Brian Lee Yung Rowe

Introduction to Reproducible Science in R

Contents

1

The reproducible scientific method

> The unexamined life is not worth living.
>
> *Socrates*

Stripped to its essence, reproducible science is just science: the ability to achieve similar results under similar conditions. Prior to the rise of the computational sciences and open data, this was the only definition of reproducible science. Due to the portability of data, reproducibility is more of a continuum today. A weak form of reproducibility uses the same data and same experimental setup (code) but by a different team. Some call this scenario reproducibility [26, 13], while others call it **replicability** [14, 8, 15]. A strong form of **reproducibility** uses different data and a different experimental setup. In between is **generalizability**: different data but same code. While reproducibility is the end goal for science, for business generalizability is typically the goal.

In the computational sciences, it's possible to achieve exactly the same results with exactly the same initial conditions. Unfortunately, due to the complexities of software, replicability has been surprisingly elusive. Thankfully, times have changed. Numerous factors have made it easier to replicate results, including open source code, open data, increased code literacy, and a plethora of collaboration tools. This book shows one approach for making your work easier to replicate, generalize, and reproduce. Reproducible science benefits not just the researcher or experimenter, but also the colleague, the peer reviewer, and the auditor. Irrespective of the role, the initial goal of reproducible science is literally replicating stated results with the same data, the same code, the same model parameters. Assuming the code and data are available, there's no point attempting to reproduce results if they cannot be replicated first. After replication, it's equally possible to focus on reproducing results or generalizing results. For data science and non-academic uses, emphasis is often on generalizability: once trained, a model will be applied to new data. For peer review, strong reproducibility is required to truly verify scientific results. But even in business and government, strong reproducibilility is important. Depending on the impact of a model, it should be vetted by an independent team. In investment management this approach is fairly common, but it is surprisingly less common in situations like determining the length of a prison sentence [34, 28].

Reproducible science requires a repeatable process. Numerous steps must be taken to achieve *practical* repeatability: datasets must be easy to acquire and construct, code must be easy to run and modify, model parameters must be explicit, and even seeds for random number generators must be specified. The ad hoc nature of data exploration must be replaced by a more disciplined and structured development process. But how do you balance exploration efficiency at the beginning with repeatability later on? Too much structure at the beginning of a project becomes an opportunity cost if a model does not have enough explanatory power. Wouldn't it better to add in structure along the way, as your confidence in your model and project increases? With functional programming, this is not only possible but practical. Software written in a functional programming style easily adapts to new uses. You'll find that it is remarkably easy to shift from one mode of work to another without many changes to your code. Your code will also be easier to understand, more reliable, and more reusable.

What makes some work more readily reproducible than other work? Like patents, some models are technically reproducible but practically impossible to reproduce. Research that is easily reproduced is **accessible**, **transparent**, and **automated**. Accessible work is either self-contained, providing all necessary artifacts, or makes it easy to obtain all necessary artifacts. Clean data consolidated into a single file is more accessible than a set of raw files with unclear transformation rules. Similarly, code that includes all dependencies is more accessible than code that only lists its dependencies. Transparency encompasses the availability and readability of code, data, and documentation. It is easier to reproduce results when the process and workflow surrounding the research is available, not just the data and methods. Open source code that includes data sets is transparent, but it loses value when lacking adequate documentation describing the approach. It also slows the advancement of science when others have to spend time just to replicate work or understand how a process works. In cases where redistribution of data is not possible, the next best option is to provide a script that downloads and constructs a dataset. To reproduce results quickly, automation is necessary. Ideally, bundled scripts are provided that can install dependencies, download data, and run a model without human intervention. Automation is also required to run models in an operational context. Introduced in Section 1.1, operational models are integrated into a business process or system making predictions and/or decisions regularly.

Accessibility and transparency are characteristics of open science, enabling anyone to run, examine, and modify experiments. On the other hand, automation can stifle openness and freedom by locking you into a particular way of doing things. Automated programs can make replication trivial but complicate generalization and reproduction to the point that it's impractical to examine or modify an existing experiment. Avoiding this perverse consequence requires a good programming practice and deliberate design decisions that preserve openness and independence. My approach to repro-

ducible science is thus based on a good programming practice. Well-written programs are *usable* and naturally accessible, transparent, and automated. The key to good programming is embracing simplicity and modularity. Like any discipline, it can take years to become proficient. This book aims to speed that process by highlighting aspects of functional programming (Section 1.2) and the UNIX philosophy (Section 1.3) that simplify code and make it more modular.

The core toolset I use for open and reproducible science leans heavily on the UNIX tradition and includes `Linux`, `make`, and `Docker`. The reason for choosing this set of tools is that the time spent learning the tools is transferable: you will encounter them over and over throughout your career. For example, much of R relies on the same suite of tools. The same is true for data visualization and web programming, data engineering, not to mention plain old systems engineering. In addition to the above tools, I also reference my `crant` utility throughout the book. This utility automates numerous development tasks, streamlining model development. Its focus is on system-level processes and works in conjunction with lower level build tools like `devtools`. Section 1.4 compares this book's approach to reproducible science with other approaches. Regardless of the choice of tools, this book provides a reusable framework for thinking about models and systems that promote reproducible science.

1.1 The rise of operational models

Back in the twentieth century, a statistician was often a hired gun. She would grace the town whenever the good townspeople were unable to tame a particularly nefarious problem. This lone statistician would ride in, assess the situation, build a model and run some tests. Problem overcome, she would submit a final report of what to do with the scoundrel and ride off into the sunset, leaving a dust trail of model artifacts and one-off code. Things are different today. Data is no longer an intermittent friend or foe walking through a dusty town. Data is now so ubiquitous that it has instead become the lifeblood of a bustling digital metropolis. In the not-so-distant future, most of our lives will run on data. The education we can access, the jobs we can interview for, the amount of money we can borrow, the type of healthcare we receive, the insurance premia we pay, and even the prison terms we get will eventually all be dictated by data. As models become a part of our daily lives, our dependence on them will increase. Consequently, there is a greater need to scrutinize the models that affect our lives. Reproducible science is an important step in making scientific research more transparent. As we enter a world where models and AI make decisions for us, reproducible science not only improves science but has the potential to improve society.

Societal implications aside, the extent to which data will influence our lives is great. As scientists, engineers, and researchers [1] working with data, the way we interact with data is changing alongside the role of data. It's gone from a periodic strategic need to a continuous, operational need. The need for reproducibility, and the transition to **operational models** impacts the modern day scientist in a number of ways. First, science is no longer a singular, isolated endeavor. Like software development and business, it has become collaborative and iterative. Collaborators range from department managers and business analysts to other data scientists to software development teams. Each role has their own requirements regarding the model and its results. Managers will likely want summaries of results in a flat report or dashboard. Business analysts may want a more interactive interface to investigate relationships and tinker with data on their own. Peers may want a detailed view presented in code or as a notebook. Systems engineering will need to review the architecture, system characteristics, and operational needs of a model. If a separate software team intends to reimplement the model for a production system, they need a model specification, a reference implementation, and test cases. Each audience has their own needs in terms of visibility into how the model works and how well it works. Irrespective of the audience, models also need to be easy to run with clearly defined inputs and outputs. In other words, repeatable research is an essential trait of modeling, no matter the audience.

Second, expectations of code quality have changed. A standalone analysis or research project has few requirememts outside of the immediate goal of obtaining results from a dataset and model. While both the data and model need to be robust, the underlying quality of the code is often ignored. Isolated projects are not meant to have a long shelf-life nor many users. It's therefore understandable if the corresponding code is disorganized, monolithic, inflexible, poorly documented, slow, etc. The moment that someone else wants to use a model or portion of a model, low quality code is no longer viable. To avoid wasting countless hours helping a collaborator reproduce your results in their environment, your model needs to be portable, robust, and well-documented. When a model is destined for operational use in a business process, the quality of the code must be even higher. Operational models need to be resilient to the constant turbulence of production systems. Whether it's missing data, bad data, updates to data, regime change, or other unforeseen factors affecting the model process and/or result, a production-quality model must anticipate and handle these scenarios.

Finally, the responsibilities of a data scientist have increased, but often the available resources have not. In some ways, modelers haven't fully shed the guise of the lone wolf. Sometimes a team of one, model teams are often

[1] There are many professions that engage in quantitative modeling, numerical analysis, etc. Collectively, I refer to anyone engaging in data modeling as a scientist or modeler. For variation, I will occasionally use other titles to mean the same thing. I refer to the research that scientists engage in as data modeling, modeling, or data science.

small. Modelers are like the homesteaders of yesteryear and need to be self-sufficient. In addition to building models, scientists can be responsible for acquiring and managing data, visualizing data, creating reports and dashboards, deploying and maintaining operational models. The key to juggling so many responsibilities is writing robust, reliable software, so models and systems take care of themselves. A poorly designed model system is no better than a straw house that needs constant repair and upkeep. How can you possibly tend to the fields when all your energy and time is wasted trying to keep a dry roof over your head?

The software that realizes a model thus plays a prominent role in this story. It is software that extracts data from disparate sources, sometimes on demand and sometimes on a regular schedule. It is software that processes data and implements models. It is software that runs and tests models. It is software that generates reports and visualizes results. And it is the quality of the software you write that dictates the pace of your research and reliability of your model.

Software is the universal tool to make data analysis and modeling tractable, efficient, and reproducible. Like any tool or machinery, if the software breaks, you're forced to *fix* the tool preventing you from *using* the tool. When models don't produce desired or expected results, you need to take apart the software machine to identify the problem. Bugs can appear in three places: the data, the model, and the code. Debugging is tedious and time-consuming. To debug data, you need tools to inspect data structures. To debug models, you need tools to partition data. Sometimes you need to modify a process to inject a different dataset into a model. To debug code, you need tools to interrupt the flow of a program and taps to peer into the transient state of our machine. We may find some of these tools ready-made. Other times you may need to create our own tools from scratch. The conclusion? The majority of a data scientist's time is spent programming. And the cruel reality is this: the worse you are at programming, the more time you spend doing it. Given this perverse imbalance, it's worth embarking on the journey ahead to hone your programming skills to even out the time you spend modeling versus the time you spend writing code. This is your *call to adventure*.

If you have prior experience programming, you may be tempted to ignore this call and stay within the comfortable world you know. Unfortunately, even if you've taken programming classes, it's likely that most of what you know about programming and software design doesn't apply to data science. Many academic programs focus on web development or mobile app development. These curricula tend to focus on object-oriented programming, which is well suited for GUI applications that deal with digital analogues to real world phenomena. Even server-side systems may use classes to organize groups of operations and maintain state. While this approach can be effective, it works less well for systems built for data science. To be clear, I am talking about model pipelines, batch processing jobs, and even real-time predictive models

built by modelers. This distinction is important because systems built by professional programmers have design goals different from computational systems, which may warrant the use of object-oriented programming.

In contrast, model systems almost always benefit from adhering to functional programming principles. This simple design philosophy emphasizes writing many small functions. A system is built by composing these reusable functions into a **pipeline**. Another way of describing this structure is that a model system can be represented as a **computational graph**, discussed in Section 1.2.4. When done well, these systems are easy to understand, modify, automate, reuse, and exchange.

To achieve reproducibility, is it really necessary to let go of other programming styles and adopt functional programming techniques? Put another way, is programming for data science really that different from systems or app development? Consider the programming lifecycle of model development versus systems or app development. Professional software development focuses on automated operation of systems. The system is designed with this intention from the start, so attention is paid to how bits of code are organized. Data science programs typically start as an exploratory exercise. Many experiments will result in nothing conclusive, so the code will be thrown away. Some survive this gauntlet of viability and emerge as the vehicle for reproducible science. Others reach a higher level of transcedence, becoming operational code that runs continuously as part of a business process. Hence, data science programs often have a changing raison d'être depending on the maturity of the research and the model. How we write the code determines how long it takes for code to metamorphose from one stage to the next. Of course, this pupa feeds on our time to fully transform, which can be quite costly and painful.

As a model matures, a strange phenomenon occurs: the surrounding environment changes with the model. This can impact the reproducibility of the model and must be accounted for. Projects usually start as an idea or hypothesis that warrants investigation. This first phase is exploratory in nature. At this stage, data is probably stored in flat files with few external resources used. If the analysis yields a promising result, more time and energy might be invested. Datasets will likely grow, triggering additional data processing pipelines to appear. Model outputs may shift from a binary `Rdata` file to CSVs to tables in a database. Eventually the model might mature into code that runs in a production environment. Production environments have more stringent requirements than development environments. Data will no longer be extracted from local files and must instead be pulled from a source datastore. Other resources may also be moved from a single process to other standalone microservices. Supporting a production environment adds a new layer of software to support the integration of the model with other systems. The environments surrounding the model are stacked like parallel universes, where the model shifts between the universes depending on who is using the model. Hence, the different methods of data access must all be supported

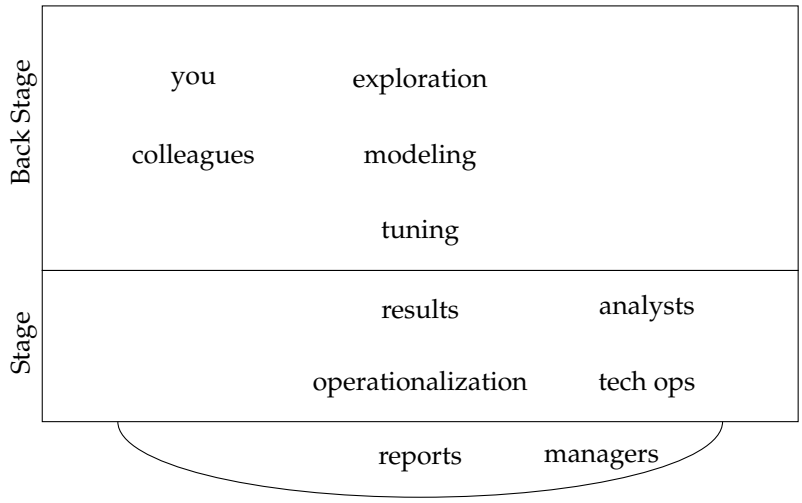


FIGURE 1.1: The model development workflow and the actors interacting with models. As a model matures, more actors join the stage. The model system must support all the actors.

and be internally consistent given the environment. To support these shifting realities, it's important that model systems remain simple and flexible. Part II of this book discusses these workflows in detail.

At the component level, the data structures and objects present in the system also differ from conventional software systems. The reason is that much of data science programming is manipulating data and transforming it with mathematical operations. It's quite possible that a model script or program can be reduced to a long sequence of repeated function application or extended function composition. One reason this is possible is because once a model system starts, there is just one use case and rarely any human interaction. Another reason is that model systems rarely need shared state. In contrast, GUIs mediate a workflow where a human is an integral part of the process. A GUI element needs to know what happened previously, what's happening around it, as well as what's inside it. GUIs support complex interactions, so data structures and operations must be defined in advance. Without this design stage, many object-oriented systems end up being difficult to understand, difficult to change, and ultimately fragile.

Components in model systems usually represent some sort of transformation, mathematical or otherwise. Data might be converted from one format to another, features added, a series normalized or transformed, partitioned, etc. These operations keep to themselves and don't care about what happens around them. For example, a linear operator doesn't need to know anything more than its operand to function. It doesn't need to know about what hap-

pened previously, nor what will happen later on in the processing. This selfish ignorance helps to simplify model systems.

1.2 The importance of being functional

What exactly is functional programming? I think of it mainly as a philosophy for writing software that uses functions as the primary building block. Functional programming has a rich mathematical theory underpinning it, but it is largely unnecessary to tap the benefits of the philosophy. Programs written in a functional programming style (a.k.a. functional programs) exhibit a number of properties. Functional programs are:

- declarative: algorithms focus on what transformations they make, not how;
- deterministic: functions avoid side effects so that their behavior can be fully deduced;
- modular: individual functions are small and can be used in various contexts;
- composable: programs are simply functions chained together (i.e. function composition);
- type-free: functions and programs generally use a handful of simple types.

Functional programming is well suited to model development because model systems often resemble transformation pipelines, which are no more than a chain of function transformations.

Much of machine learning and big data leverages functional programming in one way or another. In fact, the rise of big data is one of the driving forces behind the renewed interest in functional programming. Starting with the MapReduce paradigm for big data processing, it uses two core higher-order functions map and reduce (a.k.a. fold) as the foundation for arbitrary data transformations. Parallelization and distributed computing largely rely on being able to apply the same function to subsets of data on different compute nodes. This is simply a map operation.

1.2.1 Determinism of programs

Mathematical functions are deterministic by definition. Recall that the graph of a function comprises a (theoretical) table associating an input value with an output value. Functions defined on the real numbers will have an infinitely long table. Irrespective of the size of the table, it is completely deterministic: given a specific input x, the output will always be y. Some "functions" are problematic and violate this definition. Consider the square root function. For a given positive number x, the square root $\sqrt{x}$ is y, where both y^2 as well as $(-y)^2$ equal x. Multiple values for the same input is problematic

because it's unclear which value is expected in an operation. In other words, the function is not deterministic. Imagine how inefficient it woud be if you had to check whether the square root meant the positive or negative root every time you encountered it. Even worse is if its value was randomly the positive or negative root. Mathematics gets around this annoying problem by declaring the principal square root as the default square root, unless otherwise specified. This function limits the range of $\sqrt{\cdot}$ to the positive numbers, thus skirting the issue of two output values altogether.

Mathematics carefully defines functions to be deterministic, but the same is not true in computer science. Programming functions are essentially step-by-step algorithms that manipulate or move data from one place to another. Technically, these functions are completely deterministic given all initial conditions and inputs are known. The difficulty arises when initial conditions are unknown. Consider the function `read_line(fd)` that reads a line from a file. This file takes a single argument, which is a file descriptor. What is the result of calling `read_line(fd)`? Without knowing what file the file descriptor references and the contents of that file, it is impossible to deduce the output. This is not the only way that non-determinism seeps into functions. The use of global variables introduce that prevent programs from being deterministic. Global variables typically involve more than one function. Consider function `write_state` and function `read_state` that write to and read from a global variable, respectively. The function `read_state` is non-deterministic since its return value depends on more than its function arguments.

Determinism is not only a necessary ingredient of deduction; it is necessary for repeatability as well. To preserve determinism, the same care taken in mathematics must be applied to computing. This means avoiding the use of global variables and shared state in general. Sometimes shared state is unavoidable in programs. The functional programming approach uses closures to manage shared values. Instead of declaring global variables accessible to everyone and anyone, variables are wrapped within the execution environment of a function. Changes to a shared variable are mediated by a function, which limits how a variable can be changed. Functions related to I/O necessarily depend on side effects (with the exception of pure functional languages). These functions should also be isolated from the rest of the system to limit the reach of side effects. Scientific computing introduces another area of non-determinism, this time deliberately.

Pseudo random number generators pose an interesting conundrum. On the one hand, they are meant to be random. But for repeatability, they need to be deterministic. RNGs are technically deterministic given an initial seed. Each successive value is known based on the initial seed and previous value. The sequence appears random and satisfies the properties of randomness, despite being deterministic. However, if the initial seed is unknown, the pseudo random numbers are effectively random. Therefore, to facilitate repeatability, the initial seed is typically set explicitly.

1.2.2 Lazy evaluation

Imagine a time before computing. In this age, there were no electronic calculators to compute roots or logarithms. Instead, lookup tables were used to find the value of a function based on an input value. Suppose no table yet exists. How do you go about constructing this table? One approach is to collect a set of input values and construct the image of the function on that set. For example, you can construct a table for the base 10 logarithm of the integers 1–100. Whenever you need the value of the logarithm, you just look it up and use the value. There are two drawbacks to this approach. First, the table contains a small set of possible input values. Any numbers that don't exist need to be computed. One solution is to add more numbers to the table. But the input domain is infinite, so there will always be numbers that aren't present in the table. Worse, it takes time to compute the logarithm. Calculating a table up with 100 numbers requires a lot of time upfront, before starting an analysis.

There's no guarantee that any of the precalculated values will actually be used, it's a waste to compute this upfront. To reduce the table construction time, it's possible to construct the table incrementally instead. Values will only be added to the table only when they are computed for the first time. Any subsequent calculations will use the lookup table. This technique is known as memoization and is an example of a process.

Another example of a JIT process is lazy evaluation. Like the calculation of the lookup table, variables or functions that are lazily evaluated are not evaluated until needed. Lazy evaluation eliminates a lot of computing overhead, since unused values are not computed. Most functional programming languages are lazy and R is mostly lazy. Lazy evaluation and just-in-time processes are not limited to computing. Toyota introduced JIT manufacturing in the 1960s [**?**] to minimize inventory overhead. During the model development process, being lazy can save the scientist precious time by minimizing unused work.

1.2.3 Generic versus specific types

Many diverse fields of mathematics are described in terms of sets. In addition to sets, there are tuples (ordered sets), vectors, matrices, and of course, functions. These basic types are used over and over. Some disciplines of mathematics may introduce their own types, but they usually build on these basic types. For example, a random variable is a special type of function, as are sequences.

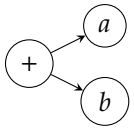
Matrices can represent all sorts of real-world phenomena, from asset returns to economic utility to social networks to a corpus of documents. This flexibility of matrices to represent arbitrary concepts maximizes reuse and compatibility of work built on matrices. Any field that utilizes matrices can reuse analysis methods from another field that utilizes matrices. This compatibility is great for science since it minimizes duplication of effort.

Unlike object-oriented languages that include a massive framework of types, most functional programming languages rely on just a few general types. R also includes a limited number of built-in types. This spartan set of types is usually sufficient to create a complete model and surrounding system. The UNIX philosophy also embraces a minimal set of tyoes. Most programs support a single input format: a text stream. Core UNIX tools read from the so-called standard input `stdin` and write to the standard output `stdout`. This simple interface is highly modular. Programs that adhere to this simple protocol can be used and arranged in any number of arbitrary ways to accomplish a task. Put another way, a small set of generic types promotes modularity and reuse.

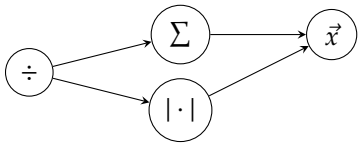
Taking a lazy approach, most model systems should rely on the base types first. Only when it is clear that type specialization is required should custom types be created. Compare this to object-oriented programming, where the definition of custom classes is central to the programming work. The simple act of creating a class hierarchy imposes a structure that is difficult to change. Any changes to the underlying model requires refactoring of the class structure, which takes away valuable time from the work of analysis.

1.2.4 Computational graphs

Frameworks for deep learning, like TensorFlow, describe transformations on data as a computational graph. These graphs leverage two principles of functional programming: declarative notation and function composition. Formally, computational graphs are **directed acyclic graphs (DAGs)** that represent a computation. For example, the simple calculation $a + b$ can be represented as a computational graph:



In this representation, functions (operators) and variables are represented as nodes. Variables are usually terminal. Edges show the dependency from the function to the operand. More complicated computations can also be represented. Take the mean of a vector x, which can be described by:



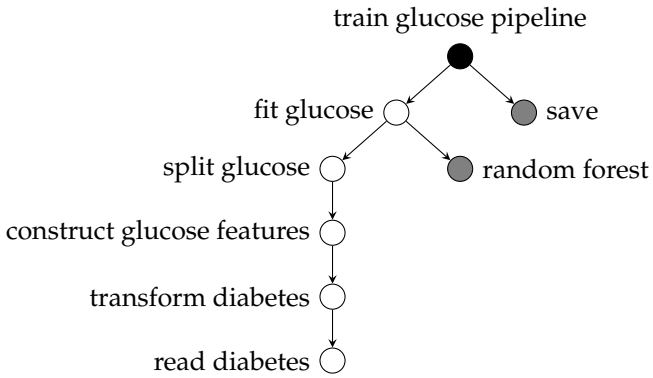


FIGURE 1.2: Functions in the diabetes package drawn as a dependency graph. The black node represents the function itself. Gray nodes represent function dependencies in the body of a function, while whilte nodes represent dependencies from the function arguments.

