Technical Foundations of Informatics

Michael Freeman and Joel Ross

June 01, 2017

Contents

1 Setting up your Machine **9**
1.1 Git . . . 10
1.1.1 GitHub . . . 10
1.2 Command-line Tools (Bash) . . . 10
1.2.1 Command-line on a Mac . . . 11
1.2.2 Command-line on Windows . . . 11
1.3 Text Editors . . . 11
1.3.1 Atom . . . 12
1.3.2 Visual Studio Code . . . 12
1.3.3 SublimeText . . . 12
1.4 RStudio . . . 13

2 The Command Line **15**
2.1 Accessing the Command-Line . . . 15
2.2 Navigating the Command Line . . . 17
2.2.1 Changing Directories . . . 18
2.2.2 Listing Files . . . 18
2.2.3 Paths . . . 19
2.3 File Commands . . . 20
2.3.1 Learning New Commands . . . 21
2.4 Dealing With Errors . . . 23

3 Markdown **25**
3.1 Writing Markdown . . . 25
3.1.1 Text Formatting . . . 25
3.1.2 Text Blocks . . . 26
3.2 Rendering Markdown . . . 27

4 Git and GitHub **29**
4.1 What is this *git* thing anyway? . . . 30
4.1.1 Git Core Concepts . . . 31
4.1.2 Wait, but what is GitHub then? . . . 32
4.2 Installation & Setup . . . 32
4.2.1 Creating a Repo . . . 33

4.2.2 Checking Status 34
4.3 Making Changes 34
4.3.1 Adding Files 34
4.3.2 Committing 35
4.3.3 Commit History 36
4.3.4 Reviewing the Process 36
4.3.5 The `.gitignore` File 36
4.4 GitHub and Remotes 37
4.4.1 Forking and Cloning 38
4.4.2 Pushing and Pulling 39
4.4.3 Reviewing The Process 40
4.5 Course Assignments on GitHub 40
4.6 Command Summary 40

5 Introduction to R 43
5.1 Programming with R 43
5.2 Running R Scripts 44
5.2.1 Command-Line 44
5.2.2 RStudio 46
5.3 Comments 47
5.4 Variables 48
5.4.1 Basic Data Types 49
5.5 Getting Help 51

6 Functions 53
6.1 What are functions? 53
6.1.1 R Function Syntax 53
6.2 Built-in R Functions 55
6.3 Loading Functions 55
6.4 Writing functions 56
6.5 Conditional Statements 58

7 Vectors 61
7.1 What is a Vector? 61
7.1.1 Creating Vectors 61
7.2 Vectorized Operations 63
7.2.1 Recycling 64
7.2.2 Everything is a Vector! 64
7.2.3 Vectorized Functions 65
7.3 Vector Indices 66
7.3.1 Multiple Indicies 67
7.4 Vector Filtering 68
7.5 Modifying Vectors 69

8 Lists 71
8.1 What is a List? 71

8.1.1 Creating Lists 72
8.1.2 Accessing Lists 72
8.1.3 List Indicies 74
8.1.4 Modifying Lists 75
8.2 The `lapply()` Function 76

About this Book

This book covers the foundation skills necessary to start ***writing computer programs to work with data*** using modern and reproducable techniques. It requires no technical background. These materials were developed for the **INFO 201: Technical Foundations of Informatics** course taught at the University of Washington Information School; however they have been structured to be an online resource for anyone hoping to learn to work with information using programmatic approaches.

This book is currently in **beta** status. Visit us on GitHub to contribute improvements.

Chapter 1

Setting up your Machine

We'll be using a variety of different software programs to write, manage, and execute the code that we write. Unfortunately, one of the most frustrating and confusing barriers to working with code is simply getting your machine properly set up. This chapter aims to provide sufficient information for setting up your machine, and troubleshooting the process.

Note that iSchool lab machines should have all appropriate software already installed and ready to use.

In short, you'll need to install the following programs: see below for more information / options.

- **Git**: A set of tools for tracking changes to computer code (especially when collaborating with others). This program is already installed on Macs.
  - **GitHub**: A web service for hosting code online. You don't actually need to *install* anything GitHub (it uses `git`), but you'll need to sign up for the service.
- **Bash**: A *command-line interface* for controlling your computer. `git` is a command-line program so you'll need a command shell to use it. Macs already have a Bash program called *Terminal*. On Windows, installing `git` will also install a Bash shell called *Git Bash*, or you can try the (experimental) Linux subsystem for Windows 10.
- **Atom**: A lightweight text editor that supports programming in lots of different languages.
  - You are welcome to use another text editor if you wish; some further suggestions are included.
- **R**: a programming language commonly used for working with data. This will be our primary language for the quarter. "Installing R" actually means installing tools that will let your computer understand and run R code.

- **RStudio**: An graphical editor for writing and running R code. This will soon become our primary development application.

The following sections have additional information about the purpose of each component, how to install it, and alternative configurations.

If you run into any installation/configuration challenges, please let others know on the slack channel so that others can anticipate the same issues.

1.1 Git

`git` is a version control system that provides a set of commands that allow you to manage changes to written code, particularly when collaborating with other programmers (much more on this in chapter 4). To start, you'll need to download and install the software. If you are on a Mac, `git` should already be installed.

If you are using a Windows machine, this will also install a program called Git Bash, which provides a text-based interface for executing commands on your computer. For alternative/additional Windows command-line tools, see below.

1.1.1 GitHub

GitHub is a website that is used to store copies of computer code that are being managed with `git` (think "Imgur for code"). Students in the INFO 201 course will use GitHub to turn in programming assignments.

In order to use GitHub, you'll need to create a free GitHub account, if you don't already have one. You should register a username that is identifiable as you (e.g., based on your name or your UW NetID). This will make it easier for others to determine out who contributed what code, rather than needing to figure out who 'LeetDesigner2099' is. This can be the start of a professional account you may use for the rest of your career!

1.2 Command-line Tools (Bash)

The command-line provides a text-based interface for giving instructions to your computer (much more on this in chapter 2). With this book, you'll largely use the command-line for navigating your computer's file structure, and executing commands that allows you to keep track of changes to the code you write (i.e., version control with `git`).

In order to use the command-line, you will need to use a **command shell** (a.k.a. a *command prompt*). This is a program that provides the interface to type commands into. In particular, we'll be working with the Bash shell, which provides a particular common set of commands common to Mac and Linux machines.

1.2.1 Command-line on a Mac

On a Mac you'll want to use the built-in app called **Terminal**. You can open it by searching via Spotlight (hit Cmd (⌘) and Spacebar together, type in "terminal", then select the app to open it), or by finding it in the Applications/Utilities folder.

1.2.2 Command-line on Windows

On Windows, we recommend using **Git Bash**, which you should have installed along with `git` (above). Open this program to open the command-shell. This works great, since you'll primarily be using the command-line for performing version control.

- Note that Windows does come with its own command-prompt, called the *DOS Prompt*, but it has a different set of commands and features. *Powershell* is a more powerful version of the DOS prompt if you really want to get into the Windows Management Framework. But Bash is more common in open-source programming like we'll be doing, and so we will be focusing on that set of commands.

Alternatively, the 64-bit Windows 10 Anniversary update (August 2016) *does* include a beta version of an integrated Bash shell. You can access this by enabling the subsystem for Linux and then running `bash` in the command prompt. This is currently (May 2017) "beta" technology, but will suffice for our purposes if you can get it running.

1.3 Text Editors

In order to produce computer code, you need somewhere to write it (and we don't want to write it in MS Word!). There are a variety of available programs that provide an interface for editing code. A major advantage of these programs is that they provide automatic formatting/coloring for easier interpretation of the code, along with cool features like auto-completion and integration with version control.

RStudio has a great built-in text editor, but you'll sometimes want to use another text editor which is lighter weight (e.g., runs faster), more robust, or supports a different programming language. There are lots of different text editors out there, all of which have slightly different appearances and features. You only need to download and use one of the following programs (we recommend

Atom as a default), but feel free to try out different ones to find something you like (and then evangelize about it to your friends!)

1.3.1 Atom

Atom is a text editor built by the folks at GitHub and has been gaining in popularity. As an open source project, people are continually building (and making available) interesting/useful extensions. Its built-in spell-check is a great feature, especially for documents that require lots of written text. It also has excellent support for Markdown, a markup language you'll be using regularly in this course.

Click the "Download" button to download the installer `.exe` file, then double-click on that to install the application.

Once you've installed Atom, the trick to using it effectively is to get comfortable with the Command Palette. If you hit `Cmd+Shift+P`, Atom will open a small window where you can search for whatever you want the editor to do. For example, if you type in `markdown` you can get list of commands related to Markdown files (including the ability to open up a preview).

For more information about using Atom, see the manual.

1.3.2 Visual Studio Code

Visual Studio Code (or VS Code; not to be confused with Visual Studio) is a free, open-source editor developed by Microsoft—yes, really. While it focuses on web programming and JavaScript, it readily supports lots of languages including Markdown and R and provides a number of extensions for adding even more features. It has a similar *command palette* to Atom, but isn't quite as nice for editing Markdown specifically. Although fairly new, it is updated regularly and has become one of my main editors for programming.

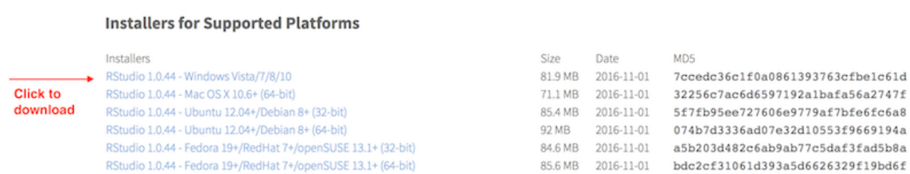
1.3.3 SublimeText

SublimeText is a very popular text editor with excellent defaults and a variety of available extensions (though you'll need to manage and install extensions to achieve the functionality offered by other editors out of the box). While the software can be used for free, every 20 or so saves it will prompt you to purchase the full version. This is my application of choice for when just want to write a plain text file.

1.4 RStudio

The primary programming language you will use throughout the course is `R`. It's a very powerful statistical programming language that is built to work well with large and diverse datasets. While you are able to execute `R` scripts without an interface, the **RStudio** application provides a wonderful way to engage with the `R` language.

To install the RStudio program, select the **installer** for your operating system from the downloads page. Make sure to download the ***free*** version:



Installers for Supported Platforms

Installers	Size	Date	MD5
RStudio 1.0.44 - Windows Vista/7/8/10	81.9 MB	2016-11-01	7ccedc36c1f0a0861393763cfbe1c61d
RStudio 1.0.44 - Mac OS X 10.6+ (64-bit)	71.1 MB	2016-11-01	32256c7ac6d6597192a1bafa56a2747f
RStudio 1.0.44 - Ubuntu 12.04+/Debian 8+ (32-bit)	85.4 MB	2016-11-01	5f7fb95ee727606e9779af7bfe6fc6a8
RStudio 1.0.44 - Ubuntu 12.04+/Debian 8+ (64-bit)	92 MB	2016-11-01	074b7d3336ad07e32d10553f9669194a
RStudio 1.0.44 - Fedora 19+/RedHat 7+/openSUSE 13.1+ (32-bit)	84.6 MB	2016-11-01	a5b203d482c6ab9ab77c5daf3fad5b8a
RStudio 1.0.44 - Fedora 19+/RedHat 7+/openSUSE 13.1+ (64-bit)	85.6 MB	2016-11-01	bdc2cf31061d393a5d6626329f19bd6f

Figure 1.1: File to choose for downloading RStudio. Image may not show the latest version.

Once the download completes, double-click on the `.exe` or `.dmg` file to run the installer. Simply follow the steps of the installer, and you should be prepared to use RStudio.

By downloading RStudio, you will also install the R programming language, if it is not already present in your operating system. However, if there are any problems with that you can also install R separately. Click on the download page for your operating system in order to find a link to the installer. On a Mac you're looking for the `.pkg` file, and for Windows you want to look in the `base` section (follow the link to "install R for the first time").

Resources

Links to the recommended software are collected here for easy access:

- git (and Git Bash)
  - GitHub (sign up)
  - optional: Bash on Windows
- Atom
- R
- RStudio

Chapter 2

The Command Line

The **command-line** is an *interface* to a computer—a way for you (the human) to communicate with the machine. But unlike common graphical interfaces that use windows, icons, menus, and pointers, the command-line is *text-based*: you type commands instead of clicking on icons. The command-line lets you do everything you'd normally do by clicking with a mouse, but by typing in a manner similar to programming!

The command-line is not as friendly or intuitive as a graphical interface: it's much harder to learn and figure out. However, it has the advantage of being both more powerful and more efficient in the hands of expert users. (It's faster to type than to move a mouse, and you can do *lots* of "clicks" with a single command). Thus all professional developers interact with the command-line, particularly when working with large amounts of data or files.

This chapter will give you a brief introduction to basic tasks using the command-line: enough to get you comfortable navigating the interface and able to interpret commands.

2.1 Accessing the Command-Line

In order to use the command-line, you will need to open a **command shell** (a.k.a. a *command prompt*). This is a program that provides the interface to type commands into. You should have installed a command shell (hereafter "the terminal") as part of setting up your machine.

