Technical Foundations of Informatics

Michael Freeman and Joel Ross

March 30, 2018

Contents

About the Book

About the Book

This book covers the foundation skills necessary to start ***writing computer programs to work with data*** using modern and reproducible 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 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 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 is the primary programming language used throughout this book. "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.

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 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** (also called 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 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 Terminal 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 latest updates to Windows 10 (August 2016 or later) *does* includes 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.

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.

While we will primarily use RStudio (below) which has a great built-in text editor, 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 coding 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!)

Programming involves working with many different file types, each detailed by their extension. It is useful to specify that your computer should show these extensions in the File Explorer or Finder; see instructions for Windows or for Mac to enable this.

1.3.1 Atom

Atom is a text editor built by the folks at GitHub. 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 used regularly in this book (see Chapter 3).

To download Atom, visit their **webpage** and click the "Download" button to download the program. On Windows you will download the installer `AtomSetup.exe` file; double-click on that icon to install the application). On a Mac you will download a zip file; open that file and drag the `Atom.app` file to

Once you've installed Atom, you can open the program and create a new text file. When you save a document that is a particular file-type (i.e., `file-name.R`, or `file-name.md`), Atom (or any other modern text-editor) will apply a language specific color scheme to your text, making it easier to read.

The trick to using Atom more efficiently 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 right in Atom).

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 our main editors for programming.

1.3.3 Sublime Text

Sublime Text 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.

1.4 R Language

The primary programming language you will use throughout this book is called **R**. It's a very powerful statistical programming language that is built to work well with large and diverse datasets. See Chapter 5 for a more in-depth introduction to the language.

In order to program with R, you will need to install the ***R Interpreter*** on your machine. This is a piece of software that is able to "read" code written in R and use that code to control your computer, thereby "programming" it.

The easiest way to install R is to download it from the Comprehensive R Archive Network (CRAN) at **https://cran.rstudio.com/**. Click on the appropriate link for your operating system in order to find a link to the installer.

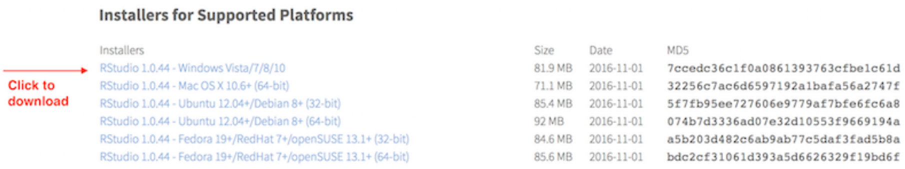
- On a Mac, you're looking for the `.pkg` file—get the latest version supported by your computer.
- On Windows, follow the link to the `base` subdirectory (or follow the link to "*install R for the first time*"), then click the link to download the latest version of R for Windows. You will need to double-click on the `.exe` file to install the software.

1.5 RStudio

While you are able to execute R scripts without a dedicated application, the **RStudio** program provides a wonderful way to engage with the R language. RStudio is described in more detail in Chapter 5.

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



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.

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!

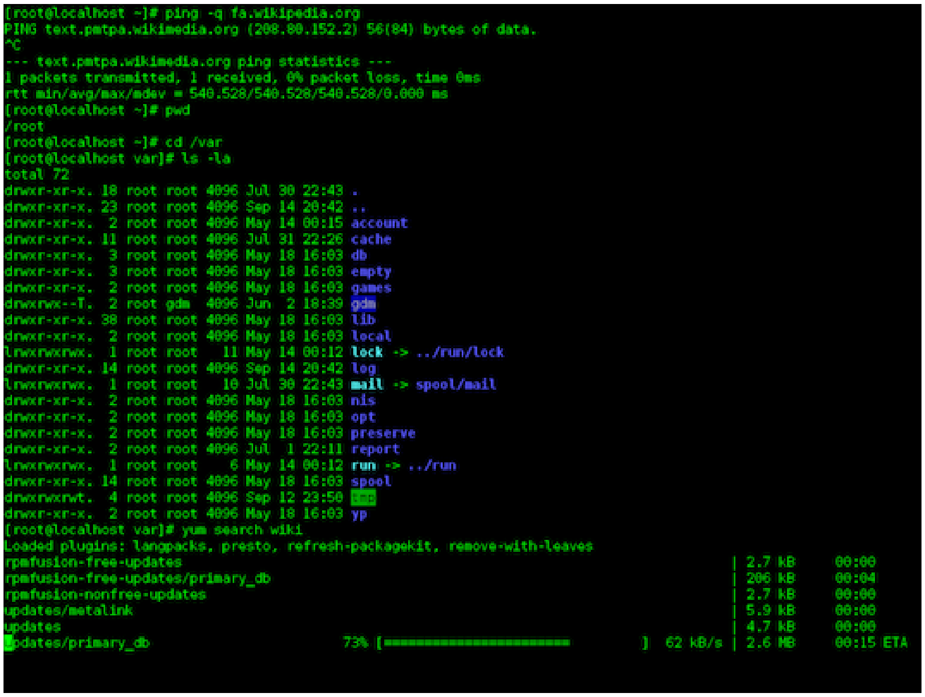
The command-line is not as friendly or intuitive as a graphical interface: it's much harder to learn and figure out. However, it has the advantage of being both more powerful and more efficient in the hands of expert users. (It's faster to type than to move a mouse, and you can do *lots* of "clicks" with a single command). The command-line is also used when working on remote servers or other computers that for some reason do not have a graphical interface enabled. Thus, command line is an essential tool for all professional developers, 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

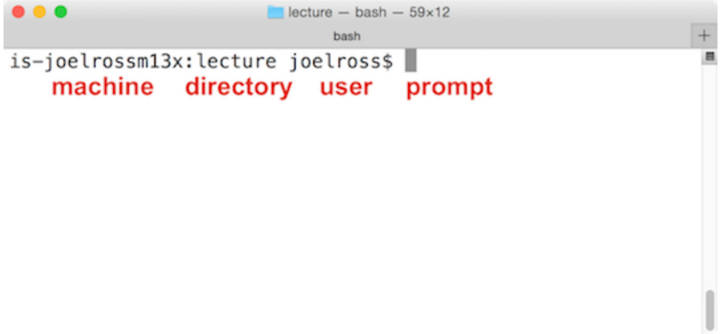
In order to use the command-line, you will need to open a **command shell** (a.k.a. a *command prompt*). This is a program that provides the interface to type commands into. You should have installed a command shell (hereafter "the terminal") as part of setting up your machine.

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



```
[root@localhost ~]# ping -q fa.wikipedia.org
PING text.pmtpa.wikimedia.org (208.80.152.2) 56(84) bytes of data.
^C
--- text.pmtpa.wikimedia.org ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 540.528/540.528/540.528/0.000 ms
[root@localhost ~]# pwd
/root
[root@localhost ~]# cd /var
[root@localhost var]# ls -la
total 72
drwxr-xr-x. 18 root root 4096 Jul 30 22:43 .
drwxr-xr-x. 23 root root 4096 Sep 14 20:42 ..
drwxr-xr-x.  2 root root 4096 May 14 00:15 account
drwxr-xr-x. 11 root root 4096 Jul 31 22:26 cache
drwxr-xr-x.  3 root root 4096 May 18 16:03 db
drwxr-xr-x.  3 root root 4096 May 18 16:03 empty
drwxr-xr-x.  2 root root 4096 May 18 16:03 games
drwxrwx--T.  2 root gdm  4096 Jun  2 18:39 gdm
drwxr-xr-x. 38 root root 4096 May 18 16:03 lib
drwxr-xr-x.  2 root root 4096 May 18 16:03 local
lrwxrwxrwx.  1 root root   11 May 14 00:12 lock -> ../run/lock
drwxr-xr-x. 14 root root 4096 Sep 14 20:42 log
lrwxrwxrwx.  1 root root   10 Jul 30 22:43 mail -> spool/mail
drwxr-xr-x.  2 root root 4096 May 18 16:03 nis
drwxr-xr-x.  2 root root 4096 May 18 16:03 opt
drwxr-xr-x.  2 root root 4096 May 18 16:03 preserve
drwxr-xr-x.  2 root root 4096 Jul  1 22:11 report
lrwxrwxrwx.  1 root root    6 May 14 00:12 run -> ../run
drwxr-xr-x. 14 root root 4096 May 18 16:03 spool
drwxrwxrwt.  4 root root 4096 Sep 12 23:50 tmp
drwxr-xr-x.  2 root root 4096 May 18 16:03 yp
[root@localhost var]# yum search wiki
Loaded plugins: langpacks, presto, refresh-packagekit, remove-with-leaves
rpmfusion-free-updates                                                                    | 2.7 kB     00:00
rpmfusion-free-updates/primary_db                                                         | 206 kB     00:04
rpmfusion-nonfree-updates                                                                 | 2.7 kB     00:00
updates/metalink                                                                          | 5.9 kB     00:00
updates                                                                                   | 4.7 kB     00:00
updates/primary_db                        73% [=======================         ]  62 kB/s | 2.6 MB     00:15 ETA
```

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



```
is-joelrossm13x:lecture joelross$
```

Figure 2.2: A newly opened command-line.

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:

- 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** (typically denoted as the `$` symbol), 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 you'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 usually 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 which on a Mac may look like:

You describe what folder you 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

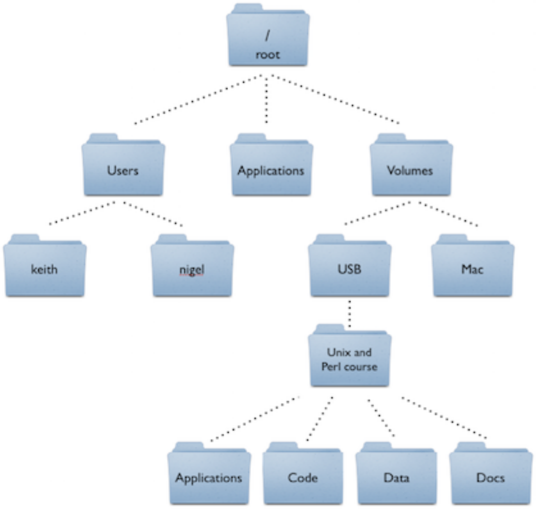


Figure 2.3: A Directory Tree, from Bradnam and Korf.

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 current 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 written 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.

Warning: 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 `/`:

```
cd /Users/iguest/Desktop/
```

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

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:

```
cd iguest/Desktop/
```

Because this path doesn't have the leading slash, it just says to "go to the `iguest/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/` path will only refer to the correct location 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! Because you'll almost always be managing multiples files in a project, you should refer to the files *relatively* within your project. That way, you program can easily work across computers. For example, if your code refers to `/Users/your-user-name/project-name/data`, it can only run on the `your-user-name` account. However, if you use a *relative path* within your code (i.e., `project-name/data`), the program will run on multiple computers (crucial for collaborative projects).

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`.

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.

Additionally, 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 (e.g., `~/Desktop` is an *absolute path* to the desktop for the current user).

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. Here is an short list of commands to get you started using the command prompt, though there are many more:

Command	Behavior
mkdir	**m**ake 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	opens a file or folder (Mac only)
start	opens a file or folder (Windows only)
cat	con**cat**enate (combine) file contents and display the results
history	show previous commands executed

Warning: The command-line makes it **dangerously easy** to *permanently delete* multiple files or folders and *will not* ask you to confirm that you want to delete them (or move them to the "recycling bin"). Be very careful when using the terminal to manage your files, as it is very powerful.

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 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 (though *not* Git Bash, unfortunately) also include their own manual you can use to look up commands!

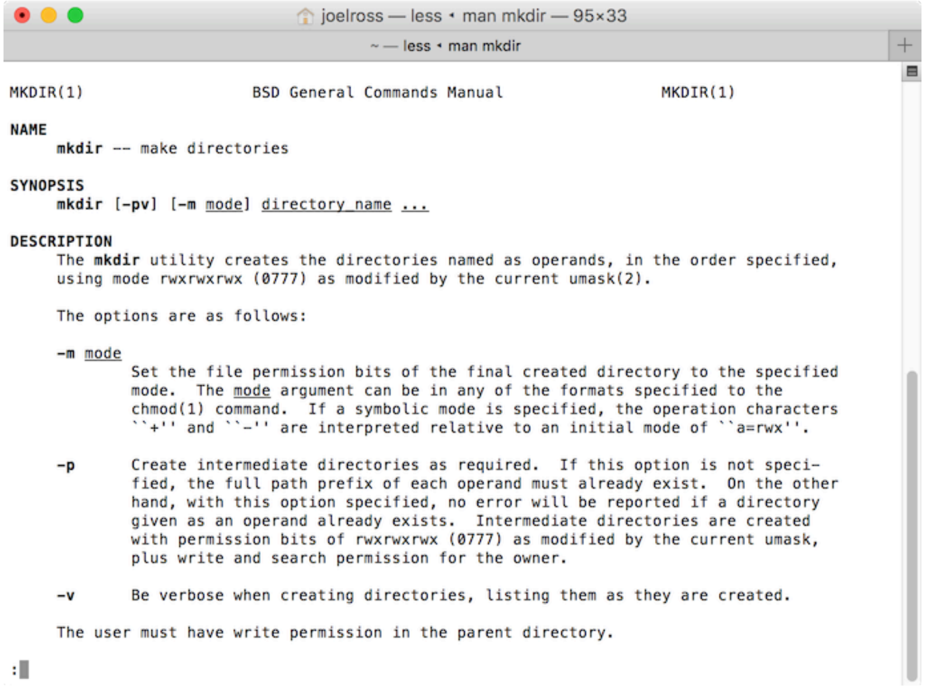
```
man mkdir
```

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. Op-



```
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 2.4: The `mkdir` man page.

tions 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 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; forgetting a space may result in an entirely different action.

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).

Throughout this book, we'll discuss a variety of approaches to handling errors in computer programs. While it's tempting to disregard dense error messages, many programs do provide **error messages** that explain what went wrong. If you enter an unrecognized command, the terminal will inform you of your mistake:

```
lx
> -bash: lx: command not found
```

However, forgetting arguments yields different results. In some cases, there will be a default behavior (see what happens if you enter `cd` without any arguments). If more information is *required* to run a command, your terminal will provide you with a brief summary of the command's usage:

```
mkdir
> usage: mkdir [-pv] [-m mode] directory ...
```

Take the time to read the error message and think about what the problem might be before you try again.

Resources

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

Chapter 3

Markdown

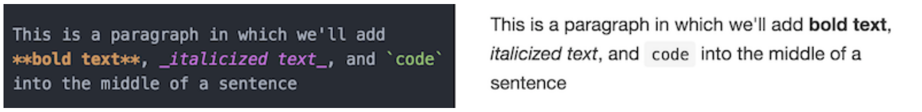
Markdown syntax provides a simple way to describe the desired formatting of text documents. In fact, this book was written using Markdown! With only a small handful of options, Markdown allows you to format to your text (like making text **bold**, or *italics*), as well as provide structure to a document (such as headers or bullet-points). There are a number of programs and services 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). As a programmer, you'll use Markdown to create documentation and other supplementary materials that help explain your projects.

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 text to be rendered as *italiccs*, you would surround that text with underscores (_): you would type _italics_, and a program would know to render that text as *italics*. You can see how this looks in the below example (code on the left, rendered version on the right):



```
This is a paragraph in which we'll add
**bold text**, _italicized text_, and `code`
into the middle of a sentence
```

This is a paragraph in which we'll add **bold text**, *italicized text*, and `code` into the middle of a sentence

Figure 3.1: Markdown text formatting.

There are a few different ways you can format text:

Syntax	Formatting
text	*italicized* using underscores (_)
text	**bolded** using two asterisks (*)
\`text\`	inline `code` with backticks (\`)
\~\~text\~\~	~~strike-through~~ using tildes (~)

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:

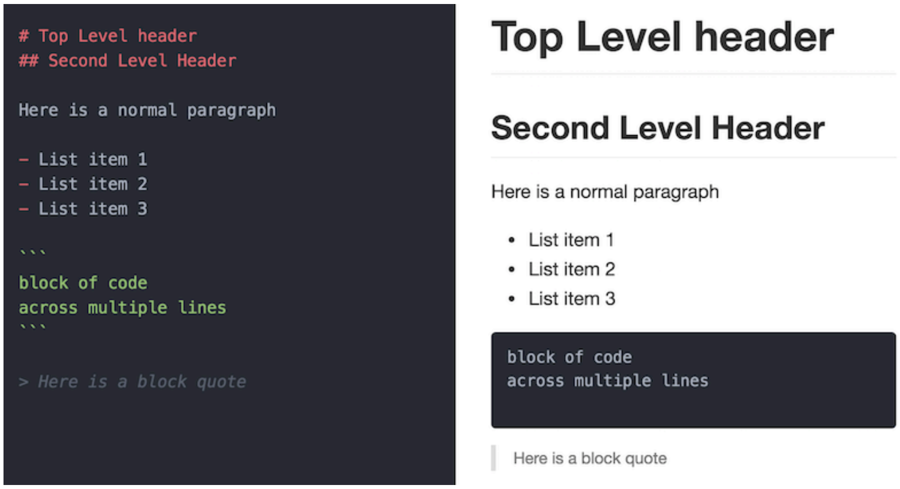
As you can see, the document (right) is produced using the following Markdown shorthand:

Syntax	Formatting
#	Header (use ## for 2nd-level, ### for 3rd, etc.)
\`\`\`	Code section (3 back ticks) that encapsulate the code
-	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:



```
# Top Level header
## Second Level Header

Here is a normal paragraph

- List item 1
- List item 2
- List item 3

```
block of code
across multiple lines
```

> Here is a block quote
```

Top Level header

Second Level Header

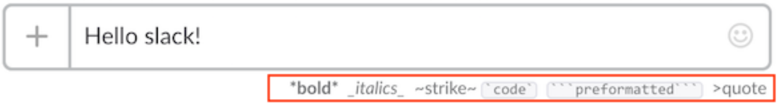
Here is a normal paragraph

- List item 1
- List item 2
- List item 3

```
block of code
across multiple lines
```

> Here is a block quote

Figure 3.2: Markdown block formatting.



Hello slack!

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

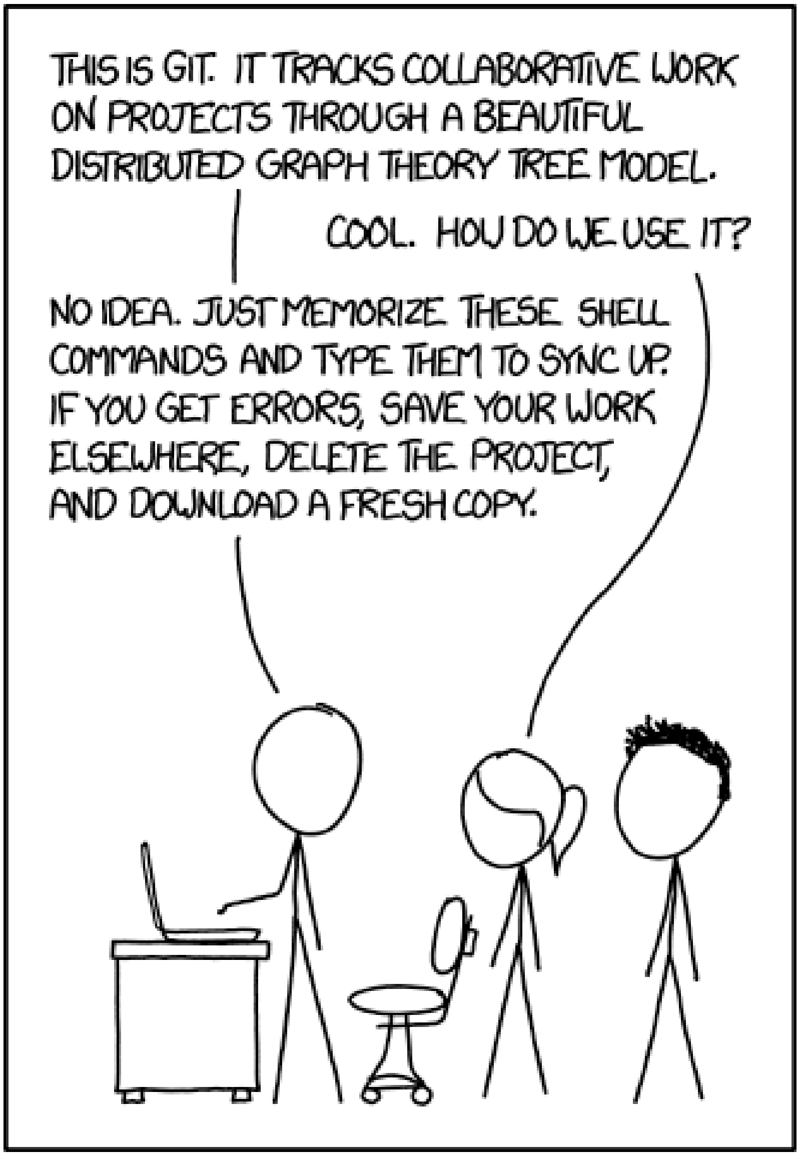
Git and GitHub

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?


THIS IS GIT. IT TRACKS COLLABORATIVE WORK ON PROJECTS THROUGH A BEAUTIFUL DISTRIBUTED GRAPH THEORY TREE MODEL.
COOL. HOW DO WE USE IT?
NO IDEA. JUST MEMORIZE THESE SHELL COMMANDS AND TYPE THEM TO SYNC UP. IF YOU GET ERRORS, SAVE YOUR WORK ELSEWHERE, DELETE THE PROJECT, AND DOWNLOAD A FRESH COPY.

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 almost 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 a project folder 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 (sets of changes) 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.

The first time you use `git` on your machine, you'll need to 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 (i.e., running the `git config` command):

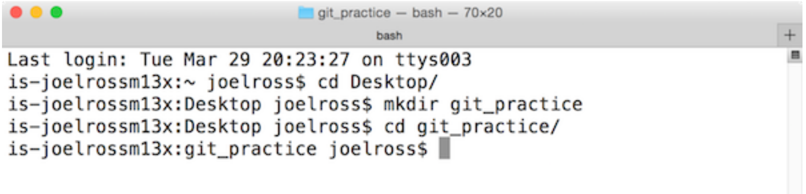
```
# enter your full name (without the dashes)
git config --global user.name "your-full-name"

# 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. If you don't set up the key, you'll need to enter your GitHub password each time you want to push changes up to GitHub (which may be multiple times a day). Simply follow the instructions on this page to set up a key, and make sure to only do this on *your machine*.

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 programmatically:



```
Last login: Tue Mar 29 20:23:27 on ttys003
is-joelrossm13x:~ joelross$ cd Desktop/
is-joelrossm13x:Desktop joelross$ mkdir git_practice
is-joelrossm13x:Desktop joelross$ cd git_practice/
is-joelrossm13x:git_practice joelross$
```

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
# you can easily check this with the "pwd" command
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 synechoche).

- 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 radical 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.

If you are on a Mac, we **strongly suggest** *globally ignoring* your `.DS_Store` file. There's no need to ever share or track this file. To always ignore this file on your machine, simply run these lines of code:

```
# Run these lines on your terminal to configure git to ignore .DS_Store
git config --global core.excludesfile ~/.gitignore
echo .DS_Store >> ~/.gitignore
```

See this article for more information.

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 this link. 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:

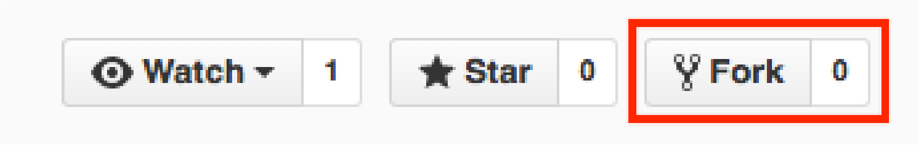


Figure 4.2: The fork button on GitHub's web portal.

  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
- Ignore DS_Store on a Mac

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 data science tools because of its ability to efficiently perform statistical analysis, implement machine learning algorithms, and create data visualizations. R is the primary programming language used throughout this book, and understanding its foundational operations is key to being able to perform more complex tasks.

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 (or set of files), 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:

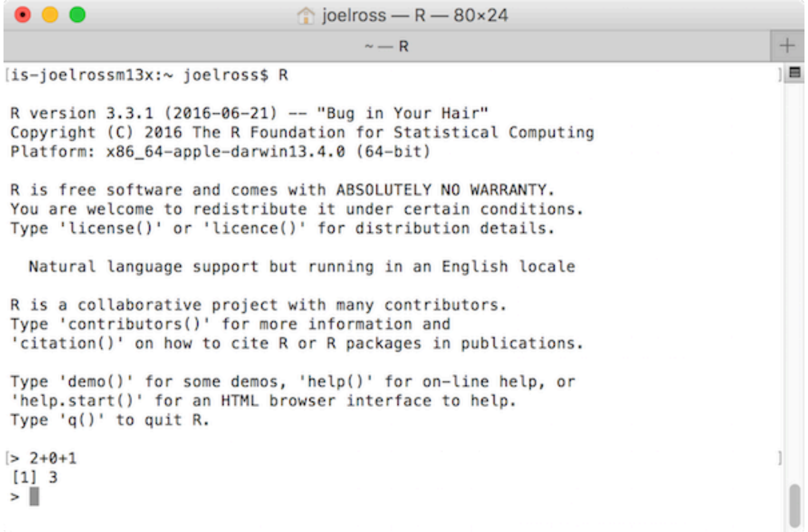
Notice that this description also include *instructions on what to do next*—most importantly `"Type 'q()' to quit R."`.

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. For example, try doing some math at the command prompt (i.e., enter `1 + 1` and see the output).

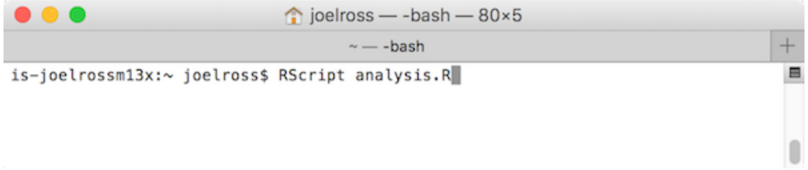
- Note that RStudio also provides an interactive console that provides the exact same functionality.

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:



```
is-joelrossm13x:~ joelross$ R

R version 3.3.1 (2016-06-21) -- "Bug in Your Hair"
Copyright (C) 2016 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin13.4.0 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> 2+0+1
[1] 3
> 
```

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



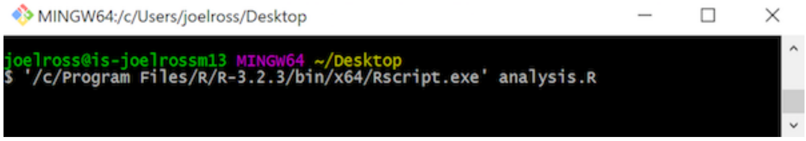
```
is-joelrossm13x:~ joelross$ RScript analysis.R
```

Figure 5.2: Using RScript from the terminal

Entering this command in the terminal would execute each line of R code written in the `analysis.R` file, performing all of the instructions that you had save there. This is helpful if your data has changed, and you want to reproduce the results of your analysis using the same instructions.

5.2.1.1 Windows Command-Line

On Windows, you need to tell the computer where to find the `R.exe` and `RScript.exe` programs to execute—that is, what is the **path** to these programs. You can do this by specifying the *absolute path* to the R program when you execute it, for example:



```
joelross@is-joelrossm13 MINGW64 ~/Desktop
$ '/c/Program Files/R/R-3.2.3/bin/x64/Rscript.exe' analysis.R
```

Figure 5.3: Using RScript from a Windows shell

If you plan to run R from the command-line regularly (*not* a requirement for this course), a better solution is to add the folder containing these programs to your computer's **`PATH` variable**. This is a *system-level* variable that contains a list of folders that the computer searches when finding programs to execute execute. The reason the computer knows where to find the `git.exe` program when you type `git` in the command-line is because that program is "on the `PATH`".

In Windows, You can add the `R.exe` and `RScript.exe` programs to your computer's `PATH` by editing your machine's **Environment Variables** through the *Control Panel*:

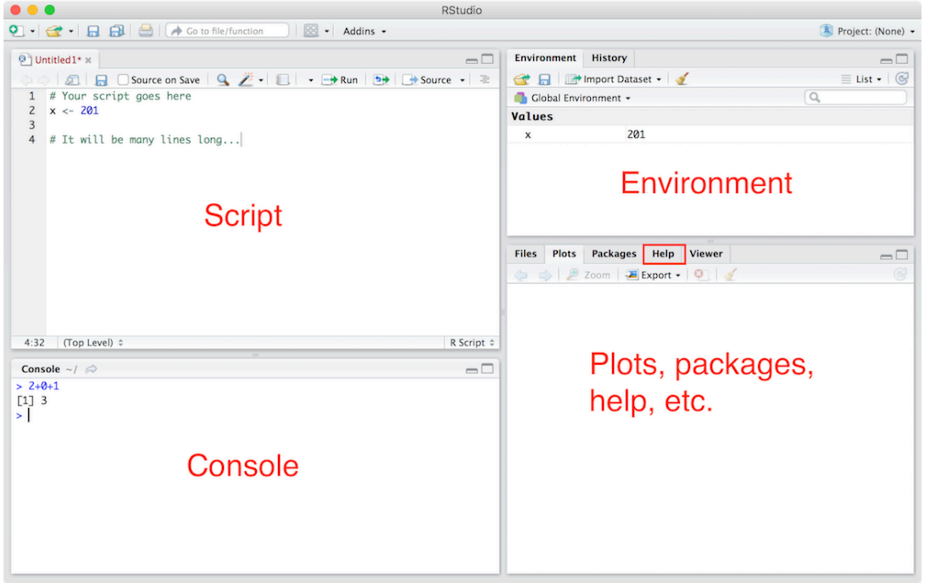
- Open up the "Advanced" tab of the "System Properties". In Windows 10, you can find this by searching for "environment". Click on the "Environment Variables…" button to open the settings for the Environment Variables.
- In the window that pops up, select the "PATH" variable (either per user or for the whole system) and click the "Edit" button below it.
- In the next window that pops up, click the "Browse" button to select the *folder that contains* the `R.exe` and `RScript.exe` files. See the above screenshot for one possible path.
- You will need to close and re-open your command-line (Git Bash) for the `PATH` changes to take effect.

Overall, using R from the command-line can be tricky; we recommend you just use RStudio instead.

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:



```
# Your script goes here
x <- 201

# It will be many lines long...
```

```
> 2+0+1
[1] 3
>
```

Values
x 201

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

An 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 it. This is particularly imporant when someone else will be looking at your work—whether that person is a collaborator, or is simply 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):

```
# 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:

```
# 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
# Store values in variables
x <- 3
y <- 3.15

# Compare values in x and y
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, allowing you to make more complex logical expressions. These include `&` (and), `|` (or), and `!` (not).

```r
# Store values in variables
x <- 3.1
y <- 3.2
pet <- "dog"
weather <- "rain"

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

# Check if pet is "cat" OR "dog"
pet == "cat" | pet == "dog"  # TRUE
```

```
# Check if 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 put the time into trying to understand them.
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 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
- aRrgh: a newcomer's (angry) guide to R opinionated but clear introduction
- 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. For example, imagine a function that can determine the largest number in a set of numbers. That function's **input** would be the set of numbers, and the **output** would be the largest number in the set.

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()`.

```r
# 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:

```r
# 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 have likely noticed, R comes with a variety 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 (make stringr() functions available in this R session/program)
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 value 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

> wherever 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:

```
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!")
}
```

Note, we can extend the set of conditions evaluated using an `else if` statement. For example:

```
# Function to determine if you should eat porridge
FoodTempTest <- function(temp) {
  if (temp > 120) {
    status <- "This porridge is too hot!"
  } else if (temp < 70) {
    status <- "This porridge is too cold!"
  } else {
    status <- "This porridge is just right!"
  }
  return(status)
}
# Use function on different temperatures
FoodTempTest(119)  # "This porridge is just right!"
FoodTempTest(60)   # "This porridge is too cold!"
FoodTempTest(150)  # "This porridge is too hot!"
```

See more about `if` and `else` in Appendix C.

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 `people` that contains the character strings "Sarah", "Amit", and "Zhang". Alternatively, you could make 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 `people` 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:

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

```r
people <- c("Sarah", "Amit", "Zhang")
people.length <- length(people)
print(people.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 in each step:

```r
# 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`:

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

Once created, you are **unable to change** the number of elements in a vector (elements cannot be added or removed). However, you can create a *new vector* by **c**ombining a new element with an existing vector:

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

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

7.2 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:

```
# Create the people vector
people <- c("Sarah", "Amit", "Zhang")

# access the element at index 1
first.person <- people[1]
print(first.person)  # [1] "Sarah"

# access the element at index 2
second.person <- people[2]
print(second.person)  # [1] "Amit"

# You can also use variables inside the brackets
last.index <- length(people)  # last index is the length of the vector!
last.person <- people[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 `people`, but what index in the *extracted* result (e.g., stored in `first.person`) 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.

```r
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:

```r
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"
```

7.2.1 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):

```r
# 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"*.

7.3 Modifying Vectors

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!

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):

```r
# 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:

```r
# 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*:

```r
# 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 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 *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.