Notice that both the summation and the length operator depend on x. One reason for using computational graphs is that updates to values can automatically propagate up the edges to dependent nodes. In the graph for the mean of x, if x changes, then the value of $\sum x$ and $|x|$ will change. Once these values change, the result of the division operation will update to reflect the new values of its operands.

Another benefit of computational graphs (and DAGs in general) is that it's easy to partition a computation into smaller subgraphs. These subgraphs can then be moved to a different compute node, paving the way to distributed computing. Numerous approaches exist for distributed computing. Functional programming approaches have become popular because they are easy to implement and easy to reason about.

Computational graphs are not limited to deep learning. Indeed, so long as the functions used are determinstic and acyclic, any chain of function composition can be viewed as a graph. Data processing pipelines can often be represented as a graph. Figure 1.2 shows the training pipeline for a diabetes analysis as a computational graph. Two types of dependencies are represented. Within the body of a function, a function may call another function. This dependency is colored gray. White nodes represent a dependency in a function argument itself. Formulating model systems as computational graphs imposes certain restrictions on the structure of the program. These restrictions promote determinism by eliminating global variables and complicated dependencies. Representing existing systems as a graph can identify unnecessary interdependencies. The DAG also gives clues about what needs testing.

1.3 The UNIX philosophy

In an age of planned obsolescence, it's remarkable that the Internet runs on 40-year-old technology: UNIX. What has made UNIX stand the test of time? UNIX started its life in 1969 by Ken Thompson. There have been numerous changes to UNIX over the years (and its successors, e.g. Linux), but the philosophy surrounding UNIX has changed remarkably little. The UNIX philosophy started as a collection of programming principles based on Thompson's insights creating UNIX. Over time, the philosophy has been refined.

Key elements of the UNIX philosophy echo functional programming such as modularity, composability, and simplicity. Other aspects echo the goals of reproducible science: transparency, extensibility. Accessibility appears in the guise of representation, least surprise, and separation.

While the UNIX philosophy is targeted at the operating system (OS) and the ecosystem of programs surrounding the OS, the principles equally apply to functions within a program. This book shows how to leverage the UNIX philosophy for data science.

1.4 Other approaches to reproducible science

One pillar of the UNIX philosophy is codified in [32] as the Rule of Diversity: "Distrust all claims for 'one true way'". There are plenty approaches to reproducible science, all equally valid. What sets this book apart from other approaches is that it is less a framework and more a philosophy. Frameworks offer convenience and simplicity early on. They act as training wheels for for beginners to become productive in a short amount of time.

At some point training wheels start getting in the way. Maybe the sensible defaults no longer seem sensible; maybe the automagical functionality has lost its appeal; maybe you want to move a little faster; or maybe you've started forming your own opinions. This book is for those ready to shed your training wheels and become truly independent. Programming is an expression of the programmer. You don't talk nor write like others. Instead you develop your own voice. Programming is no different, and this book guides the way.

If you've been on training wheels for a while, your initial reaction may be to reject this call to adventure. After all, your process works fine as it is. And that's okay. Eventually though, you may find certain operations difficult to do using the approach of your favorite framework. Once this happens, it's necessary to go deeper.

Another drawback to frameworks is that the time spent learning the framework is usually not transferable. For example, the `pandas` package in Python has an idiosyncratic syntax that is difficult to learn. Once you've

mastered these idiosyncracies, that knowledge cannot be applied anywhere else. Time is finite, so we have to spend our time wisely. Do we want to spend time learning idiosyncratic knowledge or core principles and tools that have unlimited application?

1.4.1 The tidyverse

The so-called tidyverse is a collection of packages with consistent syntax and semantics. It is good for getting started with data science and R. For the beginner, it makes R less scary with the promise of becoming productive quickly. The tidyverse has strong and consistent opinions, stripping away functionality and providing usually sensible defaults. [2]

According to [37], there are four principles of the tidyverse. They state that you should:

- reuse existing data structures,
- compose simple functions with the pipe;
- embrace functional programming, and
- design for humans.

These principles are generally consistent with this book and help prepare you for the next stage of functional programming proficiency. In other words, the tidyverse acts as training wheels for the budding data scientist, either new to modeling, programming, or both.

1.4.2 ReproZip

Taking a different approach, ReproZip is minimally invasive and doesn't require any changes to your programming style nor adhering to other people's opinions. ReproZip focuses on tracing the execution path of your model to produce a self-contained package to re-run the model. Behind the scenes, it uses virtual machines or containers to execute the code.

While ReproZip doesn't require any changes to your code, you do have to learn how to use their toolchain. You need to decide if you want to spend time learning one-off syntax or general syntax (e.g. `make` and `Docker`) that can be more broadly applied. The facade of simplicity also hides the underlying implementation, creating a certain amount of vendor lock-in.

1.4.3 REANA

Initially developed to facilitate repeatable experiments in particle physics, REANA leverages Python virtual environments and containers (via Kubernetes) to manage and run scientific workflows. REANA is comprehensive

[2] Too many defaults and "automagic" behavior can cause confusion later on. An example is ggplot2, where it "will only use six shapes at a time. By default, additional groups will go unplotted when you use the shape aesthetic." [16]

and spans deploying to a compute cluster. However, REANA is very Python-centric and support for R is undocumented. There is also a fair amount of upfront learning and configuration just to get started. For example, you need to understand the syntax of the configuration files in addition to the specific workflow defined by REANA. For computationally intensive models, this upfront investment is probably justified, as REANA abstracts a lot of complexities of reproducible science.

1.5 Coding conventions

To improve readability, different fonts are used to represent different entities within the text. When referring to the language, R is typeset using a sans serif font. Code listings use a `fixed width` font. References to variables, functions, and packages within the prose also use a fixed width font. The three canonical higher-order functions, *map*, *fold*, and *filter* are also italicized to underscore their importance. This helps avoid confusion with other meanings for these words and reinforces the canonical nature of these higher-order functions. Mathematical expressions and objects are *italicized*. When switching between mathematical objects and their code counterparts, I maintain font consistency to help distinguish the context. In general, code is provided inline, as this is more conversational. That said, breaking up a function into too many little pieces can impede understanding. Thankfully, functions are meant to be short, so they can usually be listed in toto. Code examples may also depict the result of a command in the R console. In these situations, executed code is preceded by a `>` character, which represents the console prompt.

1.6 Package dependencies and datasets

A number of packages are referenced in this book. Many are written by me to support my own model development. Table 1.1 lists the packages used in the book and their purpose. Readers are expected to know how to install and load libraries. If not, an introductory text on R should be read prior to reading this book.

Most chapters set the stage with a single dataset. This helps focus the discussion on a single analysis theme. For example, this chapter focused on the *diabetes* dataset. Some datasets are available within R, such as *iris*, *trees*, *swiss*, *fgl* (in *MASS*), and *baseball* (in *plyr*). Others, like *adult*, *congress*, and *diabetes* need to be downloaded and constructed locally. Some of these datasets

Package	Purpose
`caret`	Model developent and tuning
`dplyr`	Data manipulation
`futile.logger`	Logging
`lambda.r`	Functional programming dispatching and type system
`lambda.tools`	Collection of functions for functional programming
`magrittr`	Infix pipe notation
`MASS`	Contains useful datasets
`plyr`	Data manipulation
`purrr`	Collection of functions for standardized development
`rmr2`	MapReduce implementation for RHadoop project
`zeallot`	Pattern matching for assignment

TABLE 1.1: Packages used in the book

have non-standard data formats. They are intentionally chosen to provide real world examples of writing code to extract, assemble, and normalize the data.

The *diabetes* dataset [20] is a collection of readings for 70 diabetics. This dataset is different from the Pima Indian diabetes dataset that comes with R. A file in the dataset corresponds to various measurements for a patient. These include glucose and insulin levels. The entries are key-value pairs tagged with a date and time. The name of each key is provided in a separate data dictionary.

The *adult* dataset [24] is designed for binary classification. This dataset contains demographic information on a population of people. The response variable indicates whether each person has an annual income greater or less than $50,000. It is a single file with a slightly bespoke format. There is no header, and the response variable isn't coded as a standard boolean value.

The *congress* [**?**] dataset provides information about the legislators of the United States Congress. It includes demographic information for both houses.

The *bills* [30] dataset comprises all of the bills and resolutions introduced (and possibly passed) in the US Congress. This dataset is an example of hierarchical data.

1.7 Summary

Like any tool, programming can either make your work easier or harder. It's harder to cut with a dull knife than a sharp knife and is even more dangerous. Dull programming skills are similar. The extra effort required to cut through a problem often leaves a trail of blood and tears. It's therefore important to be honest about the role of programming in science and also your skill level.

Recognizing some of the traps of procedural and imperative programming is your first call to adventure. Before stepping over the threshold, Parts I and II will help you prepare for your journey to ensure a successful return.

1.8 Exercises

Exercise 1.1. Find a paper on `https://arxiv.org/` that describes an experiment. Provide the URL of the paper and a paragraph discussion on its reproducibility. What information do you need to reproduce the results of the paper? How difficult is it to reproduce?

Exercise 1.2. In theory, any practitioner of a relevant field can take a patent application and reproduce the claim. Does this hold in practice? Use the USPTO patent search at `http://appft.uspto.gov/netahtml/PTO/search-bool.html` to find two patents: one on the keyword "Bayesian" and one on the keyword "machine learning". Provide the URLs to both patent applications and discuss how practical it is to reproduce the claimed results. Which application do you think is easier to reproduce?

Exercise 1.3. Define an algorithm for a daily routine in your life. What is your expected outcome? What assumptions are embedded in your algorithm? How difficult would it be for someone else to achieve the same outcome?

For example, an algorithm for how you clothe yourself in the morning is not guaranteed to produce the same results (i.e. outfit) given two different people. Both the clothes and body are embedded assumptions.

Part I

Tools for Model Development

2

Core tools and requirements

> We shape our buildings;
> thereafter they shape us.
>
> *Winston S. Churchill*

Before embarking on this journey, proper preparation is necessary. It's important to have a model development environment designed for reproducible science and incremental development typical of data science. Our environment dictates how we work, so it's important to start with an environment that fosters workflows typical of data science. A minimally working environment consists of three building blocks: the GNU/Linux [1] and corresponding build tools, R, and Docker. You also need to consider the operating system that hosts your model system. Personal workstations may run Linux, Mac, or Windows, but automated system and scientific computing environments almost always run Linux. An overview of Linux essentials is provided in Section 2.1. Container technology is useful to reconcile these different computing environments. Section 2.4 discusses using Docker to run Linux environments within a host system running a different operating system. While this book is about R, the reality is that scientific work is also conducted in Python. At times, it's useful to use libraries or packages in Python. Section 2.5 shows how to achieve interoperability between R and Python. My `crant` utility automates this process and can create an isolated Docker environment for each project you work on. Discussed in Section 2.6, this tool can also build and install R packages from the command line.

2.1 GNU/Linux

Some journeys are taken on foot, though it's usually faster with a trusty steed. The same is true of this journey, where the steed takes the form of a self-contained and reusable model environment. Linux makes a solid foundation for model development. It continues the rich heritage of UNIX and is the

[1] Colloquially known as Linux, the complete operating systems is technically GNU/Linux. For sake of brevity, I will often just say Linux.

operating system of choice for production models. Doing development in a Linux-like environment removes potential friction later on. And Linux has never been easier to use. Modern Linux distributions sport user-friendly GUIs and one-click installers for technical novices, while preserving the power of the command line to more experienced users. There are two dominant lineages of Linux: Debian and RedHat. [2] Either are suitable for data science, although my examples and code assume a Debian environment. The `bash` scripts in this book should be portable across these two strains of Linux, and even Mac OS X and Windows (using Cygwin). If you're not convinced that Linux is preferable for model development, you can use Docker to create a Linux environment to run your models. There is good reason to do this even if you develop in Linux, which I discuss further in Chapter 6.

Most UNIX tools appear simple but are dizzyingly complex and powerful. It can take years to fully master the text editor `vim`, let alone the `bash` command shell and the stream editor `sed`. The steep learning curve often discourages novices from fully embracing Linux. This is a shame, because UNIX and its philosophy has much to offer. Understanding it provides the backdrop and context for much of programming as we know it, not to mention R. An effective approach to using and learning UNIX tools is to start small. It's okay not to understand how everything works from day one. Focus on solving the specific problems you have. From there commit some time to learning incrementally. Use new problems as opportunities to learn a little bit more. You'll be surprised at how quickly you become proficient.

2.1.1 The command shell

In a UNIX system, the shell is how you interact with the operating system. Colloquially known as the terminal or command line, it has a text-based interface. The command line provides a prompt to issue commands and execute programs. Most of UNIX is modular, and the shell is no different. Due to its near ubiquity, a good starting point is to use `bash`. While `bash` is most common, the older `sh` also comes standard with Linux. Other shells, like `ksh` and `zsh` offer different features and conveniences but must be installed explicitly.

Virtually every command in UNIX is actually an executable program. Some useful commands in UNIX are given in Table 2.1. We'll use many of these commands throughout this chapter. Any file with executable permissions is deemed a program. Whether it runs is dependent on the file. Programs are executed by specifying their path (location in the file system) on the command line and hitting enter. This can be an absolute path (starting at root, or `/`) or a relative path, based on the current directory. For example, to list all files in a directory you execute the command `/bin/ls`. This command is normally

[2] Popular distributions like Ubuntu, Kali Linux, and Tails are all derived from Debian. Fedora and RHEL are RedHat distributions, while CentOS is derived from RHEL.

Command	Mnemonic	Description
`man`	Manual	View the manual for a given command/topic
`set`	N/A	Set/view shell options
`date`	N/A	Get or set the system date
`pwd`	Print working directory	View which directory the current shell is in
`cd`	Change directory	Change the directory of the current shell
`ls`	List	Show the files in a directory
`rm`	Remove	Delete a file, files, and/or directories
`touch`	Touch a file	Create a file with no contents
`mkdir`	Make directory	Create a directory
`rmdir`	Remove directory	Delete a directory
`chmod`	Change file mode	Change the permissions of a file
`chown`	Change owner	Change the owner of a file
`su`	Superuser or switch user	Switch to another user, possibly the superuser
`sudo`	Pseudo	Temporarily run a command as another user
`.`	N/A	Execute a file as a program
`source`	N/A	Execute lines in a file in current shell
`xargs`	Execute with arguments	Execute a command using stdin as argument list
`du`	Disk usage	View the size of a file or directory
`top`	Top processes	View running processes interactively
`ps`	Process	Report on some running processes
`kill`	Kill	Stop a process
`find`	N/A	Find files matching conditions
`echo`	N/A	Display a line of text
`cat`	Concatenate	Concatenate files and print to stdout
`head`	N/A	Print the first n lines of a file
`tail`	N/A	Print the last n lines of a file
`grep`	Regular expression	Search files using regular expressions
`cut`	Cut file	Remove columns from delimited files
`sort`	Sort file	Sort lines in a file
`uniq`	Unique	Remove duplicate lines
`diff`	Difference	Find differences between two files
`tr`	Translate	Replace one character with another
`sed`	Stream editor	Modify files using scripted rules
`awk`	N/A	A language for manipulating text files

TABLE 2.1: Some useful commands available in Linux distributions. Commands range from navigating the file system to managing processes to processing text files. The `man` command provides reference documentation for each command.

executed using just the name of the program, in this case `ls`. Most commands don't require the full path to be specified because they are located in the `PATH` environment variable. Linux uses this variable to find the full path of a program entered on the command line. A common trick is to type in a few letters of a program and hit the tab key to see what programs in the path match this string. To view the complete contents of the path, type

```
$ echo $PATH
/home/brian/google-cloud-sdk/bin:/home/brian/bin:/home/brian/.local/b
in:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/
games:/usr/local/games:/snap/bin:/home/brian/workspace/crant
```

Each element in the `PATH` is separated by a `:` and searched in order. The first matching program will be the one used.

Any text following a command is treated as its argument list. Each argument is separated by a space. For example `ls workspace` shows the contents of the `workspace` directory. Notice that these commands simply denote functions. The above command denoted as a function in a programming language resembles `ls("workspace")` and has equivalent meaning. Commands also support options that are orthogonal to its arguments. Technically there is no difference between an argument and an option, but tools tend to treat each differently. Arguments act as input data to a function, whereas options tweak the behavior. This is similar to a model that takes a training set as its input and also has hyperparameters that control the behavior of the algorithm.

UNIX commands support two option styles: short and long. Short options are abbreviated using a single letter, (typically) preceded by a hyphen. The `-l` option of `ls` lists files and displays the output as a table with extra information. Short options can be combined as a single string following the hyphen. To list files and sort by reverse chronological order, use `ls -ltr`. The `-t` option sorts the files in chronological order, newest first. The `-r` reverses the view. Combined they produce reverse chronological order. Even with command options, the modularity and composition make an appearance.

Some programs don't require a hyphen for their options. Examples include `ps` and `tar`. Options for these commands are provided as a single string, such as `ps aux`, which lists all running processes in a user-oriented table. Options are generally commutative, so the order doesn't matter. The command `ps axu` is therefore equivalent to `ps aux`. However, we'll see more complex commands later on where the order of options do matter. In particular, `docker` has order-specific arguments, as does `git`.

Options can also include a value that represents a key-value pair. In this form, the option key is followed by its value, such as `ps -C R`, which shows all running R processes. These options are typically separated from the options without arguments, such as `ps w -C R`. The option value can follow the switch without a space. Consider the `awk` language, which can be used to extract a specific column from a delimited text file. The `-F` switch sets the delimiter field, which can be any single character. It doesn't matter whether a space

separates the switch from the actual delimiter field or not. For example, `awk -F`, behaves the same as `awk -F ,`.

Long options are meant to be self-documenting at the expense of conciseness. Instead of a single hyphen, long options start with two hyphens, followed by the option name. The name can span multiple words, where hyphens replace spaces. [3] For example, `ls -l --author` includes the author of each file in the long listing. Long options can also represent key-value pairs. For this usage, the option name is followed by an equals sign and then the option value. The displayed file size units in `ls` can be controlled using `ls -l --block-size=M`, which changes the units to megabytes.

2.1.2 File permissions and scripts

All files in UNIX have an owner and a group. The owner is initially the creator of the file, though it can be changed to anyone. A group represents a group of users sharing the same permissions. Depending on the system, the group might just be the owner initially. All other users are considered a third group, exclusive to the union of the file's user and group. Each of these three roles has a set of permissions: read (r), write (w), and execute (x). A bit mask represents these nine permissions, where 1 means granted and 0 means not granted. In the file system, letter abbreviations are used to indicate that a permission is granted, while a hyphen means not granted. Each set of permissions is grouped together to form an equivalent octal number. For example, `rw-` implies 110, which is shortened to 6 in octal. A read-only file is marked `r--`, which is 4 in octal. The permissions for all three roles represent a three digit octal number. A file that everyone can read but only written to by the owner or group member is represented as `rw-rw-r--`, which translates to 664.

Marking a file as executable with `chmod +x` tells the system the file *may* be executed. Whether it *can* be executed depends on the file. Shell scripts are the easiest way to create a custom executable file. The simplest `bash` scripts just list out a sequence of commands to run. When called on the command line, the `bash` interpreter will execute each command in the script in order. Executable scripts require a header that specifies the interpreter that actually runs the script. This header must be the first line of the file [4] and begins with a `#!` (read she-bang). What follows is the full path of the interpreter to use for the remainder of the file. Aside from `/bin/bash`, `/usr/bin/env python` and `/usr/bin/env Rscript` are commonvi interpreters.

Example 2.1. A toy script just prints "Hello, world" to the shell. We can create this directly in `bash` using the `echo` command and a file redirect.

```
$ echo '#!/bin/bash
> echo "Hello, world"' > hello_world.sh
```

[3] This is a convention and not enforced.

[4] The only exception is that comments and newlines can precede it.

The output of `echo` is sent to `hello_world.sh` instead of `stdout`. This script is not executable yet because files are not executable by default. The `ls` command can verify this statement.

```
$ ls -l hello_world.sh
-rw-rw-r-- 1 brian brian 31 Sep 17 12:07 hello_world.sh
```

The first character (a hyphen) specifies the file mode as an octal number. The following nine characters represent the complete file permissions, which are 664.

A script can be made executable using the `chmod` command. File permissions can be set by specifying the permissions in octal. A common permission is 755, which means the owner can read, write, and execute the file, while everyone else can read and execute. This means only you can modify the file, which is a safe default.

```
$ chmod 755 hello_world.sh
$ ls -l hello_world.sh
-rwxr-xr-x 1 brian brian 31 Sep 17 12:07 hello_world.sh
```

Alternatively, the file can be made executable by using the `+x` command switch. The `hello_world.sh` script can now be executed. To run the script in the current directory, it must be prefaced with `./`.

```
$ ./hello_world.sh
Hello, world
```

If this file is accessible via the `PATH`, it can be called without the preceding `./`□

Scripts help codify and automate manual procedures. It's generally a good idea to write a script if you find yourself doing the same thing more than once. Scripts are also useful for tasks that happen infrequently. If not documented, you waste time remembering how to do an infrequent task each time you need to do the task. This process knowledge must be re-learned over and over. A more efficient approach is to use a script that documents the steps involved until it can be run as a standalone script.

2.1.3 Linking commands with the pipe

UNIX is meant to be modular and programs composable. The UNIX pipe operator `|` is the glue that joins programs together. Given commands `a` and `b`, the syntax `a | b` directs the `stdout` of `a` into the `stdin` of `b`. Multiple programs can be composed together using the pipe. This is possible since UNIX programs adhere to text as a common data format. If we ignore command options and only look at `stdin`, UNIX commands are univariate functions. Pipes thus represent simple function composition with left-to-right precedence. Traditional function composition has right-to-left precedence and looks like $(b \circ a)(x) = b(a(x))$. This duality is useful when designing systems and data processing pipelines as graphs (see Section 1.2.4).

The pipe concept appears in programming languages in a few guises.

Haskell gives the most formal treatment, showing how the operator is actually a **monad**. [5] Object oriented languages accomplish composition using method chaining. The `magrittr` package implements a version of pipes in R. These pipes support multivariate functions, creating semantics similar to method chaining. When functions are chained together, the first function argument is assumed to receive the return value of the prior function in the chain. The remaining function arguments are specified using **partial application**. The pipe operator therefore rewrites the function call to obtain the correct argument order prior to calling the function.

Example 2.2. One way to generate random numbers in `bash` is to read from `/dev/urandom`. This special device is a kernel-based random number generator that gathers noise from device drivers and elsewhere to create reliably random numbers. [3] Reading from `/dev/urandom` can generate any random sequence of bytes. We can emulate 10 rolls of a die using

```
$ cat /dev/urandom | tr -dc '1-6' | fold -w 10 | head -1
3633546154
```

Like `grep`, `tr` supports specifying the inverse, or complement, of a pattern. The `-c` specifies the complement of the range 1 to 6, while the `-d` option deletes those characters. □

Example 2.3. When your hard drive fills up, it's necessary to delete files to free up space. It's inefficient to randomly delete things. Wouldn't it be easier to delete or archive the largest files first? Let's find all directories from the current directory that are larger than a gigabyte.

```
du -sh * | grep '[[:digit:]]G$'
```

This approach uses `du` to summarize the disk usage of files and directories. The output is piped to `grep`, which filters the results using a regular expression. If the working directory contains more than directories, individual files may also appear in the output. We can use `grep` a second time to filter out entries that end in a file extension.

```
du -sh * | grep '[[:digit:]]G' | grep -v '\.\w\+$'
```

□

Example 2.4. The `ps` command shows running processes. Suppose you want to find all running `R` processes and see how long they've been running. Maybe you want to check any scripts running longer than 10 hours.

```
$ ps u -C R | grep brian | awk '{print $2, $10}'
22112 8:49
28387 6:10
29438 2:32
```

This command finds all R processes owned by `brian` and prints the process ID (PID) and the elapsed time of each R process.