Once you open up the shell (Terminal or Git Bash), you should see something like this (red notes are added):

This is the textual equivalent of having opened up Finder or File Explorer and having it show you the user's "Home" folder. The text shown lets you know:

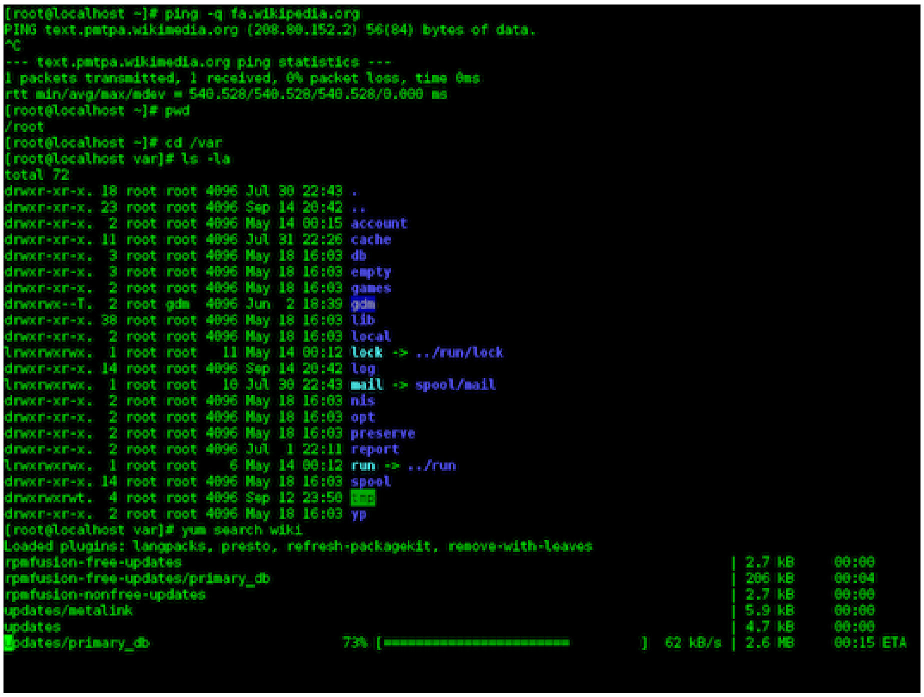


Figure 2.1: An example of the command-line in action (from Wikipedia).

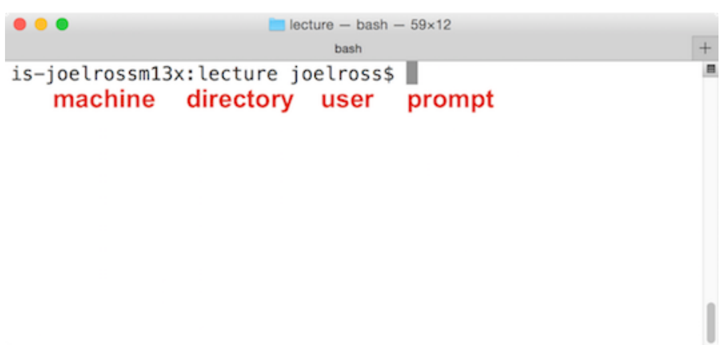


Figure 2.2: A newly opened command-line.

- What machine you're currently interfacing with (you can use the command-line to control different computers across a network or the Internet).
- What **directory** (folder) you are currently looking at (~ is a shorthand for the "home directory").
- What user you are logged in as.

After that you'll see the prompt, which is where you will type in your commands.

2.2 Navigating the Command Line

Although the command-prompt gives you the name of the folder we're in, you might like more detail about where that folder is. Time to send your first command! At the prompt, type:

```
pwd
```

This stands for **p**rint **w**orking **d**irectory (shell commands are highly abbreviated to make them faster to type), and will tell the computer to print the folder you are currently "in".

Fun fact: technically this command starts a tiny program (app) that does exactly one thing: prints the working directory. When you run a command, you're actually executing a tiny program! And when you run programs (tiny or large) on the command-line, it looks like you're typing in commands.

Folders on computers are stored in a hierarchy: each folder has more folders inside it, which have more folders inside them. This produces a tree structure:

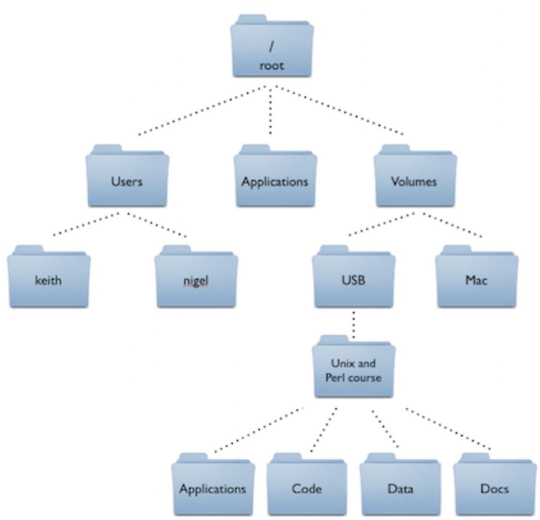


Figure 2.3: A Directory Tree, from Bradnam and Korf.

We describe what folder we are in putting a slash `/` between each folder in the tree: thus `/Users/iguest` means "the `iguest` folder, which is inside the `Users` folder".

At the very top (or bottom, depending on your point of view) is the **root** `/` directory-which has no name, and so is just indicated with that single slash. So `/Users/iguest` really means "the `iguest` folder, which is inside the `Users` folder, which is inside the root folder".

2.2.1 Changing Directories

What if you want to change folders? In a graphical system like Finder, you would just double-click on the folder to open it. But there's no clicking on the command-line.

This includes clicking to move the cursor to an earlier part of the command you typed. You'll need to use the left and right arrow keys to move the cursor instead!

Protip: The up and down arrow keys will let you cycle though your previous commands so you don't need to re-type them!

Since you can't click on a folder, you'll need to use another command:

```
cd folder_name
```

The first word is the **command**, or what you want the computer to do. In this case, you're issuing the command that means **c**hange **d**irectory.

The second word is an example of an **argument**, which is a programming term that means "more details about what to do". In this case, you're providing a *required* argument of what folder you want to change to! (You'll of course need to replace `folder_name` with the name of the folder).

- Try changing to the `Desktop` folder, which should be inside the home folder you started in—you could see it in Finder or File Explorer!
- After you change folders, try printing your currently location. Can you see that it has changed?

2.2.2 Listing Files

In a graphical system, once you've double-clicked on a folder Finder will show you the contents of that folder. The command-line doesn't do this automatically; instead you need another command:

```
ls [folder_name]
```

This command says to **lis**t the folder contents. Note that the *argument* here is in brackets (`[]`) to indicate that it is *optional*. If you just issue the **`ls`** command without an argument, it will list the contents of the current folder. If you include the optional argument (leaving off the brackets), you can "peek" at the contents of a folder you are not currently in.

The command-line can be not great about giving **feedback** for your actions. For example, if there are no files in the folder, then `ls` will simply show nothing, potentially looking like it "didn't work". Or when typing a **password**, the letters you type won't show (not even as `*`) as a security measure.

Just because you don't see any results from your command/typing, doesn't mean it didn't work! Trust in yourself, and use basic commands like `ls` and `pwd` to confirm any changes if you're unsure. Take it slow, one step at a time.

2.2.3 Paths

Note that both the **`cd`** and **`ls`** commands work even for folders that are not "immediately inside" the current directory! You can refer to *any* file or folder on the computer by specifying its **path**. A file's path is "how you get to that file": the list of folders you'd need to click through to get to the file, with each folder separated by a `/`:

`/Users/iguest/Desktop/myfile.txt`

This says to start at the root directory (that initial `/`), then go to `Users`, then go to `iguest`, then to `Desktop`, and finally to the `myfile.txt` file.

Because this path starts with a specific directory (the root directory), it is referred to as an **absolute path**. No matter what folder you currently happen to be in, that path will refer to the correct file because it always starts on its journey from the root.

Contrast that with:

`iguest/Desktop/myfile.txt`

Because this path doesn't have the leading slash, it just says to "go to the Desktop folder *from the current location*". It is known as a **relative path**: it gives you directions to a file *relative to the current folder*. As such, the relative path `iguest/Desktop/myfile.txt` path will only refer to the correct file if you happen to be in the `/Users` folder; if you start somewhere else, who knows where you'll end up!

You should **always** use relative paths, particularly when programming! That way file directions are more likely to work across computers (e.g., in case the username is different, making your home folder `janesmith` instead of `iguest`; with a relative path, `Desktop/myfile.txt` will work for either person).

You can refer to the "current folder" by using a single dot **`.`**. So the command

```
ls .
```

means "list the contents of the current folder" (the same thing you get if you leave off the argument).

If you want to go *up* a directory, you use *two* dots: **..** to refer to the **parent** folder (that is, the one that contains this one). So the command

```
ls ..
```

means "list the contents of the folder that contains the current folder".

Note that **.** and **..** act just like folder names, so you can include them anywhere in paths: `../../my_folder` says to go up two directories, and then into `my_folder`.

Super Protip: Most command shells like Terminal and Git Bash support **tab-completion**. If you type out just the first few letters of a file or folder name and then hit the `tab` key, it will automatically fill in the rest of the name! If the name is ambiguous (e.g., you type `Do` and there is both a `Documents` and a `Downloads` folder), you can hit `tab` *twice* to see the list of matching folders. Then add enough letters to distinguish them and tab to complete! This will make your life better.

Also remember that you can use a tilde **~** as shorthand for the home directory of the current user. Just like `.` refers to "current folder", ~ refers to the user's home directory (usually `/Users/USERNAME`). And of course, you can use the tilde as part of a path as well.

2.3 File Commands

Once you're comfortable navigating folders in the command-line, you can start to use it to do all the same things you would do with Finder or File Explorer, simply by using the correct command:

Command	Behavior
mkdir	**m**a**k**e a **dir**ectory
rm	**r**e**m**ove a file or folder
cp	**c**o**p**y a file from one location to another
open	Mac: opens a file or folder
start	Windows: opens a file or folder
cat	con**cat**enate (combine) file contents and display the results
history	show previous commands executed

Be aware that many of these commands **won't print anything** when you run them. This often means that they worked; they just did so quietly. If it *doesn't* work, you'll know because you'll see a message telling you so (and why, if you read the message). So just because you didn't get any output doesn't mean you

Figure 2.4:

did something wrong—you can use another command (such as **`ls`**) to confirm that the files or folders changed the way you wanted!

2.3.1 Learning New Commands

How can you figure out what kind of arguments these commands take? You can look it up! This information is available online, but many command shells (but *not* Git Bash, unfortunately) also include their own manual you can use to look up commands!

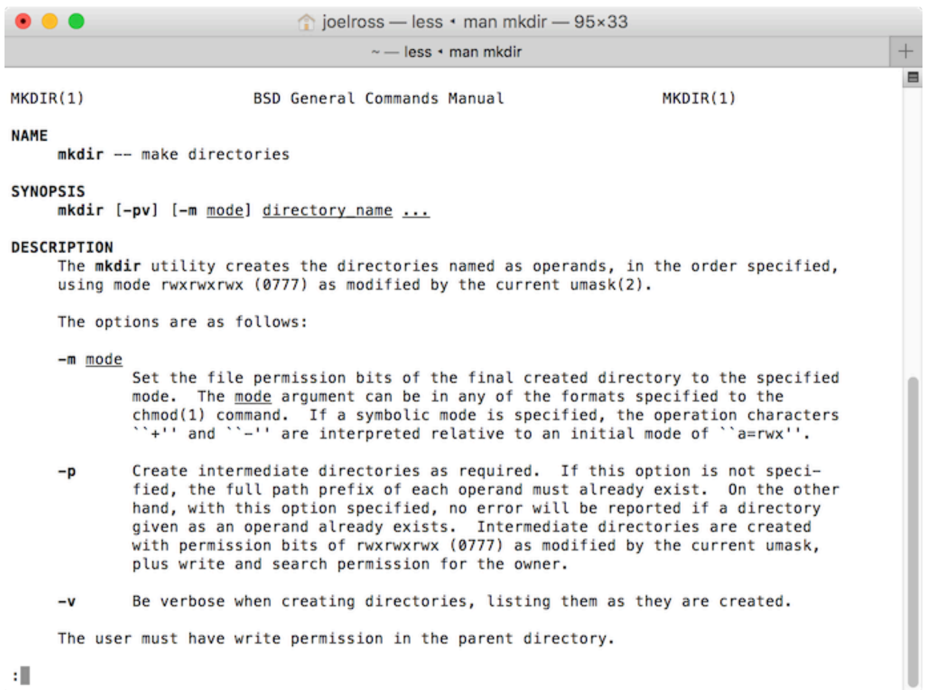
```
man mkdir
```

Will show the **man**ual for the **`mkdir`** program/command.

Because manuals are often long, they are opened up in a command-line viewer called `less`. You can "scroll" up and down by using the arrow keys. Hit the `q` key to **q**uit and return to the command-prompt.

If you look under "Synopsis" you can see a summary of all the different arguments this command understands. A few notes about reading this syntax:

- Recall that anything in brackets `[]` is optional. Arguments that are not in brackets (e.g., `directory_name`) are required.
- **"Options"** (or "flags") for command-line programs are often marked with a leading dash `-` to make them distinct from file or folder names. Options may change the way a command-line program behaves—like how you might set "easy" or "hard" mode in a game. You can either write out



joelross — less ◂ man mkdir — 95×33

~ — less ◂ man mkdir

```
MKDIR(1)                  BSD General Commands Manual                 MKDIR(1)

NAME
     mkdir -- make directories

SYNOPSIS
     mkdir [-pv] [-m mode] directory_name ...

DESCRIPTION
     The mkdir utility creates the directories named as operands, in the order specified,
     using mode rwxrwxrwx (0777) as modified by the current umask(2).

     The options are as follows:

     -m mode
             Set the file permission bits of the final created directory to the specified
             mode.  The mode argument can be in any of the formats specified to the
             chmod(1) command.  If a symbolic mode is specified, the operation characters
             ``+'' and ``-'' are interpreted relative to an initial mode of ``a=rwx''.

     -p      Create intermediate directories as required.  If this option is not speci-
             fied, the full path prefix of each operand must already exist.  On the other
             hand, with this option specified, no error will be reported if a directory
             given as an operand already exists.  Intermediate directories are created
             with permission bits of rwxrwxrwx (0777) as modified by the current umask,
             plus write and search permission for the owner.

     -v      Be verbose when creating directories, listing them as they are created.

     The user must have write permission in the parent directory.

:
```

Figure 2.5: The `mkdir` man page.

each option individually, or combine them: **`mkdir -p -v`** and **`mkdir -pv`** are equivalent.