```
# 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.

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

# Note: look up the `paste0()` function if it's not familiar!
```

```
band <- paste0(colors, spaces)  # returns "Greensky", "Bluegrass"
# http://greenskybluegrass.com/
```

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

7.5.1 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).

7.5.2 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.

```r
# 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.

7.5.3 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.

  ```r
  # 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).

```r
# 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.

```r
# 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*!

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!

Note: technically, you can create named elements in a vector, but this is very rarely done. Compared to vectors, lists provide a simpler syntax for accessing named elements.

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 (and should) 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 (which can't be done with vectors).

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 `$`, followed by the element's tag:

```r
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).

```r
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 = TR
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 Indices

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

Watch 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 indices, just as single brackets on a vector return a subset of elements in that vector:

```
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 list element**. 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

Similarly to vectors, you can add and modify list elements. However, lists also enable you to *remove* elements.

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:

```
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
people <- list("Sarah", "Amit", "Zhang")

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

# apply the `paste()` function to each element in `people`,
# with an addition argument `"dances!"` to each call
dance.party <- lapply(people, 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). This makes it a **mapping** operation (applied to the maps that are lists).

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 people
people <- list("Sarah", "Amit", "Zhang")

# greet each name
greetings <- lapply(people, 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

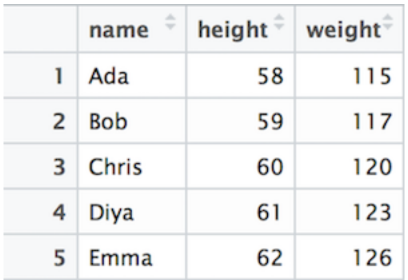
Chapter 9

Data Frames

This chapter introduces **data frame** objects, which are the primary data storage type used in R. In many ways, data frames are similar to a two-dimensional row/column layout that you should be familiar with from spreadsheet programs like Microsoft Excel. Rather than interact with this data structure through a UI, we'll learn how to programmatically and reproducibly perform operations on this data type. This chapter covers various ways of creating, describing, and accessing data frames, as well as how they are related to other data types in R.

9.1 What is a Data Frame?

At a practical level, **Data Frames** act like *tables*, where data is organized into rows and columns. For example, consider the following table of names, weights, and heights:



	name	height	weight
1	Ada	58	115
2	Bob	59	117
3	Chris	60	120
4	Diya	61	123
5	Emma	62	126

Figure 9.1: A table of data (people's weights and heights).

In this table, each *row* represents a **record** or **observation**: an instance of a single thing being measured (e.g., a person). Each *column* represents a **feature**:

a particular property or aspect of the thing being measured (e.g., the person's height or weight). This structure is used to organize lots of different *related* data points for easier analysis.

In R, you can use **data frames** to represent these kinds of tables. Data frames are really just **lists** (see Lists) in which each element is a **vector of the same length**. Each vector represents a **column, *not* a row**. The elements at corresponding indices in the vectors are considered part of the same record (row).

- This makes sense because each row may have a different type of data—e.g., a person's `name` (string) and `height` (number)—and vector elements must all be of the same type.

For example, you can think of the above table as a *list* of three *vectors*: `name`, `height` and `weight`. The name, height, and weight of the first person measured are represented by the first elements of the `name`, `height` and `weight` vectors respectively.

You can work with data frames as if they were lists, but data frames include additional properties as well that make them particularly well suited for handling tables of data.

9.1.1 Creating Data Frames

Typically you will *load* data sets from some external source (see below), rather than writing out the data by hand. However, it is important to understand that you can construct a data frame by combining multiple vectors. To accomplish this, you can use the `data.frame()` function, which accepts **vectors** as *arguments*, and creates a table with a column for each vector. For example:

```r
# vector of names
name <- c('Ada','Bob','Chris','Diya','Emma')

# Vector of heights
height <- 58:62

# Vector of weights
weight <- c(115, 117, 120, 123, 126)

# Combine the vectors into a data.frame
# Note the names of the variables become the names of the columns!
my.data <- data.frame(name, height, weight, stringsAsFactors=FALSE)
```

- (The last argument to the `data.frame()` function is included because one of the vectors contains strings; it tells R to treat that vector as a *vector* not as a **factor**. This is usually what you'll want to do. See below for details about factors).

Because data frame elements are lists, you can access the values from `my.data` using the same **dollar notation** and **double-bracket notation** as lists:

```r
# Using the same weights/heights as above:
my.data <- data.frame(height, weight)

# Retrieve weights (the `weight` element of the list: a vector!)
my.weights <- my.data$weight

# Retrieve heights (the whole column: a vector!)
my.heights <- my.data[['height']]
```

9.1.2 Describing Structure of Data Frames

While you can interact with data frames as lists, they also offer a number of additional capabilities and functions. For example, here are a few ways you can *inspect* the structure of a data frame:

Function	Description
`nrow(my.data.frame)`	Number of rows in the data frame
`ncol(my.data.frame)`	Number of columns in the data frame
`dim(my.data.frame)`	Dimensions (rows, columns) in the data frame
`colnames(my.data.frame)`	Names of the columns of the data frame
`rownames(my.data.frame)`	Names of the row of the data frame
`head(my.data.frame)`	Extracts the first few rows of the data frame (as a new data frame)
`tail(my.data.frame)`	Extracts the last few rows of the data frame (as a new data frame)
`View(my.data.frame)`	Opens the data frame in as spreadsheet-like viewer (only in RStudio)

Note that many of these description functions can also be used to *modify* the structure of a data frame. For example, you can use the `colnames` functions to assign a new set of column names to a data frame:

```r
# Using the same weights/heights as above:
my.data <- data.frame(name, height, weight)

# A vector of new column names
new.col.names <- c('first.name','how.tall','how.heavy')

# Assign that vector to be the vector of column names
colnames(my.data) <- new.col.names
```

9.1.3 Accessing Data in Data Frames

As stated above, since data frames *are* lists, it's possible to use **dollar notation** (`my.data.frame$column.name`) or **double-bracket notation** (`my.data.frame[['column.name']]`) to access entire columns. However, R also uses a variation of **single-bracket notation** which allows you to access individual data elements (cells) in the table. In this syntax, you put *two* values separated by a comma (**,**) inside the brackets—the first for which row and the second for which column you wish you extract:

Syntax	Description	Example
`my.df[row.num, col.num]`	Element by row and column indices	`my.frame[2,3]` (element in the second row, third column)
`my.df[row.name, col.name]`	Element by row and column names	`my.frame['Ada','height']` (element in row *named* `Ada` and column *named* `height`; the `height` of `Ada`)
`my.df[row, col]`	Element by row and col; can mix indices and names	`my.frame[2,'height']` (second element in the `height` column)
`my.df[row, ]`	All elements (columns) in row index or name	`my.frame[2,]` (all columns in the second row)
`my.df[, col]`	All elements (rows) in a col index or name	`my.frame[,'height']` (all rows in the `height` column; equivalent to list notations)

Take special note of the 4th option's syntax (for retrieving rows): you still include the comma (`,`), but because you leave *which column* blank, you get all of the columns!

```
# Extract the second row
my.data[2, ]  # comma

# Extract the second column AS A VECTOR
my.data[, 2]  # comma

# Extract the second column AS A DATA FRAME (filtering)
my.data[2]  # no comma
```

(Extracting from more than one column will produce a *sub-data frame*; extracting from just one column will produce a vector).

And of course, because *everything is a vector*, you're actually specifying vectors of indices to extract. This allows you to get multiple rows or columns:

```r
# Get the second through fourth rows
my.data[2:4, ]

# Get the `height` and `weight` columns
my.data[, c('height', 'weight')]

# Perform filtering
my.data[my.data$height > 60, ] # rows for which `height` is greater than 60
```

9.2 Working with CSV Data

So far you've been constructing your own data frames by "hard-coding" the data values. But it's much more common to load that data from somewhere else, such as a separate file on your computer or by downloading it off the internet. While R is able to ingest data from a variety of sources, this chapter will focus on reading tabular data in **comma separated value** (CSV) format, usually stored in a `.csv` file. In this format, each line of the file represents a record (*row*) of data, while each feature (*column*) of that record is separated by a comma:

```
Ada, 58, 115
Bob, 59, 117
Chris, 60, 120
Diya, 61, 123
Emma, 62, 126
```

Most spreadsheet programs like Microsoft Excel, Numbers, or Google Sheets are simply interfaces for formatting and interacting with data that is saved in this format. These programs easily import and export `.csv` files; however `.csv` files are unable to save the formatting done in those programs—the files only store the data!

You can load the data from a `.csv` file into R by using the `read.csv()` function:

```r
# Read data from the file `my_file.csv` into a data frame `my.data`
my.data <- read.csv('my_file.csv', stringsAsFactors=FALSE)
```

Again, use the `stringsAsFactors` argument to make sure string data is stored as a *vector* rather than as a *factor* (see below). This function will return a data frame, just like those described above!

Important Note: If for whatever reason an element is missing from a data frame (which is very common with real world data!), R will fill that cell with the logical value `NA` (distinct from the string `"NA"`), meaning "**N**ot **A**vailable". There are multiple ways to handle this in an analysis; see this link among others for details.

9.2.1 Working Directory

The biggest complication when loading `.csv` files is that the `read.csv()` function takes as an argument a **path** to the file. Because you want this script to work on any computer (to support collaboration, as well as things like assignment grading), you need to be sure to use a **relative path** to the file. The question is: *relative to what*?

Like the command-line, the R interpreter (running inside R Studio) has a **current working directory** from which all file paths are relative. The trick is that ***the working directory is not the directory of the current script file!***

- This makes sense if you think about it: you can run R commands through the console without having a script, and you can have open multiple script files from separate folders that are all interacting with the same execution environment.

Just as you can view the current working directory when on the commandline (using `pwd`), you can use an R function to view the current working directory when in R:

```
# get the absolute path to the current working directory
getwd()
```

You often will want to change the working directory to be your "project" directory (wherever your scripts and data files happen to be). It is possible to change the current working directory using the `setwd()` function. However, this function would also take an absolute path, so doesn't fix the problem. You would not want to include this absolute path in your script (though you could use it from the console).

One solution is to use the tilde (~) shortcut to specify your directory:

```
# Set working directory on Desktop
setwd("~/Desktop/project-name")
```

This enables you to work across machines, as long as the project is stored in the same location on each machine.

Another solution is to use R Studio itself to change the working directory. This is reasonable because the working directory is a property of the *current running environment*, which is what R Studio makes accessible! The easiest way to do this is to use the **`Session > Set Working Directory`** menu options: you can either set the working directory `To Source File Location` (the folder containing whichever `.R` script you are currently editing; this is usually what you want), or you can browse for a particular directory with `Choose Directory`.

You should do this whenever you hit a "path" problem when loading external files. If you want to do this repeatedly by calling `setwd()` from your script to

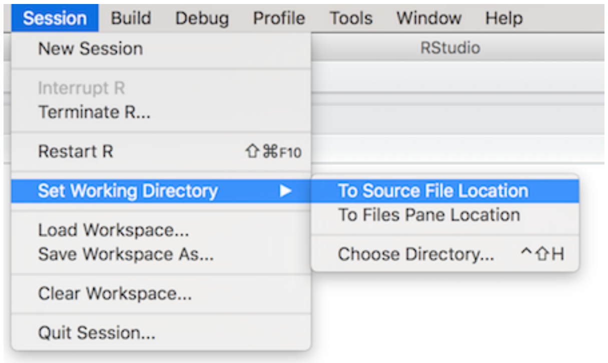


Figure 9.2: Use `Session > Set Working Directory` to change the working directory through R Studio

an absolute path, you may want to keep it commented out (`# setwd(...)`) so it doesn't cause problems for others who try to run your script.

9.3 Factor Variables

Factors are a way of *optimizing* variables that consist of a finite set of categories (i.e., they are **categorical (nominal) variables**).

For example, imagine that you had a vector of shirt sizes which could only take on the values `small`, `medium`, or `large`. If you were working with a large dataset (thousands of shirts!), it would end up taking up a lot of memory to store the character strings (5+ letters per word at 1 or more bytes per letter) for each one of those variables.

A **factor** on the other hand would instead store a *number* (called a **level**) for each of these character strings: for example, `1` for `small`, `2` for `medium`, or `3` for `large` (though the order or specific numbers will vary). R will remember the relationship between the integers and their **labels** (the strings). Since each number only takes 4 bytes (rather than 1 per letter), factors allow R to keep much more information in memory.

```
# Start with a character vector of shirt sizes
shirt.sizes <- c('small', 'medium', 'small', 'large', 'medium', 'large')

# Convert to a vector of factor data
shirt.sizes.factor <- as.factor(shirt.sizes)

# View the factor and its levels
print(shirt.sizes.factor)
```

```
# The length of the factor is still the length of the vector, not the number of leve
length(shirt.sizes.factor)  # 6
```

When you print out the `shirt.sizes.factor` variable, R still (intelligently) prints out the **labels** that you are presumably interested in. It also indicates the **levels**, which are the *only* possible values that elements can take on.

It is worth re-stating: **factors are not vectors**. This means that most all the operations and functions you want to use on vectors *will not work*:

```
# Create a factor of numbers (factors need not be strings)
num.factors <- as.factor(c(10,10,20,20,30,30,40,40))

# Print the factor to see its levels
print(num.factors)

# Multiply the numbers by 2
num.factors * 2  # Error: * not meaningful
                 # returns vector of NA instead

# Changing entry to a level is fine
num.factors[1] <- 40

# Change entry to a value that ISN'T a level fails
num.factors[1] <- 50  # Error: invalid factor level
                      # num.factors[1] is now NA
```

If you create a data frame with a string vector as a column (as what happens with `read.csv()`), it will automatically be treated as a factor *unless you explicitly tell it not to*:

```
# Vector of shirt sizes
shirt.size <- c('small', 'medium', 'small', 'large', 'medium', 'large')

# Vector of costs (in dollars)
cost <- c(15.5, 17, 17, 14, 12, 23)

# Data frame of inventory (with factors, since didn't say otherwise)
shirts.factor <- data.frame(shirt.size, cost)

# The shirt.size column is a factor
is.factor(shirts.factor$shirt.size)  # TRUE

# Can treat this as a vector; but better to fix how the data is loaded
as.vector(shirts.factor$shirt.size)  # a vector

# Data frame of orders (without factoring)
```

```r
shirts <- data.frame(shirt.size, cost, stringsAsFactors=FALSE)

# The shirt.size column is NOT a factor
is.factor(shirts$shirt.size)  # FALSE
```

This is not to say that factors can't be useful (beyond just saving memory)! They offer easy ways to group and process data using specialized functions:

```r
shirt.size <- c('small', 'medium', 'small', 'large', 'medium', 'large')
cost <- c(15.5, 17, 17, 14, 12, 23)

# Data frame of inventory (with factors)
shirts.factor <- data.frame(shirt.size, cost)

# Produce a list of data frames, one for each factor level
# first argument is the data frame to split, second is the factor to split by
shirt.size.frames <- split(shirts.factor, shirts.factor$shirt.size)

# Apply a function (mean) to each factor level
#   first argument is the vector to apply the function to,
#   second argument is the factor to split by
#   third argument is the name of the function
tapply(shirts$cost, shirts$shirt.size, mean)
```

However, in general this course is more interested in working with data as vectors, thus you should always use `stringsAsFactors=FALSE` when creating data frames or loading `.csv` files that include strings.

Resources

- R Tutorial: Data Frames
- R Tutorial: Data Frame Indexing
- Quick-R: Missing Values
- Factor Variables (UCLA)

Chapter 10

The `dplyr` Library

The **`dplyr`** ("dee-ply-er") package is the preeminent tool for data wrangling in R (and perhaps, in data science more generally). It provides programmers with an intuitive vocabulary for executing data management and analysis tasks. Learning and utilizing this package will make your data preparation and management process faster and easier to understand. This chapter introduces the philosophy behind the library and an overview of how to use the library to work with dataframes using its expressive and efficient syntax.

10.1 A Grammar of Data Manipulation

Hadley Wickham, the creator of the `dplyr` package, fittingly refers to it as a ***Grammar of Data Manipulation***. This is because the package provides a set of **verbs** (functions) to describe and perform common data preparation tasks. One of the core challenge in programming is mapping from questions about a dataset to specific programming operations. The presence of a data manipulation grammar makes this process smoother, as it enables you to use the same vocabulary to both *ask* questions and *write* your program. Specifically, the `dplyr` grammar lets you easily talk about and perform tasks such as:

- **select** specific features (columns) of interest from the data set
- **filter** out irrelevant data and only keep observations (rows) of interest
- **mutate** a data set by adding more features (columns)
- **arrange** the observations (rows) in a particular order
- **summarize** the data in terms of aspects such as the mean, median, or maximum
- **join** multiple data sets together into a single data frame

You can use these words when describing the *algorithm* or process for interrogating data, and then use `dplyr` to write code that will closely follow your "plain language" description because it uses functions and procedures that share the same language. Indeed, many real-world questions about a dataset come down to isolating specific rows/columns of the data set as the "elements of interest", and then performing a simple comparison or computation (mean, count, max, etc.). While it is possible to perform this computation with basic R functions—the `dplyr` library makes it much easier to write and read such code.

10.2 Using **dplyr** Functions

The `dplyr` package provides functions that mirror the above verbs. Using this package's functions will allow you to quickly and effectively write code to ask questions of your data sets.

Since `dplyr` is an external package, you will need to install it (once per machine) and load it to make the functions available:

```r
install.packages("dplyr")  # once per machine
library("dplyr")
```

After loading the library, you can call any of the functions just as if they were the built-in functions you've come to know and love.

For each `dplyr` function discussed here, the **first argument** to the function is a data frame to manipulate, with the rest of the arguments providing more details about the manipulation.

IMPORTANT NOTE: inside the function argument list (inside the parentheses), we refer to data frame columns **without quotation marks**—that is, we just give the column names as *variable names*, rather than as *character strings*. This is refered to as non-standard evaluation, and is described in more detail below; while it makes code easier to write and read, it can occasionally create challenges.

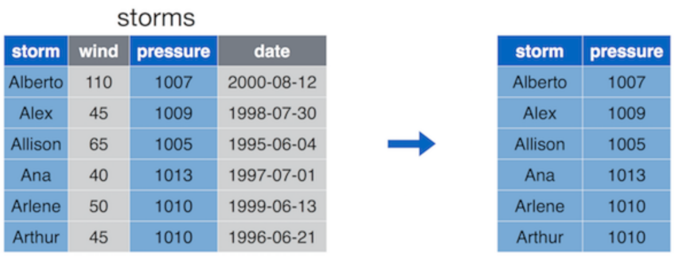
The images in this section come from the RStudio's STRATA NYC R-Day workshop, which was presented by Nathan Stephens.

10.2.1 Select

The **`select()`** operation allows you to choose and extract **columns** of interest from your data frame.