[5] My package `lambda.r` implements some common monads in R.

□

Example 2.5. In Example 2.3, we looked for gigabyte files. What if we want to find the file sizes of all CSVs instead? The `find` command will find all files matching a pattern. We'll then pipe this to `xargs`, which transforms the output of the preceding command to arguments of the succeeding command.

```
find . -name *.csv | xargs du -sh
```

All matching CSVs are piped to `xargs`, which constructs a complete argument list for the command `du` and executes it. The `xargs` command thus transforms the `stdout` of any program into an argument list. A similar technique is available in R using `do.call`. □

2.1.4 File redirects

The pipe redirects `stdout` to the `stdin` of a subsequent program. It's possible to redirect program output to a file instead. Two redirection operators control program output. The `>` operator truncates and writes to the specified file. The `>>` operator appends output to the given file, creating it as necessary.

Used less frequently is the file redirect operator `<` for `stdin`. This operator is useful for programs that expect input coming from `stdin` but you want it to read from a file instead. These operators can be used together, with `<` preceding `>`.

Example 2.6. The single `>` effectively creates a new file each time it is used by truncating existing files. Any previous data in the file are lost.

```
$ echo "hello" > hello.txt
$ echo "hello" > hello.txt
$ cat hello.txt
hello
```

This operator therefore exhibits **idempotence** since it can be called repeatedly without any change in value. The double output redirection operator `>>` is not idempotent. Every time it is called, the data will be appended to the existing file.

```
$ echo "hello" >> hello.txt
$ echo "hello" >> hello.txt
$ cat hello.txt
hello
hello
hello
```

Note that the first `hello` is the initial content of the file from the previous operation. □

Example 2.7. Before starting an analysis, there's usually a bit of time spent getting familiar with a dataset. For files in non-standard formats, it can be convenient to examine some files in the shell before writing a script to parse it in R. The *diabetes* data contain multiple files, one per patient. The files are

tab delimited, using a format similar to key-value pairs. Suppose we want to know how many records in the first patient file `data-01` that contain insulin doses, which are features 33, 34, and 35.

```
$ cut -f 3 < data-01 | grep '33\|34\|35' | wc -l
381
```

We use a file input redirect to pass a file to `cut`, which extracts the third column (the data code) of `data-01`. We then use `grep` to match all codes that are either 33, 34, or 35. Finally, we pipe this output to `wc`, which counts the total number of matching lines. □

Most programs read and write from `stdin` and `stdout`, respectively. We don't want errors and warnings to appear in `stdout` since it can corrupt the input to another program. To keep `stdout` clean, programs are supposed to write to the separate `stderr` stream. Doing so helps separate program output from error output. Both `stdout` and `stderr` are printed to the console. We can redirect any of the standard streams by prepending a file descriptor to the redirection operator. The three standard streams are denoted 0 (standard input), 1 (standard output), 2 (standard error).

Example 2.8. To suppress error messages altogether, `stderr` can be redirected to the special file `/dev/null`. A typical example is directory creation. If the directory already exists, `mkdir` emits an error.

```
$ mkdir data
$ mkdir data
mkdir: cannot create directory 'data': File exists
```

If you just want to ensure that the `data` directory exists, this is a pointless error. Printing the error in a script is poor hygiene and creates red herrings that impede future troubleshooting. One solution is to suppress the error message.

```
$ mkdir data 2> /dev/null
```

Note that there cannot be a space between the file descriptor and the redirection operator. □

2.1.5 Return codes

Error messages are helpful, but for automation, error codes are more useful. All UNIX programs return a code upon termination. Scripts can use this value to execute different logical paths. By convention, a 0 indicates a success, while a positive integer indicates a failure. This may seem counterintuitive to traditional boolean logic. However, it is convenient since there is only one possible success code but an arbitrary number of error codes. The return code of a program is given by the special variable `$?`. This variable only holds the value of the last executed program, so it must be used immediately or saved in another variable.

Example 2.9. The `which` program finds the full path for a program assuming it is accessible from the `PATH`. Normal usage looks like

```
$ which ls
/bin/ls
```

If no program is found, it returns nothing:

```
$ which s
```

We could test that the output was empty, but it's more explicit to examine the return code directly.

```
$ echo $?
1
```

According to `which`, the input `s` resulted in an error, since no program was found. □

Example 2.10. Even if error messages are suppressed, the error code can still be accessed. Suppose I list files in a directory. If this directory doesn't exist, I'll see an error message.

```
$ ls ksdf
ls: cannot access 'ksdf': No such file or directory
```

In a script, this message isn't so useful and can be hidden using a redirect.

```
$ ls ksdf 2> /dev/null
```

Now the output is empty, since nothing is written to `stdout`. Within a script, I can check for an error by inspecting the return code, which in this case is 2.

```
$ ls ksdf 2> /dev/null
$ echo $?
2
```

□

2.1.6 Processes

Every program runs as a distinct process in UNIX. When the program starts, it is assigned a process ID (PID) and a priority. The scheduler uses this information to decide which program gets CPU cycles at any given time. When UNIX boots, a number of processes start automatically. These long-lived processes are typically daemons and servers. They run in the background, out of sight. In the shell, programs run in the foreground. A foreground process will get its input from the shell and display to the shell. This is necessary and useful to interact with a program. In the background, a program won't get input from the shell.

Running processes can be temporarily suspended by typing Control-Z. The foreground process of a shell will be paused and control will return to you via the shell. To resume the process, type `fg`. To send the process to the background type `bg`. Once in the background, a process must be stopped using `kill`. This program sends a termination signal to a program, which then performs any necessary clean up before shutting down.

```
#!/bin/bash

i=1
while [ "$i" -gt "0" ]
do
  echo $i
  sleep 1
  ((i+=1))
done
```

LISTING 2.1: A simple script `count.sh` that counts from one.

Example 2.11. Let's create a toy script that counts the whole numbers from one on. Listing 2.1 gives one such implementation. We'll use this script to show that a program is running even when it is in the background. Name the script `count.sh` and update the permissions so it is executable. Executing the script will start printing the natural numbers to the console. Hit Ctrl-Z to pause the program. You can use this shell for other purposes now. When ready, type `fg` to resume the program. □

For long running commands or to start a server, it is useful to execute a command in the background. The `&` at the end of a command tells the shell to run a command in the background. When the program runs in the background, you can still use the shell for other acitivities. But be aware that the background command still has access to `stdout`, so your shell may get periodic updates from the program! To completely separate the program from your shell, use the `disown` built-in command and specify the PID of the process to disown. The terminal window can be closed and the process will still run.

Example 2.12. In a GUI environment, sending programs to the background isn't very compelling. It's just as easy to open a new shell. However, the story is different on a remote server. Running a model in a remote shell is a common source of consternation. Connections are fragile and when they drop, your job is killed. Ouch! It's easy to underestimate how long a model will take to run. One approach is to use `screen`, which creates a virtual `tty`. But this is somewhat involved. A simpler alternative is to send the process to the background. Then it will keep running regardless of whether your current shell is active or not.

We can emulate a remote server connection by logging into our existing workstation via SSH.

```
$ ssh localhost
brian@localhost's password:
Welcome to Ubuntu 16.04.5 LTS (GNU/Linux 4.4.0-134-generic x86_64)
```

Now run the `count.sh` script from Example 2.11 in the foreground, as though we're running a model job.

```
./count.sh
```

Imagine that after some time, you realize the job will take quite a while longer (in this case an infinite amount of time). It's no longer practical to sit and monitor its progress. It's time to send the job to the background and avoid accidental termination due to an inactive session. Use Ctrl-Z to suspend the process and then `bg` to send it to the background. While `bg` sends a process to the background, it is still tied to the current shell. The `disown` command dissociates the process from the current shell. Once that command is executed, it is safe to close the shell.

```
$ disown $(ps ux | grep 'count\.sh' | awk '{print $2}')
$ exit
```

In a separate terminal window, use `ps` to verify `count.sh` is still running.

When you've completed this experiment, it's a good idea to clean up and terminate the program. The same `ps` command can be used to find the process and `kill` used to terminate it. For variation, we can pipe the PID to `xargs`, which constructs the command using `stdin` as the argument list.

```
$ ps ux | grep 'count\.sh' | awk '{print $2}' | xargs kill
```

Respecting different approaches is another part of the UNIX philosophy, which is empbodied by the Rule of Diversity: distrust all claims for "one true way". [32] □

2.1.7 Variables

As a Turing complete language, `bash` provides variables, functions, and control structures. Variables are defined using an identifier followed by an equals sign and then the value, such as `a=1`. Variable names in scripts typically use lowercase letters, while environment variables use uppercase letters. For example, most standard environment variables in the shell are uppercase. These include `PATH`, `LANG` and `TERM`, which can be found using the `set` command. Following this convention makes it visually clear where a variable was defined. For assignment, a quirk of `bash` is that no spaces are allowed surrounding the equals sign. Another interesting behavior is that strings do not always need to be wrapped in quotation marks. For example, the string 'cali' can be assigned to the variable `a` as a string literal:

```
$ a=cali
```

Multi-word strings do need to be wrapped in quotation marks, otherwise the second token will be treated as a command (see Section 2.2 for how this is used).

To reference a variable, a dollar sign is placed before the identifier. The variable `a` can be used in the assignment of `b`:

```
$ b="super, $a"
$ echo $b
super, cali
```

Variables are often used to construct strings. If a variable is embedded within a string without spaces, there's no way for `bash` to detect the end of the variable name. To properly delimit the variable from the string, they can be wrapped in braces.

```
$ echo "Super${a}fragilisticexpialidocious"
Supercalifragilisticexpialidocious
```

Both single and double quotes can wrap a string, but they have different behavior. Variables in a double-quoted string are interpolated as one might expect. In single quotes, the variables are treated as-is and don't perform any **interpolation**. With single quotes, the variable `a` isn't evaluated.

```
$ echo 'super, $a'
super, $a
```

Compare what happens when we assign the uninterpolated string to another variable. Calling `echo` on this variable forces interpolation. This behavior is useful when chaining commands together, since a variable can be interpolated in the future. Delayed interpolation is partially possible thanks to dynamic scoping. Of course, it should be used sparingly as misuse can cause unnecessary confusion.

2.1.8 Functions

As with other languages, functions help organize and modularize code. Function definitions begin with an identifier followed by a set of parentheses and then braces demarcating the body of the function. Functions must have at least one line in the body, otherwise `bash` will complain. Listing 2.2 shows a simple function that determines whether a program with the given PID is running. The function gets a process list from `ps`. This output is visually tabular. To make it machine readable, `sed` replaces a contiguous sequence of whitespace with a comma. Operating on the resultant CSV, `cut` extracts the second column, which is the PID. Finally, `grep` searches for a process ID that matches the given ID. All output sent to `stdout` is ignored by redirecting it to the special device `/dev/null`. We only care about the return code, so superfluous output just adds noise. If not given explicitly, the return code of the function is dictated by the last program called, which is `grep`.

```
1 pid_exists() {
2   ps ux | sed -e 's/\s\+/,/g' | cut -d , -f 2 | grep "^$1\$" >> /dev/
        null
3 }
```

LISTING 2.2: A function to see whether a process is running.

Unlike other languages, no function arguments are allowed between the parentheses in the function signature. The function `pid_exists` might be a function of one argument or a function of four arguments. We only know based on the arguments referenced within the function. Positional arguments

are captured using the special variables `$n`, where `n` can range from 1 to 9. [6] Only `$1` is used within `pid_exists`, which represents a process ID.

Functions are called like any other program and are passed positional arguments after the function name. To check whether the PID 29049 exists (i.e. is running), the function can be pasted into a shell and executed:

```
$ pid_exists 29049
```

Like programs, the function output is different from the return code. This function does not write to `stdout`, so it appears to have done nothing. It does have a return code though, which can be found by examining `$?`.

```
$ echo $?
0
```

Remember that a return code of 0 indicates that the program was successful. For `pid_exists`, that means the PID 29049 was found (on my system, at time of writing).

The output of a program can be captured in a variable. The final line of a function often calls `echo` to produce a text output that can be piped to another program or stored in a variable. Using `return` is ineffective as this only sets the return code. The special operator `$()` executes a command as a subprocess and capturing its output. Whatever is written to `stdout` can thus be assigned to a variable. The following command gets the name of the current directory.

```
$ dir=$(pwd | tr / '\n' | tail -1)
```

Example 2.13. Configuration data can often be inferred from one location and used to automatically set variables in other places. Doing so streamlines configuration, eliminates syntax errors, and documents a process all at once. In an R package, the package name is used in multiple places, such as in the `DESCRIPTION` and the package source file. For example, if a package is named `bestmodel`, there is supposed to be a `bestmodel-package.R` file in the `R` directory. This boilerplate code can be generated automatically by getting the name of the current directory and using that to construct other file names.

```
1 get_directory_name() {
2   pwd | tr / '\n' | tail -1
3 }
```

This can now be used in a script to create the package file.

```
touch R/$(get_directory_name)-package.R
```

□

Example 2.14. The `echo` command transforms newlines into spaces. This can be confusing to novices because the behavior may seem inconsistent. However, it only happens when a variable is unquoted. Consider the random number generator described in Example 2.2. Let's create a function to roll a die n times. This output can be easily written to a file to be read by R.

[6] There is also a special `$0`, which holds the name of the script being called.

```
roll_die() {
  cat /dev/urandom | tr -dc '1-6' | fold -w 1 | head -$1
}
```

LISTING 2.3: A function to simmulate rolling a die n times.

Calling this function directly from the shell preserves newlines.

```
$ roll_die 4
4
4
2
4
```

Let's capture the rolls in a variable and print it.

```
$ x=$(roll_die 4)
$ echo $x
6 5 1 5
```

This transformation of newline to space is a function of `echo`. Quoting the variable preserves the newlines.

```
$ echo "$x"
6
5
1
5
```

We'll see in Section 2.1.12 that this strange behavior is useful for looping over a sequence of values. □

2.1.9 Conditional expressions

Conditional expressions are defined using either `test` or the equivalent `[` notation. Conditional expressions are just command line programs, and conditional operators are simply options to the program. Numerical equality is tested using the `-eq` option, such as `test $x -eq 5`. Similarly, the greater than and less than comparators use `-gt` and `-lt`, respectively. The `=` operator can compare both numbers and strings. Conjunction and disjunction are expressed as the options `-a` and `-o`. The order of options is important for these operations. Clearly `$y -gt 2` is different from `2 -gt $y`.

The `test` notation is interchangeable with `[` notation, so `test $x -eq 5` is equivalent to `[ $x -eq 5 ]`. Extended tests use `[[` notation. This operator enables the use of more familiar operators like `>`, `<`, `&&`, and `||`.

Example 2.15. Syntax errors in tests can fail silently, leading to unexpected behavior. Consider the variable

```
$ y=3
```

To check if `y` is less than 2, the `-lt` switch must be used in standard test notation.

```
$ [ $y -lt 2 ] && echo true
```

Since this expression evaluates to false, nothing is printed. The same test works using extended notation.

```
[[ $y -lt 2 ]] && echo true
```

With extended notation, it's possible to use the more common `<` comparator.

```
[[ $y < 2 ]] && echo true
```

If this comparator is used in a regular test, the result is unexpected.

```
$ [ $y < 2 ] && echo true
true
```

□

In addition to testing equality and making comparisons, `Bash` offers some non-standard operators as well. The `-z` and `-n` tests whether a variable is empty or not. Hence, these operators test for existence. When a variable is undefined it is equivalent to an empty string. [7] These switches are useful when coupled with program flags. A behavior can be enabled or disabled depending on whether a variable is set or not.

Example 2.16. A common option in a script is whether it should print log messages to the screen. A variable `verbose` can be defined and set to any non-empy value to indicate the script should print log messages, such as `verbose=yes`. The script simply needs to check whether this variable exists and print a log message if it does.

```
[ -n "$verbose" ] && echo "Step 1"
```

Variables can be unset by assigning a variable to nothing (i.e. an empty string).

```
verbose=
if [ -n "$verbose" ]; then echo "Step 2"; fi
```

Alternatively, some scripts are verbose by default. These scripts need to ensure `quiet` has not been set.

```
[ -z "$quiet" ] && echo "Step 1"
```

In either case, the output verbosity can be controlled using a variable. □

2.1.10 Conditional statements

Conditional statements control the branching of different logical paths. Rather than using braces or indentation to represent alternate code blocks, `Bash` tends to use beginning and ending keywords. Conditional blocks start with `if`, followed by the conditional expression. The keyword `then` indicates the beginning of the true block, which is followed by an optional `else` keyword. The end of the block is marked with `fi`, which is just `if` backwards. Here's a block that shows whether the variable `$verbose` was set or not. Multiple statements can appear in either block.

[7] It may seem seem strange to conflate these two concepts, but R does something similar in terms of undefined vector indices. Given a vector `x` with length `n`, accessing `x[n+1]` yields `NA`.

```
$ if [ "$verbose" = "yes" ]
> then
>   echo verbose
> else
>   echo not verbose
> fi
```

If more than one condition needs to be tested, the `elif` keyword is used. This syntax is equivalent to starting a new if-then block after the `else` keyword.

In cases where there is only a true case, the conditional can be condensed to one line using a `;` to indicate the end of each statement.

```
$ if [ "$verbose" = "yes" ]; then echo verbose; fi
```

Even more concise is the use of the `&&` operator that creates a conjunction with the two statements.

```
$ [ "$verbose" = "yes" ] && echo verbose
```

2.1.11 Case statements

When testing whether a single variable matches one of many values, it is more readable to use a switch (or case) statement. The syntax is similar to a conditional block, where `case` and its reverse, `esac`, indicate the beginning and end of the block. The body of the block is delimited by the keyword `in`. Each alternative is defined by a pattern followed by `)`, giving the appearance of a list of choices. A sequence of statements follows each choice. The final statement (even if there's just one) must be delimited with `;;`. For example, some scripts support multiple levels of verbosity. A case statement can determine which level is desired.

```
$ case $opt in
>   0) verbose=0;;
>   1|verbose) verbose=1;;
>   2|loud) verbose=2;;
>   *) verbose=0;;
> esac
```

Choices support limited pattern matching. Alternation of multiple equivalent values is possible with the `|` operator. Hence, the pattern `1|verbose` indicates that either `1` or `verbose` matches this case. Since `bash` supports string literals, notice that no quotation marks are needed around `verbose`. The `*` matches anything and acts as a default case.

2.1.12 Loops

Loops can be constructed using `for` or `while`. For-loops iterate over a list of values. In `bash` this list is explicitly itemized with whitespace delimiting each item.

```
$ for i in H T T H; do echo Coin is $i; done
```

```
Coin is H
Coin is T
Coin is T
Coin is H
```

Explicitly listing all the values to loop over is rather limiting. A variable can represent the list instead. This time, we'll just echo the value so it can be easily read as data.

```
$ x="H T T H"
$ for i in $x; do echo $i; done
H
T
T
H
```

Example 2.17. Sequences can be generated more conveniently using a function. Using the random number generator in Example 2.14, let's simulate n rolls of a die to use as a random variable. We can compute the expected value of this random variable by iterating over its values.

```
1 expected_value() {
2   sum=0
3   for i in $(roll_die $1)
4   do
5     ((sum+=i))
6   done
7   echo $((sum / $1))
8 }
```

The special `((` notation on line 5 represents an arithmetic expression evaluated by `bash`. This approach can only yield an integer result and is therefore imprecise. See Exercise 2.5 for a more precise approach. □

Example 2.18. The *diabetes* dataset consists of a number of tab-delimited files. A `for` loop can be used to convert each file into a CSV.

```
$ for f in $(ls data-*)
> do
>   tr '\t' , < $f > ${f}.csv
> done
```

Within the loop, `tr` translates all tab characters into commas. The input file is passed to `stdin` using the `<` redirection operator. The output is also written to a file using the `>` redirection operator. □

While-loops work similar to for-loops but stop when an expression no longer evaluates to true. Behind the scenes, `while` is just testing the return code for a success value (i.e. 0). Therefore, `while` can evaluate any function or program call, so long as it conforms to the return code convention. A function call replaces the test expression and the return code is tested directly by `while`.

Example 2.19. It's useful to know how many resources an application consumes. We can monitor the CPU utilization and memory consumption directly from `bash`. Let's write a function that monitors the CPU and memory usage of a process given a PID. This function will continue to monitor the process until it either quits or is cancelled. The former behavior is useful for monitoring a model job with an unknown run time.

```
1 monitor_usage() {
2   while $(pid_exists $1)
3   do
4     usage=$(ps ux | sed -e 's/\s\+/,/g' | grep ",$1," | grep -v grep |
          cut -d , -f 3,4)
5     echo "$usage"
6     sleep 1
7   done
8 }
```

This loop tests the return code of the `pid_exists` function, which was defined in Listing 2.2. The loop will continue until the process terminates or Ctrl-C is typed into the terminal. □

Example 2.20. The `getopts` function is used to collect options passed to a script. It takes two arguments: a string that represents all possible options, followed by the name of a variable that holds the current option name. Any option not specified in the argument list will result in an error. Only short options are supported, where each letter represents an option/flag to your script. A colon following a letter indicates that the option is a key-value pair. The value is populated in the temporary variable `$OPTARG`. The while-loop iterates over all arguments given. A case statement tests each option and performs the appropriate action.

Listing 2.4 extends Example 2.19 and creates a complete script to monitor the CPU and memory usage of a process. It's customary to provide documentation on using a script or program. The `-h` and `-?` switches are aliases for printing the script usage syntax. Alternation is used on line 27 to support both flags in a single case. The `do_help` function on line 3 provides the basic syntax for using the script. The actual usage line is printed using a here document [12], which is a special form of input redirection.

Line 22 sets `delta_t` to one second, which is used as the default sleep interval between iterations. Without a sleep interval, the loop will run too fast, potentially affecting the system performance. The `-t` switch enables changing the sleep interval by assigning `delta_t` with the current value of `$OPTARG`. No type checking is performed, so an invalid value will result in an error.

The actual `getopts` call is on line 23. The options listed in the case statement should match those specified in getopts. If an option is specified in `getopts` but not in a case, the case statement will throw an error. In the opposite situation, `getopts` will throw an error. After a `getopts` block, it is customary

```
#!/bin/bash

do_help() {
  cat <<EOT
USAGE: $0 [-t <wait time>] pid
EOT
}

pid_exists() {
  ps ux | sed -e 's/\s\+/,/g' | cut -d , -f 2 | grep "^$1\$" >> /dev/null
}

monitor_usage() {
  while $(pid_exists $1)
  do
    usage=$(ps ux | sed -e 's/\s\+/,/g' | grep ",$1," | grep -v grep | cut -d , -f 3,4)
    echo "$usage"
    sleep $delta_t
  done
}

delta_t=1
while getopts "t:h?" opt
do
  case $opt in
  t) delta_t=$OPTARG;;
  h|?) do_help; exit 0;;
  esac
done
shift $(($OPTIND - 1))
pid=$1

monitor_usage $pid
```

LISTING 2.4: A complete script to monitor CPU usage and memory utilization. The output can easily be redirected into a CSV for analysis.

to `shift` the array of input arguments. Line 30 does just this, wih the effect that the first non-option argument becomes `$1`. □

2.2 The `make` build tool

Modularity and reuse is good for software, but it doesn't come free. Within your code modularity has few negative consequences, but external modularity can complicate building your software. When a program depends on other programs or libraries, those dependencies must be available on every system your code runs on. To guarantee repeatability, you must install the same versions of all software used to build your code. Dependencies themselves have dependencies, so it's turtles all the way down. And that's the best case scenario. Dependencies are usually not linear and look more like a web of interconnections. In perverse situations circular dependencies might prevent your dependencies from ever being met. To stop the insanity, a script or tool is needed to make this process more reliable. Numerous tools solve this problem, and the standard tool in UNIX is `make`. [8]

Two fundamental problems are solved by `make`: understanding application dependencies and automating the process of building a program. For sake of brevity I'll skip most of the dependency management features since they are unnecessary for interpreted languages like R and Python. Besides, both R and Python have their own package managers, so it's usually unnecessary to use `make` to mediate this process. In Section 2.4 we'll see how these requirements can be managed in Docker. Makefiles define independent targets that can be linked together. A target is an identifier followed by `:` and optional targets it depends on. These targets are run first, to ensure dependencies are satisfied. Targets can either model a dependency graph or simply codify commands needed to build and run a program. The statements within a target are indented with a *tab character*. Each statement is a line of shell code that gets executed by `make`. This code is run in a separate environment and will not necessarily inherit variables defined in the current shell.