  - Some options may require an additional argument beyond just indicating a particular operation style. In this case, you can see that the `-m` option requires you to specify an additional `mode` parameter; see the details below for what this looks like.

- Underlined arguments are ones you choose: you don't actually type the word `directory_name`, but instead your own directory name! Contrast this with the options: if you want to use the `-p` option, you need to type `-p` exactly.

Command-line manuals ("man pages") are often very difficult to read and understand: start by looking at just the required arguments (which are usually straightforward), and then search for and use a particular option if you're looking to change a command's behavior.

For practice, try to read the man page for `rm` and figure out how to delete a folder and not just a single file. Note that you'll want to be careful, as this is a good way to break things.

2.4 Dealing With Errors

Note that the syntax of these commands (how you write them out) is very important. Computers aren't good at figuring out what you meant if you aren't really specific; you can't forget spaces or anything.

Try another command: **`echo`** lets you "echo" (print out) some text. Try echoing `"Hello World"` (which is the traditional first computer program):

```
echo "Hello world"
```

What happens if you forget the closing quote? You keep hitting "enter" but you just get that `>` over and over again! What's going on?

- Because you didn't "close" the quote, the shell thinks you are still typing the message you want to echo! When you hit "enter" it adds a *line break* instead of ending the command, and the `>` marks that you're still going. If you finally close the quote, you'll see your multi-line message printed!

IMPORTANT TIP If you ever get stuck in the command-line, hit **`ctrl-c`** (The `control` and `c` keys together). This almost always means "cancel", and will "stop" whatever program or command is currently running in the shell so that you can try again. Just remember: "**`ctrl-c`** to flee".

(If that doesn't work, try hitting the `esc` key, or typing `exit`, `q`, or `quit`. Those commands will cover *most* command-line programs).

Resources

- Learn Enough Command Line to be Dangerous
- Video series: Bash commands
- List of Common Commands (also here)

Chapter 3

Markdown

Markdown syntax provides a simple way to describe the desired formatting of text documents. In fact, this book was written using Markdown! With only a small handful of options, Markdown allows you to format to your text (like making text **bold**, or *italics*), as well as provide structure to a document. There are a number of programs and service that support the *rendering* of Markdown, including GitHub, Slack, and StackOverflow (though note the syntax may vary slightly across programs). In this chapter, you'll learn the basics of Markdown syntax, and how to leverage it to produce readable code documents.

3.1 Writing Markdown

Markdown is a lightweight markup language that is used to format and structure text. It is a kind of "code" that you write in order to *annotate* plain text: it lets the computer know that "this text is bold", "this text is a heading", etc. Compared to other markup languages, Markdown is easy to write and easy to read without getting in the way of the text itself. And because it's so simple to include, it's often used for formatting in web forums and services (like Wikipedia or StackOverflow).

3.1.1 Text Formatting

At its most basic, Markdown is used to declare text formatting options. You do this by adding special symbols (punctuation) *around* the text you wish to "mark". For example, if you want to make text *italiced*, you would surround that text with underscores (`_`) so it looks like `_italicized text_`. You can see how this looks in the below example (code on the left, rendered version on the right):

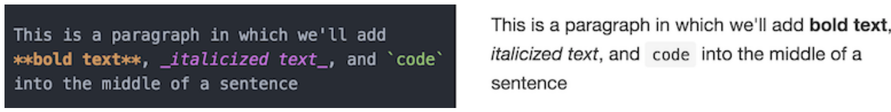


Figure 3.1: Markdown text formatting.

There are a few different ways you can format text:

Syntax	Formatting
`_text_`	*italicized* with underscores
`**text**`	**bolded** with two asterisks
`` `text` ``	inline `code` with backticks

3.1.2 Text Blocks

But Markdown isn't just about adding **bold** and *italics* in the middle of text—it also enables you to create distinct blocks of formatted content (such as a header or a chunk of code). You do this by adding a single symbol in front of the text. Consider the below example:

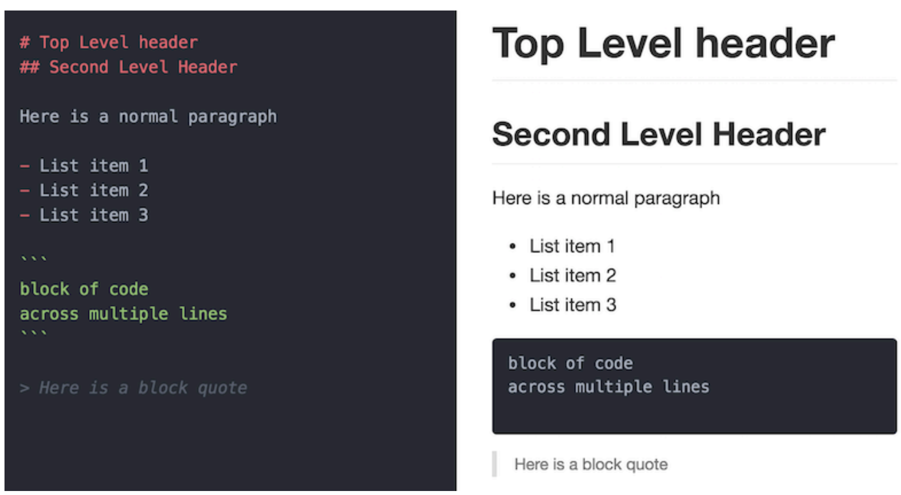


Figure 3.2: Markdown block formatting.

As you can see, the document (right) is produced using the following Markdown shorthand:

Syntax	Formatting
`#`	Header (use `##` for 2nd-level, `###` for 3rd, etc.)
```` ``` ````	Code section (3 back ticks)
`-`	Bulleted/unordered lists (hyphens)
`>`	Block quote

And as you might have guessed from this document, Markdown can even make tables, create hyperlinks, and include images!

For more thorough lists of Markdown options, see the resources linked below.

Note that Slack will allow you to use Markdown as well, though it has slightly different syntax. Luckily, the client gives you hints about what it supports:

Figure 3.3: Markdown in Slack.

3.2 Rendering Markdown

In order to view the *rendered* version of your Markdown-formatted syntax, you need to use a program that converts from Markdown into a formatted document. Luckily, GitHub will automatically render your Markdown files (which end with the **`.md`** extension), and Slack or StackOverflow will automatically format your messages.

However, it can be helpful to preview your rendered Markdown before posting code. The best way to do this is to write your marked code in a text-editor that supports preview rendering, such as **Atom**.

- To preview what your rendered content will look like, simply open a Markdown file (`.md`) in Atom. Then use the command palette (or the shortcut `ctrl-shift-m`) to toggle the **Markdown Preview**. And once this preview is open, it will automatically update to reflect any changes to the text!
- Note that you can also use the command palette to **Toggle Github Style** for the Markdown preview; this will make the rendered preview look the same as it will when uploaded to GitHub!

Other options for rendering Markdown include:

- Many editors (such as Visual Studio Code) include automatic Markdown rendering, or have extensions to provide that functionality.
- Stand-alone programs such as Macdown (Mac only) will also do the same work, often providing nicer looking editor windows.
- There are a variety of online Markdown editors that you can use for practice or quick tests. Dillinger is one of the nicer ones, but there are plenty of others if you're looking for something more specific.

- There are also a number of Google Chrome Extensions that will render Markdown files for you. For example, Markdown Reader, provides a simple rendering of a Markdown file (note it may differ slightly from the way GitHub would render the document). Once you've installed the Extension, you can drag-and-drop a `.md` file into a blank Chrome tab to view the formatted document. Double-click to view the raw code.

Resources

- Original Markdown Source
- GitHub Markdown Basics
- Slack Markdown
- StackOverflow Markdown

Chapter 4

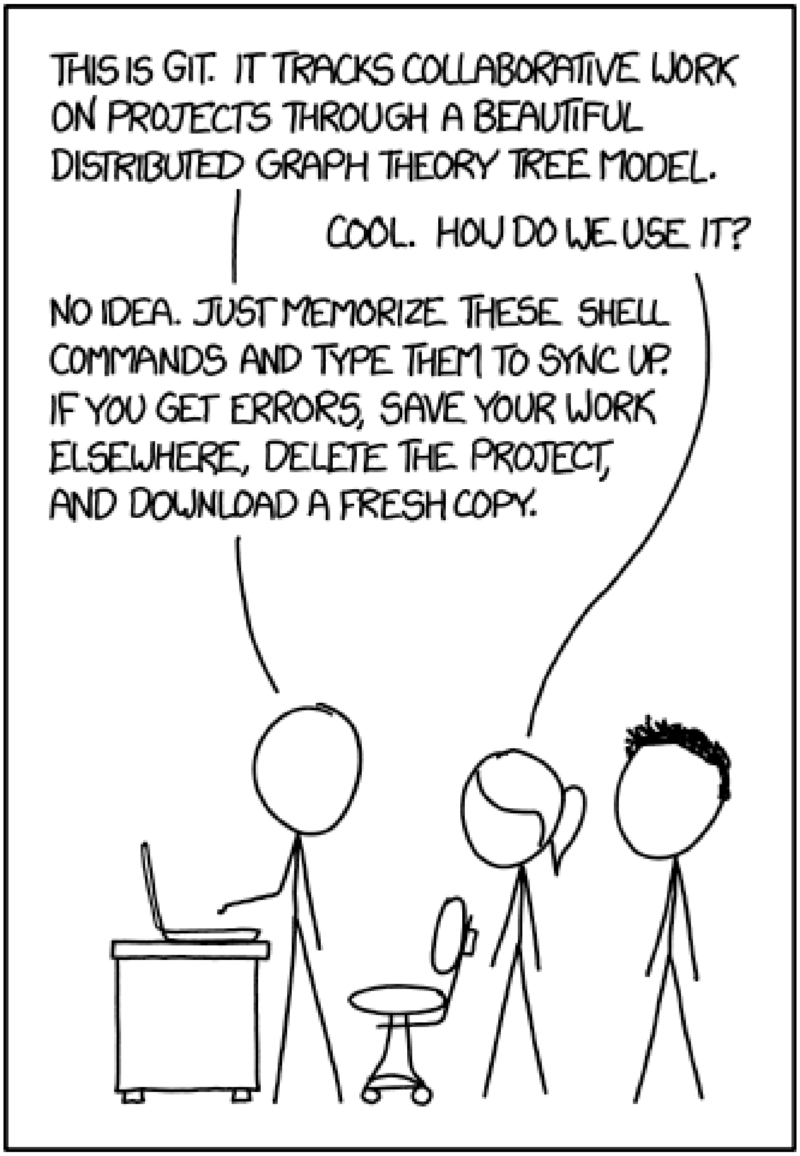
Git and GitHub

A frightening number of people still email their code to each other, have dozens of versions of the same file, and lack any structured way of backing up their work for inevitable computer failures. This is both time consuming and error prone.

And that is why they should be using **git**.

This chapter will introduce you to `git` command-line program and the GitHub cloud storage service, two wonderful tools that track changes to your code (`git`) and facilitate collaboration (GitHub). Git and GitHub are the industry standards for the family of tasks known as **version control**. Being able to manage changes to your code and share it with others is one of the most important technical skills a programmer can learn, and is the focus of this (lengthy) chapter.

4.1 What is this *git* thing anyway?



Git is an example of a **version control system**. Eric Raymond defines version control as

> A version control system (VCS) is a tool for managing a collection of program code that provides you with three important capabilities: **reversibility**, **concurrency**, and **annotation**.

Version control systems work a lot like Dropbox or Google Docs: they allow multiple people to work on the same files at the same time, to view and "roll back" to previous versions. However, systems like git different from Dropbox in a couple of key ways:

1. New versions of your files must be explicitly "committed" when they are ready. Git doesn't save a new version every time you save a file to disk. That approach works fine for word-processing documents, but not for programming files. You typically need to write some code, save it, test it, debug, make some fixes, and test again before you're ready to save a new version.
2. For text files (which most all programming files are), git tracks changes *line-by-line*. This means it can easily and automatically combine changes from multiple people, and gives you very precise information what what lines of code changes.

Like Dropbox and Google Docs, git can show you all previous versions of a file and can quickly rollback to one of those previous versions. This is often helpful in programming, especially if you embark on making a massive set of changes, only to discover part way through that those changes were a bad idea (we speak from experience here).

But where git really comes in handy is in team development. Almost all professional development work is done in teams, which involves multiple people working on the same set of files at the same time. Git helps the team coordinate all these changes, and provides a record so that anyone can see how a given file ended up the way it did.

There are a number of different version control systems in the world, but git is the de facto standard—particularly when used in combination with the cloud-based service GitHub.

4.1.1 Git Core Concepts

To understand how git works, you need to understand its core concepts. Read this section carefully, and come back to it if you forget what these terms mean.

- **repository (repo)** A database containing all the committed versions of all your files, along with some additional metadata, stored in a hidden subdirectory named `.git` within your project directory. If you want to sound cool and in-the-know, call this a "repo."
- **commit** A set of file versions that have been added to the repository (saved in the database), along with the name of the person who did the commit, a message describing the commit, and a timestamp. This extra tracking information allows you to see when, why, and by whom changes were made to a given file. Committing a set of changes creates a "snapshot" of what that work looks like at the time—it's like saving the files, but more so.

- **remote** A link to a copy of this same repository on a different machine. Typically this will be a central version of the repository that all local copies on your various development machines point to. You can push (upload) commits to, and pull (download) commits from, a remote repository to keep everything in sync.
- **merging** Git supports having multiple different versions of your work that all live side by side (in what are called **branches**), whether those versions are created by one person or many collaborators. Git allows the commits saved in different versions of the code to be easily *merged* (combined) back together without you needing to manually copy and paste different pieces of the code. This makes it easy to separate and then recombine work from different developers.

4.1.2 Wait, but what is GitHub then?

Git was made to support completely decentralized development, where developers pull commits from each other's machines directly. But most professional teams take the approach of creating one central repository on a server that all developers push to and pull from. This repository contains the authoritative version the source code, and all deployments to the "rest of the world" are done by downloading from this centralized repository.

Teams can setup their own servers to host these centralized repositories, but many choose to use a server maintained by someone else. The most popular of these in the open-source world is GitHub. In addition to hosting centralized repositories, GitHub also offers other team development features, such as issue tracking, wiki pages, and notifications. Public repositories on GitHub are free, but you have to pay for private ones.

In short: GitHub is a site that provides as a central authority (or clearing-house) for multiple people collaborating with git. Git is what you use to do version control; Github is one possible place where repositories of code can be stored.

4.2 Installation & Setup

This chapter will walk you through all the commands you'll need to do version control with git. It is written as a "tutorial" to help you practice what you're reading!

If you haven't yet, the first thing you'll need to do is install git. You should already have done this as part of setting up your machine.

You'll need configure the installation, telling git who you are so you can commit changes to a repository. You can do this by using the `git` command with the `config` option (e.g., running the `git config` command):