```r
# Select `storm` and `pressure` columns from `storms` data frame
storm.info <- select(storms, storm, pressure)
```

The `select()` function takes in the data frame to select from, followed by the names of the columns you wish to select (quotation marks are optional!)



storm	wind	pressure	date
Alberto	110	1007	2000-08-12
Alex	45	1009	1998-07-30
Allison	65	1005	1995-06-04
Ana	40	1013	1997-07-01
Arlene	50	1010	1999-06-13
Arthur	45	1010	1996-06-21

storm	pressure
Alberto	1007
Alex	1009
Allison	1005
Ana	1013
Arlene	1010
Arthur	1010

Figure 10.1: Diagram of the `select()` function (by Nathan Stephens).

This function is equivalent to simply extracting the columns:

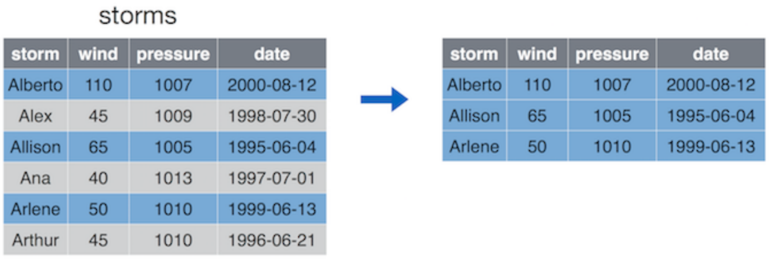
```
# Extract columns by name
storm.info <- storms[, c("storm", "pressure")]  # Note the comma!
```

But easier to read and write!

10.2.2 Filter

The **`filter()`** operation allows you to choose and extract **rows** of interest from your data frame (contrasted with `select()` which extracts *columns*).

```
# Select rows whose `wind` column is greater than or equal to 50
some.storms <- filter(storms, wind >= 50)
```



storm	wind	pressure	date
Alberto	110	1007	2000-08-12
Alex	45	1009	1998-07-30
Allison	65	1005	1995-06-04
Ana	40	1013	1997-07-01
Arlene	50	1010	1999-06-13
Arthur	45	1010	1996-06-21

storm	wind	pressure	date
Alberto	110	1007	2000-08-12
Allison	65	1005	1995-06-04
Arlene	50	1010	1999-06-13

Figure 10.2: Diagram of the `filter()` function (by Nathan Stephens).

The `filter()` function takes in the data frame to filter, followed by a comma-separated list of conditions that each returned *row* must satisfy. Note again that columns are provided without quotation marks!

- R will extract the rows that match **all** conditions. Thus you are specifying that you want to filter down a data frame to contain only the rows that meet Condition 1 **and** Condition 2.

This function is equivalent to simply extracting the rows:

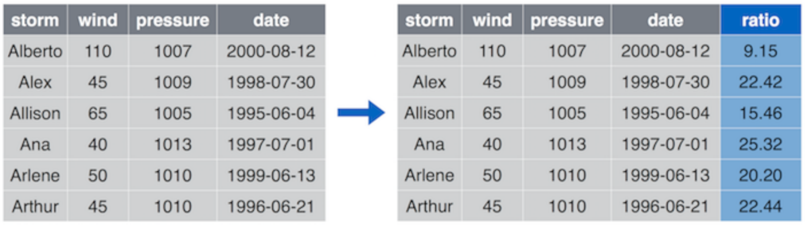
```
# Extract rows by condition
some.storms <- storms[storms$wind >= 50, ]  # Note the comma!
```

As the number of conditions increases, it is **far easier** to read and write `filter()` functions, rather than squeeze your conditions into brackets.

10.2.3 Mutate

The **`mutate()`** operation allows you to create additional **columns** for your data frame.

```
# Add `ratio` column that is ratio between pressure and wind
storms <- mutate(storms, ratio = pressure/wind)  # Replace existing `storms` frame w
```



storm	wind	pressure	date
Alberto	110	1007	2000-08-12
Alex	45	1009	1998-07-30
Allison	65	1005	1995-06-04
Ana	40	1013	1997-07-01
Arlene	50	1010	1999-06-13
Arthur	45	1010	1996-06-21

storm	wind	pressure	date	ratio
Alberto	110	1007	2000-08-12	9.15
Alex	45	1009	1998-07-30	22.42
Allison	65	1005	1995-06-04	15.46
Ana	40	1013	1997-07-01	25.32
Arlene	50	1010	1999-06-13	20.20
Arthur	45	1010	1996-06-21	22.44

Figure 10.3: Diagram of the `mutate()` function (by Nathan Stephens).

The `mutate()` function takes in the data frame to mutate, followed by a comma-separated list of columns to create using the same **`name = vector`** syntax you used when creating **lists** or **data frames** from scratch. As always, the names of the columns in the data frame are used without quotation marks.

- Despite the name, the `mutate()` function doesn't actually change the data frame; instead it returns a *new* data frame that has the extra columns added. You will often want to replace the old data frame variable with this new value.

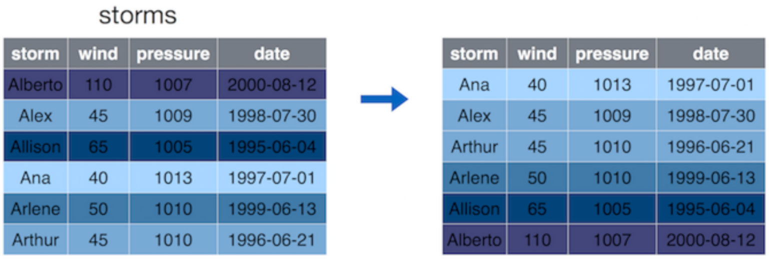
In cases where you are creating multiple columns (and therefore writing really long code instructions), you should break the single statement into multiple lines for readability. Because you haven't closed the parentheses on the function arguments, R will not treat each line as a separate statement.

```
# Generic mutate command
more.columns <- mutate(my.data.frame,
                       new.column.1 = old.column * 2,
                       new.column.2 = old.column * 3,
                       new.column.3 = old.column * 4
                      )
```

10.2.4 Arrange

The **arrange()** operation allows you to **sort the rows** of your data frame by some feature (column value).

```
# Arrange storms by INCREASING order of the `wind` column
sorted.storms <- arrange(storms, wind)
```



storms

storm	wind	pressure	date
Alberto	110	1007	2000-08-12
Alex	45	1009	1998-07-30
Allison	65	1005	1995-06-04
Ana	40	1013	1997-07-01
Arlene	50	1010	1999-06-13
Arthur	45	1010	1996-06-21

storm	wind	pressure	date
Ana	40	1013	1997-07-01
Alex	45	1009	1998-07-30
Arthur	45	1010	1996-06-21
Arlene	50	1010	1999-06-13
Allison	65	1005	1995-06-04
Alberto	110	1007	2000-08-12

Figure 10.4: Diagram of the `arrange()` function (by Nathan Stephens).

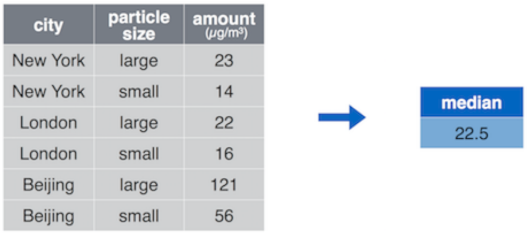
By default, the `arrange()` function will sort rows in **increasing** order. To sort in **reverse** (decreasing) order, place a minus sign (`-`) in front of the column name (e.g., `-wind`). You can also use the `desc()` helper function (e.g, `desc(wind)`).

- You can pass multiple arguments into the `arrange()` function in order to sort first by `argument.1`, then by `argument.2`, and so on.
- Again, this doesn't actually modify the argument data frame—instead returning a new data frame you'll need to store.

10.2.5 Summarize

The **summarize()** function (equivalently `summarise()` for those using the British spelling) will generate a *new* data frame that contains a "summary" of a **column**, computing a single value from the multiple elements in that column.

```
# Compute the median value of the `amount` column
summary <- summarize(pollution, median = median(amount))
```



city	particle size	amount (μg/m³)
New York	large	23
New York	small	14
London	large	22
London	small	16
Beijing	large	121
Beijing	small	56

median
22.5

Figure 10.5: Diagram of the `summarize()` function (by Nathan Stephens).

The `summarize()` function takes in the data frame to mutate, followed by the values that will be included in the resulting summary table. You can use multiple arguments to include multiple summaries in the same statement:

```
# Compute statistics for the `amount` column
summaries <- summarize(pollution,
                       median = median(amount),  # median value
                       mean = mean(amount),      # "average" value
                       sum = sum(amount),        # total value
                       count = n()               # number of values (neat trick!)
                       )
```

Note that the `summarize()` function is particularly useful for grouped operations (see below), as you can produce summaries of different groups of data.

10.2.6 Distinct

The **`distinct()`** operation allows you to extract distinct values (rows) from your data frame—that is, you'll get one row for each different value in the dataframe (or set of selected **columns**). This is a useful tool to confirm that you don't have **duplicate observations**, which often occurs in messy datasets.

For example (no diagram available):

```
# Create a quick data frame
x <- c(1, 1, 2, 2, 3, 3, 4, 4)  # duplicate x values
y <- 1:8                        # unique y values
my.df <- data.frame(x, y)

# Select distinct rows, judging by the `x` column
distinct.rows <- distinct(my.df, x)
                                    #   x
                                    # 1 1
                                    # 2 2
                                    # 3 3
                                    # 4 4

# Select distinct rows, judging by the `x` and `y`columns
distinct.rows <- distinct(my.df, x, y)  # returns whole table, since no duplicate rows
```

While this is a simple way to get a unique set of rows, **be careful** not to lazily remove rows of your data which may be important.

10.3 Multiple Operations

You've likely encountered a number of instances in which you want to take the results from one function and pass them into another function. Your approach thus far has often been to create *temporary variables* for use in your analysis. For example, if you're using the `mtcars` dataset, you may want to ask a simple question like,

> Which 4-cylinder car gets the best milage per gallon?

This simple question actually requires a few steps:

1. *Filter* down the dataset to only 4-cylinder cars
2. Of the 4-cylinder cars, *filter* down to the one with the highest mpg
3. *Select* the car name of the car

You could then implement each step as follows:

```
# Preparation: add a column that is the car name
mtcars.named <- mutate(mtcars, car.name = row.names(mtcars))

# 1. Filter down to only four cylinder cars
four.cyl <- filter(mtcars.named, cyl == 4)

# 2. Filter down to the one with the highest mpg
best.four.cyl <- filter(four.cyl, mpg == max(mpg))

# 3. Select the car name of the car
best.car.name <- select(best.four.cyl, car.name)
```

While this works fine, it clutters the work environment with variables you won't need to use again, and which can potentially step on one anothers toes. It can help with readability (the results of each step is explicit), but those extra variables make it harder to modify and change the algorithm later (you have to change them in two places).

An alternative to saving each step as a distinct, named variable would be to utilize **anonymous variables** and write the desired statements **nested** within other functions. For example, you could write the algorithm above as follows:

```
# Preparation: add a column that is the car name
mtcars.named <- mutate(mtcars, car.name = row.names(mtcars))

# Write a nested operation to return the best car name
best.car.name <- select(  # 3. Select car name of the car
                  filter(  # 2. Filter down to the one with the highest mpg
                    filter( # 1. Filter down to only four cylinder cars
                      mtcars.named,  # arguments for the Step 1 filter
                      cyl == 4
                    ),
                    mpg == max(mpg)  # other arguments for the Step 2 filter
                  ),
                  car.name  # other arguments for the Step 3 select
                )
```

This version uses *anonymous variables*—result values which are not assigned to names (so are anonymous), but instead are immediately used as the arguments to another function. You've used these frequently with the `print()` function and with filters (those vectors of `TRUE` and `FALSE` values)—and even the `max(mpg)` in the Step 2 filter is an anonymous variable!

This *nested* version performs the same results as the *temporary variable* version without creating the extra variables, but even with only 3 steps it can get quite complicated to read—in a large part because you have to think about it "inside out", with the stuff in the middle evaluating first. This will obviously become undecipherable for more involved operations.

10.3.1 The Pipe Operator

Luckily, `dplyr` provides a cleaner and more effective way of achieving the same task (that is, using the result of one function as an argument to the next). The **pipe operator** (**`%>%`**) indicates that the result from the first function operand should be passed in as **the first argument** to the next function operand!

As a simple example:

```
# nested version: evaluate c(), then max(), then print()
print(max(c(2, 0, 1)))

# pipe version
c(1,2,3) %>%   # do first function
  max() %>%   # which becomes the _first_ argument to the next function
  print()  # which becomes the _first_ argument to the next function
```

Or as another version of the above data wrangling:

```
# Preparation: add a column that is the car name
mtcars.named <- mutate(mtcars, car.name = row.names(mtcars))

best.car.name <- filter(mtcars.named, cyl == 4) %>%  # Step 1
  filter(mpg == max(mpg)) %>%  # Step 2
  select(car.name)  # Step 3
```

- Yes, the `%>%` operator is awkward to type and takes some getting use to (especially compared to the command-line's use of `|` to pipe). However, you can ease the typing by using the RStudio keyboard shortcut `cmd + shift + m`.

The pipe operator is part of the `dplyr` package (it is only available if you load that package), but it will work with *any* function, not just `dplyr` ones! This syntax, while slightly odd, can completely change and simplify the way you write code to ask questions about your data!

10.4 Grouped Operations

`dplyr` functions are powerful, but they are truly awesome when you can apply them to **groups of rows** within a data set. For example, the above use of `summarize()` isn't particularly useful since it just gives a single summary for a given column (which you could have done anyway). However, a **grouped** operation would allow you to compute the same summary measure (`mean`, `median`, `sum`, etc.) automatically for multiple groups of rows, enabling you to ask more nuanced questions about your data set.

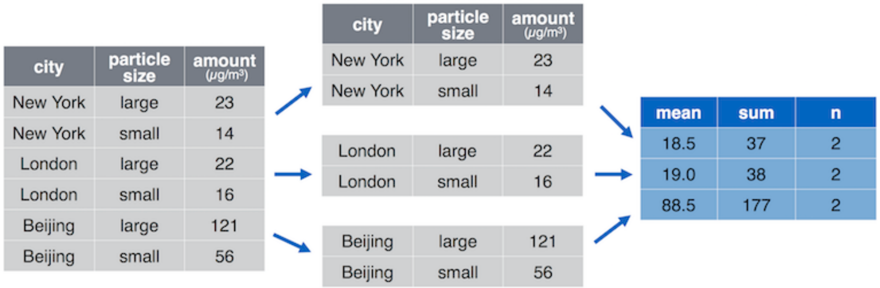
The **`group_by()`** operation allows you to break a data frame down into *groups* of rows, which can then have the other verbs (e.g., `summarize`, `filter`, etc). applied to each one.