To build a project, `make` is typically run from the base of the project directory, where a Makefile is assumed to exist. The name of the target to run is given afterward, such as `make all`. If no target is given, `make` runs the first defined target. Makefiles require at least one target. A common target is `all`, [9] which usually lists a set of targets it depends on. Calling `make` without a target thus runs every target necessary to build the software.

Example 2.21. Suppose you implemented a model and created a package

[8] Package managers accomplish the same thing for system dependencies. Programming languages also typically provide a package manager to install packages and their dependencies.

[9] The `all` target is considered "phony" since it doesn't utilize `make`'s compilation features. It can be marked as such, but it isn't necessary since for R all targets are phony.

`bestmodel`. A simple Makefile for the package might contain targets for building the package, testing the package with a dataset, and running the model on new data to generate a report. We can assume a standard process for building, such as with my `crant` tool (see Section 2.6). To test the model, we can use `Rscript` to call a specific function within the package. Generating a report probably requires arbitrary data that we cannot know about in advance. Only the static targets are worth including in `all`.

```
all: build test

build:
  crant

test:
  Rscript -e 'library(bestmodel); run_model("private/test_set.csv")'

run:
  Rscript -e 'library(bestmodel); out <- run_model("$(INPUT)"); run_
      report(out)'
```

□

2.2.1 Variables

While technically Turing complete [36], Makefiles are impractical Universal Turing Machines. That said, a number of useful programming constructs are available, such as variables, conditional expressions, and functions. Variables are assigned using the `:=` operator or the `=` operator. The former operator assigns the variable immediately, where as `=` assigns the variable lazily. [29] In `make` terminology the latter is known as a recursive variable. Any variables referenced within the definition will only be evaluated when the variable is used. This delayed evaluation can be useful when defining variables based on context.

Example 2.22. Suppose you want to distribute a model for use in a runtime system. The model contains a version number dictated by `git`. This version is incremented by `crant` when a build is successful. If the `VERSION` variable is set prematurely it will have the wrong value. Hence, lazy evaluation is appropriate.

```
VERSION = $(git tag | tail -1)
build:
  crant
  tar jcf data model_$(VERSION).tbz2
```

□

When executing `make`, variables can be specified on the command line. Variables can either be defined temporarily in the environment or just for `make`. In the first form, a local environment variable is set first. Here's a Makefile that simply prints a variable `X`:

```
all:
  @echo X=$(X)
```

We can provide this variable via the environment, such as

```
$ X=5 make
X=5
```

The script then has access to this variable. This approach works for any program.

```
$ X=5 sh -c 'echo "X=$X"'
X=5
```

The second syntax calls `make` and sets a local variable within `make`'s environment.

```
$ make X=5
X=5
```

Furthermore, multiple variables can be specified this way. Let's add another variable to the Makefile.

```
all:
  @echo X=$(X), Y=$(Y)
```

And again:

```
$ make X=5 Y=4
X=5, Y=4
```

This approach works for `make` but not necessarily other programs.

```
$ sh X=5 -c 'echo "X=$X"'
sh: 0: Can't open X=5
```

Some variables are only assigned if they don't have a value. The `?=` operator is used for this purpose. In our toy Makefile we can set a default value for `Y`.

```
Y ?= 1
all:
>-@echo X=$(X), Y=$(Y)
```

We can now run the Makefile without specifying `Y`.

```
$ make X=5
X=5, Y=1
```

Example 2.23. In the *diabetes* Makefile, a number of optional variables are defined, including the `PORT`. This variable is used to set up network address translation (NAT) between the host machine and a running Docker container. Since this file is auto-generated, multiple packages will share the same port, causing a conflict. A second instance can run by overriding `PORT` on the command line.

```
$ make PORT=8090 run
```

□

2.2.2 Conditional blocks

Conditional directives can evaluate a block of code depending on whether a variable is defined (`ifdef`) or not (`ifndef`). Both blocks end with `endif`. For example, to support different operating systems, sometimes a slightly different configuration is needed. This behavior can be controlled depending on the existence of a specific environment variable. Another common situation is where behavior differs between two versions of a library. The build process might be different between these two versions. Here, the use of `ifeq` or `ifneq` can test whether a variable is equal to a particular string value or not, respectively.

Example 2.24. When debugging code it's useful to output extra log messages. This can be controlled in the Makefile by checking if a variable `VERBOSE` is defined. If so, we can change the call to docker to turn on debugging, as well as pass additional environment variables to indicate debugging is active.

```
ifdef VERBOSE
  DOCKER := docker -D -e THRESHOLD=DEBUG
else
  DOCKER := docker
endif
```

□

2.2.3 Macros and functions

Variables in `make` are actually macros in disguise. Macros are essentially subroutines that can be referenced later. [29] Unlike simple variable assignment, macros support multiple statements. The `define` directive, defines a macro.

```
TIMESTAMP = $(shell date +%s)
define archive
  mkdir old.$(TIMESTAMP)
  mv *.tar.Z *.tar.gz old.$(TIMESTAMP)
endef
```

Once a macro is defined it can be used like any other variable.

```
all:
  $(archive)
```

Macros essentially define a replacement rule, where the name of the macro is replaced by the body. Arguments can be passed to a macro, making them indistinguishable from functions. Within the body, these arguments can be referenced. Like `bash`, arguments are positional and referenced by `$n`, where $n \in [1, 9]$. To call a function, the same variable dereferencing syntax is used, except it starts with `call`, followed by the name of the macro. Any arguments to the macro come next and are delimited by commas, such as `$(call macro, arg1, arg2)`.

Rather than assuming the directory uses a timestamp as a suffix, let's pass the suffix to the macro as the first argument.

```
TIMESTAMP = $(shell date +%s)
define archive
  mkdir old.$1
  mv *.tar.Z *.tar.gz old.$1
endef
```

The call to the macro must now include the suffix. To preserve the previous functionality, the timestamp can be passed to the macro.

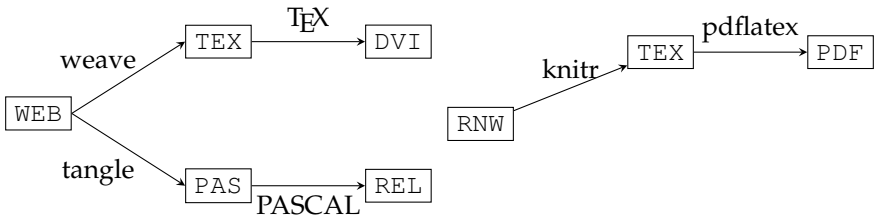
```
$(call archive, $(TIMESTAMP))
```

This approach is a common pattern for refactoring a function to make it more general, while ensuring backwards compatibility.

One final point is in order. Tools are only useful if they are used. Most tools fall into disrepair when not used. The same holds true for Makefiles. To ensure they stay useful, they need to be used and maintained. In a team setting, all members need to use the Makefile to ensure consistent results. For larger projects, one trick is to use the Makefile in an automated build system (see Chapter 5.5). This way it is always in use. Any changes to the build process that aren't reflected in the Makefile will fail, thus prompting an update. Like code, Makefiles will evolve over time to accomodate more situations. Cavalier additions to the file will eventually turn a simple and clean script into a complicated mess. Complex systems are brittle and are difficult to understand, preventing change. When software is difficult to change, it goes from being empowering to being oppressive. To avoid this prison, it's important to allocate at least some time to refactoring (i.e. cleaning up) the file whenever new changes are required. While simplicity and modularity are the keys to removing complexity, without the discipline to refactor, it's hard to maintain a simple system.

2.3 Literate programming and LaTeX

Created by Donald Knuth in 1978, TeX is a language used for typesetting documents. TeX is the cornerstone of literate programming: the idea that software should be written for humans and not machines. Meant to be more than documentation, it is intended to dictate the structure of a program. Literate programs treat exposition and documentation as the primary focus. A special compiler weaves the parts of a program together into an executable. LaTeX, written by Leslie Lamport, extends TeX. It provides numerous conveniences and formatting options for academic and scientific writing. Formal papers and presentations often use LaTeX for typesetting. R package documentation is rendered as a PDF using LaTeX. The older `Sweave` and newer `knitr` allow LaTeXto be used in conjunction with R code outside of package documentation. In theory, literate programs can be written this way, but it is largely impractical. Figure 2.1 shows the difference between the approaches. Donald



(a) The original literate programming produced both a TEXdocument to describe typesetting and a source code file.

(b) Only the `weave` branch is present in the R interpretation of literate programming.

FIGURE 2.1: A comparison of the original vision for literate programming and how it appears in R.

Knuth's original approach for literate programming began with a WEB file that comprised both exposition and code. The `weave` program transformed this file into a TEX source file, which could then be compiled using standard tools to produce a DVI typesetting document. In the other branch, the `tangle` program produced a PASCAL source file, which was compiled to produce an executable binary file. The approach used in R mimics the `weave` path, which is reflected in the precursor to `knitr`, `Sweave`. Instead of producing a DVI, these days a PDF or HTML file is produced.

Three places in the R universe exhibit literate programming concepts: R documentation, papers/reports, and notebooks. The Roxygen2 package enables the mixing of documentation with code to produce standalone documentation files. This process resembles a weave stage, except the source file is simply code and therefore doesn't require a corresponding tangle transformation. Papers and reports can be written in LATEX or Rmarkdown with embedded R code. This time it's exposition that hosts code. Instead of producing a standalone executable, the code generates data to support the exposition. In some ways, notebooks are the most faithful interpretation of literate programming. Both documentation and code live in a single document. The notebook is interactive, so code is evaluated on the fly with results appearing in the notebook.

2.3.1 R documentation

The `Roxygen2` package is inspired by `doxygen`. This framework enables documentation to be embedded in code and rendered to a separate document. Unfortunately, the documentation is not first-class, as is the ideal. It also doesn't support modules in the sense that a function can be broken up into chunks, surrounded by exposition. Despite these limitations, it is valuable to bring together code and documentation. `Roxygen2` processes

all comment lines that begin with `#'`. These lines are read with the code to produce package documentation. `Roxygen2` scans the code following documentation, conveniently filling in required metadata. For usable documentation, standard functions only require a few elements. The first line is treated as the title. An empty documentation line separates a description section. All documentation sections can be specified in the code comments. Function arguments are given using the special tag `@param`. These tags are collected to generate the Usage section. The `@return` tag is used to produce the Value section of the documentation. Similarly, the `@examples` tag produces the Examples section.

When I document a function, I usually begin with this bare minimum of information. Particularly important are the examples, which help others understand how to use a function. I usually start with the examples, even before I've implemented the function. The examples help me work out the design of the function signature. I often change it a few times until I get a nice balance between convenience, flexibility, and functionality.

Example 2.25. Let's create a toy function to simulate rolling a six-sided die. This function simply calls `sample` with some parameters filled in and provides a custom, specific function name. When first writing a function, I typically don't document it thoroughly. The reason is that functions can be in flux for a while, and I don't want to waste time re-writing documentation. Initially, I'll write just enough to remind myself what the function does and how to call it. In Listing 2.5 that means focusing on just a title and an example or two. Over time, the function signature and behavior will stabilize, and I'll add more description and documentation on function arguments. During the operationalization phase, I'll fill in additional documentation necessary for public consumption of a package.

```
#' Simulate rolling a die
#'
#' Roll a fair six-sided die a given number of times. This
#' function gives a probability of 1/6 for each number.
#'
#' @param n The number of times to roll the die
#' @return A vector representing n rolls of the die
#'
#' @examples
#' roll_die(4)
roll_die <- function(n) sample(1:6, n, replace=TRUE)
```

LISTING 2.5: An example of "just enough" documentation to inform people of the purpose of a function and what types of inputs it expects.

While the function is only one line of code, the documentation is much longer. This is often the case and highlights the fact that documentation can often take longer to write than the code itself. Rather than shun documentation as a waste of time, literate programming emphasizes that documentation is the *primary* goal of writing software. □

`Roxygen2` documentation can be generated from R using the command `roxygen2::roxygenize` or `devtools::document`. [1] Alternatively, the documentation can be built using `crant` as part of the package building process. Two options are available: `-x` generates documentation and also builds the package, while `-X` just generates the documentation. The latter option can be useful when you just need to update documentation without making code changes.

2.3.2 Articles, papers, and reports

Papers often need to include data and charts from R. The `knitr` package can produce a working LaTeX file from a file that contains both LaTeX and R code. Both TeX and LaTeX are extremely powerful yet complex tools. To get started with LaTeX, it can be helpful to install a GUI-based editor that provides hints and code completion. Online editors can also be valuable.

The most basic LaTeX document defines a document class and specifies the beginning and end of the document. The document class acts like a template, defining page dimensions, margins, etc. Commands in LaTeX begin with a backslash character. These commands sometimes look like tags, with a starting and ending tag, like `\begin{document}` and `\end{document}`. Others look more like function calls, where the term in curly braces are the parameters to the function. Consider a simple document in Listing 2.6. Line 1 calls `\documentclass` and passes the argument `article` to it. What about the square brackets, such as `[11pt]`? These terms are optional arguments and can be positional or named. Multiple options are separated by commas.

```
1  \documentclass[11pt]{article}
2  \title{An Initial Report}
3  \author{Anonymous Data Scientist}
4
5  \begin{document}
6  \maketitle
7  \section{Introduction}
8
9  \section{Method}
10
11 \section{Conclusion}
12 \end{document}
```

LISTING 2.6: A simple LaTeX document for writing an article.

To create a PDF document, the command `pdflatex` can be used. Assuming Listing 2.6 is in a file `basic_article.tex`, the command

```
$ pdflatex basic_article.tex
This is pdfTeX, Version 3.14159265-2.6-1.40.16 (TeX Live 2015/Debian)
    (preloaded format=pdflatex)
 restricted \write18 enabled.
entering extended mode
(./basic_article.tex
...
Output written on basic_article.pdf (1 page, 39028 bytes).
```

```
Transcript written on basic_article.log.
```

will produce a PDF, along with supporting metadata and a log file. If an error exists, the output will make it clear. Usually the corresponding log file (the transcript) will provide clues regarding the nature of the error.

If you want to embed R code within LaTeX, the file extension `.Rnw` is typically used. [19] An example appears in Listing 2.7. The file is basically the same as the original file but now includes code chunks that `knitr` will evaluate. A code chunk starts using a token surrounded by `<<` and `>>=`. This delimiter accepts options, separated by commas. The initial setup chunk on line 7 includes two such options. Lines 17-19 shows a regular code chunk. The contents will be evaluated by `knitr`. Both the code and the output will replace the code chunk in the resulting TeX output file.

```
1  \documentclass[11pt]{article}
2  \title{An Initial Report}
3  \author{Anonymous Data Scientist}
4
5  \begin{document}
6  \maketitle
7  <<setup, include=FALSE, cache=FALSE>>=
8  library(knitr)
9  set.seed(6)
10 roll_die <- function(n) sample(1:6, n, replace=TRUE)
11 @
12
13 \section{Introduction}
14
15 \section{Method}
16
17 <<die>>=
18 roll_die(5)
19 @
20
21
22 \section{Conclusion}
23 \end{document}
```

LISTING 2.7: An R noweb document that embeds code within TeX.

This file cannot be compiled by `pdflatex`, which is reflected in Figure 2.1. Instead, `knitr:knit` must be run first in R, which will produce a TeX file compatible with `pdflatex`.

As with most sufficiently complex tools, learning LaTeX can be intimidating. A simpler alternative is Markdown, which is popular for web documents. The `rmarkdown` package leverages `knitr` to generate documents in various output formats from a combination of languages. Chapter 11 discusses the use of `rmarkdown` and other aspects of creating reports and visualizations.

2.3.3 Notebooks

A notebook is an interactive document where both exposition and code coexist. Interactive notebooks have found renewed popularity thanks to the Jupyter project. While associated with Python, Jupyter notebooks support R as well. Notebooks are similar to R noweb documents, but both exposition and code are distinct chunks. Unlike Roxygen2 and knitr, using a notebook requires more infrastructure. Section 6.4 shows how to run a notebook from a container, while Section 11.2 discusses some of the benefits and drawbacks of notebooks.

2.4 Containerization and Docker

With a familiar grasp of GNU/Linux, the development environment is almost complete. Modern systems tend to use **container** technology to promote portability and repeatability. Containers also offer isolation, separating the requirements for running a model from the requirements for running your system. Sometimes these requirements are incompatible. For example, prior to publishing a package to CRAN, it should be tested against three versions of R. The system is only designed to have one version running at a time. While it's possible to install three versions of R, it has unintended consequences that hinders repeatability. Other times you may want to test two versions of a model, using different versions of dependencies. Or you may want to run two different models in the same workstation that have conflicting requirements. How do you resolve these issues? Models need to be self-contained and run independent of the host workstation and other models. This isolation is what ensures repeatability. Rather than worry about writing flexible code that works across heterogeneous platforms and environments, containers allow us to define and construct a single reference environment. Others can use an exact replica of the environment we create. This environment is self-contained and includes the operating system, directory structure, and build tools.

These days I use a Docker container for all model development, even if I'm just working locally. Containers can be created and destroyed quickly, so there's little cost in creating isolated computing environments on demand. Proper use of containers nearly guarantees repeatable science, and I always assume that my work will be shared with others. I also always assume that one day my model will reach a production environment. These optimistic assumptions help put me in the mindset of collaboration. The benefit far outweighs the cost, even if this assumption doesn't hold and my model ends up dying in the valley of statistical insignificance. And if my model is destined for the garbage heap, it's likely that I'll cannabalize certain parts of it before

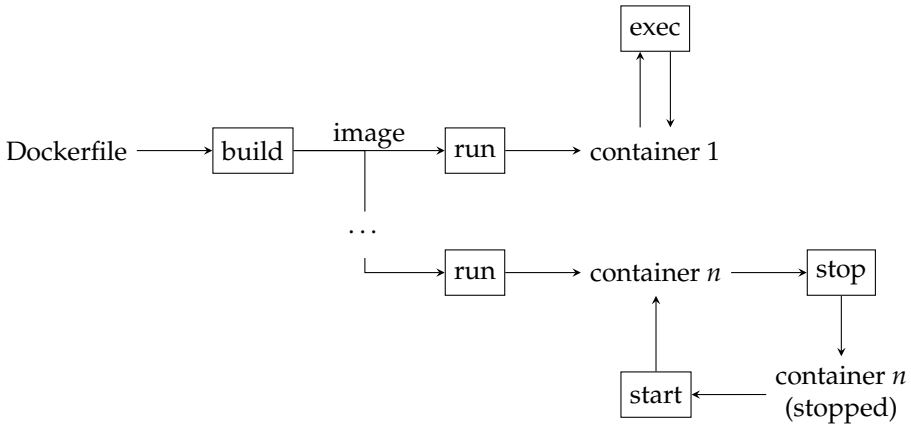


FIGURE 2.2: Relationship of Docker commands and objects. While only one image can be created from a given Dockerfile, multiple containers can be created from the same image.

the model completely rots away. Functional programming principles help easily harvest the good bits from the code. When it is already modular and with minimal dependencies, it's much easier to derive some value out of it regardless of the final state of the project that birthed the code.

Docker can be installed on Linux, Mac OS X, and Windows. For debian users, installation follows the standard approach, using

```
apt-get install docker-engine
```

in a `bash` session. Mac OS X users will want to use the standalone installation package for Docker CE (Community Edition). Download from `https://www.docker.com/docker-mac` and follow the instructions. Windows installation is similar. To test your installation from `bash`, simply run

```
docker run --rm -it zatonovo/r-base R
```

This command will download the `zatonovo/r-base` image and create a running container from the **image**. If you are developing within a container, Docker is the only other dependency required for your host system. Everything else is provided within the image. I created this image to consolidate dependencies for notebooks and OpenCPU, so it can be used during exploration and potentially as a live server. It contains all of the dependencies necessary for model development in R, and is the base image my company uses. [10] One of the benefits of container technology is that images can be derived from other images. This image inherits from the Jupyter project, providing many tools and dependencies transparently.

[10] If you want to use a different image, other public images are available at `https://hub.docker.com`.

Containers are essentially lightweight virtual machines. Rather than emulating all hardware, they reuse some services from the OS. Both virtual machines and container technology rely on templates (images) that define a specific configuration (initial state) of a system. Instances (containers) are created from these templates. One key feature of container technology is how images are defined and shared. Inspired by Makefiles, the Dockerfile defines the exact steps necessary to build an image. Each command in the Dockerfile represents a layer that can be cached and reused. Images are thus a stack of layers. And rather than starting from scratch, an image can be derived from other images. A custom image only needs to define the steps you care about. Many public images are available on Docker Hub. Images exist for virtually every operating system, making it easy to experiement with other platforms.

Building images and running containers all rely on the `docker` command. Figure 2.2 shows how different Docker commands relate to images and containers. Once an image is built, it can be run in a container. Additional commands can be executed on a running container using the `docker exec` command. Like Linux, Docker provides a `ps` command to view all running containers. The `docker stop` command stops a running container. Docker will save this container, which can be `start`ed later.

Container technology promotes repeatability: build an image and you're guaranteed to get an exact replica of the original image. But once a container has run, won't the system state be different from the initial state? While containers can save state, containers are meant to be disposable and stateless. This approach ensures repeatability. What if you do want the system state to evolve? Do you have to rebuild the image every time? If so, this would be awfully inefficient and hinder the development process. One approach is to `commit` changes to the container. This approach works in a pinch but is less repeatable than updating the Dockerfile. If the changes to system state are limited to *data*, a better approach exists: volume mapping. Volumes (i.e. logical disks) can be mapped from the host to the container. A directory on the host therefore appears as a directory in the container. Changes to the container directory are reflected in the mapped folder on the host. Section 2.6 shows how to use containers for model development, while Chapter 6 discusses container technology in more detail.

2.5 R, Python, and interoperability

As a book on R, it may seem surprising to talk about other languages. The reality of data science is that you need to be comfortable with both R and Python and possibly other languages. Sometimes a model may require code from both languages. This is particularly true of deep learning frameworks, which are predominantly written in Python. It's also useful to embrace diversity

R	Python
`NULL`	`None`
`NA`	`numpy.nan`
Vector of length 1	scalar
Vector	`list`
`list`	`list`
`array`	`numpy.ndarray`
`matrix`	`numpy.ndarray`
`data.frame`	`pandas.DataFrame`

TABLE 2.2: Equivalent data structures between Python and R using `reticulate`.

as it is a vehicle for learning. Many of the programming concepts discussed in this book apply equally well to programming in either language. Some projects comprise multiple languages. These tend to be more challenging to automate. One approach is to glue the parts together using a `bash` script. Another approach is to use an interoperability library. Both the `rPython` [11] library and `reticulate` [**?**] enable running Python code from R. The newer `reticulate` library offers more seamless integration with Python than the older `rPython` library. It also provides interoperability in the opposite direction: Python accessing R. Prior to `reticulate`, a separate library, `Rpy2` [**?**] was used for this purpose.