```
# enter your full name (without the dashes)
git config --global user.name "your-full-name"

# enter your email address (the one associated with your GitHub account)
git config --global user.email "your-email-address"
```

Setting up an SSH key for GitHub on your own machine is also a huge time saver; just follow the instructions on that page.

4.2.1 Creating a Repo

The first thing you'll need in order to work with git is to create a **repository**. A repository acts as a "database" of changes that you make to files in a directory.

In order to have a repository, you'll need to have a directory of files. Create a new folder `git_practice` on your computer's Desktop. Since you'll be using the command-line for this course, you might as well practice creating a new directory with that:

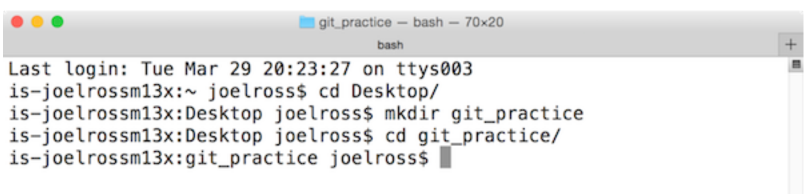


Figure 4.1: Making a folder with the command-line.

You can turn this directory *into* a repository by telling the `git` program to run the `init` action:

```
# run IN the directory of project (make sure pwd is correct!)
git init
```

This creates a new *hidden* folder called `.git` inside of the current directory (it's hidden so you won't see it in Finder, but if you use `ls -a` (list with the **a**ll option) you can see it there). This folder is the "database" of changes that you will make—git will store all changes you commit in this folder. The presence of the `.git` folder causes that directory to become a repository; we refer to the whole directory as the "repo" (an example of synechdoche).

- Note that because a repo is a single folder, you can have lots of different repos on your machine. Just make sure that they are in separate folders; folders that are *inside* a repo are considered part of that repo, and trying to treat them as a separate repository causes unpleasantness. **Do not put one repo inside of another!**

4.2.2 Checking Status

Now that you have a repo, the next thing you should do is check its **status**:

```
git status
```

The `git status` command will give you information about the current "state" of the repo. For example, running this command tells us a few things:

- That you're actually in a repo (otherwise you'll get an error)
- That you're on the `master` branch (think: line of development)
- That you're at the initial commit (you haven't committed anything yet)
- That currently there are no changes to files that you need to commit (save) to the database
- *What to do next!*

That last point is important. Git status messages are verbose and somewhat awkward to read (this is the command-line after all), but if you look at them carefully they will almost always tell you what command to use next.

If you are ever stuck, use `git status` to figure out what to do next!

This makes `git status` the most useful command in the entire process. Learn it, use it, love it.

4.3 Making Changes

Since `git status` told you to create a file, go ahead and do that. Using your favorite editor, create a new file `books.md` inside the repo directory. This Markdown file should contain a *list* of 3 of your favorite books. Make sure you save the changes to your file to disk (to your computer's harddrive)!

4.3.1 Adding Files

Run `git status` again. You should see that git now gives a list of changed and "untracked" files, as well as instructions about what to do next in order to save those changes to the repo's database.

The first thing you need to do is to save those changes to the **staging area**. This is like a shopping cart in an online store: you put changes in temporary storage before you commit to recording them in the database (e.g., before hitting "purchase").

We add files to the staging area using the `git add` command:

```
git add filename
```

(Replacing `filename` with the name/path of the file/folder you want to add). This will add a single file *in its current saved state* to the staging area. If you change the file later, you will need to re-add the updated version.

You can also add all the contents of the directory (tracked or untracked) to the staging area with:

```
git add .
```

(This is what I tend to use, unless I explicitly don't want to save changes to some files.)

Add the `books.md` file to the staging area. And of course, now that you've changed the repo (you put something in the staging area), you should run `git status` to see what it says to do. Notice that it tells you what files are in the staging area, as well as the command to *unstage* those files (remove them from the "cart").

4.3.2 Committing

When you're happy with the contents of your staging area (e.g., you're ready to purchase), it's time to **commit** those changes, saving that snapshot of the files in the repository database. We do this with the `git commit` command:

```
git commit -m "your message here"
```

The `"your message here"` should be replaced with a short message saying what changes that commit makes to the repo (see below for details).

WARNING: If you forget the `-m` option, git will put you into a command-line *text editor* so that you can compose a message (then save and exit to finish the commit). If you haven't done any other configuration, you might be dropped into the ***vim*** editor. Type **`:q`** (**colon** then **q**) and hit enter to flee from this horrid place and try again, remembering the `-m` option! Don't panic: getting stuck in *vim* happens to everyone.

4.3.2.1 Commit Message Etiquette

Your commit messages should be informative about what changes the commit is making to the repo. `"stuff"` is not a good commit message. `"Fix critical authorization error"` is a good commit message.

Commit messages should use the **imperative mood** (`"Add feature"` not `"added feature"`). They should complete the sentence:

> If applied, this commit will **{your message}**

Other advice suggests that you limit your message to 50 characters (like an email subject line), at least for the first line—this helps for going back and looking at previous commits. If you want to include more detail, do so after a blank line.

A specific commit message format may also be required by your company or project team. See this post for further consideration of good commit messages.

Finally, be sure to be professional in your commit messages. They will be read by your professors, bosses, coworkers, and other developers on the internet. Don't join this group.

After you've committed your changes, be sure and check `git status`, which should now say that there is nothing to commit!

4.3.3 Commit History

You can also view the history of commits you've made:

```
git log [--oneline]
```

This will give you a list of the *sequence* of commits you've made: you can see who made what changes and when. (The term **HEAD** refers to the most recent commit). The optional `--oneline` option gives you a nice compact version. Note that each commit is listed with its SHA-1 hash (the random numbers and letters), which you can use to identify each commit.

4.3.4 Reviewing the Process

This cycle of "edit files", "add files", "commit changes" is the standard "development loop" when working with git.

In general, you'll make lots of changes to your code (editing lots of files, running and testing your code, etc). Then once you're at a good "break point"—you've got a feature working, you're stuck and need some coffee, you're about to embark on some crazy changes—you will add and commit your changes to make sure you don't lose any work and you can always get back to that point.

4.3.4.1 Practice

For further practice using git, perform the following steps:

1. **Edit** your list of books to include two more books (top 5 list!)
2. **Add** the changes to the staging area
3. **Commit** the changes to the repository

Be sure and check the status at each step to make sure everything works!

4.3.5 The `.gitignore` File

Sometimes you want git to always ignore particular directories or files in your project. For example, if you use a Mac and you tend to organize your files in the Finder, the operating system will create a hidden file in that folder named `.DS_Store` (the leading dot makes it "hidden") to track the positions of icons, which folders have been "expanded", etc. This file will likely be different from machine to machine. If it is added to your repository and you work from multiple machines (or as part of a team), it could lead to a lot of merge conflicts (not to mention cluttering up the folders for Windows users).

You can tell git to ignore files like these by creating a special *hidden* file in your project directory called `.gitignore` (note the leading dot). This file contains a *list* of files or folders that git should "ignore" and pretend don't exist. The file uses a very simple format: each line contains the path to a directory or file to ignore; multiple files are placed on multiple lines. For example:

```
# This is an example .gitignore file

# Mac system file; the leading # marks a comment
.DS_Store

# example: don't check in passwords or ssl keys!
secret/my_password.txt

# example: don't include large files or libraries
movies/my_four_hour_epic.mov
```

Note that the easiest way to create the `.gitignore` file is to use your preferred text editor (e.g., Atom); select `File > New` from the menu and choose to make the `.gitignore` file *directly inside* your repo.

4.4 GitHub and Remotes

Now that you've gotten the hang of git, let's talk about GitHub. GitHub is an online service that stores copies of repositories in the cloud. These repositories can be *linked* to your **local** repositories (the one on your machine, like you've been working with so far) so that you can synchronize changes between them.

- The relationship between git and GitHub is the same as that between your camera and Imgur: **git** is the program we use to create and manage repositories; GitHub is simply a website that stores these repositories. So we use git, but upload to/download from GitHub.

Repositories stored on GitHub are examples of **remotes**: other repos that are linked to your local one. Each repo can have multiple remotes, and you can synchronize commits between them.

Each remote has a URL associated with it (where on the internet the remote copy of the repo can be found), but they are given "alias" names (like browser bookmarks). By convention, the remote repo stored on GitHub's servers is named **`origin`**, since it tends to be the "origin" of any code you've started working on.

Remotes don't need to be stored on GitHub's computers, but it's one of the most popular places to put repos.

4.4.1 Forking and Cloning

In order to use GitHub, you'll need to **create a free GitHub account**, which you should have done as part of setting up your machine.

Next, you'll need to download a copy of a repo from GitHub onto your own machine. **Never make changes or commit directly to GitHub**: all development work is done locally, and changes you make are then uploaded and *merged* into the remote.

Start by visiting **https://github.com/info201/github_practice**. This is the web portal for an existing repository. You can see that it contains one file (`README.md`, a Markdown file with a description of the repo) and a folder containing a second file. You can click on the files and folder to view their source online, but again you won't change them there!

Just like with Imgur or Flickr or other image-hosting sites, each GitHub user has their own account under which repos are stored. The repo linked above is under the course book account (`info201`). And because it's under our user account, you won't be able to modify it—just like you can't change someone else's picture on Imgur. So the first thing you'll need to do is copy the repo over to *your **own** account on GitHub's servers*. This process is called **forking** the repo (you're creating a "fork" in the development, splitting off to your own version).

- To fork a repo, click the **"Fork"** button in the upper-right of the screen:

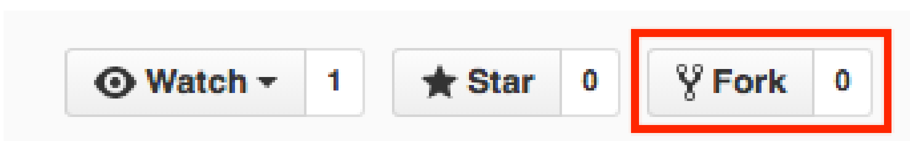


Figure 4.2: The fork button on GitHub's web portal.

  This will copy the repo over to your own account, so that you can upload and download changes to it!

  Students in the INFO 201 course will be forking repos for class and lab execises, but *not* for homework assignments (see below)

Now that you have a copy of the repo under your own account, you need to download it to your machine. We do this by using the `clone` command:

```
git clone [url]
```

This command will create a new repo (directory) *in the current folder*, and download a copy of the code and all the commits from the URL you specify.

- You can get the URL from the address bar of your browser, or you can click the green "Clone or Download" button to get a popup with the URL. The little icon will copy the URL to your clipboard. **Do not** click "Open in Desktop" or "Download Zip".
- Make sure you clone from the *forked* version (the one under your account!)

  Warning also be sure to `cd` out of the `git_practice` directory; you don't want to `clone` into a folder that is already a repo; you're effectively creating a *new* repository on your machine here!

Note that you'll only need to `clone` once per machine; `clone` is like `init` for repos that are on GitHub—in fact, the `clone` command *includes* the `init` command (so you do not need to init a cloned repo).

4.4.2 Pushing and Pulling

Now that you have a copy of the repo code, make some changes to it! Edit the `README.md` file to include your name, then `add` the change to the staging area and `commit` the changes to the repo (don't forget the `-m` message!).

Although you've made the changes locally, you have not uploaded them to GitHub yet—if you refresh the web portal page (make sure you're looking at the one under your account), you shouldn't see your changes yet.

In order to get the changes to GitHub, you'll need to `push` (upload) them to GitHub's computers. You can do this with the following command:

```
git push origin master
```

This will push the current code to the `origin` remote (specifically to its `master` branch of development).

- When you cloned the repo, it came with an `origin` "bookmark" to the original repo's location on GitHub!

Once you've **pushed** your code, you should be able to refresh the GitHub webpage and see your changes to the README!

If you want to download the changes (commits) that someone else made, you can do that using the `pull` command, which will download the changes from GitHub and *merge* them into the code on your local machine:

```
git pull
```

Because you're merging as part of a `pull`, you'll need to keep an eye out for **merge conflicts**! These will be discussed in more detail in chapter 14.

Pro Tip: always `pull` before you `push`. Technically using `git push` causes a merge to occur on GitHub's servers, but GitHub won't let you push if that merge might potentially cause a conflict. If you `pull` first, you can make sure your local version is up to date so that no conflicts will occur when you upload.

4.4.3 Reviewing The Process

Overall, the process of using git and GitHub together looks as follows:

4.5 Course Assignments on GitHub

For students in INFO 201: While class and lab work will use the "fork and clone" workflow described above, homework assignments will work slightly differently. Assignments in this course are configured using GitHub Classroom, which provides each student *private* repo (under the class account) for the assignment.

Each assignment description in Canvas contains a link to create an assignment repo: click the link and then **accept the assignment** in order to create your own code repo. Once the repository is created, you should **`clone`** it to your local machine to work. **Do not fork your asssignment repo**.

DO NOT FORK YOUR ASSIGNMENT REPO.

After `cloning` the assignment repo, you can begin working following the workflow described above:

1. Make changes to your files
2. **Add** files with changes to the staging area (`git add .`)
3. **Commit** these changes to take a repo (`git commit -m "commit message"`)
4. **Push** changes back to GitHub (`git push origin master`) to turn in your work.

Repeat these steps each time you reach a "checkpoint" in your work to save it both locally and in the cloud (in case of computer problems).

4.6 Command Summary

Whew! You made it through! This chapter has a lot to take in, but really you just need to understand and use the following half-dozen commands:

- `git status` Check the status of a repo
- `git add` Add file to the staging area
- `git commit -m "message"` Commit changes
- `git clone` Copy repo to local machine
- `git push origin master` Upload commits to GitHub

- `git pull` Download commits from GitHub

Using git and GitHub can be challenging, and you'll inevitably run into issues. While it's tempting to ignore version control systems, **they will save you time** in the long-run. For now, do your best to follow these processes, and read any error messages carefully. If you run into trouble, try to understand the issue (Google/StackOverflow), and don't hesitate to ask for help.

Resources

- Git and GitHub in Plain English
- Atlassian Git Tutorial
- Try Git (interactive tutorial)
- GitHub Setup and Instructions
- Official Git Documentation
- Git Cheat Sheet

Chapter 5

Introduction to R

R is an extraordinarily powerful open-source software program built for working with data. It is one of the most popular tools for performing advanced data techniques, including statistical analysis, machine learning, and data visualization. R is be the primary programming language for this course, which will help you develop a strong understanding of how to leverage the power of R.

5.1 Programming with R

R is a **statistical programming language** that allows you to write code to work with data. It is an **open-source** programming language, which means that it is free and continually improved upon by the R community. The R language has a number of functionalities that allow you to read, analyze, and visualize datasets.

- *Fun Fact:* R is called “R” because it was inspired by and comes after the language “S”, a language for **S**tatistics developed by AT&T.

So far you’ve leveraged formal language to give instructions to your computers, such as by writing syntactically-precise instructions at the command-line. Programming in R will work in a similar manner: you will write instructions using R’s special language and syntax, which the computer will **interpret** as instructions for how to work with data.

However, as projects grow in complexity, it will become useful if you can write down all the instructions in a single place, and then order the computer to *execute* all of those instructions at once. This list of instructions is called a **script**. Executing or “running” a script will cause each instruction (line of code) to be run *in order, one after the other*, just as if you had typed them in one by one. Writing scripts allows you to save, share, and re-use your work—by saving instructions in a file, you can easily check, change, and re-execute the list of instructions as you figure out how to use data to answer questions. And because R is an *interpreted* language rather than a *compiled* language like Java, R programming environments will also give you the ability to execute each individual line of code in your script if you desire (though this will become cumbersome as projects become large).

As you begin working with data in R, you will be writing multiple instructions (lines of code) and saving them in files with the **`.R`** extension, representing R scripts. You can write this R code in any text editor (such as Atom), but we recommend you usually use a program called **RStudio** which is specialized for writing and running R scripts.

5.2 Running R Scripts

R scripts (programs) are just a sequence of instructions, and there are a couple of different ways in which we can tell the computer to execute these instructions.

5.2.1 Command-Line

It is possible to issue R instructions (run lines of code) one-by-one at the command-line by starting an **interactive R session** within your terminal. This will allow you to type R code directly into the terminal, and your computer will interpret and execute each line of code (if you just typed R syntax directly into the terminal, your computer wouldn't understand it).

With R installed, you can start an interactive R session on a Mac by typing `R` into the terminal (to run the `R` program), or on Windows by running the "R" desktop app program. This will start the session and provide you with lots of information about the R language:

Notice that this description also include *instructions on what to do next*—most importantly `"Type 'q()' to quit R."`.

Always read the output when working on the command-line!

Once you've started running an interactive R session, you can begin entering one line of code at a time at the prompt (`>`). This is a nice way to experiment with the R language or to quickly run some code.

- Note that RStudio also provides an interactive console that provides the exact same functionality.

It is also possible to run entire scripts from the command-line by using the `RScript` program, specifying the `.R` file you wish to execute:

- On Windows, you would need to find the **absolute path** to the `RScript.exe` program *on your particular machine* and use that.

  This can be tricky; we recommend you just use RStudio instead.

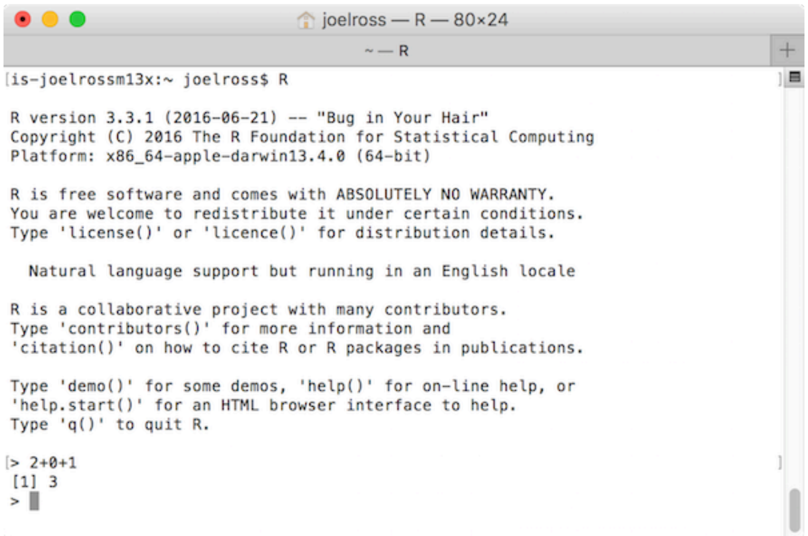


Figure 5.1: An interactive R session running in the terminal.

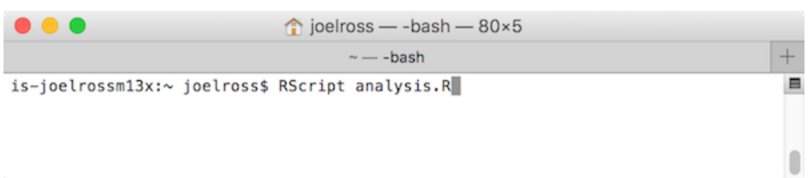


Figure 5.2: Using RScript from the terminal

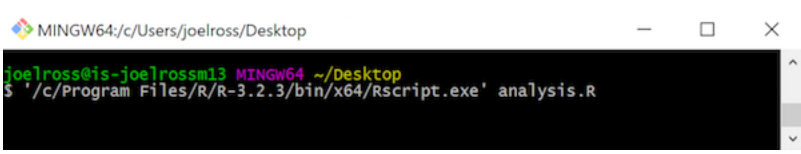


Figure 5.3: Using RScript from a Windows shell

5.2.2 RStudio

RStudio is an open-source **integrated development environment (IDE)** that provides an informative user interface for interacting with the R interpreter. IDEs provide a platform for writing *and* executing code, including viewing the results of the code you have run. If you haven't already, make sure to download and install the free version of RStudio.

When you open the RStudio program (either by searching for it, or double-clicking on a desktop icon), you'll see the following interface:

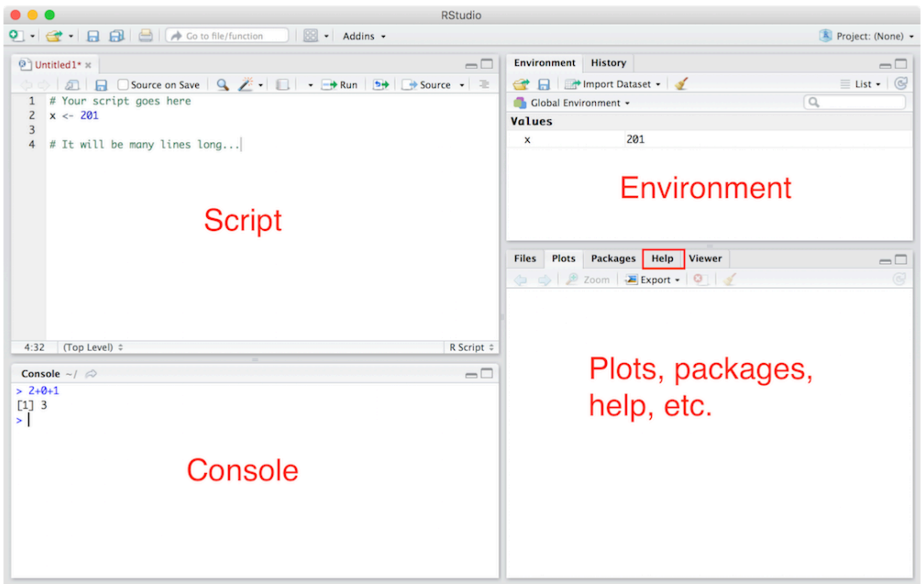


Figure 5.4: RStudio's user interface. Annotations are in red.

A RStudio session usually involves 4 sections ("panes"), though you can customize this layout if you wish:

- **Script**: The top-left pane is a simple text editor for writing your R code. While it is not as robust as a text editing program like Atom, it will colorize code, "auto-complete" text, and allows you to easily execute your code. Note that this pane is hidden if there are no open scripts; select `File > New File > R Script` from the menu to create a new script file.

  In order to execute (run) the code you write, you have two options:

  1. You can execute a section of your script by selecting (highlighting) the desired code and pressing the "Run" button (keyboard shortcut: `ctrl` and `enter`). If no lines are selected, this will run the line

currently containing the cursor. This is the most common way to execute code in R.

- *Protip:* use `cmd + a` to select the entire script!

2. You can execute an entire script by using the `Source` command to treat the current file as the "source" of code. Press the "Source" button (hover the mouse over it for keyboard shortcuts) to do so. If you check the "Source on save" option, your entire script will be executed every time you save the file (this may or may not be appropriate, depending on the complexity of your script and its output).

- **Console**: The bottom-left pane is a console for entering R commands. This is identical to an inetractive session you'd run on the command-line, in which you can type and execute one line of code at a time. The console will also show the printed results from executing the code you execute from the Script pane.
  - *Protip:* just like with the command-line, you can **use the up arrow** to easily access previously executed lines of code.
- **Environment**: The top-right pane displays information about the current R environment—specifically, information that you have stored inside of *variables* (see below). In the above example, the value `201` is stored in a variable called `x`. You'll often create dozens of variables within a script, and the Environment pane helps you keep track of which values you have stored in what variables. *This is incredibly useful for debugging!*
- **Plots, packages, help, etc.**: The bottom right pane contains multiple tabs for accessing various information about your program. When you create visualizations, those plots will render in this quadrant. You can also see what packages you've loaded or look up information about files. *Most importantly*, this is also where you can access the official documentation for the R language. If you ever have a question about how something in R works, this is a good place to start!

Note, you can use the small spaces between the quadrants to adjust the size of each area to your liking. You can also use menu options to reorganize the panes if you wish.

5.3 Comments

Before discussing how to program with R, we need to talk about a piece of syntax that lets you comment your code. In programming, **comments** are bits of text that are *not interpreted as computer instructions*—they aren't code, they're just notes about the code! Since computer code can be opaque and difficult to understand, we use comments to help write down the meaning and *purpose* of our code. While a computer is able to understand the code, comments are there to help *people* understand. This is particularly imporant when someone else will be looking at your work—whether that person is a collaborator, or is simple a future version of you (e.g., when you need to come back and fix something and so need to remember what you were even thinking).

Comments should be clear, concise, and helpful—they should provide information that is not otherwise present or "obvious" in the code itself.

In R, we mark text as a comment by putting it after the pound/hashtag symbol (`#`). Everything from the `#` until the end of the line is a comment. We put descriptive comments *immediately above* the code it describes, but you can also put very short notes at the end of the line of code (preferably following two spaces):

```r
# Set how many bottles of beer are on the wall
bottles <- 99 - 1  # 98 bottles
```

(You may recognize this `#` syntax and commenting behavior from the commmand-line and git chapters. That's because the same syntax is used in a Bash shell!)

5.4 Variables

Since computer programs involve working with lots of *information*, we need a way to store and refer to this information. We do this with **variables**. Variables are labels for information; in R, you can think of them as "boxes" or "nametags" for data. After putting data in a variable box, you can then refer to that data by the name on the box.

Variable names can contain any combination of letters, numbers, periods (`.`), or underscores (`_`). Variables names must begin with a letter. Note that like everything in programming, variable names are case sensitive. It is best practice to make variable names descriptive and information about what data they contain. `a` is not a good variable name. `cups.of.coffee` is a good variable name. To comply with Google's Style Guidelines variables should be **all lower-case letters, separated by periods (`.`)**.

We call putting information in a variable **assigning** that value to the variable. We do this using the *assignment operator* `<-`. For example:

```r
# Stores the number 7 into a variable called shoe.size
shoe.size <- 7
```

- *Notice:* variable name goes on the left, value goes on the right!

You can see what value (data) is inside a variable by either typing that variable name as a line of code, or by using R's built-in `print()` function (more on functions later):

```r
print(shoe.size)
# [1] 7
```

- We'll talk about the `[1]` in that output later.

You can also use **mathematical operators** (e.g., `+`, `-`, `/`, `*`) when assigning values to variables. For example, you could create a variable that is the sum of two numbers as follows:

```r
x <- 3 + 4
```

Once a value (like a number) is *in* a variable, you can use that variable in place of any other value. So all of the following are valid:

```r
x <- 2  # store 2 in x
y <- 9  # store 9 in y
z <- x + y  # store sum of x and y in z
print(z)  # 11
z <- z + 1  # take z, add 1, and store result back in z
print(z)  # 12
```

5.4.1 Basic Data Types

In the example above, we stored **numeric** values in variables. R is a **dynamically typed language**, which means that we *do not* need to explicitly state what type of information will be stored in each variable we create. R is intelligent enough to understand that if we have code `x <- 7`, then `x` will contain a numeric value (and so we can do math upon it!)

There are a few "basic types" (or *modes*) for data in R:

- **Numeric**: The default computational data type in R is numeric data, which consists of the set of real numbers (including decimals). We use use **mathematical operators** on numeric data (such as `+`, `-`, `*`, `-`, etc.). There are also numerous functions that work on numeric data (such as calculating sums or averages).

- **Character**: Character data stores *strings* of characters (things you type with a keyboard) in a variable. You specify that some information is character data by surrounding it in either single quotes (`'`) or double quotes (`"`).

