer's viewpoint.

Keeping track of who owns which datasets (and which versions thereof) is key for big teams as a way to discover and learn more about the data (besides documentation, which we will cover later in this section). Similarly, keeping track of who consumes which versions of your datasets is a good way to keep a communication channel open in case there's a bug fix, a version that will be obsolete, or any other change that requires coordination between producers and consumers of the data.

Changes in data cover both schema and semantic changes; the latter is subtle and should be always tested by running consumers' acceptance tests that cover how the data are being used. Changing the meaning of a field, with or without breaking the dataset's schema, can be catastrophic; for example, imagine a dataset that includes a field called `length` whose data changed systems from imperial to metric at a point of time unknown to the consumers of said dataset.

15.3.4 Documentation

Documentation should serve the same purpose it does for code: adding meaning to the unclear. Many data management systems allow for comments on data schemas like in data description language (DDL). However, said comments are mainly meant for the data producers and schema designers; in order for data consumers to read data documentation, it has to be easily accessible. Avro schemas support comments[37] that many languages, like Java, incorporate into the compiled schema contract so that both producers and consumers can read them easily.

[37] https://avro.apache.org/docs/1.7.6/idl.html#minutiae_comments.

A data schema is basically a way to describe a data structure; hence, all the guidelines that apply for code documentation apply here. For example, schema fields' names should be self-explanatory and specific; naming a field `durationInSeconds` is orders of magnitude better than naming it `duration` with an adjacent comment that indicates its unit. The better—and sometimes, required—option is to have the schema define a type for the unit and assign it to the `duration` field.

15.3.5 Debuggability

Debugging a data issue can be more complex than debugging code. For starters, the time it takes to run some big-data queries can be too long that it becomes prohibitive to debug data issues in a single setting. Since debugging may require many attempts to identify an issue, the speed of iteration here is key. Imagine if every time you need to test a hypothesis about some data issue, the query you run takes tens of minutes to return; how productive can you be? The first step to improving data debuggability is to make debugging steps move faster. For example, if your big data live in Hadoop and you use Pig to run analytical queries, it will scan each and every record in the dataset to find the ones in which you're interested; that can be cumbersome. Instead, loading the dataset of interest first into a columnar query engine (like Presto or a relational database) that supports indexing can bring down the query execution time from tens of minutes to a few seconds. In addition to the speed improvement, such systems can bring data from various stores (e.g., Hadoop, Cassandra, SQL Server, etc.) together in a single query. According to Presto's website, "Facebook uses Presto for interactive queries against several internal data stores, including their 300PB data warehouse. Over 1,000 Facebook employees use Presto daily to run more than 30,000 queries that in total scan over a petabyte each per day."

Another step to take to improve debuggability is making sure data is human-readable; for example, one may choose a binary storage format to save space but when it comes time to debug issues a text-based format is necessary. One approach to take here is providing tools that converts between various storage formats; it may also help to start a new project using the human-readable format and eventually switch to a binary one after stabilizing both the code and the data.

15.3.6 Quality

Data quality subsumes more than just correctness, which can be verified using testing techniques similar to those employed in testing code; ensuring the quality of big datasets involves statistical testing, monitoring for shifts in data distributions, and anomaly detection. Data contracts and schemas are the first line of defense as they prevent egregious syntax and data issues. Correctness testing is the second line

of defense as it verifies the expected behavior of data producers; for example, if a program must set a field x to a value y when condition z is met, a simple test case can verify that expectation. At the throughput of billions of data records per day, it's prohibitive to test for each and every possible data value and combinations thereof. That said, the level of scrutiny required when testing a big data system depends on what it does; for example, a machine learning system that recognizes faces can tolerate a certain percentage of low-quality data while the New York Stock Exchange cannot.

At big companies, it's common to have many producers and consumers of shared datasets from various teams; organizational structures play a big role in determining goals and metrics each team tracks. It's also common that said teams don't share the same business priorities and have different standards for data quality. Because of that sort of decoupling, it's even more crucial to establish a strong culture of demanding high standards of data quality; the following guidelines can help with that:

- Proactively overcommunicating changes that might affect another team.
- Alignment of business goals, which need to include data quality goals and metrics.
- Fostering a sense of comradery among partner teams—helping others succeed should be part of your team's success criteria.
- Conducting effective postmortems for data issues in which one should eschew finger pointing and blame shifting.
- Clear commitment making; shared accountability of maintaining quality without a clear owner and timeframe means no accountability.
- Quality is everyone's job; that doesn't conflict with clear commitment assignment; it encourages individuals to take ownership of quality tasks.

The earlier data quality measures are set in motion the better; cleaning up data after they get polluted is cost-prohibitive and in some cases impossible. Take for example data that track how users interact with an application, a data bug might cause an entire dataset to be useless and unrecoverable. In certain cases, like when the entire business of a company is being a data broker, such cases could be business-ending. In all cases, decisions taken based on collected data can be—at their best—as good as the data.

Consumers of big data have to perform their own testing as they can't rely on correctness testing performed by producers, if any, which can be outside of their team or company altogether. This is a good usecase for statistical analysis of the data. Take for example elections poll data and the infamous case of Research 2000, a company that published polling data for the 2008 US elections, which exhibited extreme statistical anomalies inconsistent with random polling.[38] Different applications may require different methods of statistical analysis but most

[38] https://en.wikipedia.org/wiki/Research_2000.

of them can apply a set of validation on a sample of the data to monitor the overall quality of the population. For details on how to deal with data issues, see Chap. 9.

References

Wolfram Schultz, Peter Dayan, and P. Read Montague. A neural substrate of prediction and reward. *Science*, 275(5306):1593–1599, 1997. ISSN 0036-8075. doi: 10.1126/science.275.5306.1593. URL http://science.sciencemag.org/content/275/5306/1593.

H. Scarlett. *Neuroscience for Organizational Change: An Evidence-Based Practical Guide to Managing Change*. Kogan Page, 2016. ISBN 9780749474881.

M. Feathers. *Working Effectively with Legacy Code*. Robert C. Martin Series. Pearson Education, 2004. ISBN 9780132931755.

W. Isaacson. *Steve Jobs*. Simon & Schuster, 2011. ISBN 9781451648539.

M.T. Nygard. *Release It!: Design and Deploy Production-ready Software*. Pragmatic Bookshelf Series. Pragmatic Bookshelf, 2007. ISBN 9780978739218.

Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. ISBN 0-201-61622-X.

M. M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, Sept 1980. ISSN 0018-9219. doi: 10.1109/PROC.1980.11805.

Melissa R. Dale and Clemente Izurieta. Impacts of design pattern decay on system quality. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '14, pages 37:1–37:4, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2774-9. doi: 10.1145/2652524.2652560. URL http://doi.acm.org/10.1145/2652524.2652560.

R.C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Robert C. Martin series. Prentice Hall, 2009. ISBN 9780132350884.

D. Kahneman. *Thinking, Fast and Slow*. Farrar, Straus and Giroux, 2011. ISBN 9781429969352.

David N. Card and Robert L. Glass. *Measuring Software Design Quality*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990. ISBN 0-13-568593-1.

R. W. Selby and V. R. Basili. Analyzing error-prone system structure. *IEEE Transactions on Software Engineering*, 17(2):141–152, Feb 1991. ISSN 0098-5589. doi: 10.1109/32.67595.

S. Levy. *In The Plex: How Google Thinks, Works, and Shapes Our Lives*. Simon & Schuster, 2011. ISBN 9781416596714.