Packages like `reticulate` attach a Python process to a running R process. It manages the conversion of data structures from one language to another. It's important to understand how data structures get converted between the two languages because types are not always compatible, though `reticulate` does a good job of marshalling between the two languages. Table 2.2 shows how types are converted between the two languages.

To use `reticulate`, you just need a Python interpreter discoverable by R. If you have custom packages that you want to use, they need to be accessible via the `PYTHONPATH`. In R, the Python session becomes available when loading the library. Python modules can be imported and any functions defined within those modules can be called. Here's a list of the functions and constants within the Python `os` module.

```
> library(reticulate)
> os <- import("os")
> ls(os)
> head(names(os))
[1] "abc"    "abort"  "access" "altsep" "chdir"  "chmod"
```

Objects have properties and instance methods as defined in Python. In R, both modules and objects are implemented as environments. It makes it easy to inspect objects.

[11] There is also the package `RPython`, which aims to build an R interpreter within Python.

```
> np <- import("numpy")
> x <- np$array(1:4)
> x
[1] 1 2 3 4
> class(x)
[1] "array"
```

Eventually you'll want to transform these objects into their R counterparts. For simple types and objects, you can usually use them as is. For composite objects, like a `pandas DataFrame`, I find it easier to convert subsets or views of an object as opposed to the complete object.

Example 2.26. For natural language processing, the `spacy` library is popular for being fast and feature rich. Let's use it to extract named entities from a sentence. First, create a new Docker container and enter it via `bash`.

```
$ docker run -it --rm zatonovo/r-base bash
```

Now install spacy and download NLP models for English. Then start an R session within the container.

```
$ pip install spacy
$ python -m spacy download en_core_web_sm
$ R
```

Within R, load `reticulate` and import `spacy`. Load the model we downloaded, which is a callable object. We'll use this `nlp` object to parse actual text.

```
> library(reticulate)
> spacy <- import('spacy')
> nlp <- spacy$load('en_core_web_sm')
```

Let's parse some text about Stanford AI professor Li Fei-Fei.

```
> text <- "Fei-Fei is working as Chief Scientist of AI/ML of Google
    Cloud while being on leave from Stanford till the second half of
    2018."
> doc <- nlp(text)
```

We can work with the parsed document like any other R object. For example, let's iterate over all found entities and produce a matrix showing the matched text and the named entity label.

```
> t(sapply(doc$ents, function(x) c(entity=x$text, label=x$label_)))
     entity                    label
[1,] "Fei-Fei"                 "PERSON"
[2,] "AI"                      "GPE"
[3,] "Google Cloud"            "LOC"
[4,] "Stanford"                "ORG"
[5,] "the second half of 2018" "DATE"
```

When you exit from this container, remember that all changes are temporary and will be lost. This includes `spacy` and its English model. To make a permanent change to the image, modify the Dockerfile or create a new one that extends the existing image. □

Embedding a language runtime is convenient for development but not usually suitable in a production environment. The reason is that memory management and resource management in general is not as stable nor reliable as within a single language. Queue technology like RabbitMQ or raw sockets like `zeromq` offer a different strategy for connecting runtimes in two languages. Another approach is to wrap each runtime in a standalone web service, one for each language. These approaches are more involved than reticulate but are more reliable and easier to scale.

Part of the reticulate philosophy is to encourage this weaving of different languages, particularly within Rmarkdown documents. Just because you can write Rmarkdown in R and Python doesn't mean you should. One drawback to this approach is that it's easy to become complacent. Instead of implementing code cleanly, it can be alluring to just do what you know how to do. That may mean crossing language borders repeatedly just because you don't know how to do something in R. The cost of marshalling can be high when data must be regularly converted between the two languages. Not only will it eventually affect performance, it will make testing more difficult. Your code will also be less portable and harder to reuse.

If a library or method is only available in Python, this is a good reason to embed it. Similarly, if a production model is implemented in Python, it may make more sense to use these routines directly in R rather than try to keep two independent implementations synchronized. The R code can be used for interactive data analysis, evaluating new models, and so forth. The key is to maintain proper modularity, so explicit boundaries exist between different parts of the system.

2.6 The `crant` utility

John Chambers, the creator of S, describes statistical analysis as a journey that begins with ad hoc exploration and transitions into structured model development. The S3 and and S4 class systems were designed with this idea in mind: only add structure and process when needed. [10] The reason? Efficiency. Our time is finite and design is an investment. When you don't know if an analysis will bear fruit, why spend time designing a system? My `crant` tool (along with this book) extends this idea to the model development process overall. Focused on R, `crant` provides tools for formalizing the model development workflow once a model looks promising. Rather than focusing on a structured process too early, `crant` tools are designed to be used only when necessary. This saves time and avoids premature structure that can hinder exploration.

Leaning on the UNIX philosophy, `crant` provides a handful of simple utilities to ease model development. These tools can be integrated into custom

scripts to set up Docker images or remote machines. The following scripts come with `crant`:

- `init_package` creates packages,
- `crant` builds them, and
- `rpackage` installs them, while
- `init_shiny` initializes Shiny projects.

The first three tools are described in the following sections. For a more detailed discussion on creating and building packages, see Chapter 5.

To install `crant`, simply clone it from `https://github.com/zatonovo/crant`. Once cloned, you can add the project to your `PATH` environment variable. Then you can issue any of the commands without specifying the full path. Asumming you cloned `crant` into your `~/workspace` directory, you can update your `PATH` with

```
$ export PATH=$PATH:~/workspace/crant
```

This change will only affect your current terminal session. A more permanent approach is to put this in your `~/.profile`, which is loaded for every login `bash` session:

```
$ echo "PATH=$PATH:~/workspace/crant" >> ~/.profile
```

To see the change in your current terminal, you need to `source` the file.

```
$ source ~/.profile
```

2.6.1 Creating projects and packages

The philosophy of `crant` mirrors this book: develop within a custom Docker image. While not required, `init_package` provides configuration files for various tools to streamline development using Docker. The workflow is illustrated in Figure 2.3 .

Project initialization can occur any time. I usually do this once I finish some initial data exploration. At this point I usually have a few functions defined to automate repetitive tasks. From the perspective of Chambers, this happens after the "strict cut-and-paste stage". [10] Functions may encompass data cleaning and normalization. They may also include filtering and some basic feature extraction (e.g. PCA). I may also have some functions that help me debug a model. This is usually the point where the complexity of the code gets in the way of development itself. One common symptom is an environment polluted by temporary variables. Due to the interactive nature of the REPL, these variables may not be in the same state you expect them to. Creating a package helps to clean up this sprawl. The built-in `package.skeleton` can initialize a package based on your code. I find it too finicky for my tastes. Another option is `devtools:create`, but like Goldilocks, I find that it also doesn't suit me. [12]

[12] It's important to stress that these are just my *opinions*. There is nothing wrong with these tools, which benefit scores of people.

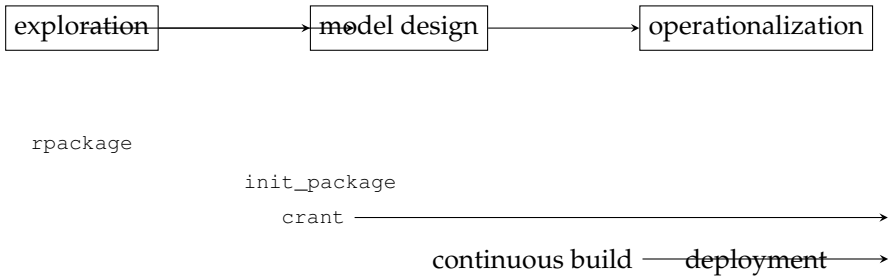


FIGURE 2.3: A portion of the model development process and where different tools are used. The `rpackage` script is used during exploration and model design to install new packages. The `init_package` script is called once, when the model has begun to stabilize. `crant` builds a package and can be used as part of a continuous build process. Deployments occur when a model has been operationalized or packaged for distribution.

I wrote the `init_package` script to be just right. It creates an immediately buildable package with some extra features including a `.gitignore`, a `.Rbuildignore`, a `.travis.yml` build file, a `Dockerfile`, and a `Makefile`. Not only do you get the batteries, you get the full accessory kit for free. These accessories reflect the practical realities of modern model development: source code management (Chapter 4), containers (Chapter 6), and continuous integration (Chapter 5.5). The `init_package` script attempts to be flexible enough to support most data science workflows. This includes working in Python and using notebooks. The `make` target `python` starts a Python 3 interpreter. Libraries can be added using `pip` and its `requirements.txt`. Simply add the line

```
RUN pip3 install -r requirements.txt
```

to the package Dockerfile to install the packages in the image. Similarly, the `notebook` target starts the Jupyter notebook server.

The `.gitignore` tells `git` to ignore specified files, such as project configuration, temporary files, and data. These artifacts should not be committed to a repository, and the generated `.gitignore` ensures they cannot be added. The `.dockerignore` works similarly but for Docker image builds. Data or other large artifacts can slow down the build process, since the data must be copied to the Docker engine. Telling Docker to ignore these files can therefore reduce the wait time. During the build process, R also supports its own ignore file, called `.Rbuildignore`. It works similarly and ensures R does not erroneously include files within the package. The syntax of these files are all similar. File patterns are added one per line. Any matching files will be ignored by the appropriate command.

2.6.2 Building packages

When you are ready to share your work, documentation needs to be generated, tests run, code committed, etc. The built-in command is

```
R CMD build
```

but the syntax can be difficut to remember. I always forget if `CMD` or `build` is capitalized. These inconsistencies chip away at my memory and distract me from my actual goals. It is also just one step in a complete workflow. To simplify this process, the eponymous `crant` tool streamlines this workflow. Without arguments, `crant` will build and run tests like `R CMD build` but with friendlier syntax. A number of options control the specific behavior of `crant`. When I'm in development mode, I use

```
crant -SCi
```

which allows a dirty (uncommitted) repository, skips checks and tests, and installs the package. This is similar to using `devtools:load_all` to emulate building a package, except that it actually builds the package. Another common workflow is to update the documentation based on changes to the Roxygen2 comments. For this task I use

```
crant -xi
```

The `-x` option will build documentation and commit the changes, while `-i` installs the package locally.

Running code as a package is more repeatable and reliable than individual scripts. Unfortunately, this reliability comes at a cost. Developing with packages is less efficient than free-flowing interactive development. The time it takes to build a package can interrupt flow. The `devtools` package addresses this issue by emulating package building using `load_all`. This is faster than a formal build, but also skips some of the safety mechanisms. It's therefore useful during exploration and initial model development, but eventually it's more rigorous to formally build a package.

2.6.3 Developing multiple packages

For large projects, multiple R packages might be necessary. For my work, I tend to split packages based on functionality (e.g. `futile.logger` versus `lambda.r`) and also license (open source versus proprietary). Each package needs to be built separately. In an R session, packages can be indiviually removed and reloaded. The caveat is that a package can only be removed if no other loaded package in the session depends on it. If so, those must be removed first. The `detach` function removes a package from the current R session. In order to load a new version, it must also be unloaded. Otherwise, the previous version will remain cached and be used. For example, the diabetes package can be unloaded using

```
detach('package:diabetes', unload=TRUE)
```

```
.reload <- function() {
  packages <- c('package.c', 'package.b', 'package.a')
  sapply(packages,
    function(p) detach(paste('package',p,sep=':'), unload=TRUE)
  sapply(packages[length(packages):1], library)
```

LISTING 2.8: A private function to reload a package and its dependencies.

Custom model packages shouldn't be too deep in the dependency graph, so it should be relatively painless to reload a package. Even so, I find it useful to write a private package function that can reload packages for me, such as `.reload`. By default, functions starting with a `.` are not publicly visible in the package, so this function won't be exported. A hypothetical example is shown in Listing 2.8. In this listing, `package.c` is the current package. It depends on `package.b`, which itself depends on `package.a`. To reload a change in `package.b` requires unloading `package.c` first. For convenience, the function can unload all local packages in one go. Once unloaded, the updated packages can be re-loaded. The loading process works in the reverse order of the unloading process. In other words, `package.a` is loaded first, then `package.b`, and finally `package.c`. When the function returns, all local packages will reflect changes from a command line build/install. This approach ensures consistent behavior.

2.6.4 Installing packages

It's easy to install packages from an R session. It's less easy to do it from the command line. While it's possible to execute an ad hoc one-liner

```
Rscript -e 'install.packages()'
```

or

```
Rscript -e 'library(devtools); install_github("user/package")'
```

these approaches are inconvenient to type. The `rpackage` utility solves this by providing a simple command line interface to install both packages published on CRAN and those available via `git`. For example, it's possible to install multiple packages at once, such as my functional programming language extension, functional programming tools, and my logging package.

```
rpackage lambda.r lambda.tools https://github.com/zatonovo/futile.logger.git
```

In this sense, it works similar to a package manager that can install more than one package at a time. However, `rpackage` is dumb, and doesn't manage dependencies. It's up to you to list packages in the correct order.

When does it make sense to use packages on CRAN versus Github? Versions of packages on CRAN are typically more stable than versions on Github. CRAN enforces rigorous rules to ensure software available on CRAN is reliable and well documented. Stable software is also older, so versions on CRAN

generally don't include the latest features. If you want newer features, install Github versions over CRAN versions. Another important difference is that a version on Github can change arbitrarily. These changes may break your code, and there will be little advance warning if this happens.

2.7 Exercises

Exercise 2.1. Use `bash` commands to print each component of the `PATH` on a separate line. For example, given the path

```
/home/brian/bin:/home/brian/.local/bin:/usr/local/sbin:/usr/local/bin:/
usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin:/hom
e/brian/workspace/crant
```

use bash to produce the following output:

```
/home/brian/bin
/home/brian/.local/bin
/usr/local/sbin
/usr/local/bin
/usr/sbin
/usr/bin
/sbin
/bin
/usr/games
/usr/local/games
/snap/bin
/home/brian/workspace/crant
```

Hint: use `tr`.

Exercise 2.2. Write a `bash` script to generate the `DESCRIPTION` file for an R package. Use Example 2.13 as a guide. What command line arguments does your script accept?

Exercise 2.3. In the following `bash` code, explain why the value of `xy_test` is empty instead of 0.

```
$ xy_test=$(test $x -eq 5 -o $y -eq 3)
$ echo $xy_test
```

Exercise 2.4. Listing 2.1 gave a script that counts from one. Modify the script to accept an optional argument for the starting number. Show an example of the script running. Does it make sense to use `getopts` to manage the optional argument? Why or why not?

Exercise 2.5. In Example 2.17, the expected value of a die was computed in `bash`. The approach is limited to integers and is not useful. To rectify this, the `bc` program can be used. For example, a fraction can be computed to two decimal places with

```
echo "scale=2; 39/10" | bc
```

Modify the code listing in Example 2.17 to compute the expected value using `bc`.

Exercise 2.6. Create a script that runs indefinitely. Start the script in the foreground. Disown the script and close the window. Open another shell and verify the process is still running. Show your script, the commands you ran, and the output that shows the process is still running.

Exercise 2.7. Use the `kill` command to stop the process you started in Exercise 2.6.

Exercise 2.8. Use Docker to run the `r-base` public image. What operating system is this image using? Show the commands you used to determine this.

Exercise 2.9. By default `crant` requires a clean working directory (all changes committed in `git`). This can be disabled with the `-S` switch. Why do you think `crant` requires a clean working directory before building the package?

3

Project conventions

> No important institution is ever merely what the law makes it. It accumulates about itself traditions, conventions, ways of behaviour, which are not less formidable in their influence.
>
> *Harold Laski*

Repetition in everyday life can become a mind-numbing sludge. It's also the grease that makes machines efficient and processes automated. A repeatable process is the foundation of repeatable science. To keep things structured and predictable, repeatable processes rely on various conventions. Little details like the directory structure, file names, function names, variable names, and how to format source code are shaped by the conventions we use. Conventions even appear in the architecture of a program, dictating how modules fit together and interact. Without conventions, code loses its structure and its predictability. More time is spent figuring out what things are and where they go instead of being productive. This chapter recommends some conventions, many of which come from R itself. Using the same conventions as R limits surprises because the conventions are consistent. Maintaining consistency extends to function design, particularly in the choice of default values of optional arguments. Known as **sensible defaults**, default behaviors and conventions shouldn't surprise the end user.

In programming circles, there is a software design philosophy known as "convention over configuration". [17] This approach tries to limit the number of decisions that must be made by a developer to use a function/library/framework. Rather than offering a lot of configuration options, certain conventions are used to simplify both the system and its configuration. These act as default choices that can be overridden as needed. But sometimes these conventions are so ingrained in software that changing them can prevent the software from working! It's usually possible to stray from conventions without such dire consequences, but things can get difficult quickly.

How R tests a package is an example of convention over configuration. During the build process, R runs any scripts in the `tests` directory, assuming they are tests. Any number of testing packages can be used for testing. A convention is to name the main testing script `test-all.R`, creating a subdirectory

named after the test package used. Hence, if using `testthat`, there would be a directory `tests/testthat`, while if using `testit`, the directory would be named `tests/testit`. This is just a convention, though, and can be named anything.

In some cases, conventions can be overridden by a configuration. This depends on the software designer and whether she chose to expose a particular parameter as configurable. In this case, the convention becomes the *default value*. The effectiveness of this approach is dictated by how sensible i.e. probable the defaults are. An example in R is *ggplot2*, which uses many defaults to set colors, legend placement, etc. My logging package *futile.logger* also uses many sensible defaults to simplify usage of the package. I provide sensible defaults that correspond to a common use case, so users don't have to worry about configuring every aspect of the logging system just to use the package. Contrast this approach to similar logging packages in Python and Java that require many lines of configuration code just to print a message to the console!

The folllowing sections describe different aspects of software development that tend to have conventions. Some are conventions dictated by software while others are dictated by best practices. I suggest following these conventions until you've mastered development and are comfortable creating your own conventions.

3.1 Directory structure

New projects should start in their own directory within your filesystem. I generally put all my projects in a folder called `workspace`, which is a leftover from an old **integrated development environment (IDE)**. I don't use IDEs any more, and this book is written from this perspective. It may seem arcane, but disavowing IDEs will ultimately increase your productivity. One reason for this ascetic approach is to limit complexity by eliminating unnecessary dependencies. Another reason is that the conveniences provided by IDEs are training wheels that eventually become a wheelchair. While initially useful to get you to your destination, overreliance on IDEs transforms into dependence. When this happens, it becomes difficult to be productive without the IDE, creating vendor lock-in and difficulty adapting to other work environments. Software written without an IDE is more portable and can be shared more readily with different audiences.

IDEs generally introduce their own workflow for development. If this is the only workflow you support when sharing your work, it is a big ask of others to change how they work. Minimizing dependencies makes it easier for others to review or adopt your work, since you aren't imposing your

Path	Description
~/workspace/diabetes	Project home
~/workspace/diabetes/data	Package data
~/workspace/diabetes/inst	Other package files
~/workspace/diabetes/man	Package documentation
~/workspace/diabetes/notebooks	Jupyter notebook files
~/workspace/diabetes/private	Non-package data
~/workspace/diabetes/R	R source code
~/workspace/diabetes/tests	Test scripts

TABLE 3.1: Directories within an R project

workflow on them. [1] Whether you choose to use an IDE is your choice, though you may need to adjust some of the conventions in the book based on the conventions your IDE imposes.

The directory structure conventions I use are dictated by the package building process. Doing so minimizes friction later and doesn't cost me anything upfront aside from educating myself about how packages in R are structured. A complete directory structure appears in Table 3.1. The `private` and `notebooks` directories are my personal conventions and are not required for building packages. The `tests`, `data`, and `inst` directories are used by R but are also optional.

Example 3.1. Let's create a new project for the diabetes analysis. The project home directory is `diabetes` and located at `~/workspace/diabetes`. This directory needs to be created in your host operating system. We expect to add functions in `.R` files, so we'll create the `R` subdirectory as well. We can do this in one shot with the following command in `bash`:

```
$ mkdir -p ~/workspace/diabetes/R
$ cd ~/workspace/diabetes
```

Next, download and extract the diabetes data into `private`.

```
$ mkdir private
$ cd private
$ curl -O https://archive.ics.uci.edu/ml/machine-learning-databases/
    diabetes/diabetes-data.tar.Z
$ tar zxf diabetes-data.tar.Z
$ mv Diabetes-Data diabetes
```

It's easy to execute some commands, download your data, and get on with your life. But what about reproducibility? How will someone else acquire the same dataset? It's best to codify this process in a script, leaving a trail of breadcrumbs for others to follow. Not only is it easier for someone else to

[1] Of course, one can argue that I'm merely imposing a new workflow to replace the IDE workflow. This is true, up to a point. However, the workflow I introduce minimizes new dependencies and is designed for freedom and portability, as opposed to dependence and lock-in.

reproduce your work, it documents the process in case you forget. A simple approach is to add these commands into a Makefile.

```
r:
	docker run $(VARS) -it $(XVARS) -u jovyan $(REPOSITORY) R

notebook: all
```

The data can now be downloaded at any time by anyone by simply running `make data`. □

When we convert source code into a formal package, the `private` directory will be ignored by R and `git`. It's important to segregate data from your source code. Ownership and licenses are typically different for data and source code, so it's good practice to keep them separate. Even if you collected and compiled the data yourself, you probably want to adhere to this practice. The reason is that the nature of data differs from the nature of code. The size of data files is usually much larger than source code. Massive repositories take a long time to download, update, and change. This can lower adoption of a package or model, simply because it's inconvenient to download a bloated repository. Data also changes differently from source code, making it harder to track changes in either the data or the code. Some users may not even care about your data. They may be interested in using your code for a completely different dataset, making the included data superfluous.

The drawback to using separate repos for source code and data are that it's less convenient. Now two repositories need to be downloaded and maintained. This inconvenience can be scripted away in the Makefile, though. The script can download data directly into the `private` directory, which is ignored by `git`. By using the Makefile we get an effective compromise between convenience and correctness. [2]

3.2 Creating R files

During the exploratory phase, it's easy to spend all your time in a notebook or interactive session. The allure of immediate feedback is strong, so much that it's easy to lose yourself in the rapture of convenience. Too much freewheeling accumulates what's known as **technical debt**. And like Shylock coming to redeem his pound of flesh, there's never a good time to repay this debt. Better to prevent it from accumulating in the first place. An effective first step is to encapsulate algorithms in functions within `.R` source files. The interactive sessions are thus limited to calling functions instead of implementing algorithms. This simple approach makes your work more repeatable. Notebooks

[2] It's also possible to use `git` submodules, but they introduce their own complexities.

are also easier to follow when procedural code is kept to a minimum. Even if your model isn't intended to go into production, writing scripts is more permanent than a transient session. Crashes due to memory can result in unsaved history. It's easy to rely on the history within an interactive session, but one accidental printing of a large variable can wipe out the scrollback buffer of your terminal. Accidental shutdowns of a session can also cause unnecessary stress. Scripts protect against these dangers. Furthermore, code in source files is first-class and can be unit tested and tracked in a version control system.