  ```r
  # Create character variable `famous.poet` with the value "Bill Shakespeare"
  famous.poet <- "Bill Shakespeare"
  ```

  Note that character data *is still data*, so it can be assigned to a variable just like numeric data!

  There are no special operators for character data, though there are a many built-in functions for working with strings.

- **Logical**: Logical (a.k.a Boolean) data types store "yes-or-no" data. A logical value can be one of two values: `TRUE` or `FALSE`. Importantly, these **are not** the strings `"TRUE"` or `"FALSE"`; logical values are a different type! If you prefer, you can use the shorthand `T` or `F` in lieu of `TRUE` and `FALSE` in variable assignment.
  - *Fun fact:* logical values are called "booleans" after mathematician and logician George Boole.

  Logical values are most commonly the result of applying a **relational operator** (also called a **comparison operator**) to some other data. Comparison operators are used to compare values and include: `<` (less than), `>` (greater than), `<=` (less-than-or-equal), `>=` (greater-than-or-equal), `==` (equal), and `!=` (not-equal).

```r
x <- 3
y <- 3.15

# compare numbers
x > y  # returns logical value FALSE (x IS NOT bigger than y)
y != x  # returns logical value TRUE (y IS not-equal to x)

# compare x to pi (built-in variable)
y == pi  # returns logical value FALSE

# compare strings (based on alphabetical ordering)
"cat" > "dog"  # returns FALSE
```

  Logical values have their own operators as well (called **logical operators** or **boolean operators**). These apply to logical values and produce logical values, and allow you to make more complex logical expressions. They include `&` (and), `|` (or), and `!` (not).

```r
x <- 3.1
y <- 3.2
pet <- "dog"
weather <- "rain"

# x is less than pi AND y is greater than pi
x < pi & y > pi  # TRUE

# pet is "cat" OR "dog"
pet == "cat" | pet == "dog"  # TRUE

# pet is "dog" AND NOT weather is "rain"
pet == "dog" & !(weather == "rain")  # FALSE
```

  Note that it's easy to write complex expressions with logical operators. If you find yourself getting lost, I recommend rethinking your question to see if there is a simpler way to express it!

- **Complex**: Complex (imaginary) numbers have their own data storage type in R, are are created using the `i` syntax: `complex.variable <- 2i`. We will not be using complex numbers in this course.
- **Integer**: Integer values are technically a different data type than numeric values because of how they are stored and manipulated by the R interpreter. This is something that you will rarely encounter, but it's good to know that you can specify a number is of integer type rather than general numeric type by placing a capital `L` (for "long integer") after an value in variable assignment (`my.integer <- 10L`).

5.5 Getting Help

As with any programming language, when working in R you will inevitably run into problems, confusing situations, or just general questions. Here are a few ways to start getting help.

1. **Read the error messages**: If there is an issue with the way you have written or executed your code, R will often print out a red error message in your console. Do you best to decipher the message (read it carefully, and think about what is meant by each word in the message), or you can put it directly into Google to get more information. You'll soon get the hang of interpreting these messages if you don't panic when one comes up.
2. **Google**: When you're trying to figure out how to do something, it should be no surprise that Google is often the best resource. Try searching for queries like `"how to <DO THING> in R"`. More frequently than not, your question will lead you to a Q/A forum called StackOverflow (see below), which is a great place to find potential answers.
3. **StackOverflow**: StackOverflow is an amazing Q/A forum for asking/answering programming questions. Indeed, most basic questions have already been asked/answered here. However, don't hesitate to post your own questions to StackOverflow. Be sure to hone in on the specific question you're trying to answer, and provide error messages and sample code. I often find that, by the time I can articulate the question clearly enough to post it, I've figured out my problem anyway.
   - There is a classical method of debugging called rubber duck debugging, which involves simply trying to explain your code/problem to an inanimate object (talking to pets works too). You'll usually be able to fix the problem if you just step back and think about how you would explain it to someone else!
4. **Documentation**: R's documentation is actually quite good. Functions and behaviors are all described in the same format, and often contain a

helpful examples. To search the documentation within R (or in RStudio), simply type `?` followed by the function name you're using (more on functions coming soon). You can also search the documentation by typing two questions marks (`??SEARCH`).

- You can also look up help by using the `help()` function (e.g., `help(print)` will look up information on the `print()` function, just like `?print` does). There is also an `example()` function you can call to see examples of a function in action (e.g., `example(print)`). This will be more important in the next module!
- rdocumentation.org has a lovely searchable and readable interface to the R documentation.

Resources

- Google's R Style Guide
- DataCamp (awesome resource for interactive tutorials in R)
- R Tutorial: Introduction
- R Tutorial: Basic Data Types
- R Tutorial: Operators
- RStudio Keyboard Shortcuts
- R Documentation searchable online documentation
- R for Data Science online textbook
- The Art of R Programming print textbook

Chapter 6

Functions

This chapter will explore how to use **functions** in R to perform advanced capabilities and actually ask questions about data. After considering a function in an abstract sense, it will discuss using built-in R functions, accessing additional functions by loading R packages, and writing your own functions.

6.1 What are functions?

In a broad sense, a **function** is a named sequence of instructions (lines of code) that you may want to perform one or more times throughout a program. They provide a way of *encapsulating* multiple instructions into a single "unit" that can be used in a variety of different contexts. So rather than needing to repeatedly write down all the individual instructions for "make a sandwich" every time you're hungry, you can define a `MakeSandwich()` function once and then just **call** (execute) that function when you want to perform those steps.

In addition to grouping instructions, functions in programming languages like R also tend to follow the mathematical definition of functions, which is a set of operations (instructions!) that are performed on some **inputs** and lead to some **outputs**. Function inputs are called **arguments** or **parameters**, and we say that these arguments are **passed** to a function (like a football). We say that a function then **returns** an ouput to use.

6.1.1 R Function Syntax

R functions are referred to by name (technically, they are values like any other variable). As in many programming languages, we **call** a function by writing the name of the function followed immediately (no space) by parentheses `()`.

Inside the parentheses, we put the **arguments** (inputs) to the function separated by commas (`,`). Thus computer functions look just like multi-variable mathematical functions, but with names longer than `f()`.

```
# call the print() function, pass it "Hello world" value as an argument
print("Hello world")  # "Hello world"

# call the sqrt() function, passing it 25 as an argument
sqrt(25)  # 5, square root of 25

# call the min() function, pass it 1, 6/8, AND 4/3 as arguments
# this is an example of a function that takes multiple args
min(1, 6/8, 4/3)  # 0.75, (6/8 is the smallest value)
```

- *Note:* To keep functions and variables distinct, we try to always include empty parentheses `()` when referring to a function by name. This does not mean that the function takes no arguments, it is just a useful shorthand for indicating that something is a function.

If you call any of these functions interactively, R will display the **returned value** (the output) in the console. However, the computer is not able to "read" what is written in the console—that's for humans to view! If you want the computer to be able to *use* a returned value, you will need to give that value a name so that the computer can refer to it. That is, you need to store the returned value in a variable:

```
# store min value in smallest.number variable
smallest.number <- min(1, 6/8, 4/3)

# we can then use the variable as normal, such as for a comparison
min.is.big <- smallest.number > 1  # FALSE

# we can also use functions directly when storing to variables
phi <- .5 + sqrt(5)/2  # 1.618...

# we can even pass the result of a function as an argument to another!
# watch out for where the parentheses close!
print(min(1.5, sqrt(3)))  # prints 1.5
```

- In the last example, the resulting *value* of the "inner" function (e.g., `sqrt()`) is immediately used as an argument. Because that value is used immediately, we don't have to assign it a separate variable name. It is thus known as an **anonymous variable**.

6.2 Built-in R Functions

As you may have noticed, R comes with a large number of functions that are built into the language. In the above example, we used the `print()` function to print a value to the console, the `min()` function to find the smallest number among the arguments, and the `sqrt()` function to take the square root of a number. Here is a *very* limited list of functions you can experiment with (or see a few more here).

Function Name	Description	Example
`sum(a,b,...)`	Calculates the sum of all input values	`sum(1, 5)` returns `6`
`round(x,digits)`	Rounds the first argument to the given number of digits	`round(3.1415, 3)` returns `3.142`
`toupper(str)`	Returns the characters in uppercase	`toupper("hi there")` returns `"HI THERE"`
`paste(a,b,...)`	*Concatenate* (combine) characters into one value	`paste("hi", "there")` returns `"hi there"`
`nchar(str)`	Counts the number of characters in a string	`nchar("hi there")` returns `8` (space is a character!)
`c(a,b,...)`	*Concatenate* (combine) multiple items into a *vector* (see chapter 7)	`c(1, 2)` returns `1, 2`
`seq(a,b)`	Return a sequence of numbers from a to b	`seq(1, 5)` returns `1, 2, 3, 4, 5`

To learn more about any individual function, look them up in the R documentation by using `?FunctionName` account as described in the previous chapter.

"Knowing" how to program in a language is to some extent simply "knowing" what provided functions are available in that language. Thus you should look around and become familiar with these functions… but **do not** feel that you need to memorize them! It's enough to simply be aware "oh yeah, there was a function that sums up numbers", and then be able to look up the name and argument for that function.

6.3 Loading Functions

Although R comes with lots of built-in functions, you can always use more functions! **Packages** (or **libraries**) are additional sets of R functions that are written and published by the R community. Because many R users encounter the same data management/analysis challenges, programmers are able to use these libraries and thus benefit from the work of others (this is the amazing thing about the open-source community—people solve problems and then make those solutions available to others). R packages **do not** ship with the R software by default, and need to be downloaded (once) and then loaded into your interpreter's environment (each time you wish to use them). While this may seem cumbersome, the R software would be huge and slow if you had to install and load *all* available packages to use it.

Luckily, it is quite simple to install and load R packages from within R. To do so, you'll need to use the *built-in* R functions `install.packages` and `library`. Below is an example of installing and loading the `stringr` package (which contains more handy functions for working with character strings):

```
# Install the `stringr` package. Only needs to be done once on your machine
install.packages("stringr")

# Load the package (tell R functions are available for use)
library("stringr") # quotes optional here
```

- Note that when you load a package, you may receive a warning message about the package being built under a previous version of R. In all likelihood this shouldn't cause a problem, but you should pay attention to the details of the messages and keep them in mind (especially if you start getting unexpected errors).

After loading the package with the `library` function, you have access to functions that were written as part of that package (see the documentation for a list of functions included with the `stringr` library).

6.4 Writing functions

Even more exciting than loading other peoples' functions is writing your own. Any time that you have a task that you may repeat throughout a script—or you simply want to organize your thinking—it's good practice to write a function to perform that task. This will limit repetition and reduce the likelihood of errors… as well as make things easier to read and understand (and thus identify flaws in your analysis).

Functions are named like any other variable, so we use the *assignment operator* (`<-`) to store a new `function` in a variable. It is best practice to assign functions names in *CamelCase* without any periods (`.`) in the name. This helps distinguish functions from other variables.

The best way to understand the syntax for defining a function is to look at an example:

```r
# A function named `MakeFullName` that takes two arguments
# and returns the "full name" made from them
MakeFullName <- function(first.name, last.name) {
  # Function body: perform tasks in here
  full.name <- paste(first.name, last.name)

  # Return: what you want the function to output
  return(full.name)
}

# Call the MakeFullName function with the values "Alice" and "Kim"
my.name <- MakeFullName("Alice", "Kim")  # "Alice Kim"
```

Functions have a couple of pieces to them:

- **Arguments**: the data assigned to the function variable uses the syntax `function(...)` to indicate that you are creating a function (as opposed to a number or character string). The values put betweeen the parentheses are variables that *will contain* the values passed in as **arguments**. For example, when we call `MakeFullName("Alice", "Kim")`, the value of the first argument (`"Alice"`) will be assigned to the first variable (`first.name`), and the value of the second argument (`"Kim"`) will be assigned to the second variable (`last.name`).

  Importantly, we could have made the argument names anything we wanted (`name.first`, `given.name`, etc.), just as long as we then use *that variable name* to refer to the argument while inside the function. Moreover, these argument variable names *only apply* while inside the function. You can think of them like "nicknames" for the values. The variables `first.name`, `last.name`, and `full.name` only exist within this particular function.

- **Body**: The body of the function is a **block** of code that falls between curly braces **`{}`** (a "block" is represented by curly braces surrounding code statements). Note that cleanest style is to put the opening `{` immediately after the arguments list, and the closing `}` on its own line.

  The function body specifies all the instructions (lines of code) that your function will perform. A function can contain as many lines of code as you want—you'll usually want more than 1 to make it worth while, but if you have more than 20 you might want to break it up into separate functions. You can use the argument variables in here, create new variables, call other functions… basically any code that you would write outside of a function can be written inside of one as well!

- **Return value**: You can specify what output a function produces by calling the `return()` function and passing that the value that you wish *your function* to return (output). The `return()` function will execute instructions that end the current function and *return* the flow of code execution to whereever this function was called from. Note that even though we returned a variable called `full.name`, that variable was *local* to the function and so doesn't exist outside of it; thus we have to take the returned value and assign it to a new variable (as with `name <- MakeFullName("Alice", "Kim")`).

Because the `return()` call exits the function, it is usually the last line of code in the function.

We can call (execute) a function we defined the same way we called built-in functions. When we do so, R will take the **arguments** we passed in (e.g., `"Alice"` and `"Kim"`) and assign them to the *argument variables*. Then it executes each line of code in the **function body** one at a time. When it gets to the `return()` call, it will end the function and return the given value, which can then be assigned to a different variable outside of the functions.

6.5 Conditional Statements

Functions are a way to organize and control the flow of execution (e.g., what lines of code get run in what order). In R, as in other languages, we have one other way of controlling program flow, and that is by specifying different instructions that can be run based on a different set of conditions. **Conditional statements** allow us to specify different chunks of code to run when given different contexts, which is often valuable within functions.

In an abstract sense, an conditional statement is saying:

```
IF something is true
  do some lines of code
OTHERWISE
  do some other lines of code
```

In R, we write these conditional statements using the keywords **`if`** and **`else`** and the following syntax:

```r
if(condition){
  # lines of code to run if condition is TRUE
} else {
  # lines of code to run if condition is FALSE
}
```

(Note that the the `else` needs to be on the same line as the closing `}` of the `if` block. It is also possible to omit the `else` and its block).

The `condition` can be any variable or expression that resolves to a logical value (`TRUE` or `FALSE`). Thus both of the below conditional statements are valid:

```r
porridge.temp <- 115  # in degrees F
if(porridge.temp > 120) {
```

```
  print("This porridge is too hot!")
}

too.cold <- porridge.temp < 70
if(too.cold) {  # a logical value
  print("This porridge is too cold!")
}
```

Resources

- R Function Cheatsheet
- User Defined R Functions

Chapter 7

Vectors

This chapter covers the foundational concepts for working with vectors in R. Vectors are *the* fundamental data type in R: in order to use R, you need to become comfortable with vectors. This chapter will discuss how R stores information in vectors, the way in which operations are executed in *vectorized* form, and how to extract subsets of vectors. These concepts are key to effectively programming in R.

7.1 What is a Vector?

Vectors are *one-dimensional collections of values* that are all stored in a single variable. For example, you can make a vector `names` that contains the character strings "Sarah", "Amit", and "Zhang", or a vector `one.to.hundred` that stores the numbers from 1 to 100. Each value in a vector is refered to as an **element** of that vector; thus the `names` vector would have 3 elements: `"Sarah"`, `"Amit"`, and `"Zhang"`.

- Importantly, all the elements in a vector need to have the same *type* (numeric, character, logical, etc.). You can't have a vector whose elements include both numbers and character strings.

7.1.1 Creating Vectors

The easiest and most common syntax for creating vectors is to use the built in `c()` function, which is used to ***c****ombine* values into a vector. The `c()` function takes in any number of **arguments** of the same type (separated by commas as usual), and **returns** a vector of that contains those elements:

```
# Use the combine (`c`) function to create a vector.
names <- c("Sarah", "Amit", "Zhang")
print(names)  # [1] "Sarah" "Amit"  "Zhang"

numbers <- c(1,2,3,4,5)
print(numbers)  # [1] 1 2 3 4 5
```

You can use the `length()` function to determine how many **elements** are in a vector:

```
names <- c("Sarah", "Amit", "Zhang")
names.length <- length(names)
print(names.length)  # [1] 3

numbers <- c(1,2,3,4,5)
print( length(numbers) )  # [1] 5
```

Other functions can also help with creating vectors. For example, the `seq()` function mentioned in chapter 6 takes 2 arguments and produces a vector of the integers between them. An *optional* third argument specifies how many numbers to skip:

```
# Make vector of numbers 1 to 100
one.to.hundred <- seq(1,100)
print(one.to.hundred)

# Make vector of numbers 1 to 10, counting by 2
odds <- seq(1, 10, 2)
print(odds)  # [1] 1 3 5 7 9
```

- When you print out `one.to.hundred`, you'll notice that in addition to the leading `[1]` that you've seen in all printed results, there are additional bracketed numbers at the start of each line. These bracketed numbers tells you from which element number (**index**, see below) that line is showing the elements of. Thus the `[1]` means that the printed line shows elements started at element number `1`, a `[20]` means that the printed line shows elements starting at element number `20`, and so on. This is to help make the output more readable, so you know where in the vector you are when looking at in a printed line of elements!

As a shorthand, you can produce a sequence with the **colon operator** (**`a:b`**), which returns a vector `a` to `b` with the element values incrementing by `1`:

```
one.to.hundred <- 1:100
```

Once created, you are unable to change the number of elements in a vector. However, you can create a *new vector* by **c**ombining a new element with an existing vector:

```r
# Use the combine (`c()`) function to create a vector.
names <- c("Sarah", "Amit", "Zhang")

# Use the `c()` function to combine the `people` vector and the name 'Josh'.
more.names <- c(names, 'Josh')
print(more.names)  # [1] "Sarah" "Amit"  "Zhang" "Josh"
```

7.2 Vectorized Operations

When performing operations (such as mathematical operations `+`, `-`, etc.) on vectors, the operation is applied to vector elements **member-wise**. This means that each element from the first vector operand is modified by the element in the **same corresponding position** in the second vector operand, in order to determine the value of at *the corresponding position* of the resulting vector. E.g., if you want to add (`+`) two vectors, then the value of the first element in the result will be the sum (`+`) of the first elements in each vector, the second element in the result will be the sum of the second elements in each vector, and so on.

```r
# Create two vectors to combine
v1 <- c(1, 1, 1, 1, 1)
v2 <- c(1, 2, 3, 4, 5)

# Create arithmetic combinations of the vectors
v1 + v2  # returns 2, 3, 4, 5, 6
v1 - v2  # returns 0, -1, -2, -3, -4
v1 * v2  # returns 1, 2, 3, 4, 5
v1 / v2  # returns 1, .5, .33, .25, .2

# Add a vector to itself (why not?)
v3 <- v2 + v2  # returns 2, 4, 6, 8, 10

# Perform more advanced arithmetic!
v4 <- (v1 + v2) / (v1 + v1)  # returns 1, 1.5, 2, 2.5, 3
```

While we can't apply mathematical operators (namely, `+`) to combine vectors of character strings, we can use functions like `paste()` to concatenate the elements of two vectors.

```r
colors <- c('Green', 'Blue')
spaces <- c('sky', 'grass')

# Note: look up the `paste0()` function if it's not familiar!
bands <- paste0(colors, spaces)  # returns "Greensky", "Bluegrass"
# http://greenskybluegrass.com/
```

Notice the same *member-wise* combination is occuring: the `paste0()` function is applied to the first elements, then to the second elements, and so on.

7.2.1 Recycling

Recycling refers to what R does in cases when there are an unequal number of elements in two operand vectors. If R is tasked with performing a vectorized operation with two vectors of unequal length, it will reuse (*recycle*) elements from the shorter vector. For example:

```
# Create vectors to combine
v1 <- c(1, 3, 5)
v2 <- c(1, 2)

# Add vectors
v3 <- v1 + v2  # returns (2, 5, 6)
```

In this example, R first combined the elements in the first position of each vector (`1+1=2`). Then, it combined elements from the second position (`3+2=5`). When it got to the third element (which only was present in `v1`), it went back to the **beginning** of `v2` to select a value, yielding `5+1=6`.

- Recycling will occur no matter if the longer vector is the first or second operand.
- R may provide a warning message, notifying you that the vectors are of different length. This warning doesn't necessarily mean you did something wrong, but you should pay attention to it because it may be indicative of an error (i.e., you thought the vectors were of the same length, but made a mistake somewhere).

7.2.2 Everything is a Vector!

What happens if you try to add a vector and a "regular" single value (a **scalar**)?

```
# create vector of numbers 1 to 5
v1 <- 1:5
result <- v1 + 4  #add scalar to vector
print(result)  # [1] 5 6 7 8 9
```

As you can see (and probably expected), the operation added `4` to every element in the vector.

The reason this sensible behavior occurs is because, in truth, **everything in R is a vector**. Even when you thought you were creating a single value (a scalar), you were actually just creating a vector with a single element (length 1). When you create a variable storing the number `7` (with `x <- 7`), R creates a vector of length 1 with the number `7` as that single element.

- This is why R prints the `[1]` in front of all results: it's telling you that it's showing a vector (which happens to have 1 element) starting at element number 1.
- This is also why you can't use the `length()` function to get the length of a character string; it just returns the length of the array containing that string (`1`). Instead, use the `nchar()` function to get the number of characters in a character string.

```
# Create a vector of length 1 in a variable x
x <- 7  # equivalent to `x <- c(7)`

# Print out x: R states the vector index (1) in the console
print(x)  # [1] 7
```

Thus when you add a "scalar" such as `4` to a vector, what you're really doing is adding a vector with a single element `4`. As such the same *recycling* principle applies, and that single element is "recycled" and applied to each element of the first operand.

7.2.3 Vectorized Functions

> *Vectors In, Vector Out*

Because *everything is a vector*, it means that pretty much every function you've used so far has actually applied to vectors, not just to single values. These are referred to as **vectorized functions**, and will run significantly faster than non-vector approaches. You'll find that functions work the same way for vectors as they do for single values, because single values are just instances of vectors!

- *Fun fact:* The mathematical operators (e.g., `+`) are actually functions in R that take 2 arguments (the operands). The mathematical notation we're used to using is just a shortcut.

  ```
  # these two lines of code are the same:
  x <- 2 + 3  # add 2 and 3
  x <- '+'(2, 3)  # add 2 and 3
  ```

This means that you can use any function on a vector, and it will act in the same **vectorized**, *member-wise* manner: the function will result in a new vector where the function's transformation has been applied to each individual element in order.

For example consider the `round()` function described in the previous chapter. This function rounds the given argument to the nearest whole number (or number of decimal places if specified).

```
# round number to 1 decimal place
round(1.67, 1)  # returns 1.6
```

But recall that the `1.6` in the above example is *actually a vector of length 1*. If we instead pass a vector as an argument, the function will perform the same rounding on each element in the vector.

```
# Create a vector of numbers
nums <- c(3.98, 8, 10.8, 3.27, 5.21)

# Perform the vectorized operation
whole.nums <- round(nums, 1)

# Print the results (each element is rounded)
print(whole.nums)  # [1]  4.0  8.0 10.8  3.3  5.2
```

This vectorization process is ***extremely powerful***, and is a significant factor in what makes R an efficient language for working with large data sets (particularly in comparison to languages that require explicit iteration through elements in a collection). Thus to write really effective R code, you'll need to be comfortable applying functions to vectors of data, and getting vectors of data back as results.

Just remember: *when you use a function on a vector, you're using that function **on each item** in the vector*!

7.3 Vector Indices

Vectors are the fundamental structure for storing collections of data. Yet you often want to only work with *some* of the data in a vector. This section will discuss a few ways that you can get a **subset** of elements in a vector.

In particular, you can refer to individual elements in a vector by their **index**, which is the number of their position in the vector. For example, in the vector:

```
vowels <- c('a','e','i','o','u')
```

The `'a'` (the first element) is at *index* 1, `'e'` (the second element) is at index 2, and so on.

- Note in R vector elements are indexed starting with `1`. This is distinct from most other programming languages which are *zero-indexed* and so reference the first element at index `0`.

You can retrieve a value from a vector using **bracket notation**: you refer to the element at a particular index of a vector by writing the name of the vector, followed by square brackets (`[]`) that contain the index of interest:

```
names <- c("Sarah", "Amit", "Zhang")

# access the element at index 1
name.first <- names[1]
print(name.first)  # [1] "Sarah"
```

```
# access the elemnt at index 2
name.second <- names[2]
print(name.second)  # [1] "Amit"

# You can also use variables inside the brackets
last.index <- length(names)  # last index is the length of the vector!
name.last <- names[last.index]  # returns "Zhang"
```

- Don't get confused by the `[1]` in the printed output—it doesn't refer to which index you got from `names`, but what index in the *extracted* result (e.g., stored in `name.first`) is being printed!

If you specify an index that is **out-of-bounds** (e.g., greater than the number of elements in the vector) in the square brackets, you will get back the value `NA`, which stands for **N**ot **A**vailable. Note that this is *not* the *character string* `"NA"`, but a specific logical value.