Example 3.2. Let's start by creating a function to read the diabetes data, as shown in Listing 3.1. This dataset comprises multiple files, one per patient. To produce a single data frame, all the files need to be merged together.

```
1  read_diabetes <- function(bad=c(2,27,29,40), base='private/diabetes'){
2    read_one <- function(i) {
3      path <- sprintf('%s/data-%02i',base,i)
4      flog.info("Loading file %s", path)
5      classes <- c('character','character','numeric','numeric')
6      o <- read.delim(path, header=FALSE, colClasses=classes)
7      colnames(o) <- c('date','time','feature','value')
8      o$date <- as.Date(o$date, format='%m-%d-%Y')
9      o$id <- i
10     o
11   }
12   idx <- 1:70
13   do.call(rbind, lapply(idx[-bad], read_one))
14 }
```

LISTING 3.1: A function that concatenates individual patient data into a single data frame.

The first thing this function does is define a **closure** on line 2 that is responsible for reading a single file from the filesystem. It takes an index, constructs a path (line 3) and reads the file (line 6). The data frame `o` is then cleaned up and returned. Line 13 iterates over a sequence of indices via `lapply`, less some indices corresponding to files with bad data. The output of `lapply` is a list of data frames, each one produced by the closure `read_one`. Finally, `do.call` is a special function that dynamically calls its first operand with the second operand as arguments. In this case, `rbind` is called with the list of data frames returned by `lapply`. The end result is a single dataframe containing the events from all patients.

Note that `read_diabetes` depends on *futile.logger*. Be sure to add `require('futile.logger')` to the top of your file. □

How should R source files be organized and how many files should be created? Let's answer this question with another question: in which file should `read_diabetes` of Listing 3.2 go? A good heuristic is to group functions in files based on their role. Grouping similar functions together is akin to modules in

Python, where functions related to a particular task live in the same module. Functions responsible for reading in data can go in a file called `read.R`. A second file called `explore.R` can codify functions you write for data exploration. Grouping functions based on their purpose also answers the question of how many files to create. Initially, though, the lower bound of one source file is fine. When you have collaborators, it becomes inconvenient if everyone is modifying the same file simultaneously. This is a good time to start grouping functions by role or context and create different source files for each. Why not use the upper bound of one file per function? Nothing is theoretically wrong with this approach. However, with large systems it is inconvenient having functions spread across so many files.

Example 3.3. Let's explore the *diabetes* dataset. The data are organized as key-value pairs, such as

```
> head(df)
        date  time feature value id
1 1991-04-21  9:09      58   100  1
2 1991-04-21  9:09      33     9  1
3 1991-04-21  9:09      34    13  1
4 1991-04-21 17:08      62   119  1
5 1991-04-21 17:08      33     7  1
6 1991-04-21 22:51      48   123  1
```

The meaning of each feature is given in the `Data-Codes` file. For example, feature 58 corresponds to "Pre-breakfast blood glucose measurement" and feature 33 is "Regular insulin dose". No units are given for the values. The `id` is not part of the original dataset and is added by the `read_diabetes` function to distinguish readings from one patient to another.

The data for each patient varies greatly. We need to get an understanding of which features have enough data for an analysis. A quick way to determine the counts of each feature is to create a contingency table. Listing 3.2 shows how to do this. The `get_event_table` function should be defined in `explore.R`.

```
1 get_event_table <- function(df) {
2   table(df$id, df$feature)
3 }
```

LISTING 3.2: Create a contingency table of events within the data.

The body of this function is just one line. Do we really need a function for this? One purpose of a function to provide context via their name. If we just use `table(df$id, df$feature)` in an interactive session or in a larger function, we may forget what the purpose of this line is. Wrapping it in a function provides more information that helps your recall. Code is also a living entity. It evolves over time. A one line function may become multiple lines in the near future. When code is encapsulated in a function, it's easy to modify the code since it's only in a single place. The alternative is searching all instances of a particular line of code and changing each one. It only takes a few times of this before you'll want to write functions for everything!

Another example of a one line function is in Listing 3.3. If you weren't convinced that you may forget the purpose of a line of code, this one may prompt you to reconsider. Even if you are adept at reading code, all you learn is *what* a function does. To understand the *why*, you usually need more context. A function name is a simple device to achieve that.

```
get_patient_duration <- function(df) {
  tapply(df$date, df$id, function(date) max(date) - min(date))
}
```

LISTING 3.3: Get the duration of each patient's participation in the study.

So what does this function do? We already know that the count of features in the dataset vary quite a bit. Since each patient is different, we can't make assumptions about how many readings each patient has. For each patient, the `get_patient_duration` function computes the duration of the study. Initially, we'll assume that data exists for each day, but that will need to be verified as well. □

Creating multiple `.R` files is good practice. What do you do if you change multiple files and need to reload them? It gets annoying having to `source` each file individually. If you're not ready to create a package, a simple trick is to create a separate file that is responsible for `source`ing all the other files. I generally call this file `.init.R` and only keep it around locally. To load all your `.R` files now just requires a single call: `source("R/.init.R")`. This temporary file is only needed until you convert your project into a formal package, which is discussed in Chapter 5.

3.3 Working with dependencies

Good programmers are known for being "lazy" because they minimize the work to be done. For example, reusing a function is an example of being lazy, since you aren't rewriting the same code over and over. This concept is known as the Don't Repeat Yourself (DRY) principle. Taken to the extreme, DRY dictates that you should never implement something that has already been implemented. Unchecked, this philosophy can lead to massive amounts of dependencies in your code that ultimately add complexity to software. Writing software is ultimately a multi-objective optimization problem, where DRY is but one objective. It needs to be balanced with simplicity and ease-of-use to be truly effective.

Too many dependencies can be counterproductive, increasing build times and the possibility of unforeseen dependency issues, like conflicts or breaking changes. For fairly trivial operations, I tend to re-implement a function in lieu of adding a dependency. For example, I rarely use a library to calculate

precision, recall, or my confusion matrix. These are only a few lines of code, and I benefit from knowing exactly how it is implemented. In short, to decide whether you should include a dependency or implement something yourself, optimize on both DRY and simplicity.

Package dependencies are listed in the `DESCRIPTION` file. The convention is to list dependencies under the `Imports` section and not the `Depends` section. Imports do not expose the package's dependencies to the user of the package. They used to be in older versions of R, and this caused namespace collisions. Any symbols defined by a dependency could potentially overwrite a variable defined in your workspace. The `Imports` section solves this issue by only importing public symbols defined by the package in the `NAMESPACE` file. Within this file, a similar convention exists: only export functions that end-users care about. When users have access to all of the functions in your package, it creates additional dependencies that are difficult to remove later. By only exposing public facades (the API), as a package developer, you preserve flexibility in changing the implementation behind the interface without affecting users.

3.4 Documenting your work

Mastering programming is not just about writing code. It's also necessary to *read* code, whether your own or someone else's. Code literacy is directly proportional to how quickly you can understand how an algorithm works. This is necessary not just for debugging, but for evaluating the correctness of a peer's model and also for reusing someone else's code.

I discussed documentation in relation to literate programming in Section 2.3. In addition to documenting the code itself, it is helpful to provide high-level package documentation. This documentation describes the purpose of the package, how to install it, and how to use it. Some of this may appear in the R package documentation or in a vignette, though a more common location is a `README`-like file in the project root. R looks for either a plain text `README` or a `README.md`. If the latter, it will use `pandoc` to parse and generate corresponding HTML documentation for the package.

One benefit of using a `README.md` is that it is compatible with Github and will be rendered automatically there as well. If your source code is hosted on Github and you are using Travis CI (see Chapter 5.5), another convention is to add the build status to the `README.md` as an image that updates based on the build status as determined by Travis. This is a convenient way for users to know the current state of your development code and whether it is stable.

4

Source code management

> There's an old saying about those who forget history. I don't remember it, but it's good.
>
> *Stephen Colbert*

Depending on your disposition, accidents are either something to be avoided or a chance for serendipity. Accidents with source code are usually something to be avoided and relate to unintentional deletion or replacement of code. It can be painful trying to recover or recreate code previously written. To avoid this frustration, it's wise to use a source code repository, or *repo* for short. **Source code management** tools provide such a repository for tracking changes to source code. They typically live in the cloud, on a server with appropriate redundancy to protecte against failure and deletion. These days, most people use `git` as their SCM of choice. Written by the creator of Linux, `git` is orders of magnitude more powerful than older SCMs. The drawback is that it's also more difficult to learn. This section covers a few core concepts to get you familiar with `git`. For a more thorough discussion, see [9].

`Git` is based on a model of distributed development, which is how Linux is developed. Each local repository is a complete repository with a full history. Users either **push** changes from their local repository to another repository or **pull** changes from another repository into their local repository. Unlike other SCMs, `git` has no concept of a master repository. In a distributed model, no repository is a single point of "truth". The closest thing to truth is the release manager's repository, which will have a record of published releases. Many people would rather have a central repository, even if it's unnecessary. A central repository acts as the single point of truth that everyone can reference. Web-based repositories like Github, GitLab, and BitBucket are used for this purpose. This remote, central repository is typically called *origin*. Figure 4.1 illustrates the relationship between multiple repositories. In this model, you make changes to your local repository. You **commit** these changes locally. When you want to share your work with others, you push your repository to another repository, usually origin. Other users then pull your changes from the central repository. If no central repository is used, only a push or a pull is needed to synchronize two repositories.

All commands in `git` start with `git`. The command is next, followed by optional arguments. For example, to initialize a repository you use the

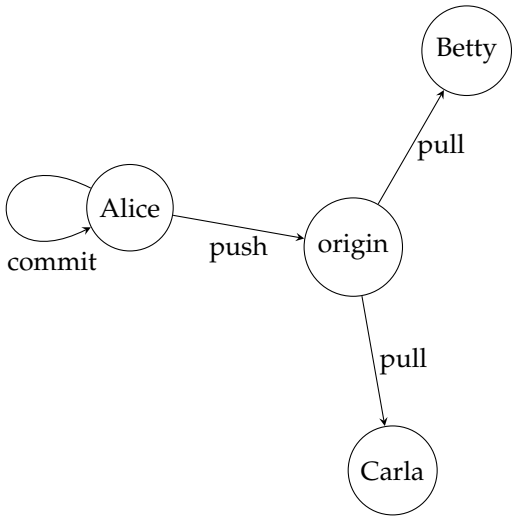


FIGURE 4.1: `Git` is a distributed source code management system. Each circle represents a complete repository. The origin is the central repository that acts as truth. While a central repository is optional, it simplifies coordination. For example, Alice commits changes to her local repository. She then pushes her changes to origin. Betty and Carla pull from origin to retrieve her changes.

command `git init`. Many commands do not have required arguments. Other commands, like `git push` need to know the repository and branch to push to. Table 4.1 summarizes some useful `git` commands.

Example 4.1. We created a few `.R` files for the diabetes example in Chapter 3. This is a good time to create a repository for the project.

```
$ cd ~/workspace/diabetes
$ git init
Initialized empty \tool{git} repository in /home/brian/workspace/diabetes/.git/
```

This command creates a hidden directory within your project called `.git`. This is the actual repository that tracks all the commits and branches. Very rarely will you need to interact with this folder directly. In fact, it's best to leave it alone; just be sure not to delete it!

Back to our project. You can now check the status of your newly created repository. There won't be anything in it, and `git` will tell you what is not tracked in the repository.

```
$ git status
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)
```

Command	Purpose
`git init`	Initialize a local repository
`git status`	Check status of current repository
`git add`	Add a file or directory to the repository
`git rm`	Remove a file or directory to the repository
`git commit`	Commit changes to the repository
`git tag`	Label a commit with a name
`git log`	View a history of commits for a file or the repository
`git blame`	View last person to commit each line within a file
`git diff`	Compare two versions of the repository
`git branch`	Show current branch
`git checkout`	Checkout a specific commit or branch
`git rebase`	Apply one branch on another
`git merge`	Merge two branches together
`git fetch`	Retrieve changes from a remote repository
`git pull`	Retrieve and merge changes from a remote repository
`git push`	Push local changes to a remote repository

TABLE 4.1: Common `git` commands and their use.

```
        R/
        private/

nothing added to commit but untracked files present (use "git add" to
    track)
```

The next step will be to add files to the repository so changes can be tracked. Version management is the essence of a source repository and is discussed in the next section. □

4.1 Version management

The primary benefit of repositories is that all changes to code are tracked. That means you can view any version of your code in time whenever you want. The unit of change in a repository is the commit. Commits can represent changes in multiple files. They can be as small or as large as you want. Just remember that commits or atomic – you can't divide them into smaller commits. [1] This is most important within a single file. If you make multiple changes to a file in the same commit, you can't retrieve intermediate changes later on.

[1] You can, however, merge multiple commits into a single commit. See `git squash` for examples.

Only files tracked by the repository can be committed. The first step is to `git add` the files you want to track. You can add specific files or complete directories. If you specify `git add .` all files and folders in the current working directory will be recursively added to the repository. I generally don't do this until I have added a `.gitignore` to tell `git` what files I don't want tracked. Otherwise I may end up with a lot of garbage in my repository, which isn't a good way to start.

Once files are tracked by `git`, `git commit` will commit the current changes to the repository. Like adding files, a commit can operate on specific files, directories, or the repository as a whole. For this operation I usually use the `-a` switch to commit all changes. When would you not want to commit all changes at once? Suppose you already have changes in your local repository. Then a collaborator asks you to make a change for her. You make the change and want to share it, but you don't want to share all your changes. In this case you might just commit the change for your collaborator, push that, and then carry on with your work.

Example 4.2. You can commit single files or all changes to the repository in a single commit. I tend to do the latter (but only by first confirming changes with `git status`) by using the `-a` switch. The `-m` switch tells `git` the message to use for the commit. This message is what you see in the `git log`, so it should be something descriptive to jog your memory. If you don't provide the `-m` switch, `git` will open up a text editor and ask you to provide a message.

```
$ git add R
$ $ git commit -am "Initial commit"
[master (root-commit) c4f2976] Initial commit
 2 files changed, 81 insertions(+)
 create mode 100644 R/explore.R
 create mode 100644 R/read.R
```

You can verify the commit by viewing the log. Notice that the message I used in the commit appears in the log output.

```
$ git log
commit c4f2976eb94093322a01dc6f32391ff15fb0f35a
Author: Brian Lee Yung Rowe <rowe@zatonovo.com>
Date:   Wed Jun 6 13:58:40 2018 -0400

    Initial commit
```

The first line of the output shows the commit hash. This hash uniquely identifies the commit. □

To view the previous state of a repository, use `git checkout`. This command takes either a commit hash (see Example 4.2), a tag, or a branch. Wherever you are in the commit graph is called `HEAD`. Usually the `HEAD` points to the latest commit in a branch. However, if you check out a specific, older commit, `HEAD` will move there leaving you in a so-called "detached HEAD" state. To revert to a normal state, just check out master again.

Example 4.3. Our commit history is pretty small, so it's difficult to see how `HEAD` works. Let's add some more functions and add them to our repository. Add the function `filter_events` to `explore.R`. This function removes unused features from the data frame produced by `read_diabetes`. It's easier to work with the dataset when we remove superfluous data. The features in `keep` are those that I determined have sufficient data for an analysis.

```
filter_events <- function(df) {
  keep <- c(33,34,58,60,62)
  df[df$feature %in% keep,]
}
```

LISTING 4.1: Remove unused features from the diabetes data frame.

Before adding the next function, let's commit this change to the repository.

```
$ git commit -am "Add filter_events"
[master 04e6ec5] Add filter_events
 1 file changed, 11 insertions(+)
```

This is strictly for pedagogical purposes, so you have a slightly larger commit history to work with. Normally, commits don't need to be this small.

Now let's add a function to analyze the time events were recorded for a given feature. Eventually, we'll want to normalize this data. A first step is just seeing what's there.

```
get_event_times <- function(df, feature, id=NULL) {
  if (is.null(id))
    df$time[df$feature==feature]
  else
    df$time[df$feature==feature & df$id==id]
}
```

LISTING 4.2: Get the time of each event of a given feature.

Let's commit this change to the repository as well.

```
$ git commit -am "Add get_event_times"
[master f40b865] Add get_event_times
 1 file changed, 12 insertions(+)
```

This gives us three commits in our history to work with. We'll use this nascent commit history throughout this chapter. □

What exactly are commits? Put simply, they are a set of changes, called **diffs**, from the previous commit. These changes are produced by the Unix utility, `diff`, which was originally written in 1974. [18] Akin to a line-oriented Levenshtein distance, `diff` shows the additions and deletions required to transform one file into another. This record of changes is used by `patch` to actually transform one file into another. Storing just the diffs is more space-efficient than storing complete files for every version. The differences between two commits can be viewed with `git diff`. As with `git checkout`, any reference to a commit can be used. The syntax is

```
git diff commit1..commit2
```

where the output indicates edits to transform `commit1` into `commit2`. If only one commit is referenced, it is assumed to be `commit1`. The other commit is assumed to be `HEAD`.

Example 4.4. Let's compare the differences between two commits. How do we get the hash of a previous commit? Calling `git log` will show us a history that provides the hashes. For a more compact view that also shows branches, we can display the history as a graph. [2]

```
$ git log --oneline --decorate --graph --all
* f40b865 (HEAD -> master) Add get_event_times
* 04e6ec5 Add filter_events
* c4f2976 Initial commit
```

A portion of each commit hash is shown followed by the commit comment. `Git` also shows us where `HEAD` is, as well as our master branch.

The diff between `HEAD` and the previous commit is obtained with

```
$ git diff 04e6ec5
diff --git a/R/explore.R b/R/explore.R
index 6e4f765..f76ddcb 100644
--- a/R/explore.R
+++ b/R/explore.R
@@ -40,3 +40,15 @@ filter_events <- function(df) {
 }


+#' table(get_event_times(df, 58))
+get_event_times <- function(df, feature, id=NULL) {
+  if (is.null(id))
+    df$time[df$feature==feature]
+  else
+    df$time[df$feature==feature & df$id==id]
+}
+
```

The `+` indicates that these lines were added to commit `04e6ec5`. Similarly, lines starting with `-` indicate a deletion. □

4.2 Branches

Suppose you have a model running somewhere. You're working on a new analysis, and then you find out your current model is broken. How do you make a fix and deploy it without introducing your new, incomplete changes? Branches solve this problem. A branch is simply a pointer to a specific commit. Effectively though, branches represent a whole chain of commits starting from

[2] If you prefer a more interactive UI, an alternative is `gitk`.

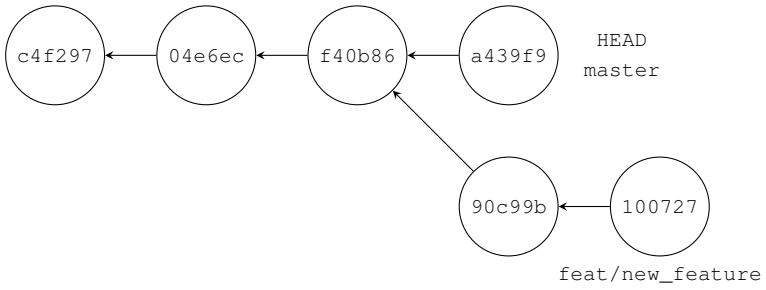


FIGURE 4.2: An example commit tree. Arrows point in the direction of the parent commit. Commits are appended to `HEAD`, which corresponds to a branch. Switching branches changes the location of `HEAD`, which changes where commits are appended.

the first commit. When a repository is initialized, there is just one branch. By default, this branch is called *master*. New branches are created using the command

```
git checkout -b <branch name>
```

Upon creation, this branch points to the same commit as the current branch. Once commits are added, though, the commit chain will be distinct from the commit chains in master. Commits will be added wherever `HEAD` is located. So switching branches requires calling `git checkout` with a different branch name. Figure 4.2 shows an example of a branch that departs from master. Both `master` and `feat/new_feature` have different commits following the commit at `f40b86`.

You can name a branch anything you want. For predictability, you may want to follow some conventions. Here are a few that I use. If the branch is personal, usually it is prefixed with your name or initials. Suppose I create a new branch to work with more features. I would call this branch `br/new_feature`. If I plan on collaborating with someone on this branch, prefixing with my initials isn't appropriate. In this case the prefix `feat` indicates that the branch is a feature branch. You can define your own conventions if you choose. The key is to be consistent in applying your conventions so others know what to expect.

Example 4.5. Let's add a branch to our diabetes project. We'll pretend that this branch is for collaboration.

```
$ git checkout -b feat/new_feature
```

With this command, `git` has created a new branch and pointed `HEAD` to it. We can verify this by viewing the log.

```
$ git log --oneline --decorate --graph --all
* f40b865 (HEAD -> feat/new_feature, master) Add get_event_times
* 04e6ec5 Add filter_events
```

```
* c4f2976 Initial commit
```

Any new commits will be along the `feat/new_feature` branch. To confirm this behavior, add a function stub to `explore.R`.

```
normalize_times <- function(df) {
  # Implement this later
}
```

Save and commit your changes to the repository. Viewing the log will show how the new commit is associated with `feat/new_feature` and not `master`.

```
$ git commit -am "Add stub for normalize_times"
[feat/new_feature 90c99b1] Add stub for normalize_times
 1 file changed, 2 insertions(+), 1 deletion(-)
$ git log --oneline --decorate --graph --all
* 90c99b1 (HEAD -> feat/new_feature) Add stub for normalize_times
* f40b865 (master) Add get_event_times
* 04e6ec5 Add filter_events
* c4f2976 Initial commit
```

This output shows that the feature branch has advanced to the new commit. The master branch is still at the older commit. Now switch back to the master branch.

```
$ git checkout master
Switched to branch 'master'
```

Any changes you make now will create a new commit along the master branch.

If you open `explore.R`, you'll find that your function stub is not there! Don't worry, you just need to switch back to the `feat/new_feature` branch to see those changes. For now, stay on `master` and edit `explore.R`. Commit the change. View the log again, and you should see a branching tree-like structure that is congruent with 4.2.

```
$ git log --oneline --decorate --graph --all
* a439f9b (HEAD -> master) Add stub for examples for section 2.4
| * 1007278 (feat/new_feature) Clean up file
| * 90c99b1 Add stub for normalize_times
|/
* f40b865 Add get_event_times
* 04e6ec5 Add filter_events
* c4f2976 Initial commit
```

This view shows the latest commits first, with the oldest commit at the bottom. The commit `f40b865` is a common ancestor to both branches. □

4.3 Merging branches

The killer feature of `git` is the disposable branch. It's cheap and easy to both create and destroy branches. Most collaboration workflows in `git` revolve

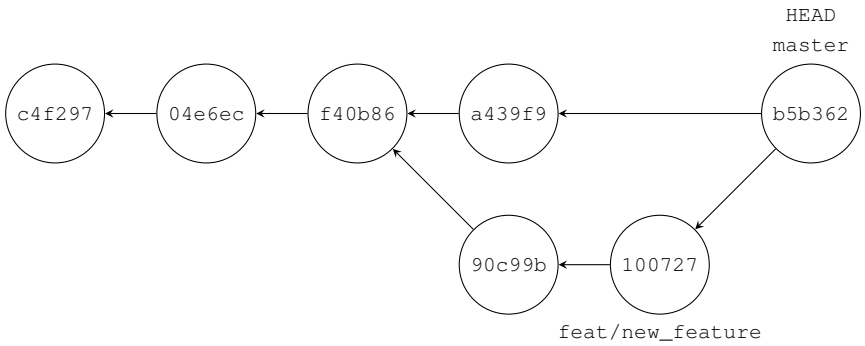


FIGURE 4.3: A merge commit has two parent commits.

around branches. These branches are typically short-lived and only exist until the code in the branch is ready to be merged with another branch, typically master. [3]

There are two approaches to merging branches. One is the eponymous `git merge`, which preserves commit history. With this approach, `git` creates a new commit that reconciles changes between your two branches and the common ancestor. This merge is known as a three-way merge and is shown in Figure 4.4.

Example 4.6. To see how a merge operation works, we can merge the branch `feat/new_feature` into `master`.

```
$ git merge feat/new_feature
Auto-merging R/explore.R
CONFLICT (content): Merge conflict in R/explore.R
Automatic merge failed; fix conflicts and then commit the result.
```

Ironically, `git` is pretty smart. Still, it won't always be able to deduce the correct way to merge the changes. When this happens, `git` will complain about a merge conflict. Each file with a conflict is indicated in the output. In this case it's our file `explore.R`. To fix the conflict, edit the file. You'll see lines similar to the output of `git diff` indicating the changes in each branch. Edit the file accordingly and save. The file has been fixed, but you still need to tell `git` that you resolved the conflict.

```
$ vi R/explore.R
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)
```

[3] Other branch management strategies exist, which are out of scope for this book.

```
        both modified:   R/explore.R

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        private/

no changes added to commit (use "git add" and/or "git commit -a")
```

Follow the instructions given by `git status` and add, then commit the change like any other commit.

```
$ git add R/explore.R
$ git commit -am "Fix merge conflicts"
[master b5b3627] Fix merge conflicts
```

Let's check the log once more to see the result of the merge.

```
$ git log --oneline --decorate --graph --all
*   b5b3627 (HEAD -> master) Fix merge conflicts
|\
| * 1007278 (feat/new_feature) Clean up file
| * 90c99b1 Add stub for normalize_times
* | a439f9b Add stub for examples for section 2.4
|/
* f40b865 Add get_event_times
* 04e6ec5 Add filter_events
* c4f2976 Initial commit
```

We can see that the special merge commit has two parents, unlike regular commits. The `master` branch points to this new commit. At this point it is safe to delete `feat/new_feature` with

```
$ git branch -d feat/new_feature
Deleted branch feat/new_feature (was 1007278).
```

□

Another approach to merging two branches is to rebase one branch onto another. Instead of preserving separate commit histories, `git rebase` collapses the two branches into a single branch. This is possible since each commit is a diff, and `git` is capable of applying the diffs in any arbitrary order. Some people prefer using a rebase over a merge since the commit history stays linear. However, one challenge with a rebase is that some commits are given a new hash, which makes it hard to trace issues later on. This happens if there is a merge conflict and `git` needs to modify the commit with your resolution.

Example 4.7. Suppose we choose to do a rebase instead of the merge in Example 4.6. For a rebase we work in the feature branch, which is opposite of the merge operation. The first thing `git` does is rewind to the common ancestor and apply each commit of `master`.

```
$ git checkout feat/new_feature
$ git rebase master
First, rewinding head to replay your work on top of it...
```

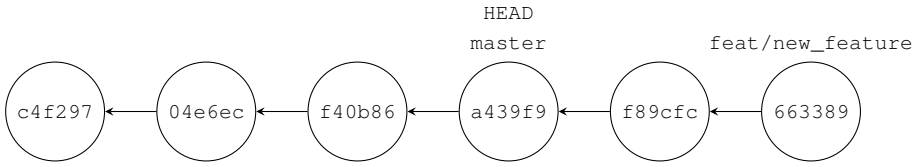


FIGURE 4.4: A rebase applies one branch onto another, creating a linear history.

```
Applying: Add stub for normalize_times
Using index info to reconstruct a base tree...
M       R/explore.R
Falling back to patching base and 3-way merge...
Auto-merging R/explore.R
CONFLICT (content): Merge conflict in R/explore.R
error: Failed to merge in the changes.
Patch failed at 0001 Add stub for normalize_times
The copy of the patch that failed is found in: .git/rebase-apply/patch

When you have resolved this problem, run "git rebase --continue".
If you prefer to skip this patch, run "git rebase --skip" instead.
To check out the original branch and stop rebasing, run "git rebase --abort".
```

Unfortunately, `git` still doesn't know how to handle our conflict, so we have to manually resolve it.

```
$ vi R/explore.R
$ git status
rebase in progress; onto a439f9b
You are currently rebasing branch 'feat/new_feature' on 'a439f9b'.
  (fix conflicts and then run "git rebase --continue")
  (use "git rebase --skip" to skip this patch)
  (use "git rebase --abort" to check out the original branch)

Unmerged paths:
  (use "git reset HEAD <file>..." to unstage)
  (use "git add <file>..." to mark resolution)

        both modified:   R/explore.R

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        private/

no changes added to commit (use "git add" and/or "git commit -a")
```

We resolve the conflict the same way as with a merge by `git add`ing the conflicted file. Instead of committing the change, we tell `git` to continue the rebase operation. It will then apply all of the commits from the common ancestor to `feat/new_feature` in order.

```
$ git add R/explore.R
$ git rebase --continue
```

```
Applying: Add stub for normalize_times
Applying: Clean up file
```

The end result is a linear history, where the `feat/new_feature` is now a descendant of `master`.

```
$ git log --oneline --decorate --graph --all
* 6633890 (HEAD -> feat/new_feature) Clean up file
* f89cfcf Add stub for normalize_times
* a439f9b (master) Add stub for examples for section 2.4
* f40b865 Add get_event_times
* 04e6ec5 Add filter_events
* c4f2976 Initial commit
```

Due to the conflict during the rebase, some of the commits differ from the original commits in `feat/new_feature`. This results in new commit hashes, which is sometimes problematic. For now we'll safely ignore this detail.

To fully integrate `feat/new_feature` into `master`, we need to merge the branch. The first step is to check out `master`.

```
$ git checkout master
Switched to branch 'master'
```

Now merge in `feat/new_feature`. Notice that this is a so-called fast-forward merge. To merge in the branch, the branch pointer for `master` just needs to be moved to coincide with the pointer for `feat/new_feature`.

```
$ git merge feat/new_feature
Updating a439f9b..6633890
Fast-forward
 R/explore.R | 5 ++---
 1 file changed, 2 insertions(+), 3 deletions(-)
```

Once the branch has been merged, it can be safely deleted.

```
$ git branch -d feat/new_feature
Deleted branch feat/new_feature (was 6633890).
```

It's good practice to delete merged branches. Like anything else, periodic maintenance goes a long way in keeping things organized and manageable.□

So rebasing branches takes more steps than a simple merge. There's also a chance that commit hashes will change. Why in the world would we prefer this to a simple merge? One reason is that fast-forward merges are easier on maintainers. As a dedicated role, maintainers don't have the context related to the changes a developer makes to her code. With a rebase operation, merge conflicts are moved upstream to the person responsible for resolving the conflict. This is a more efficient process.

The rebase operation is also a very fine instrument. It's not uncommon for repositories to go horribly wrong for one reason or another. We're human after all. With `git rebase` it's possible to perform surgery on a repository and save countless hours or days necessary to recreate changes lost in a repo due to cavalier commits.

4.4 Remote repositories

So far, all the operations we've performed in `git` have been local to our repository. What if we use a remote repository as our public repo? There are two common scenarios to consider. If we create a package and want to share it, it's likely we already have a repository and just need to push it to a remote repository. To do this, we need to create a remote target repository first. Alternatively, if we are using someone else's code, we'll first **clone** the repository into our local workspace.

Example 4.8. Suppose you want to share the diabetes project with a collaborator. Any of the web-based `git` providers can host your repository. Create the repository in your host of choice. Since you already have a local `git` repository, all you need to do is tell your local repository where your remote repository is.

```
$ git remote add origin https://github.com/profbrowe/diabetes.git
```

This command tells `git` to associate the remote repository to the name "origin". Assuming you have an account on Github and have properly authenticated, you can now push your current branch to origin, specifying the target branch. [4]

```
$ git push -u origin master
Counting objects: 29, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (22/22), done.
Writing objects: 100% (29/29), 3.11 KiB | 0 bytes/s, done.
Total 29 (delta 6), reused 0 (delta 0)
To https://github.com/profbrowe/diabetes.git
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
updating local tracking ref 'refs/remotes/origin/master'
```

The `-u` switch tells your local repository to track this remote repository, creating a link between the two using the name `origin`. You only need to use this option the first time.

If a collaborator pushes changes to origin, you may want to retrieve her changes. You can retrieve the contents of a remote repository with `git fetch`. This command downloads the commit history of a remote repository in a special namespace called `remotes`. Fetching a remote branch does not change your local working copy. Integrating remote changes thus requires a merge operation, because `git` treats this remote branch (living in your local repository) like any other branch. These two operations are "conveniently" combined in the `git pull` command. However, I rarely use this because I may accidentally merge `master` into the wrong branch or a dirty workspace. This creates

[4] If you are following along, you need to create your own repository. Pushing to my repository will fail.

avoidable headaches, such as fixing merge conflicts or deleting commits. I almost always fetch first to see if there are any changes to worry about. If so I do `git status` to make sure my repository is in the right state to merge the remote branch. If not, I'll either create a new branch or `git stash` my changes. Once my local repository is ready, I merge in the remote branch.

Example 4.9. To see how remote merges work, find a friend to help you or clone the repository a second time into a new directory. Make some changes and push to origin. Now go to your original repository and fetch and merge.

```
$ git checkout master
$ git fetch
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 3 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/profbrowe/diabetes
   6633890..b025354  master     -> origin/master
$ git merge origin/master
```

□

Merge syntax is different from push and pull. With a merge, you specify the *local* branch corresponding to the remote branch. This is a single branch name. Push and pull operations specify a remote repository followed by the name of the branch in the remote repository.

Example 4.10. Consider this alternative scenario. Your collaborator is working on a feature branch and shares it with you via origin. This is the first time you'll work on it, so you need to check out the branch. The first step is to fetch the latest changes of the remote repository.

```
$ git fetch
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 4 (delta 0), reused 4 (delta 0), pack-reused 0
Unpacking objects: 100% (4/4), done.
From https://github.com/profbrowe/diabetes
 * [new branch]      feat/visualization -> origin/feat/visualization
```

Remember, when you call `git fetch`, it's downloading the remote commit history into your local repository. The last line of the output tells you exactly this: that it's taking the remote branch called `feat/visualization` and linking it to a local branch called `origin/feat/visualization`. The `checkout` command operates on this local branch that mirrors the remote branch. This is why the branch name is prefixed with `origin`.

```
$ git checkout -t origin/feat/visualization
Branch feat/visualization set up to track remote branch feat/
    visualization from origin.
Switched to a new branch 'feat/visualization'
```

Once you check out the remote branch, `git` informs you that it's tracking it. Any changes you make will be associated with your local branch. It's only when you push to `origin` that those changes will be reflected in your local mirror of `origin`, in addition to `origin` itself. □

4.5 Exercises

Exercise 4.1. It's not necessary to provide the complete commit hash to check-out a specific commit. Starting with the left-most characters of the hash, determine the minimum numbers of characters needed for git to recognize your commit hash.

Exercise 4.2. Suppose you are on the `master` branch of the diabetes repo. What is the output of `git diff master`?
(a) Does this output make sense? Explain.
(b) Under what circumstance would the output be different?

Exercise 4.3. Suppose you clone a project directly and later want to fork it. It's unnecessary to re-clone the repo after forkkng. Instead use `git remote set url` to change the location of `origin`.

Exercise 4.4. Checkout an older commit. Create a branch from that commit and then add some commits. Use `git log --oneline --graph` to show the structure of the repository.

Exercise 4.5. A common challenge with `git` is when a file is accidentally added to a reposutory. Since the `git` repo contains all history, large files can bloat a repo and make it harder for other people to clone the repo. The `git filter` operation can remove commits completely from a repo. Remove a commit from the diabetes repo. Show the commit graph before and after the change.

5

Formalizing code in a package

> Repetition is the only form of permanence that nature can achieve.
>
> *George Santayana*

Unlike pure software engineering, it's rare for modelers to create a package at the beginning of a project. Projects may not reach a maturity level that dictates a self-contained project. It's therefore inconvenient to create a lot of structure early on. Instead, it's better to add structure when needed, usually after a model shows some promise or needs to be more widely distributed. Another good time to add structure is when you begin to lose faith in your application. When it becomes unreliable and is no longer predictable, you need to start writing tests. This is the time to transition your scripts into a formal package. In other words, you are preparing your work to be repeatable and collaborative.

Packages thus have value even if they aren't published to CRAN. For starters, packages automate testing. Rather than testing functions ad hoc, all tests can be run when a package is built. Coupled with continuous integration tools (see Section 5.5) this means less time worrying whether your software will work as expected. It's better to know that your code works, rather than hope that it works. Packages also simplify dependency management, since they explictly define your code dependencies. Installation piggy-backs off the R package installation process, making it easier to verify portability. Packaged models can usually be shared as is, without needing to provide a complete model environment. External documentation is also kept to a minimum, since this work has been bundled into the package itself.

5.1 Creating a package

There are many ways to create a package. We'll use my `crant` utility to initialize a package using the scripts you already have. `Crant` tries to minimize the amount of manual work required by inferring as much information from your existing R scripts as possible. The `init_package` script that comes with

`crant` recognizes that the decision to package code may happen later in the development process (as does `package.skeleton`). It uses this observation to generate package metadata with the correct information, minimizing the post-package creation steps to get the package to build.

Second, `crant` is largely unopinionated. [1] It doesn't force you to do things a particular way, so you retain your freedom and independence as a modeler. The `init_project` script reads all `.R` files in an `R` directory. It constructs the `DESCRIPTION` and `NAMESPACE` files required for R packages using the information in the scripts. The script is non-destructive, so if these scripts already exist, it doesn't overwrite the files. Most importantly, it finds all the dependencies and adds them to the `DESCRIPTION` file. It also provides a working stub for tests. The end result is a package that can be built with minimal work.

Example 5.1. It's possible to create a fully buildable package by specifying three arguments to `init_package`: the package title, the author of the package, and the description.

```
$ cd ~/workspace/diabetes
$ init_package -t 'Analyze Diabetes Patients' \
>   -a "Author Name <author@name.com>" \
>   -d "The description. It needs to contain at least two sentences."
Create .Rbuildignore
Create .gitignore
Create package descriptor R/diabetes.3-package.R
Create DESCRIPTION file
Create NAMESPACE file
Set up test harness
Create Travis CI build script
To complete initialization be sure to complete R/diabetes-package.R
and possibly DESCRIPTION (if you didn't provide title, author, etc.).
Then commit and run crant -x to generate documentation, test, and build
```

Like `git`, `crant` tries to be helpful and guide you through next steps, so you know what to do. An easy way to see what was created is to check the

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        .Rbuildignore
        .gitignore
        .travis.yml
        DESCRIPTION
        Dockerfile
        Makefile
        NAMESPACE
        R/diabetes-package.R
        tests/
```

[1] That said, two executive decisions are made: one for documentation and one for testing. ROxygen2 is used for documentation, and testit is used for unit testing. These decisions are optional though and can be easily changed.

```
nothing added to commit but untracked files present (use "git add" to track)
```

Lo and behold, we have a bunch of new files. Add them like any other files and commit them to your repository.

```
$ git add .
$ git commit -am "Add package metadata"
[master c425cba] Add package metadata
 10 files changed, 110 insertions(+)
 create mode 100644 .Rbuildignore
 create mode 100644 .gitignore
 create mode 100644 .travis.yml
 create mode 100644 DESCRIPTION
 create mode 100644 Dockerfile
 create mode 100644 Makefile
 create mode 100644 NAMESPACE
 create mode 100644 R/diabetes-package.R
 create mode 100644 tests/test-all.R
 create mode 100644 tests/testit/test-example.R
```

Documentation can now be generated and the package built. □

5.2 Building a package

Once the package metadata has been generated, you can build and install the package how you see fit. With `crant`, the line

```
$ crant -xi
```

will generate documentation, build, run tests, and install locally. The `-x` switch will run ROxygen2 to generate the documentation and commit the changes to the git repository. ROxygen2 generates the R package documentation inline. This switch is only necessary when the documentation changes.

When building, you'll see a NOTE in the package check. Follow the instructions to update your NAMESPACE file appropriately. Inside, you'll notice that the dependency to `futile.logger` has been already added by `crant`. The script also adds all the source files in the collate section. If you add more files, you'll need to manually add them to the DESCRIPTION file. Use

```
$ git commit -am "Add import to NAMESPACE"
$ crant -i
```

to rebuild and reinstall the package. The `-x` switch is unnecessary this time since no documentation has changed.

In your R session, load the package using the standard `library` command. If you have already loaded the package in your session, you may want to remove all variables in your R workspace first.

```
> rm(list=ls())
> library(diabetes)
```

There are a few ways to verify the package has been loaded. One way is to use the `sessionInfo` function to inspect the current state of your session. This function lists the version of R you're using, along with the versions of the packages currently loaded.

```
> sessionInfo()
R version 3.4.4 (2018-03-15)
Platform: x86_64-pc-linux-gnu (64-bit)
Running under: Ubuntu 16.04.4 LTS

Matrix products: default
BLAS: /usr/lib/libblas/libblas.so.3.6.0
LAPACK: /usr/lib/lapack/liblapack.so.3.6.0

locale:
 [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
 [3] LC_TIME=en_US.UTF-8        LC_COLLATE=en_US.UTF-8
 [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
 [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
 [9] LC_ADDRESS=C               LC_TELEPHONE=C
[11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

other attached packages:
[1] futile.logger_1.4.3

loaded via a namespace (and not attached):
[1] compiler_3.4.4       formatR_1.5          lambda.r_1.2.1
[4] futile.options_1.0.1
```

This function also shows you *how* a package was loaded. Some packages depend on other packages. These packages are imported but are not directly accessible. To use them in your session, you need to attach them by explicitly calling `library` or `require`.

During package development, you may decide to hold your version number constant. This is convenient, but it can make it more challenging to verify that the appropriate package code was loaded. One workaround is to explicitly inspect the contents of the package. The `ls` function can do this for you, showing all the defined functions in a package.

```
> ls('package:diabetes')
[1] "read_diabetes"
```

The body of an exported function can be viewed by typing the function name in the shell. For non-exported functions, the package name must be provided as a prefix followed by three colons, such as `diabetes:::read_diabetes`.

5.3 Using packages

During development, you'll be rebuilding and reinstalling your package regularly. The pure R approach to reloading a package first detaches it and then reloads it using `library` or `require`.

```
> detach('package:diabetes', unload=TRUE); library(diabetes)
```

If this is too verbose, *devtools* provides two alternatives. You can simulate a build directly in R using `load_all`. This function simulates loading functions and data within your package.

```
> load_all()
```

This can be convenient, but it's important to remember that `load_all` only simulates a package build, and doesn't actually do one. The process isn't identical and can produce unexpected behavior. This is fine for personal work. When you want to publish or share your work, it's best to correctly build and load your package to ensure proper behavior.

Another approach uses the `reload` function, also in *devtools*. This function assumes you are developing a package and operates on package metadata. It thus requires a file path to load the metadata and attempts to unload and reload packages accessible by the path. If no path is provided, it assumes `"."`.

```
> reload()
Reloading installed diabetes

Attaching package: 'diabetes'

The following object is masked _by_ '.GlobalEnv':

    read_diabetes
```

If the package has not been loaded yet, it will complain, so you'll have to use `library` to load it first. If you want to reload an actual package, an extra function call is required.

```
> reload(inst("diabetes"))
Reloading installed diabetes

Attaching package: 'diabetes'

The following object is masked _by_ '.GlobalEnv':

    read_diabetes
```

The `inst` function will find the path to the package, so `reload` can operate on it like a local source package. On the one hand, this design preserves a consistent interface for `reload`. On the other hand, the opinions embedded within the design makes usage more complicated. For the sake of convenience, you have to learn this tool's way of doing things. Most opinionated software suffers from this tradeoff: if you stick to the opinionated way of doing things, life

is simple. To do anything else requires learning idiosyncratic semantics and commands. Unfortunately this knowledge is less transferable than learning how R works and using base commands.

For completeness, recall that I discussed a compromise in Section 2.6.3 that defines a private function responsible for reloading a package. This is usually only necessary if you are developing multiple packages simultaneously.

5.4 Testing packages

Even if code runs, there is no guarantee that the results are correct. This statement is particularly true of models, where it is easy to incorrectly implement an equation or algorithm. Only the foolhardy think that the code they've written is error free. The rest of us write tests to verify that code works as expected. The simplest tests are manual and rely on human inspection of results. While this may instill some initial confidence, these tests do not promote repeatability. To establish correctness and reliability, tests must be consistent and have adequate **coverage**. Test automation ensures the same tests are executed every time and is more convenient as test coverage increases.

Numerous packages exist for managing tests, including `RUnit`, `testthat`, and `testit`. `Crant` uses `testit`, because I tend to favor simplicity over functionality. The choice of framework is less important than actually choosing one and using it. These frameworks integrate with the R package build process. If you run `init_package` on your package, you'll find a test harness at `tests/test-all.R` and an example test file at `tests/testit/test-example.R`. The harness is responsible for loading the test package and executing the tests. The example test file implements a tautology, which acts as a workable example. The syntax of tests in `testit` is simple: call the function `assert` and provide a label and a block of code to execute. Remember that code blocks are delineated by curly braces and can be used anywhere to encapsulate multiple statements. Tests are wrapped in parentheses and are expected to evaluate to `TRUE`. When the test is run, `testit` scans the code and looks for these assertions and reports the status of each one.

Example 5.2. Let's implement the successor function and test it. This function is used in the Peano axioms to construct the natural numbers.

```
> succ <- function(x) x + 1
```

The following test case defines a number of individual tests to verify the behavior of the `succ` function. The logical tests can be interspersed between normal code. The structure of a test case block typically begins with initialization of input variables, followed by a function call to collect the output result. Finally, the output is compared with the expected value.

```
> assert('Successor function adds 1', {
```

```
+   x <- 3
+   (succ(x) > x)
+   y <- succ(x)
+   (y > x)
+   (x + succ(y) == succ(x + y))
+ })
```

Running this `assert` block in the console will return nothing, which means that all tests passed. On the other hand, if a test fails, `testit` will print out an error message associated with the first test failure.

```
> assert('Successor function with incorrect test implementation', {
+   x <- 3
+   (succ(x) == x)
+   (succ(x) == x + 1)
+   y <- succ(x)
+   (y == x)
+ })
assertion failed: Successor function with incorrect test implementation
Error: (succ(x) == x) is not TRUE
```

□

To automate tests, it's best to add them as new files in the `tests` directory. To be detected by `testit`, test files need to be prepended with `test`. Since tests are run as part of the build process, all tests can be run by calling `crant -S` from the command line. The `-S` option tells crant that a dirty working directory is okay. Alternatively, a single test file can be sourced from an R session.

Knowing how to test is important. Knowing what to test will ensure that your tests are effective. Chapter 16 discusses different types of tests and how they relate to the model development process.

5.5 Continuous integration

Every change we make to a model can affect the results. The same is true of software. Even seemingly small, innocuous changes can have an out-sized effect. It would be nice to know about the effects of these changes before we start a large model job or share our model with others. **Continuous integration** is a technology that does just this. It was designed to integrate multiple branches of code into a mainline branch à la `master`, in order to identify breaking changes as quickly as possible. Coupled with test-driven development, CI tools can also run a suite of unit tests to verify system behavior and notify the offending programmer. Depending on the size of your project and the number of collaborators, continuous integration moves from being a convenience to a necessity. In many development processes, a version of code is not released into a production environment unless all tests run on a CI server pass. The concept of continuous integration can extend to continuous delivery, where code is automatically deployed to a production-like environment.

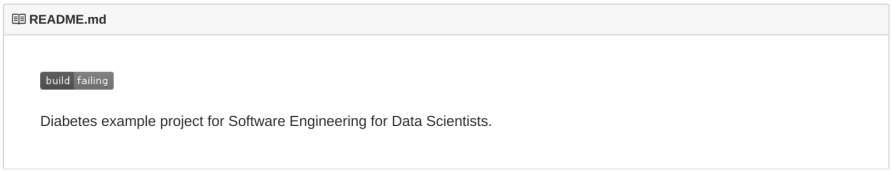


FIGURE 5.1: A special image indicates the build status of a project in Travis CI. The image automatically updates when the build status changes.

This approach simplifies integration testing and also enables non-technical stakeholders to test more quickly.

One option for continuous integration is Travis CI. It integrates easily with Github and is free for open source projects. [2] When you initialize a package with `init_package`, the script creates a hidden configuration file called `.travis.yml`. If your Github account is linked with Travis CI and your project activated, Travis CI will automatically build and test your package on three versions of R: `devel`, `release`, and `oldrel`. Testing against these three releases is a prerequisite of publishing to CRAN. It's also good practice as it increases the chance that your code is compatible with an end-user's system. The `devel` version is the upcoming version of R currently in development. The `release` is the current publicly available R version. Finally, `oldrel` is the previous public release of R. Testing against all three of these versions increases the portability of your package.

Every project in Travis CI has a build status. This status is exposed via a small image that can be added to your repository. The URL to the image has the structure `https://travis-ci.org/<user>/<project>.png`, where `<user>` is your username and `<project>` is the name of your project. A common approach is to add it to the top of the `README.md`. For your diabetes project, prepend this line to your `README.md`, replacing my username with yours.