```
vowels <- c('a','e','i','o','u')

# Attempt to access the 10th element
vowels[10]  # returns NA
```

If you specify a **negative index** in the square-brackets, R will return all elements *except* the (negative) index specified:

```
vowels <- c('a','e','i','o','u')

# Return all elements EXCEPT that at index 2
all.but.e <- vowels[-2]
print(all.but.e)  # [1] "a" "i" "o" "u"
```

7.3.1 Multiple Indicies

Remember that in R, **everything is a vector**. This means that when you put a single number inside the square brackets, you're actually putting a *vector with a single element in it* into the brackets So what you're really doing is specifying a **vector of indices** that you want R to extract from the vector. As such, you can put a vector of any length inside the brackets, and R will extract *all* the elements with those indices from the vector (producing a **subset** of the vector elements):

```
# Create a `colors` vector
colors <- c('red', 'green', 'blue', 'yellow', 'purple')

# Vector of indices to extract
indices <- c(1,3,4)
```

```
# Retrieve the colors at those indices
extracted <- colors[indices]
print(extracted)  # [1] "red"    "blue"   "yellow"


# Specify the index array anonymously
others <- colors[c(2, 5)]
print(others)  # [1] "green"  "purple"
```

It's incredibly common to use the **colon operator** to quickly specify a range of indices to extract:

```
# Create a `colors` vector
colors <- c('red', 'green', 'blue', 'yellow', 'purple')

# Retrieve values in positions 2 through 5
colors[2:5]  # [1] "green"  "blue"   "yellow" "purple"
```

This easily reads as *"a vector of the elements in positions 2 through 5"*.

7.4 Vector Filtering

In the above section, you used a vector of indices (*numeric* values) to retrieve a subset of elements from a vector. Alternatively, you can put a **vector of logical (boolean) values** inside the square brackets to specify which ones you want to extract (`TRUE` in the *corresponding position* means extract, `FALSE` means don't extract):

```
# Create a vector of shoe sizes
shoe.sizes <- c(7, 6.5, 4, 11, 8)

# Vector of elements to extract
filter <- c(TRUE, FALSE, FALSE, TRUE, TRUE)

# Extract every element in an index that is TRUE
shoe.sizes[filter]  # [1]  7 11  8
```

R will go through the boolean vector and extract every item at the same position as a `TRUE`. In the example above, since `filter` is `TRUE` and indices 1, 4, and 5, then `shoe.sizes[filter]` returns a vector with the elements from indicies 1, 4, and 5.

This may seem a bit strange, but it is actually incredibly powerful because it lets you select elements from a vector that *meet a certain criteria* (called **filtering**). You perform this *filtering operation* by first creating a vector of boolean values that correspond with the indices meeting that criteria, and then put that filter vector inside the square brackets:

```
# Create a vector of shoe sizes
shoe.sizes <- c(7, 6.5, 4, 11, 8)

# Create a boolean vector that indicates if a shoe size is greater than 6.5
shoe.is.big <- shoe.sizes > 6.5  # T, F, F, T, T

# Use the `shoe.is.big` vector to select large shoes
big.shoes <- shoe.sizes[shoe.is.big]  # returns 7, 11, 8
```

The magic here is that you are once again using *recycling*: the relational operator `>` is *vectorized*, meaning that the shorter vector (the `6.5`) is recycled and applied to each element in the `shoe.sizes` vector, thus producing the boolean vector that you want!

You can even combine the second and third lines of code into a single statement. You can think of the following statement as saying *shoe.sizes* ***where*** *shoe.sizes is greater than 6.5*:

```
# Create a vector of shoe sizes
shoe.sizes <- c(7, 6.5, 4, 11, 8)

# Select shoe sizes that are greater than 6.5
shoe.sizes[shoe.sizes > 6.5]  # returns 7, 11, 8
```

This is a valid statement because the equality inside of the square-brackets (`shoe.sizes > 6.5`) is evaluated first, producing the boolean vector which is then used to filter the `shoe.sizes` vector.

This kind of filtering is crucial for being able to ask real world questions of datasets.

7.5 Modifying Vectors

As a final note, while you are unable to change the number of elements within a vector, you *are* able to change the individual values within a vector. To achieve this, put the extracted *subset* on the **left-hand side** of the assignment operator, and then assign the element a new value:

```
# Create a vector of school supplies
school.supplies <- c('Backpack', 'Laptop', 'Pen')

# Replace 'Pen' (element at index 3) with 'Pencil'
school.supplies[3] <- 'Pencil'
```

And of course, there's no reason that you can't select multiple elements on the left-hand side, and assign them multiple values. The assignment operator is also *vectorized*!

```
# Create a vector of school supplies
school.supplies <- c('Backpack', 'Laptop', 'Pen')

# Replace  'Laptop' with 'Tablet', and 'Pen' with 'Pencil'
school.supplies[c(2,3)] <- c('Tablet', 'Pencil')
```

As a more useful example, imagine you had a vector of values in which you wanted to replace all numbers greater that 10 with the number 10 (to "cap" the values). Because the assignment operator is vectorized, you can leverage *recycling* to assign a single value to each element that has been *filtered* from the vector:

```
# Element of values
v1 <- c(1, 5, 55, 1, 3, 11, 4, 27)

# Replace all values greater than 10 with 10
v1[v1>10] <- 10  # returns 1, 5, 10, 1, 3, 10, 4, 10
```

In this example, the number `10` get recycled for each element in which `v1` is greater than 10 (`v1[v1>10]`). Presto!

Resources

- R Tutorial: Vectors

Chapter 8

Lists

This chapter covers an additional R data type called lists. Lists are somewhat similar to vectors, but can store more types of data and more details *about* that data (with some cost). Lists are R's version of a **Map**, which is a common and extremely useful way of organizing data in a computer program. Moreover: lists are used to create *data frames*, which is the primary data storage type used for working with sets of real data in R. This chapter will cover how to create and access elements in a list, as well as how to apply functions to lists or vectors.

8.1 What is a List?

A **List** is a lot like a vector, in that it is a *one-dimensional collection of data*. However, lists have two main differences from vectors:

1. Unlike a vector, you can store elements of *different types* in a list: e.g., a list can contain numeric data *and* character string data.

2. Elements in a list can be **tagged** with with names which you can use to easily refer to them—rather than talking about the list's "element #1", we can talk about the list's "`first.name` element".

The second feature is the most significant, as it allows you to use lists to create a type of **map**. In computer programming, a map (or "mapping") is a way of associating one value with another. The most common real-world example is a *dictionary* or *encyclopedia*: a dictionary associates each word with it's definition—you can "look up" a definition by using the word itself, rather than needing to look up the 3891st definition in the book. In fact, this same data structure is called a `dictionary` in the Python programming language!

Lists are extremely useful for organizing data. They allow you to group together data like a person's name (characters), job title (characters), salary (number), and whether they are in a union (logical)—and you don't have to remember whether whether the person's name or title was the first element!

8.1.1 Creating Lists

You create a list by using the `list()` function and passing it any number of **arguments** (separated by commas) that you want to make up that list—similar to the `c()` functon for vectors.

However, you can specify the *tags* for each element in the list by putting the name of the tag (which is like a variable name), followed by an equal symbol (`=`), followed by the value you want to go in the list and be associated with that tag. For example:

```
person <- list(first.name = "Ada", job = "Programmer", salary = 78000, in.union = TR
```

This creates a list of 4 elements: `"Ada"` which is tagged with `first.name`, `"Programmer"` which is tagged with `job`, `78000` which is tagged with `salary`, and `TRUE` which is tagged with `in.union`.

- Note that you can have *vectors* as elements of a list. In fact, each of these scalar values are really vectors (of length 1)!
- The use of the `=` symbol here is an example of assigning a value to a specific named argument. You can actually use this syntax for *any* function (e.g., rather than listing arguments in order, you can explicit "assign" a value to each argument), but it is more common to just use the normal order of the arguments if there aren't very many.

It is possible to create a list without tagging the elements:

```
person.alt <- list("Ada", "Programmer", 78000, TRUE)
```

But it will make code harder to read and more error-prone, so isn't as common.

8.1.2 Accessing Lists

If you printed out the above `person` list, you would see the following:

```
> print(person)
$first.name
[1] "Ada"

$job
[1] "Programmer"

$salary
[1] 78000
```

```
$in.union
[1] TRUE
```

Notice that the output lists each tag name prepended with a dollar sign (**$**) symbol, and then on the following line the vector that is the element itself. The `$` symbol is one of the easiest ways of accessing list elements.

Because list elements are (usually) tagged, you can access them by their tag name rather than by the index number you used with vectors. You do this by using **dollar notation**: you refer to the element with a particular tag in a list by writing the name of the list, followed by a `$`, following by the element's tag:

```
person <- list(first.name = "Ada", job = "Programmer", salary = 78000, in.union = TRUE)

person$first.name  # [1] "Ada"
person$salary  # [1] 78000
```

(See below for other options for accessing list elements).

You can almost read the dollar sign as like an "apostrophe s" (possessive) in English: so `person$salary` would mean "the `person` list's `salary` value".

Dollar notation allows list elements to almost be treated as variables in their own right—for example, you specify that you're talking about the `salary` variable in the `person` list, rather than the `salary` variable in some other list (or not in a list at all).

```
person <- list(first.name = "Ada", job = "Programmer", salary = 78000, in.union = TRUE)

# use elements as function or operation arguments
paste(person$job, person$first.name)   # [1] "Programmer Ada"

# assign values to list element
person$job <- "Senior Programmer"  # a promotion!
print(person$job)  # [1] "Senior Programmer"

# assign value to list element from itself
person$salary <- person$salary * 1.15  # a 15% raise!
print(person$salary)  # [1] 89700
```

Note that if you need to, you can get a *vector* of element tags using the `names()` function:

```
person <- list(first.name = "Ada", job = "Programmer", salary = 78000, in.union = TRUE)
names(person)  # [1] "first.name" "job"  "salary"  "in.union"
```

- This is useful for understanding the structure of variables that may have come from other data sources.

8.1.3 List Indicies

Whether or not a list element has a tag, you can also access it by its numeric index (i.e., if it is the 1st, 2nd, etc. item in the list). You do this by using **double-bracket notation**: you refer to the element at a particular index of a list by writing the name of the list, followed by double square brackets (`[[]]`) that contain the index of interest:

```
# note: a list and not a vector, even though elements have the same types
animals <- list("Aardvark", "Baboon", "Camel")

animals[[1]]  # [1] "Aardvark"
animals[[3]]  # [1] "Camel"
animals[[4]]  # Error: subscript out of bounds!
```

You can also use double-bracket notation to access an element by its tag if you put a character string (in `""`) of the tag name inside the brackets. This is particularly useful if you want the tag itself to be a variable!

```
person <- list(first.name = "Bob", last.name = "Wong", salary = 77000, in.union = TR

person[["first.name"]]  # [1] "Bob"
person[["salary"]]  # [1] 77000

name.to.use <- "last.name"  # choose name (i.e., based on formality)
person[[name.to.use]]  # [1] "Wong"
name.to.use <- "first.name"  # change name to use
person[[name.to.use]]  # [1] "Bob"

# Can use indices for tagged elements as well!
person[[1]]  # [1] "Bob"
person[[4]]  # [1] TRUE
```

8.1.3.1 Single vs. Double Brackets

Wach out!: vectors use ***single***-bracket notation for accessing by index, but lists use ***double***-bracket notation for accessing by index!

This is because the single-bracket syntax for vectors isn't actually selecting by index: rather it is **filtering** by whatever vector is inside the brackets (which may be just a single element: the index number to extract). In R, single brackets *always* mean to filter the collection. So if you put single-brackets after a list, what you're actually doing is getting a filtered **sub-list** of the elements that have those indicies, just as single brackets on a vector return a subset of elements in that vector:

```r
my.list <- list('A', 201, TRUE, 'rhinoceros')

# SINGLE brackets returns a list
my.list[1]
            # [[1]]
            # [1] "A"

# DOUBLE brackets returns a vector
my.list[[1]]  # [1] "A"

# can use any vector as the argument to single brackets, just like with vectors
my.list[1:3]
            # [[1]]
            # [1] "A"
            #
            # [[2]]
            # [1] 201
            #
            # [[3]]
            # [1] TRUE
```

In sum, remember that **single-brackets gives a list, double-brackets gives a vector**. You almost always want to be refering to the value itself (the vector—everything is a vector!) rather than a list, so almost always want to use ***double-brackets*** when accessing lists.

8.1.4 Modifying Lists

Unlike with vectors, you are also able to change the *number* of elements that are inside a list—you can add and remove elements from a list!

You can add elements to a list simply by assigning a value to a tag (or index) in the list that doesn't yet exist:

```r
person <- list(first.name = "Ada", job = "Programmer", salary = 78000, in.union = TRUE)

# has no `age` element
person$age  # NULL

# assign a value to the `age` tag to add it
person$age <- 40
person$age  # [1] 40

# assign using index
person[[10]] <- "Tenth field"
# elements 6-9 will be NULL
```

You can also remove elements by assiging the special value `NULL` to their tag or index:

```
a.list <- list('A', 201, True)
a.list[[2]] <- NULL  # remove element #2
print(a.list)
            # [[1]]
            # [1] "A"
            #
            # [[2]]
            # [1] TRUE
```

8.2 The `lapply()` Function

Since everything is a vector in R, and most functions are *vectorized*, you can can pass most functions (e.g., `paste()`, `round()`, etc.) a vector as an argument and the function will be applied to each item in the vector. It "just works". But if you want to apply a function to each item in a *list*, you need to put in a bit more effort.

In particular, you need to use a function called **`lapply()`** (for *list apply*). This function takes two arguments: the first is a list or vector you want to modify, and the second is a function you want to "apply" to each item in that list. For example:

```
# list, not a vector
names <- list("Sarah", "Amit", "Zhang")

# apply the `toupper()` function to each element in `names`
names.upper <- lapply(names, toupper)
            # [[1]]
            # [1] "SARAH"
            #
            # [[2]]
            # [1] "AMIT"
            #
            # [[3]]
            # [1] "ZHANG"

# apply the `paste()` function to each element in `names`,
# with an addition argument `"dances!"` to each call
dance.party <- lapply(names, paste, "dances!")
```

- Notice that the second argument to `lapply()` is just the name of the function: not a character string (it's not in `""`). You're also not actually *calling* that function (there are no `()` after it). Just put the name of the function! After that, you can put any additional arguments you want the applied function to be called with: for example, how many digits to round to, or what value to paste to the end of a string.

Note that the `lapply()` function returns a *new* list; the original one is unmodified (though if the list contains vectors or other lists as elements, it's possible for those values to be changed).

You commonly use `lapply()` with your own custom functions which define what you want to do to a single element in that list:

```
# A function that prepends "Hello" to any item
Greet <- function(item) {
  return( paste("Hello", item) )
}

# a list of names
names <- list("Sarah", "Amit", "Zhang")

# greet each name
greetings <- lapply(names, Greet)
            # [[1]]
            # [1] "Hello Sarah"
            #
            # [[2]]
            # [1] "Hello Amit"

            # [[3]]
            # [1] "Hello Zhang"
```

Additionally, `lapply()` is a member of the "`*apply()`" family of functions: a set of functions that each start with a different letter and applies to a different data structure, but otherwise all work basically the same. For example, `lapply()` is used for lists, while `sapply()` (**s**implified apply) works well for vectors.

Resources

- R Tutorial: Lists
- R Tutorial: Named List Members
- StackOverflow: Single vs. double brackets