```
[![Build Status](https://travis-ci.org/profbrowe/diabetes.png)](https:
    //travis-ci.org/profbrowe/diabetes)
```

When you push to your repository, you'll see an image showing the current build status similar to Figure 5.1. When the build status changes, the image will update showing the new status.

[2] See `https://github.com/marketplace/travis-ci` for adding Travis CI to your Github account.

5.6 Exercises

Exercise 5.1. To include a new dependency to a package, it needs to be added to the `DESCRIPTION` and `NAMESPACE` files. Let's include SVM as another classification method for the diabetes analysis. Add the `e1071` to the `DESCRIPTION` file under `Imports`. In the `NAMESPACE` file, use the `importFrom` directive to only add the `svm` function. Show the contents of your `DESCRIPTION` and `NAMESPACE` files.

Exercise 5.2. Add a test to a new file `test-explore.R` to the `tests` directory. Run `crant -S` and verify that your test was executed. What steps did you take to verify your test actually ran? Show the contents of your test and the relevant output from `crant`.

Exercise 5.3. Add Travis CI to your Github account. Update the README.md of your fork of the diabetes project with your project's build status image. Push to your repository and view the build status, which should be failing. Use the Travis CI logs to determine the reason for the failure. What needs to be corrected? Include a screenshot to support your diagnosis.

6

Developing with containers

Container technology has quickly become the de facto standard for building systems. Interestingly, containers have also infiltrated the development process, replacing virtual machines and other approaches to creating multiple isolated computing environments on a workstation. This makes repeatable science far simpler, since you can guarantee that your code will run in different environments without ever truly leaving yours. It also means that all this compatibility testing can be automated, reducing the amount of manual work the data scientist must do.

Like virtual machines, containers provide an isolated computing environment to run applications and systems. Unlike virtual machines, containers are lightweight, composable, and disposable. Containers are great for package developers, because you can test your package against different versions of R, without having to uninstall and reinstall these different versions on your actual workstation. The ability to easily create isolated computing environments is beneficial to all modelers, irrespective of whether a package will be published. Developing inside containers keeps your local environment clean. This is important, since some tools have massive dependencies that may affect your underlying system. This is less an issue with R than with Python. Historically, changes to a Python environment needed for modeling had the potential to break scripts built to manage the operating system. Another example is the deep learning framework TensorFlow. In early releases, installing TensorFlow required downloading and installing a programming language (go), a build tool (bazel), and other dependencies that added some 3 GB of additional software. If you wanted to later uninstall this system, you would be dependent on the TensorFlow maintainers to provide a correct uninstaller to restore your previous system state. Many solutions exist for managing dependencies. Containers provide a solution that transforms a headache into a trivial exercise.

Containers also facilitate repeatable science. My `init_package` script generates a Dockerfile that creates an isolated environment for your R package. This image includes all the dependencies to run your package. The script also creates a Makefile that simplifies working with the Docker image and container. The result is a repeatable model environment that can reproduce results using just one command. And your collaborators now only have one dependency to satisfy: Docker. Anyone with Docker is guaranteed to be able

```
1 FROM zatonovo/r-base
2
3 COPY . /app/diabetes
4 WORKDIR /app/diabetes
5 RUN crant -SCi
```

LISTING 6.1: A minimal `Dockerfile` for a custom package. This file is generated automatically by `crant`.

to build and run your package, [1] and ultimately reproduce the results you achieved. This approach is far more efficient than the days before containers, where any little dependency had the potential to torpedo an installation if it was missing or out of date.

Containers are popular in large systems, providing the infrastructure for so-called **microservices**. With your model already in a container, it makes it is easy to migrate your model to a production environment. This process may involve parallelizing computations. Since containers run independently and are identical, they can be used in a compute grid to speed up computations. Combined with cloud computing, a model can be created locally on a truncated dataset. When ready for a complete run, it can be easily deployed to the cloud into a virtual machine instance with far more computing resources. When the job is done, you can stop the process, so you only pay for the computing resources you need.

Are containers the silver bullet of the computing world? Unfortunately, it's not all daisies. Containers require a lot of disk space. This usually isn't an issue these days, but images can eat up a lot of storage. A bigger issue is that they add a layer of complexity. Not only do you need to have a basic understanding of container technology, but so do your collaborators. For simple package development this is usually fine, as the advanced features of containers are barely utilized. Things get more complicated when creating an interactive dashboard or notebook. Now peers need to understand how ports work and potentially volume mapping, among other things. One of the reasons that `init_package` creates a Makefile is to hide some of these details. The Makefile simplifies usage of Docker by automating and inferring many of the configuration options.

Recall that a Dockerfile defines an image. For example, the Dockerfile created by `init_package` is shown in Listing 6.1. Line 1 says that this image is derived from the image called `zatonovo/r-base`. I created this base image to hide a lot of the details related to system dependencies. The next line instructs the Docker engine to copy the contents of the current directory `.` into the image directory `/app/diabetes`. Line 4 sets the working directory to this newly created directory. Finally, the package is built using `crant`.

[1] Nothing is fully guaranteed, and there are situations where local state can seap into a Docker container. This seapage can lower the probability of repeatability.

To build the image, a Docker command must be executed. A basic workflow showing the relationship between various Docker commands is shown in Figure 6.1. The first step is to call `docker build`. This reads the Dockerfile and builds the image based on the `zatonovo/r-base` image that contains most of the dependencies you need. Alternatively, we can use the Makefile to call the command for us. The command `sudo make` must be run in the directory of the `Makefile`. [2] Without arguments, `make` runs the first target it finds, which is `all`.

```
$ sudo make
docker build -t diabetes:1.0.0 .
Sending build context to Docker daemon  156.7kB
Step 1/4 : FROM zatonovo/r-base
 ---> 3abbaada846b
Step 2/4 : COPY . /app/diabetes
 ---> 7b835946b617
Removing intermediate container f62537c91d1f
Step 3/4 : WORKDIR /app/diabetes
 ---> 2244b7431b1d
Removing intermediate container 9be82ba2f4a8
Step 4/4 : RUN crant -SCi
 ---> Running in dc762ce81c40
Running build chain on diabetes
* checking for file 'diabetes/DESCRIPTION' ... OK
* preparing 'diabetes':
* checking DESCRIPTION meta-information ... OK
* checking for LF line-endings in source and make files and shell scripts
* checking for empty or unneeded directories
Removed empty directory 'diabetes/man'
* building 'diabetes_1.0.0.tar.gz'

/app
Installing package diabetes
* installing to library '/usr/local/lib/R/site-library'
* installing *source* package 'diabetes' ...
** R
** preparing package for lazy loading
No man pages found in package  'diabetes'
** help
*** installing help indices
** building package indices
** testing if installed package can be loaded
* DONE (diabetes)
Done with package diabetes
 ---> 373e8dd13420
Removing intermediate container dc762ce81c40
Successfully built 373e8dd13420
Successfully tagged diabetes:1.0.0
docker tag diabetes:1.0.0 diabetes:latest
```

The Makefile bundles the syntax to build an image into a simpler command. It also ensures we tag the image appropriately for consistent version man-

[2] Depending on how you configured Docker, you may need to use `sudo` to temporarily grant yourself administrator rights.

agement. You can verify that the image was built by querying the Docker daemon.

```
$ sudo docker images | grep diabetes
diabetes                                  1.0.0               dd0112f87c2c
About a minute ago   3.69GB
diabetes                                  latest              dd0112f87c2c
About a minute ago   3.69GB
```

where diabetes is the default name of the image defined in the `Makefile`. Docker maintains a registry of all the images that you've built. Images are built in layers, one command at a time. This makes it easy to reuse parts of an image or create a new image based on an existing image.

6.1 Working in a container

Now that we've built an image, let's create a container. A container is a temporary or semi-permanent runtime instance of an image. Persistent containers act like traditional virtual machines where they preserve changes even when the container is stopped. This behavior is convenient but can also reduce repeatability, since a container's state will diverge from the image state. Temporary containers (specified by the switch `--rm`) are destroyed once they are stopped. This approach ensures that running containers are started in the same initial state as the image. Again, let's use the Makefile to start the container with `make run`. The Makefile always shows the underlying Docker command, so you aren't bound to the Makefile for life.

```
$ sudo make run
docker build -t diabetes:1.0.0 .
Sending build context to Docker daemon  156.7kB
Step 1/4 : FROM zatonovo/r-base
 ---> 3abbaada846b
Step 2/4 : COPY . /app/diabetes
 ---> Using cache
 ---> 7b835946b617
Step 3/4 : WORKDIR /app/diabetes
 ---> Using cache
 ---> 2244b7431b1d
Step 4/4 : RUN crant -SCi
 ---> Using cache
 ---> 373e8dd13420
Successfully built 373e8dd13420
Successfully tagged diabetes:1.0.0
docker tag diabetes:1.0.0 diabetes:latest
docker run -d -p 8004:8004 -v /home/brian/workspace/diabetes:/app/diabetes diabetes
ac1f232ee5d5b93cdadc3aa3d08e5a6233170c65427a185e221c6e42d0edb27b
```

This command also builds the image. If nothing changes, then a cached version is used. Finally, it shows the `docker run` command for creating a

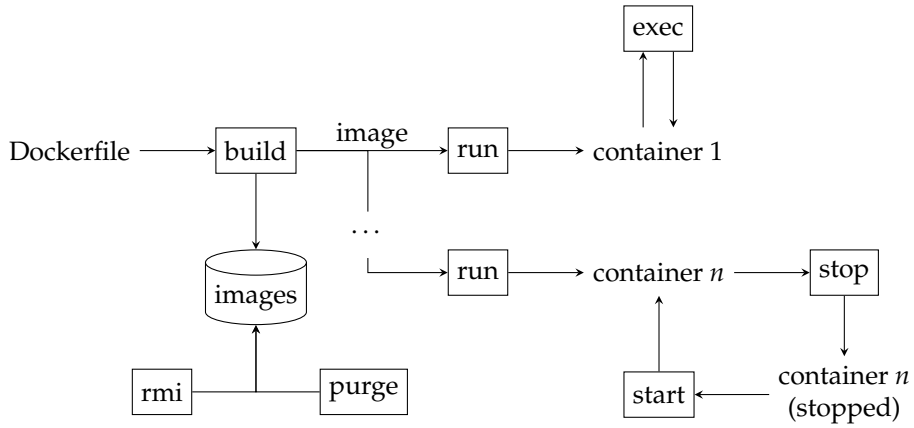


FIGURE 6.1: An extended view of Docker commands. Multiple containers can be created from the same image and commands can be `exec`uted on a running container. Built images are stored locally in a repository that must be cleaned periodically.

new container. We can verify that the container is running by using Docker's version of the `ps` command.

```
$ sudo docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS              PORTS
NAMES
ac1f232ee5d5        diabetes            "tini -- /bin/sh -..."   3 minutes ago
Up 3 minutes        80/tcp, 443/tcp, 8888/tcp, 0.0.0.0:8004->8004/tcp
xenodochial_engelbart
```

It's great that we've created a container, but what can we do with it? First, let's create a `bash` session inside the container.

```
$ sudo make bash
docker exec -it ac1f232ee5d5 bash
root@ac1f232ee5d5:/app/diabetes#
```

This commands attaches a `bash` session within your newly minted container. This looks and feels like a normal Linux environment, except that project directories are in a directory named `/app`. This is just a convention defined in the Dockerfile. Notice that the prompt has changed to indicate the user (root) and the name of the container, which is part of a hash ID that uniquely identifies the container. We can work inside this container as though it were a remote computer we logged into. For example, let's start an R session and load our `diabetes` package.

```
root@ac1f232ee5d5:/app/diabetes# R --quiet
> library(diabetes)
> df <- read_diabetes()
INFO [2018-06-12 15:03:48] Loading file private/diabetes/data-01
```

This two step dance is designed for running the container as a standalone server. If you only want to use the container for an interactive R session, just run

```
sudo make r
```

and a new temporary container will be started instead.

To shutdown the container, run the following command on your host machine.

```
$ sudo make stop
docker stop ac1f232ee5d5
ac1f232ee5d5
```

If you ran this command from a new terminal window, your existing session within the container will be terminated automatically.

6.1.1 Volume mapping

Multiple containers can be created from the same image. These containers start off as identical clones. Over time, their contents (data) may change. But when a temporary container is stopped, all changes are lost. That means upon startup, the container will revert to the initial state and will be indistinguishable from a newborn clone. A clean state is one way containers guarantee repeatability, but what if we want a process to save some data? One approach is to send the data to another server that acts as a datastore. This is appropriate for production environments but cumbersome during model development, particularly during exploration. The solution is to add a volume mapping. When starting the container, you specify a local directory that corresponds to a directory in the container. Changes to this folder made in the container will be visible to your host workstation and vice versa. This behavior violates the idea of complete isolation for the sake of convenience. The Makefile automatically maps your local diabetes folder to `/app/diabetes` within the container. That means that any changes you make to this folder on your host workstation will be visible in the container.

Example 6.1. To illustrate how volume mappings work, let's run the container with and without the volume mapping. First, start the container using `make run`. As mentioned, this command automatically creates the volume mapping. Enter the container with `make bash` and execute the following command:

```
# echo "From container" >> from_container.txt
```

This command writes the text "From container" into the file called `from_container.txt`. From your host machine, open a new terminal window and `cd` into your diabetes project. Type

```
$ cat from_container.txt
From container
```

Now let's do the same thing without the volume mapping. This time we'll execute the Docker command explicitly and start a `bash` shell directly.

```
$ docker run -it --rm diabetes bash
```

This command creates an interactive terminal session. The `--rm` switch tells Docker to remove this container after we exit. From within the container, delete the file we created above.

```
# rm from_container.txt
```

In your host system, check if the file is there or not. What do you see? □

Theoretically there is no distinction between state and data, and the system state can be changed with cavalier use of volume mapping. For example, it's possinle to map the Rlibs directory of the host to a container. Doing so streamlines development but nonlonger guarantees identical initial conditions. If you have a model working on your machine given your libraries When someone else runs the code, she may have different libraries than you that results in different results.

6.1.2 Using graphics in a container

One issue with running R or Python from Docker is that these interpreters do not have access to the host GUI by default. It's difficult to render plots, which can be inconvenient. If your host system is Linux, the easiest solution is to use X forwarding, which sends graphics output over the network. Since UNIX started life as a headless server system, clients were responsible for rendering graphics. The X11 windowing system provides this functionality out of the box and just needs to be configured.

The Makefile provided by `init_package` is already configured to forward X11 for your R session. Add the following to your `.profile` to ensure you have your X11 forwarding rules properly set up.

```
$ echo "XAUTH=$HOME/.Xauthority; touch $XAUTH" >> ~/.profile
```

Then run

```
$ sudo make r
```

and use the R console as you normally would.

6.2 Managing containers

Containers that run a foreground server process will run indefinitely until stopped. The command docker ps shows the running containers. These containers are terminated using docker stop with either the container name (if set) or the container ID. By default, containers are persistent. Stopped containers

can be viewed using `docker ps -a`. If you want to remove a stopped container and delete it completely, use the command `docker rm <container id>`. When a container is run with the `--rm` switch, it is simply deleting it after it is stopped.

Running containers are like a remote server. You can "log in" by attaching a new session to the container. The `make bash` command does just this by using `docker exec`, instead of `docker run`. Attached sessions are indispensable for debugging. If your log files are written to a folder that is not volume mapped, the easiest way to view them is to open a `bash` session within the running container. It's also convenient to directly check versions of libraries within a container and even make changes. These changes can be reflected in the image by updating the Dockerfile.

6.3 Working with images

Docker keeps a copy of every image you've built or run on your workstation. It also tracks intermediate layers, which can speed up building an image. A container can be created from any layer, which is useful if an image is broken. If a container cannot be started from an image, a common troubleshooting step is to start a `bash` session from the preceding layer and investigate. Sometimes it's necessary to go up a few layers until a container can actually be run. The container is created like any other container using `docker run --rm -it <image id>`. The only exception is that the image ID must be specified explicitly instead of using a name alias.

Image layers can take up a lot of space and may need to be cleaned up periodically. It's not uncommon for layers to take up 50-100 GB of disk space. The `docker rmi` command removes individual images. The `docker prune` command removes so-called dangling images. These images are associated with older versions of an existing image that were rebuilt using the same label. Consequently, the older version loses its name, which just displays `<none>`. Adding the `-a` switch to `docker prune` removes all unused images as well. These images are not associated with any container (running or stopped).

6.4 Running a notebook from a container

Interactive documents that supported code and rendered results was introduced to the scientific computing world in the first version of Mathematica, back in 1988. [38] Thirty years later, the popularity of the notebook interface has surged. Most data scientists are introduced to notebooks via Jupyter, the

FIGURE 6.2: The notebook home page, showing all available notebooks. If no notebooks have been created, this will be an empty table.

Python-based notebook system. Notebook technology comprise two parts: a kernel that executes code and a front-end that provides an editor and renders results. Bundled within the `zatonovo/r-base` image are the dependencies to run a Jupyter kernel for R.

To run the kernel, a different `make` command is used. First, ensure that your container is stopped. The `make notebook` command will start the notebook server and provide instructions on how to authenticate your browser with the server.

```
$ sudo make notebook
  ... truncated for brevity ...
[I 17:53:59.926 NotebookApp] The Jupyter Notebook is running at:
[I 17:53:59.926 NotebookApp] http://[all ip addresses on your system
    ]:8888/?token=dc363562c559051dc3e7a75239d47e3981a3a9d02f13a8e3
[I 17:53:59.927 NotebookApp] Use Control-C to stop this server and
    shut down all kernels (twice to skip confirmation).
[C 17:53:59.927 NotebookApp]

    Copy/paste this URL into your browser when you connect for the
        first time,
    to login with a token:
        http://localhost:8888/?token=
            dc363562c559051dc3e7a75239d47e3981a3a9d02f13a8e3
```

The end of the log output provides the URL to enter into your browser. Open this location to authenticate and then go to the notebook home page, which will look similar to Figure 6.2. The initial page is not so exciting, because you don't have any notebooks yet. To create a new notebook, click the menu button "New" and choose R. Doing so opens a new window with the active notebook. Use the file menu to save the notebook with the name "Diabetes". Notebooks are saved to the `notebooks` folder within the package. This directory is configured to be ignored by R when building the package. However, to support collaboration, it is not ignored by `git`. When you create your first notebook, `git` will show you that the `notebook` folder is not tracked. Use `git add` to add it to your repository.

Since the notebook kernel is running within the container, it has access to

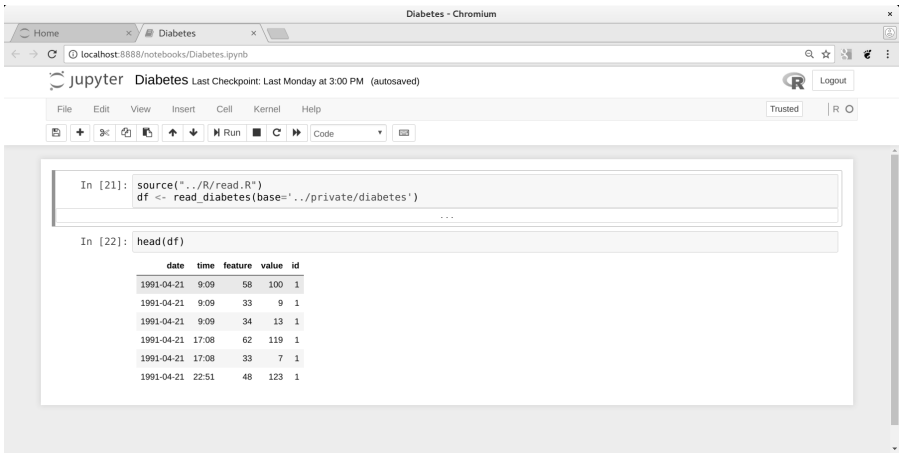


FIGURE 6.3: A notebook session that reads in the diabetes dataset and displays the first six lines.

the same R libraries as a typical R session. Assuming you have downloaded the diabetes dataset (see Section 3.1), you can read the dataset and interact with it in the notebook. It's possible to either `source` the R files explicitly or load the package. Figure 6.3 shows the result of reading in the dataset and displaying the first 6 records.

Example 6.2. One approach to development is to make all changes in package files. The notebook is used to test functions and/or use the functions on real data. Since your local package directory is visible in the container, it's possible to build the package locally and load it dynamically in the container. To do so, you need to add another volume mapping, from your local R package installation directory to the one in the container. Your local package installation directory is typically in your home directory and looks like `~/R/x86_64-pc-linux-gnu-library/3.4`. Within the container, packages are installed into `/usr/local/lib/R/site-library`.

```
$ docker run -d -p 8004:8004 \
  -v /home/brian/workspace/diabetes:/app/diabetes \
  -v ~/R/x86_64-pc-linux-gnu-library/3.4:/usr/local/lib/R/site-library \
  diabetes
```

For this to work properly, it's important that the major and minor version of R is the same in your local workstation and the container.

□

6.5 Exercises

Exercise 6.1. Start your container and then open a bash shell. Create a file in `/opt`. You can use the `touch` command to create an empty file. Stop the container and start it again. Is your file there? Why or why not?

Exercise 6.2. It's possible to create multiple containers from the same image. If you execute the command `make run` a second time to start a second container, you'll get an error message. That's because it is trying to use the same port on your workstation for another container. To assign a different port, use the command `make PORT=8005 run`. Use the command `docker ps` to show the running Docker containers. What is the output?

Exercise 6.3. The Jupyter notebook is installed in the `zatonovo/r-base` image. You can start it using the command `make notebook`. Follow the instructions to view a notebook app in your browser. Create a new notebook and try some R commands. Stop the container and start it up again. Is the notebook still there? Stop the container and then start it again using the command `make DATA=''notebook`. Repeat the process of creating a notebook, stopping the container and restarting it.

Exercise 6.4. The `zatonovo/r-base` image contains a working web server. Verify the web server is running by accessing URL in your browser. What is the output?

Exercise 6.5. The base image doesn't contain a lot of packages. If you want to use a package not included, you need to install it. However, since the container does not persist data, anything installed will be lost. The solution is to update the Dockerfile in your project directory to include the packages you want. Modify the Dockerfile to install `randomForest`. Hint: use the `rpackage` command from `crant`.

Exercise 6.6. Stopping a container deletes the container. It's possible to pause a container for a while, which effectively freezes and halts the operation. The command `docker pause` accomplishes this. Use this command on a running container. Try creating a `bash` session in your container with the command `docker exec -it <container id> bash`. What happens?