```
# Get summary statistics by city
city.summary <- group_by(pollution, city) %>%
  summarize( # first argument (the data frame) is received from the pipe
    mean = mean(amount),
    sum = sum(amount),
    n = n()
  )
```

As another example, if you were using the `mtcars` dataset, you may want to answer this question:

> What are the differences in mean miles per gallon for cars with different numbers of gears (3, 4, or 5)?

This simple question requires computing the mean for different subsets of the data. Rather than explicitly breaking your data into different groups (a.k.a.



city	particle size	amount (μg/m³)
New York	large	23
New York	small	14
London	large	22
London	small	16
Beijing	large	121
Beijing	small	56

city	particle size	amount (μg/m³)
New York	large	23
New York	small	14

London	large	22
London	small	16

Beijing	large	121
Beijing	small	56

mean	sum	n
18.5	37	2
19.0	38	2
88.5	177	2

Figure 10.6: Diagram of the `group_by()` function (by Nathan Stephens).

bins or *chunks*) and running the same operations on each, you can use the `group_by()` function to accomplish this in a single command:

```
# Group cars by gear number, then compute the mean and median mpg
gear.summary <- group_by(mtcars, gear) %>%  # group by gear
                 summarize(mean = mean(mpg))  # calculate mean
# Computing the difference between scores is done elsewhere (or by hand!)
```

Thus grouping can allow you to quickly and easily compare different subsets of your data!

10.5 Joins

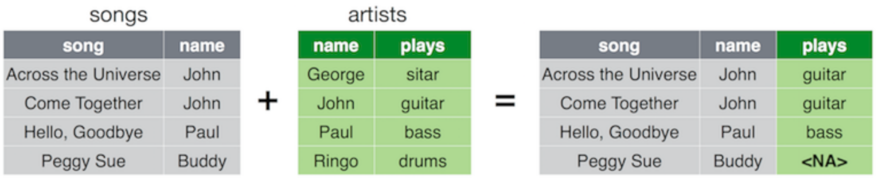
When working with real-world data, you'll often find that that data is stored across *multiple* files or data frames. This can be done for a number of reasons. For one, it can help to reduce memory usage (in the same manner as **factors**). For example, if you had a data frame containing information on students enrolled in university courses, you might store information about each course (the instructor, meeting time, and classroom) in a separate data frame rather than duplicating that information for every student that takes the same course. You also may simply want to keep your information organized: e.g., have student information in one file, and course information in another.

- This separation and organization of data is a core concern in the design of relational databases, a common topic of study within Information Schools.

But at some point, you'll want to access information from both data sets (e.g., you need to figure out a student's schedule), and thus need a way to combine the data frames. This process is called a **join** (because you are "joining" the data frames together). When you perform a join, you identify **columns** which are present in both tables. Those column values are then used as **identifiers** to determine which rows in each table correspond to one another, and thus will be combined into a row in the resulting joined table.

The **`left_join()`** operation is one example of a join. This operation looks for matching columns between the two data frames, and then returns a new data frame that is the first ("left") operand with extra columns from the second operand added on.

```
# Combine (join) songs and artists data frames
left_join(songs, artists)
```



songs

song	name
Across the Universe	John
Come Together	John
Hello, Goodbye	Paul
Peggy Sue	Buddy

+

artists

name	plays
George	sitar
John	guitar
Paul	bass
Ringo	drums

=

song	name	plays
Across the Universe	John	guitar
Come Together	John	guitar
Hello, Goodbye	Paul	bass
Peggy Sue	Buddy	<NA>

Figure 10.7: Diagram of the `left_join()` function (by Nathan Stephens).

To understand how this works, consider a specific example where you have a table of student ids and the students' contact information. You also have a separate table of student ids and the students' majors (your institution very well may store this information in separate tables for privacy or organizational reasons).

```
# Table of contact information
student.contact <- data.frame(
    student.id = c(1, 2, 3, 4),  # id numbers
    email = c("id1@school.edu", "id2@school.edu", "id3@school.edu", "id4@school.edu")
)

# Table of information about majors
student.majors <- data.frame(
  student.id = c(1, 2, 3),  # id numbers
  major = c('sociology', 'math', 'biology')
)
```

Notice that both tables have a `student.id` column, allowing you to "match" the rows from the `student.contact` table to the `student.majors` table and merge them together:

```
# Join tables by the student.id column
merged.student.info <- left_join(student.contact, student.majors)
                        #    student.id           email     major
                        # 1           1 id1@school.edu sociology
                        # 2           2 id2@school.edu      math
                        # 3          3 id3@school.edu   biology
                        # 4          4  id4@school.edu      <NA>
```

When you perform this **left join**, R goes through each row in the table on the "left" (the first argument), looking at the shared column(s) (`student.id`). For each row, it looks for a corresponding value in `student.majors$student.id`, and if it finds one then it adds any data from columns that are in `student.majors` but *not* in `student.contact` (e.g., `major`) to new columns in the resulting table, with values from whatever the matching row was. Thus student #1 is given a `major` of "sociology", student #2 is given a `major` of "math", and student #4 is given a `major` of `NA` (because that student had no corresponding row in `student.majors`!)

- In short, a **left join** returns all of the rows from the *first* table, with all of the columns from *both* tables.

R will join tables by any and all shared columns. However, if the names of your columns don't match specifically, you can also specify a `by` argument indicating which columns should be used for the matching:

```
# Use the named `by` argument to specify (a vector of) columns to match on
left_join(student.contact, student.majors, by="student.id")
```

- With the `by` argument, column name *is* a string (in quotes) because you're specifying a vector of column names (the string literal is a vector length 1).

Notice that because of how a left join is defined, **the argument order matters!** The resulting table only has rows for elements in the *left* (first) table; any unmatched elements in the second table are lost. If you switch the order of the operands, you would only have information for students with majors:

```
# Join tables by the student.id column
merged.student.info <- left_join(student.majors, student.contact)  # switched order!
                        #   student.id     major          email
                        # 1          1 sociology id1@school.edu
                        # 2          2      math id2@school.edu
                        # 3          3   biology id3@school.edu
```

You don't get any information for student #4, because they didn't have a record in the left-hand table!

Because of this behavior, `dplyr` (and relational database systems in general) provide a number of different kinds of joins, each of which influences *which* rows are included in the final table. Note that in any case, *all* columns from *both* tables will be included, with rows taking on any values from their matches in the second table.

- **`left_join`** All rows from the first (left) data frame are returned. That is, you get all the data from the left-hand table, with extra column values added from the right-hand table. Left-hand rows without a match will have `NA` in the right-hand columns.

- **`right_join`** All rows from the second (right) data frame are returned. That is, you get all the data from the right-hand table, with extra column values added from the left-hand table. Right-hand rows without a match will have `NA` in the left-hand columns. This is the "opposite" of a `left_join`, and the equivalent of switching the operands.
- **`inner_join`** Only rows in **both** data frames are returned. That is, you get any rows that had matching observations in both tables, with the column values from both tables. There will be no additional `NA` values created by the join. Observations from the left that had no match in the right, or observations in the right that had no match in the left, will not be returned at all.
- **`full_join`** All rows from **both** data frames are returned. That is, you get a row for any observation, whether or not it matched. If it happened to match, it will have values from both tables in that row. Observations without a match will have `NA` in the columns from the other table.

The key to deciding between these is to think about what set of data you want as your set of observations (rows), and which columns you'd be okay with being `NA` if a record is missing.

Note that these are all *mutating joins*, which add columns from one table to another. `dplyr` also provides *filtering joins* which exclude rows based on whether they have a matching observation in another table, and *set operations* which combine observations as if they were set elements. See the documentation for more detail on these options, but in this course we'll be primarily focusing on the mutating joins described above.

10.6 Non-Standard Evaluation vs. Standard Evaluation

One of the features that makes `dplyr` such a clean and attractive way to write code is that inside of each function, you've been able to write column variable names **without quotes**. This is called **non-standard evaluation (NSE)** (it is *not* the *standard* way that code is *evaluated*, or interpreted), and is useful primarily because of how it reduces typing (along with some other benefits when working with databases). In particular, `dplyr` will "quote" expressions for you, converting those variables (symbols) into values that can be used to refer to column names.

Most of the time this won't cause you any problems—you can either use NSE to refer to column names without quotes, or provide the quotes yourself. You can even use variables to store the name of a column of interest!

```
# Normal, non-standard evaluation version
mpg <- select(mtcars, mpg)

# "Standard-evaluation" version (same result)
mpg <- select(mtcars, 'mpg')  # with quotes! 'mpg' is a normal value!

# Make the column name a variable
which.col <- 'mpg'
my.column <- select(mtcars, which.col)
```

However, this NSE can sometimes trip you up when using more complex functions such as `summarize()` or `group_by()`, or when you want to create your own functions that use NSE.

```
which.col <- 'mpg'
summarize(mtcars, avg = mean(which.col))  # In mean.default(which.col) :
                                          # argument is not numeric or logical: returning NA
```

In this case, the `summarize()` function is trying to "quote" what we typed in (the `which.col` variable name&dmash;not it's `mpg` value), and then hitting a problem because there is no column of that name (it can't resolve that column name to a column index).

To fix this problem, there are two parts: first, you need to explicitly tell R that the *value* of `which.col` (`mpg`) is actually the value that needs to be automatically "quoted"—that is, that `mpg` is really a variable! Variable names in R are referred to as **symbols**—a symbol refers to the variable label itself. You can explicitly change a value into a symbol by using the `rlang::sym()` function (the `sym()` function found in the `rlang` library; the `::` indicates that the function belongs to a library).

```
which.col.sym <- rlang::sym(which.col)  # convert to a symbol
print(which.col.sym)  # => mpg (but not in quotes, because it's not a string!)
```

Second, you will need to tell the `summarize()` function that it should *not* quote this symbol (because you've already converted it into a variable)—what is called **unquoting**. In `dplyr`, you "unquote" a parameter to a method by including two exclamation points in front of it:

```
summarize(mtcars, avg = mean(!!which.col.sym))  # arranges by the specified column
```

There are many more details involved in this "quoting/unquoting" process, which are described in this tutorial (though that is currently being updated with better examples).

10.6.1 Explicit Standard Evaluation

Alternatively, older versions of `dplyr` supplied functions that *explicitly* performed **standard evaluation (SE)**—that is, they provide no quoting and expected you to do that work yourself. While now considered deprecated, they can still be useful if you are having problems with the new quoting system. These functions have the exact same names as the normal verb functions, except are followed by an underscore (`_`):

```
# Normal, non-standard evaluation version
mpg <- select(mtcars, mpg)

# Standard-evaluation version (same result)
mpg <- select_(mtcars, 'mpg')  # with quotes! 'mpg' is a normal value!

# Normal, non-standard evaluation version of equations
mean.mpg <- summarize(mtcars, mean(mpg))

# Standard-evaluation version of equations (same result)
mean.mpg <- summarize_(mtcars, 'mean(mpg)')

# Which column you're interested in
which.column <- 'mpg'

# Use standard evaluation to execute function:
my.column <- arrange_(mtcars, which.column)
```

Yes, it does feel a bit off that the “normal” way of using `dplyr` is the “non-standard” way. Remember that using SE is the “different” approach

The non-standard evaluation offered by `dplyr` can make it quick and easy to work with data when you know its structure and variable names, but can be a challenge when trying to work with variables. Often in that case, you may want to instead use the standard data frame syntax (e.g., bracket notation) described in Chapter 9.

Resources

- Introduction to dplyr
- dplyr and pipes: the basics (blog)
- Two-table verbs
- DPLYR Join Cheatsheet (Jenny Bryan)
- Non-standard evaluation
- Data Manipulation with DPLYR (R-bloggers)
- Data Manipulation in R (DataCamp)

Chapter 11

Accessing Web APIs

R is able to load data from external packages or read it from locally-saved `.csv` files, but it is also able to download data directly from web sites on the internet. This allows scripts to always work with the latest data available, performing analysis on data that may be changing rapidly (such as from social networks or other live events). Web services may make their data easily accessible to computer programs like R scripts by offering an **Application Programming Interface (API)**. A web service's API specifies *where* and *how* particular data may be accessed, and many web services follow a particular style known as *Representational State Transfer (REST)*. This chapter will cover how to access and work with data from these *RESTful APIs*.

11.1 What is a Web API?

An **interface** is the point at which two different systems meet and *communicate*: exchanging informations and instructions. An **Application Programming Interface (API)** thus represents a way of communicating with a computer application by writing a computer program (a set of formal instructions understandable by a machine). APIs commonly take the form of **functions** that can be called to give instructions to programs—the set of functions provided by a library like `dplyr` make up the API for that library.

While most APIs provide an interface for utilizing *functionality*, other APIs provide an interface for accessing *data*. One of the most common sources of these data apis are **web services**: websites that offer an interface for accessing their data.

With web services, the interface (the set of "functions" you can call to access the data) takes the form of **HTTP Requests**—a *request* for data sent following the ***HyperText Transfer Protocol***. This is the same protocol (way of communicating) used by your browser to view a web page! An HTTP Request represents a message that your computer sends to a web server (another computer on the internet which "serves", or provides, information). That server, upon receiving the request, will determine what data to include in the **response** it sends *back* to the requesting computer. With a web browser, the response data takes the form of HTML files that the browser can *render* as web pages. With data APIs, the response data will be structured data that you can convert into R structures such as lists or data frames.

In short, loading data from a Web API involves sending an **HTTP Request** to a server for a particular piece of data, and then receiving and parsing the **response** to that request.

11.2 RESTful Requests

There are two parts to a request sent to an API: the name of the **resource** (data) that you wish to access, and a **verb** indicating what you want to do with that resource. In many ways, the *verb* is the function you want to call on the API, and the *resource* is an argument to that function.

11.2.1 URIs

Which **resource** you want to access is specified with a **Uniform Resource Identifier (URI)**. A URI is a generalization of a URL (Uniform Resource Locator)—what you commonly think of as "web addresses". URIs act a lot like the *address* on a postal letter sent within a large organization such as a university: you indicate the business address as well as the department and the person, and will get a different response (and different data) from Alice in Accounting than from Sally in Sales.

- Note that the URI is the **identifier** (think: variable name) for the resource, while the **resource** is the actual *data* value that you want to access.

Like postal letter addresses, URIs have a very specific format used to direct the request to the right resource.

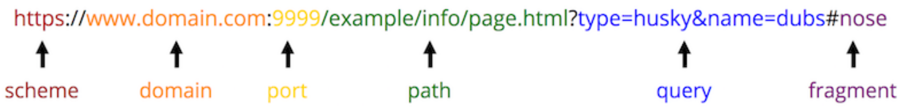

https://www.domain.com:9999/example/info/page.html?type=husky&name=dubs#nose
scheme
domain
port
path
query
fragment

Figure 11.1: The format (schema) of a URI.

Not all parts of the format are required—for example, you don't need a `port`, `query`, or `fragment`. Important parts of the format include:

- `scheme` (`protocol`): the "language" that the computer will use to communicate the request to this resource. With web services this is normally `https` (**s**ecure HTTP)
- `domain`: the address of the web server to request information from
- `path`: which resource on that web server you wish to access. This may be the name of a file with an extension if you're trying to access a particular file, but with web services it often just looks like a folder path!
- `query`: extra **parameters** (arguments) about what resource to access.

The `domain` and `path` usually specify the resource. For example, `www.domain.com/users` might be an *identifier* for a resource which is a list of users. Note that web services can also have "subresources" by adding extra pieces to the path: `www.domain.com/users/mike` might refer to the specific "mike" user in that list.

With an API, the domain and path are often viewed as being broken up into two parts:

- The **Base URI** is the domain and part of the path that is included on *all* resources. It acts as the "root" for any particular resource. For example, the GitHub API has a base URI of `https://api.github.com/`.

- An **Endpoint**, or which resource on that domain you want to access. Each API will have *many* different endpoints.

  For example, GitHub includes endpoints such as:

  - `/users/{user}` to get information about a specific `:user: id` (the `{}` indicate a "variable", in that you can put any username in there in place of the string `"{user}"`). Check out this example in your browser.
  - `/orgs/{organization}/repos` to get the repositories that are on an organization page. See an example.

Thus you can equivalently talk about accessing a particular **resource** and sending a request to a particular **endpoint**. The **endpoint** is appended to the end of the **Base URI**, so you could access a GitHub user by combining the **Base URI** (`https://api.github.com`) and **endpoint** (`/users/mkfreeman`) into a single string: `https://api.github.com/users/mkfreeman`. That URL will return a data structure of new releases, which you can request from your R program or simply view in your web-browser.

11.2.1.1 Query Parameters

Often in order to access only partial sets of data from a resource (e.g., to only get some users) you also include a set of **query parameters**. These are like extra arguments that are given to the request function. Query parameters are listed after a question mark **`?`** in the URI, and are formed as key-value pairs similar to how you named items in *lists*. The **key** (*parameter name*) is listed first, followed by an equal sign **`=`**, followed by the **value** (*parameter value*); note that you can't include any spaces in URIs! You can include multiple query parameters by putting an ampersand **`&`** between each key-value pair:

```
?firstParam=firstValue&secondParam=secondValue&thirdParam=thirdValue
```

Exactly what parameter names you need to include (and what are legal values to assign to that name) depends on the particular web service. Common examples include having parameters named `q` or `query` for searching, with a value being whatever term you want to search for: in `https://www.google.com/search?q=informatics`, the **resource** at the `/search` **endpoint** takes a query parameter `q` with the term you want to search for!

11.2.1.2 Access Tokens and API Keys

Many web services require you to register with them in order to send them requests. This allows them to limit access to the data, as well as to keep track of who is asking for what data (usually so that if someone starts "spamming" the service, they can be blocked).

To facilitate this tracking, many services provide **Access Tokens** (also called **API Keys**). These are unique strings of letters and numbers that identify a particular developer (like a secret password that only works for you). Web services will require you to include your *access token* as a query parameter in the request; the exact name of the parameter varies, but it often looks like `access_token` or `api_key`. When exploring a web service, keep an eye out for whether they require such tokens.

Access tokens act a lot like passwords; you will want to keep them secret and not share them with others. This means that you **should not include them in your committed files**, so that the passwords don't get pushed to GitHub and shared with the world. The best way to get around this in R is to create a separate script file in your repo (e.g., `apikeys.R`) which includes exactly one line: assigning the key to a variable:

```
# in `apikeys.R`
api.key <- "123456789abcdefg"
```

You can then include this file_name_ in a **`.gitignore`** file in your repo; that will keep it from even possibly being committed with your code!

In order to access this variable in your "main" script, you can use the `source()` function to load and run your `apikeys.R` script. This will execute the line of code that assigns the `api.key` variable, making it available in your environment for your use:

```r
# in `myScript.R`

# (make sure working directory is set)

source('apiKeys.R')  # load the script
print(api.key)  # key is now available!
```

Anyone else who runs the script will simply need to provide an `api.key` variable to access the API using their key, keeping everyone's account separate!

Watch out for APIs that mention using OAuth when explaining API keys. OAuth is a system for performing **authentification**—that is, letting someone log into a website from your application (like what a "Log in with Facebook" button does). OAuth systems require more than one access key, and these keys ***must*** be kept secret and usually require you to run a web server to utilize them correctly (which requires lots of extra setup, see the full `httr` docs for details). So for this course, we encourage you to avoid anything that needs OAuth

11.2.2 HTTP Verbs

When you send a request to a particular resource, you need to indicate what you want to *do* with that resource. When you load web content, you are typically sending a request to retrieve information (logically, this is a `GET` request). However, there are other actions you can perform to modify the data structure on the server. This is done by specifying an **HTTP Verb** in the request. The HTTP protocol supports the following verbs:

- `GET` Return a representation of the current state of the resource
- `POST` Add a new subresource (e.g., insert a record)
- `PUT` Update the resource to have a new state
- `PATCH` Update a portion of the resource's state
- `DELETE` Remove the resource
- `OPTIONS` Return the set of methods that can be performed on the resource

By far the most common verb is `GET`, which is used to "get" (download) data from a web service. Depending on how you connect to your API (i.e., which programming language you are using), you'll specify the verb of interest to indicate what we want to do to a particular resource.

Overall, this structure of treating each datum on the web as a **resource** which we can interact with via **HTTP Requests** is referred to as the **REST Architecture** (REST stands for *REpresentational State Transfer*). This is a standard way of structuring computer applications that allows them to be interacted with in the same way as everyday websites. Thus a web service that enabled data access through named resources and responds to HTTP requests is known as a **RESTful** service, with a *RESTful API*.

11.3 Accessing Web APIs

To access a Web API, you just need to send an HTTP Request to a particular URI. You can easily do this with the browser: simply navigate to a particular address (base URI + endpoint), and that will cause the browser to send a `GET` request and display the resulting data. For example, you can send a request to search GitHub for repositories named `d3` by visiting:

```
https://api.github.com/search/repositories?q=d3&sort=forks
```

This query accesses the `/search/repositories/` endpoint, and also specifies 2 query parameters:

- `q`: The term(s) you are searching for, and
- `sort`: The attribute of each repository that you would like to use to sort the results

(Note that the data you'll get back is structured in JSON format. See below for details).

In `R` you can send GET requests using the `httr` library. Like `dplyr`, you will need to install and load it to use it:

```r
install.packages("httr")  # once per machine
library("httr")
```

This library provides a number of functions that reflect HTTP verbs. For example, the **`GET()`** function will send an HTTP GET Request to the URI specified as an argument:

```r
# Get search results for artist
response <- GET("https://api.github.com/search/repositories?q=d3&sort=forks")
```

While it is possible to include *query parameters* in the URI string, `httr` also allows you to include them as a *list*, making it easy to set and change variables (instead of needing to do a complex `paste0()` operation to produce the correct string):

```r
# Equivalent to the above, but easier to read and change
query.params <- list(q = "d3", sort = "forks")
response <- GET("https://api.github.com", query = query.params)
```

If you try printing out the `response` variable, you'll see information about the response:

```
Response [https://api.github.com/?q=d3&sort=forks]
```

```
Date: 2017-10-22 19:05
Status: 200
Content-Type: application/json; charset=utf-8
Size: 2.16 kB
```

This is called the **response header**. Each **response** has two parts: the **header**, and the **body**. You can think of the response as a envelope: the *header* contains meta-data like the address and postage date, while the *body* contains the actual contents of the letter (the data).

Since you're almost always interested in working with the *body*, you will need to extract that data from the response (e.g., open up the envelope and pull out the letter). You can do this with the `content()` method:

```
# extract content from response, as a text string (not a list!)
body <- content(response, "text")
```

Note the second argument `"text"`; this is needed to keep `httr` from doing it's own processing on the body data, since we'll be using other methods to handle that; keep reading for details!

Pro-tip: The URI shown when you print out the `response` is a good way to check exactly what URI you were sending the request to: copy that into your browser to make sure it goes where you expected!

11.4 JSON Data

Most APIs will return data in **JavaScript Object Notation (JSON)** format. Like CSV, this is a format for writing down structured data—but while `.csv` files organize data into rows and columns (like a data frame), JSON allows you to organize elements into **key-value pairs** similar to an R *list*! This allows the data to have much more complex structure, which is useful for web services (but can be challenging for us).

In JSON, lists of key-value pairs (called *objects*) are put inside braces (`{ }`), with the key and value separated by a colon (`:`) and each pair separated by a comma (`,`). Key-value pairs are often written on separate lines for readability, but this isn't required. Note that keys need to be character strings (so in quotes), while values can either be character strings, numbers, booleans (written in lower-case as `true` and `false`), or even other lists! For example:

```
{
  "first_name": "Ada",
  "job": "Programmer",
  "salary": 78000,
  "in_union": true,
  "favorites": {
    "music": "jazz",
    "food": "pizza",
  }
}
```

(In JavaScript the period `.` has special meaning, so it is not used in key names; hence the underscores `_`). The above is equivalent to the `R` list:

```
list(first_name = "Ada", job = "Programmer", salary = 78000, in_union = TRUE,
        favorites = list(music = "jazz", food = "pizza")  # nested list in the list!
    )
```

Additionally, JSON supports what are called *arrays* of data. These are like lists without keys (and so are only accessed by index). Key-less arrays are written in square brackets (`[ ]`), with values separated by commas. For example:

```
["Aardvark", "Baboon", "Camel"]
```

which is equvalent to the `R` list:

```
list("Aardvark", "Baboon", "Camel")
```

(Like *objects* , array elements may or may not be written on separate lines).

Just as R allows you to have nested lists of lists, and those lists may or may not have keys, JSON can have any form of nested *objects* and *arrays*. This can be arrays (unkeyed lists) within objects (keyed lists), such as a more complex set of data about Ada:

```
{
  "first_name": "Ada",
  "job": "Programmer",
  "pets": ["rover", "fluffy", "mittens"],
  "favorites": {
    "music": "jazz",
    "food": "pizza",
    "numbers": [12, 42]
  }
}
```

JSON can also be structured as *arrays* of *objects* (unkeyed lists of keyed lists), such as a list of data about Seahawks games:

```
[
  { "opponent": "Dolphins", "sea_score": 12, "opp_score": 10 },
  { "opponent": "Rams", "sea_score": 3, "opp_score": 9 },
  { "opponent": "49ers", "sea_score": 37, "opp_score": 18 },
  { "opponent": "Jets", "sea_score": 27, "opp_score": 17 },
  { "opponent": "Falcons", "sea_score": 26, "opp_score": 24 }
]
```

The latter format is incredibly common in web API data: as long as each *object* in the *array* has the same set of keys, then you can easily consider this as a data table where each *object* (keyed list) represents an **observation** (row), and each key represents a **feature** (column) of that observation.

11.4.1 Parsing JSON

When working with a web API, the usual goal is to take the JSON data contained in the *response* and convert it into an `R` data structure you can use, such as *list* or *data frame*. While the `httr` package is able to parse the JSON body of a response into a *list*, it doesn't do a very clean job of it (particularly for complex data structures).

A more effective solution is to use *another* library called `jsonlite`. This library provides helpful methods to convert JSON data into R data, and does a much more effective job of converting content into data frames that you can use.

As always, you will need to install and load this library:

```r
install.packages("jsonlite")  # once per machine
library("jsonlite")
```

`jsonlite` provides a function called **`fromJSON()`** that allows you to convert a JSON string into a list—or even a data frame if the columns have the right lengths!

```r
# send request for albums by David Bowie
query.params <- list(q = "d3", sort = "forks")
response <- GET("https://api.github.com/search/repositories", query = query.params)
body <- content(response, "text")  # extract the body JSON
parsed.data <- fromJSON(body)  # convert the JSON string to a list
```

The `parsed.data` will contain a *list* built out of the JSON. Depending on the complexity of the JSON, this may already be a data frame you can `View()`… but more likely you'll need to *explore* the list more to locate the "main" data you are interested in. Good strategies for this include:

- You can `print()` the data, but that is often hard to read (it requires a lot of scrolling).
- The `str()` method will produce a more organized printed list, though it can still be hard to read.
- The `names()` method will let you see a list of the what keys the list has, which is good for delving into the data.

As an example continuing the above code:

```
is.data.frame(parsed.data)  # FALSE; not a data frame you can work with
names(parsed.data)  # "href" "items" "limit" "next" "offset" "previous" "total"
  # looking at the JSON data itself (e.g., in the browser), `items` is the
  # key that contains the value we want

items <- parsed.data$items  # extract that element from the list
is.data.frame(items)  # TRUE; you can work with that!
```

11.4.2 Flattening Data

Because JSON supports—and in fact encourages—nested lists (lists within lists), parsing a JSON string is likely to produce a data frame whose columns *are themselves data frames*. As an example:

```
# A somewhat contrived example
people <- data.frame(names = c('Spencer', 'Jessica', 'Keagan'))  # a data frame with

favorites <- data.frame(  # a data frame with two columns
                food = c('Pizza', 'Pasta', 'salad'),
                music = c('Bluegrass', 'Indie', 'Electronic')
            )
# Store second dataframe as column of first
people$favorites <- favorites  # the `favorites` column is a data frame!

# This prints nicely...
print(people)
  #     names favorites.food favorites.music
  # 1 Spencer          Pizza       Bluegrass
  # 2 Jessica          Pasta           Indie
  # 3  Keagan          salad      Electronic

# but doesn't actually work like you expect!
people$favorites.food  # NULL
people$favorites$food  # [1] Pizza Pasta salad
```

Nested data frames make it hard to work with the data using previously established techniques. Luckily, the `jsonlite` package provides a helpful function for addressing this called **`flatten()`**. This function takes the columns of each *nested* data frame and converts them into appropriately named columns in the "outer" data frame:

```
people <- flatten(people)
people$favorites.food  # this just got created! Woo!
```

Note that `flatten()` only works on values that are *already data frames*; thus you may need to find the appropriate element inside of the list (that is, the item which is the data frame you want to flatten).

In practice, you will almost always want to flatten the data returned from a web API. Thus your "algorithm" for downloading web data is as follows:

1. Use `GET()` to download the data, specifying the URI (and any query parameters).
2. Use `content()` to extract the data as a JSON string.
3. Use `fromJSON()` to convert the JSON string into a list.
4. Find which element in that list is your data frame of interest. You may need to go "multiple levels" deep.
5. Use `flatten()` to flatten that data frame.
6. …
7. Profit!

Pro-tip: JSON data can be quite messy when viewed in your web-browser. Installing a browser extension such as JSONView will format JSON responses in a more readable way, and even enable you to interactively explore the data structure.

Resources

- URIs (Wikipedia)
- HTTP Protocol Tutorial
- Programmable Web (list of web APIs; may be out of date)
- RESTful Architecture (original specification; not for beginners)
- JSON View Extension
- `httr` documentation
- `jsonlite` documentation

Chapter 12

R Markdown

R Markdown is a package that supports using `R` to dynamically create *documents*, such as websites (`.html` files), reports (`.pdf` files), slideshows (using `ioslides` or `slidy`), and even interactive web apps (using `shiny`).

As you may have guessed, R Markdown does this by providing the ability to blend Markdown syntax and `R` code so that, when executed, scripts will automatically inject your code results into a formatted document. The ability to automatically generate reports and documents from a computer script eliminates the need to manually update the *results* of a data analysis project, enabling you to more effectively share the *information* that you've produced from your data. In this chapter, you'll learn the fundamentals of the RMarkdown library to create well-formatted documents that combine analysis and reporting.

12.1 R Markdown and RStudio

R Markdown documents are created from a combination of two libraries: `rmarkdown` (which process the markdown and generates the output) and `knitr` (which runs R code and produces Markdown-like output). These packages are already included in RStudio, which provides built-in support for creating and viewing R Markdown documents.

12.1.1 Creating `.Rmd` Files

The easiest way to begin a new R-Markdown document in RStudio is to use the **File > New File > R Markdown** menu option:

RStudio will then prompt you to provide some additional details abour what kind of R Markdown document you want. In particular, you will need to choose a

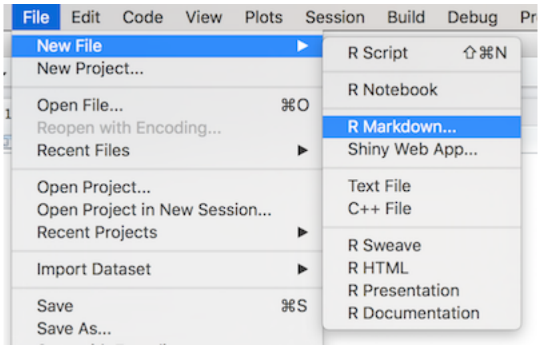


Figure 12.1: Create a new R Markdown document in RStudio.

default *document type* and *output format*. You can also provide a title and author information which will be included in the document. This chapter will focus on creating HTML documents (websites; the default format)—other formats require the installation of additional software.

Once you've chosen *R Markdown* as your desired file type, you'll be prompted to choose a default *document type* and *output format* that you would like to create. In this module, we'll discuss creating HTML documents (websites).

Once you've chosen your desired document type and output format, RStudio will open up a new script file for you. The file contains some example code for you.

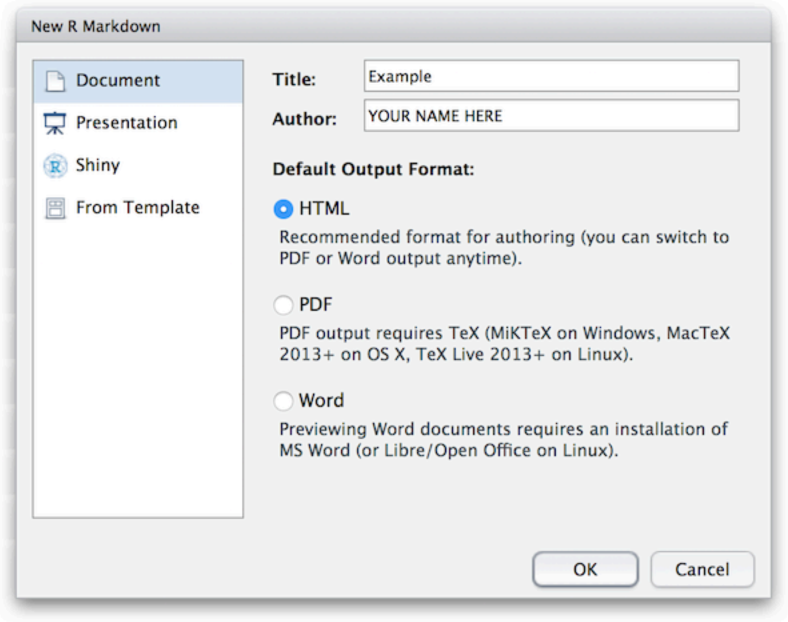
12.1.2 `.Rmd` Content

At the top of the file is some text that has the format:

```yaml
---
title: "Example"
author: "YOUR NAME HERE"
date: "1/30/2017"
output: html_document
---
```

This is the document "header" information, which tells R Markdown details about the file and how the file should be processed. For example, the `title`, `author`, and `date` will automatically be added to the top of your document. You can include additional information as well, such as whether there should be a table of contents or even variable defaults.

- The header is written in YAML format, which is yet another way of formatting structured data similar to `.csv` or JSON (in fact, YAML is a



Title: Example

Author: YOUR NAME HERE

Default Output Format:

● HTML

Recommended format for authoring (you can switch to PDF or Word output anytime).

○ PDF

PDF output requires TeX (MiKTeX on Windows, MacTeX 2013+ on OS X, TeX Live 2013+ on Linux).

○ Word

Previewing Word documents requires an installation of MS Word (or Libre/Open Office on Linux).

Figure 12.2: Specify document type.

superset of JSON and can represent the same data structure, just using indentation and dashes instead of braces and commas).

Below the header, you will find two types of content:

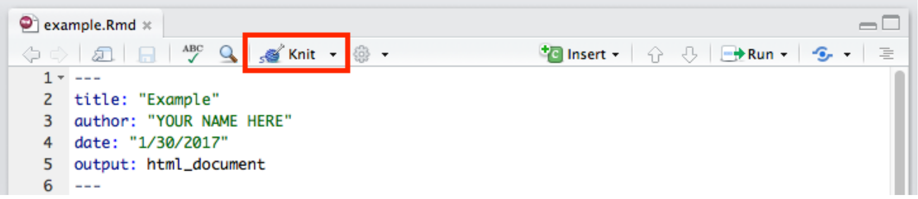
- **Markdown**: normal Markdown text like you learned in Chapter 3. For example, you can use two pound symbols (`##`) for a second-level heading.
- **Code Chunks**: These are segments (chunks) of R code that look like normal code block elements (using “‘), but with an extra `{r}` immediately after the opening backticks.

R Markdown will be able to execute the R code you include in code chunks, and render that output *in your Markdown*. More on this below.

Important This file should be saved with the extension **`.Rmd`** (for “R Markdown”), which tells the computer and RStudio that the document contains Markdown content with embedded `R` code.

12.1.3 Knitting Documents

RStudio provides an easy interface to compile your `.Rmd` source code into an actual document (a process called **“knitting”**). Simply click the **Knit** button at the top of the script panel:



```
---
title: "Example"
author: "YOUR NAME HERE"
date: "1/30/2017"
output: html_document
---
```

Figure 12.3: RStudio's Knit button

This will generate the document (in the same directory as your `.Rmd` file), as well as open up a preview window in RStudio.

While it is easy to generate such documents, the knitting process can make it hard to debug errors in your `R` code (whether syntax or logical), in part because the output may or may not show up in the document! We suggest that you write complex `R` code in another script and then `source()` that script into your `.Rmd` file for use the the output. This makes it possible to test your data processing work outside of the knit application, as well as *separates the concerns* of the data and its representation—which is good programming practice.

Nevertheless, you should still be sure and knit your document frequently, paying close attention to any errors that appear in the console.

Pro-tip: If you're having trouble finding your error, a good strategy is to systematically remove segments of your code and attempt to re-knit the document. This will help you identify the problematic syntax.

12.1.4 HTML

Assuming that you've chosen HTML as your desired output type, RStudio will knit your `.Rmd` into a `.html` file. HTML stands for ***H**yper**T**ext **M**arkup **L**anguage* and, like Markdown, is a syntax for describing the structure and formatting of content (though HTML is **far** more extensive and detailed). In particular, HTML is a markup language that can be automatically rendered by web browsers, and thus is the language used to create web pages. As such, the `.html` files you create can be put online as web pages for others to view—you will learn how to do this in a future chapter. For now, you can open a `.html` file in any browser (such as by double-clicking on the file) to see the content outside of RStudio!

- As it turns out, it's quite simple to use GitHub to host publicly available webpages (like the `.html` files you create with RMarkdown). But, this will require learning a bit more about `git` and GitHub. For instructions on publishing your `.html` files as web-pages, see chapter 14.

12.2 R Markdown Syntax

What makes R Markdown distinct from simple Markdown code is the ability to actually *execute your* `R` *code and include the output directly in the document.* `R` code can be executed and included in the document in blocks of code, or even inline in the document!

12.2.1 R Code Chunks

Code that is to be executed (rather than simply displayed as formatted text) is called a **code chunk**. To specify a code chunk, you need to include **`{r}`** immediately after the backticks that start the code block (the “‘). For example:

~~~
Write normal **markdown** out here, then create a code block:

```{r}
# Execute R code in here
x <- 201
```

Back to writing _markdown_ out here.
~~~

Note that by default, the code chunk will *render* any raw expressions (e.g., `x`)—just like you would see in the console if you selected all the code in the chunk and used `ctrl-enter` to execute it.

It is also possible to specify additional configuration **options** by including a comma-separate list of named arguments (like you've done with lists and functions) inside the curly braces following the `r`:

~~~
```{r options_example, echo=FALSE, message=TRUE}
# a code chunk named "options_example", with parameter `echo` assigned FALSE
# and parameter `message` assigned TRUE

# Would execute R code in here
```
~~~

- The first “argument” (`options_example`) is a “name” for the chunk, and the following are named arguments for the options. Chunks should be named as a variable or function, based on what code is being executed and/or rendered by the chunk. It's always a good idea to name individual code chunks as a form of documentation.

There are many options for creating code chunks (see also the reference). However some of the most useful ones have to do with how the code is outputted in the the document. These include:

- **`echo`** indicates whether you want the *R code itself* to be displayed in the document (e.g., if you want readers to be able to see your work and reproduce your calculations and analysis). Value is either `TRUE` (do display; the default) or `FALSE` (do not display).
- **`message`** indicates whether you want any messages generated by the code to be displayed. This includes print statements! Value is either `TRUE` (do display; the default) or `FALSE` (do not display).

If you only want to *show* your `R` code (and not *evaluate* it), you can alternatively use a standard Markdown codeblock that indicates the `r` language (“‘r, *not* “‘{r}), or set the `eval` option to `FALSE`.

12.2.2 Inline Code

In addition to creating distinct code blocks, you may want to execute R code *inline* with the rest of your text. This empowers you to **reference a variable** from your code-chunk in a section of Markdown—injected that variable into the text you have written. This allows you to easily include a specific result inside a paragraph of text. So if the computation changes, re-knitting your document will update the values inside the text without any further work needed.

As with code blocks, you’ll follow the Markdown convention of using single backticks (`` ` ``), but put the letter **`r`** immediately after the first backtick. For example:

```
To calculate 3 + 4 inside some text, we can use `r 3 + 4` right in the _middle_.
```

When you knit the text above, the ‘r 3 + 4‘ would be replaced with the number `7`.

Note you can also reference values computed in the code blocks preceding your inline code; it is **best practice** to do your calculations in a code block (with `echo=FALSE`), save the result in a variable, and then simply inline that variable with e.g., ‘r my.variable‘.

12.3 Rendering Data

R Markdown’s code chunks let you perform data analysis directly in your document, but often you will want to include more complex data output. This section discusses a few tips for specifying dynamic, complex output to render using R Markdown.

12.3.1 Rendering Strings

If you experiment with knitting R Markdown, you will quickly notice that using `print()` will generate a code block with content that looks like a printed vector:

~~~
```{r echo=FALSE}
print("Hello world")
```
~~~

```
## [1] "Hello world"
```

For this reason, you usually want to have the code block generate a string that you save in a variable, which you can then display with an inline expression (e.g., on its own line):

~~~
```{r echo=FALSE}
msg <- "Hello world"
```

Below is the message to see:
`r msg`
~~~

Note that any Markdown syntax included in the variable (e.g., if you had `msg <- "**Hello** world"`) will be rendered as well—the ‘r msg ‘is replaced by the value of the expression just as if you had typed that Markdown in directly. This allows you to even include dynamic styling if you construct a “Markdown string” out of your data.

Alternatively, you can use as `results` option of `'asis'`, which will cause the “output” to be rendered directly into the markdown. When combined with the `cat()` function (which con**cat**enates content without specifying additional information like vector position), you can make a code chunk effectively render a specific string:

~~~
```{r results='asis', echo=FALSE}
cat("Hello world")
```
~~~

12.3.2 Rendering Lists

Because outputted strings render any Markdown they contain, it’s possible to specify complex Markdown such as *lists* by constructing these strings to contain the `-` symbols utilized (note that each item will need to be separated by a line break or a `\n` character):

~~~
```{r echo=FALSE}
markdown.list <- "
- Lions
- Tigers
~~~

~~~
- Bears
"
```

`r markdown.list`
~~~

Would output a list that looks like:

- Lions
- Tigers
- Bears

Combined with the vectorized `paste()` function, it's to easily convert vectors into Markdown lists that can be rendered

~~~
```{r echo=FALSE}
animals <- c("Lions", "Tigers", "Bears")

# paste a `-` in front of each, then cat the items with newlines between
markdown.list <- cat(paste('-',animals), sep='\n')
```

`r markdown.list`
~~~

And of course, the contents of the vector (e.g., the text `"Lions"`) could easily have additional Markdown syntax syntax to include bold, italic, or hyperlinked text.

- Creating a "helper function" to do this conversion is perfectly reasonable; or see libraries such as `pander` which defines a number of such functions.

12.3.3 Rendering Tables

Because data frames are so central to programming with R, R Markdown includes capabilities to easily render data frames as Markdown *tables* via the **`knitr::kable()`** function. This function takes as an argument the data frame you wish to render, and it will automatically convert that value into a Markdown table:

~~~
```{r echo=FALSE}
library(knitr)  # make sure you load this library (once per doc)

# make a data frame
letters <- c('a','b','c')
numbers <- 1:3
df <- data.frame(letters = letters, numbers = numbers)

# render the table
~~~

~~~
kable(df)
```
~~~

- `kable()` supports a number of other arguments that can be used to customize how it outputs a table.
- And of courrse, if the values in the dataframe are strings that contain Markdown syntax (e.g., bold, itaic, or hyperlinks), they will be rendered as such in the table!

So while you may need to do a little bit of work to manually generate the Markdown syntax, it is possible to dynamically produce complex documents based on dynamic data sources

Resources

- R Markdown Homepage
- R Markdown Cheatsheet (really useful!)
- R Markdown Reference (really useful!)
- `knitr`

Chapter 13

The `ggplot2` Library

Being able to create **visualizations** (graphical representations) of data is a key step in being able to *communicate* information and findings to others. In this chapter you will learn to use the **`ggplot2`** library to declaratively make beautiful plots or charts of your data. Although R does provide built-in plotting functions, the `ggplot2` library implements the **Grammar of Graphics** (similar to how `dplyr` implements a *Grammar of Data Manipulation*; indeed, both packages were developed by the same person). This makes the library particularly effective for describing how visualizations should represent data, and has turned it into the preeminent plotting library in R. Learning this library will allow you to easily make nearly any kind of (static) data visualization, customized to your exact specifications.

Examples in this chapter adapted from *R for Data Science* by Garrett Grolemund and Hadley Wickham.

13.1 A Grammar of Graphics

Just as the grammar of language helps us construct meaningful sentences out of words, the ***Grammar of Graphics*** helps us to construct graphical figures out of different visual elements. This grammar gives us a way to talk about parts of a plot: all the circles, lines, arrows, and words that are combined into a diagram for visualizing data. Originally developed by Leland Wilkinson, the Grammar of Graphics was adapted by Hadley Wickham to describe the *components* of a plot, including

- the **data** being plotted
- the **geometric objects** (circles, lines, etc.) that appear on the plot
- the **aesthetics** (appearance) of the geometric objects, and the *mappings* from variables in the data to those aesthetics

- a **statistical transformation** used to calculate the data values used in the plot
- a **position adjustment** for locating each geometric object on the plot
- a **scale** (e.g., range of values) for each aesthetic mapping used
- a **coordinate system** used to organize the geometric objects
- the **facets** or groups of data shown in different plots

Wickham further organizes these components into **layers**, where each layer has a single *geometric object*, *statistical transformation*, and *position adjustment*. Following this grammar, you can think of each plot as a set of layers of images, where each image's appearance is based on some aspect of the data set.

All together, this grammar enables you to discuss what plots look like using a standard set of vocabulary. And like with `dplyr` and the *Grammar of Data Manipulation*, `ggplot2` uses this grammar directly to declare plots, allowing you to more easily create specific visual images.

13.2 Basic Plotting with **ggplot2**

The **ggplot2** library provides a set of *declarative functions* that mirror the above grammar, enabling you to efficaciously specify what you want a plot to look like (e.g., what data, geometric objects, aesthetics, scales, etc. you want it to have).

`ggplot2` is yet another external package (like `dplyr` and `httr` and `jsonlite`), so you will need to install and load it in order to use it:

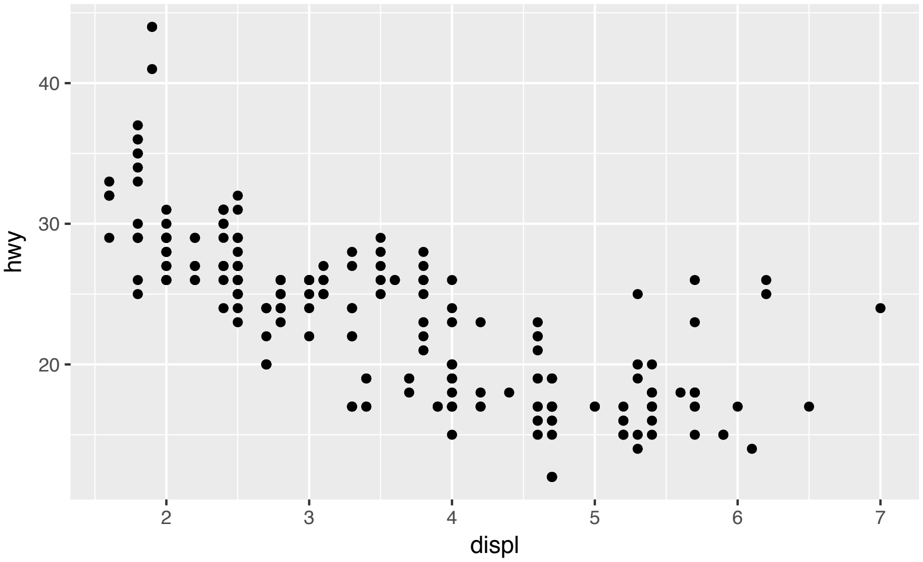
```
install.packages("ggplot2")  # once per machine
library("ggplot2")
```

This will make all of the plotting functions you'll need available.

- Note that the library also comes with a number of built-in data sets. This chapter will use the provided `mpg` data set as an example, which is a data frame containing information about fuel economy for different cars.

In order to create a plot, you call the `ggplot()` function, specifying the **data** that you wish to plot. You then add new *layers* that are **geometric objects** which will show up on the plot:

```
# plot the `mpg` data set, with highway mileage on the x axis and
# engine displacement (power) on the y axis:
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy))
```



To walk through the above code:

- The `ggplot()` function is passed the data frame to plot as the `data` argument.

- You specify a geometric object (`geom`) by calling one of the many`geom` functions, which are all named `geom_` followed by the name of the kind of geometry you wish to create. For example, `geom_point()` will create a layer with "point" (dot) elements as the geometry. There are a large number of these functions; see below for more details.

- For each `geom` you must specify the **aesthetic mappings**, which is how data from the data frame will be mapped to the visual aspects of the geometry. These mappings are defined using the `aes()` function. The `aes()` function takes a set of arguments (like a list), where the argument name is the visual property to map *to*, and the argument value is the data property to map *from*.

- Finally, you add `geom` layers to the plot by using the addition (`+`) operator.

Thus, basic simple plots can be created simply by specifying a data set, a `geom`, and a set of aesthetic mappings.

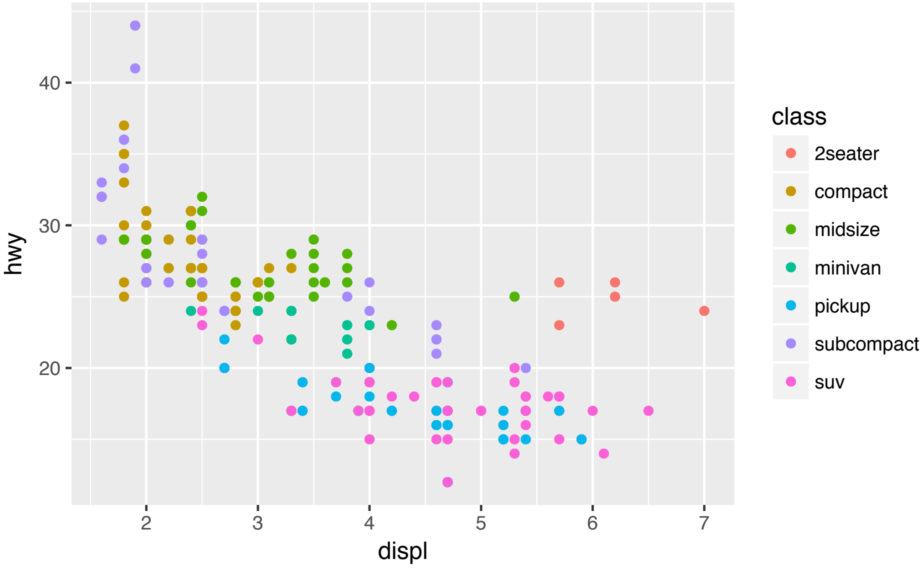
- Note that `ggplot2` library does include a `qplot()` function for creating "quick plots", which acts as a convenient shortcut for making simple, "default"-like plots. While this is a nice starting place, the strength of `ggplot2` is in it's *customizability*, so read on!

13.2.1 Aesthetic Mappings

The **aesthetic mappings** take properties of the data and use them to influence **visual channels**, such as *position*, *color*, *size*, or *shape*. These are the representations that you want to *drive with your data properties*, rather than set for all markers. Each visual channel can therefore encode an aspect of the data and be used to express underlying patterns.

The data-driven aesthetics for a plot are specified in the `aes()` function call for that `geom` layer. For example, you can add a mapping from the `class` of the cars to the *color* channel:

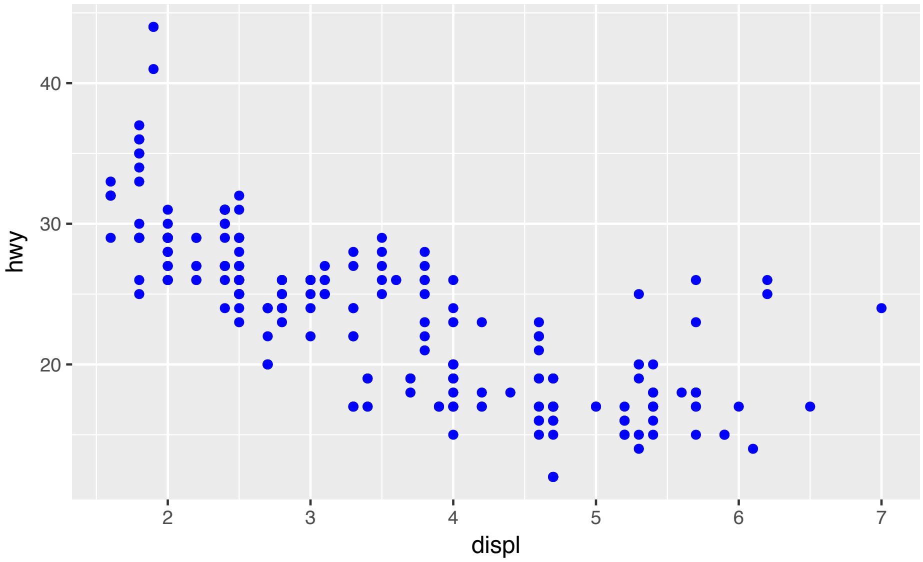
```
# color the data by car type
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy, color = class))
```



(`ggplot2` will even create a legend for you!)

Note that using the `aes()` function will cause the visual channel to be based on the data specified in the argument. For example, using `aes(color = "blue")` won't cause the geometry's color to be "blue", but will instead cause the visual channel to be mapped from the *vector* `c("blue")`—as if you only had a single type of engine that happened to be called "blue". If you wish to apply an aesthetic property to an entire geometry, you can ***set*** that property as an argument to the `geom` method, outside of the `aes()` call:

```
ggplot(data = mpg) +                                # note where parentheses are closed
  geom_point(mapping = aes(x = displ, y = hwy), color = "blue")  # blue points!
```



13.3 Complex Plots

Building on these basics, `ggplot2` can be used to build almost any kind of plot you may want. These plots are declared using functions that follow from the *Grammar of Graphics*.

13.3.1 Specifying Geometry

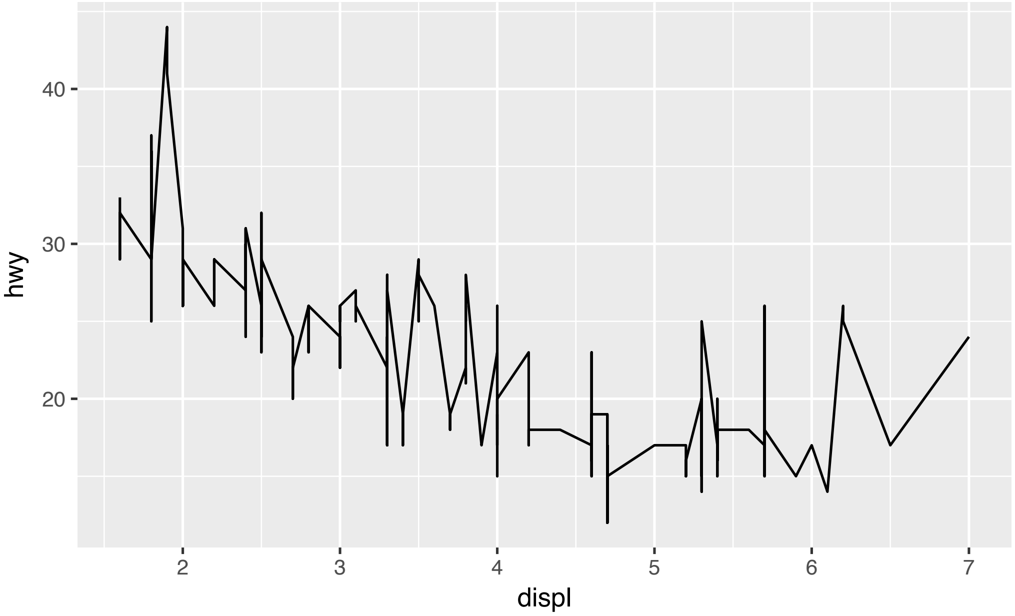
The most obvious distinction between plots is what **geometric objects** (`geoms`) they include. `ggplot2` supports a number of different types of `geoms`, including:

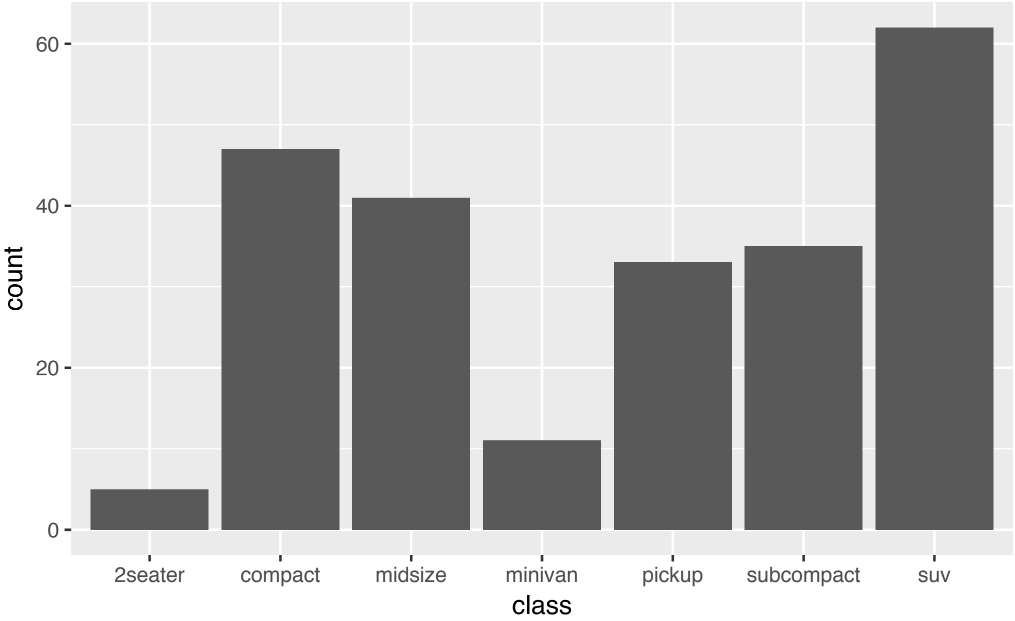
- **`geom_point`** for drawing individual points (e.g., a scatter plot)
- **`geom_line`** for drawing lines (e.g., for a line charts)
- **`geom_smooth`** for drawing smoothed lines (e.g., for simple trends or approximations)
- **`geom_bar`** for drawing bars (e.g., for bar charts)
- **`geom_polygon`** for drawing arbitrary shapes (e.g., for drawing an area in a coordinate plane)
- **`geom_map`** for drawing polygons in the shape of a map! (You can access the *data* to use for these maps by using the `map_data()` function).

Each of these geometries will need to include a set of **aesthetic mappings** (using the `aes()` function and assigned to the `mapping` argument), though the specific *visual properties* that the data will map to will vary. For example, you can map data to the `shape` of a `geom_point` (e.g., if they should be circles or squares), or you can map data to the `linetype` of a `geom_line` (e.g., if it is solid or dotted), but not vice versa.

- Almost all `geoms` **require** an `x` and `y` mapping at the bare minimum.

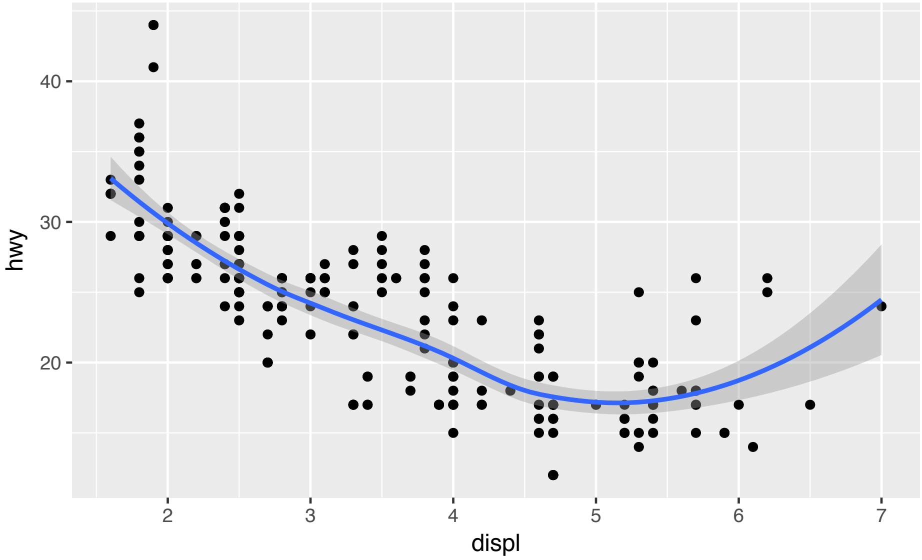
```
# line chart of mileage by engine power
ggplot(data = mpg) +
  geom_line(mapping = aes(x = displ, y = hwy))

# bar chart of car type
ggplot(data = mpg) +
  geom_bar(mapping = aes(x = class))  # no y mapping needed!
```





What makes this really powerful is that you can add **multiple geometries** to a plot, thus allowing you to create complex graphics showing multiple aspects of your data

```r
# plot with both points and smoothed line
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy)) +
  geom_smooth(mapping = aes(x = displ, y = hwy))
```



Of course the aesthetics for each `geom` can be different, so you could show multiple lines on the same plot (or with different colors, styles, etc). It's also possible to give each `geom` a different `data` argument, so that you can show multiple data sets in the same plot.

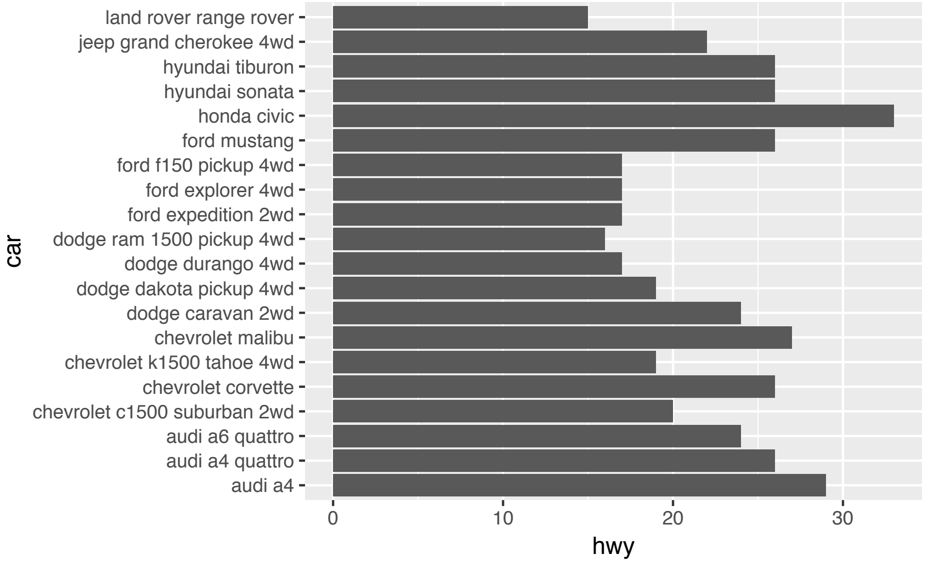
- If you want multiple `geoms` to utilize the same data or aesthetics, you can pass those values as arguments to the `ggplot()` function itself; any `geoms` added to that plot will use the values declared for the whole plot *unless overridden by individual specifications*.

13.3.1.1 Statistical Transformations

If you look at the above `bar` chart, you'll notice that the the `y` axis was defined for you as the `count` of elements that have the particular type. This `count` isn't part of the data set (it's not a column in `mpg`), but is instead a **statistical transformation** that the `geom_bar` automatically applies to the data. In particular, it applies the `stat_count` transformation, simply summing the number of rows each `class` appeared in the dataset.

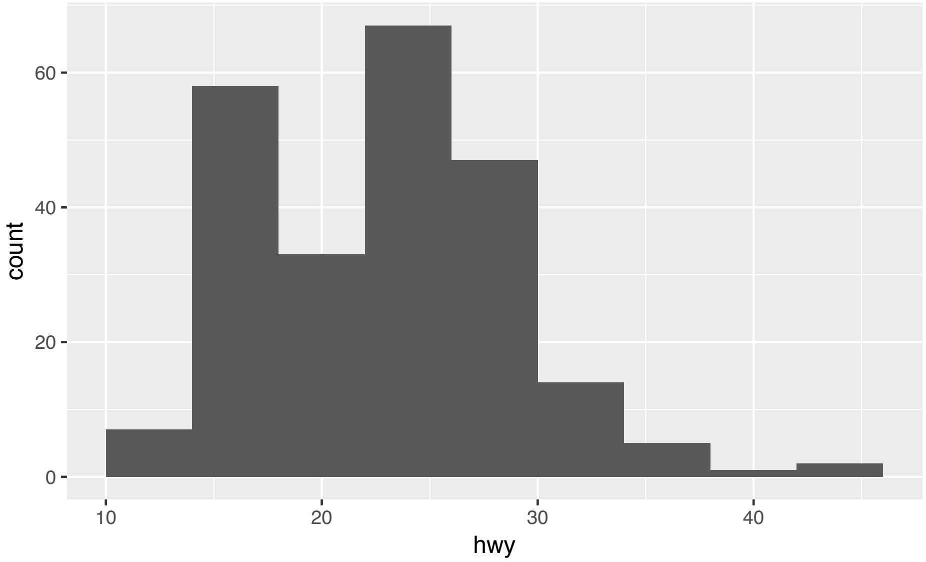
`ggplot2` supports many different statistical transformations. For example, the "identity" transformation will leave the data "as is". You can specify which statistical transformation a `geom` uses by passing it as the **`stat`** argument:

```
# bar chart of make and model vs. mileage

# quickly (lazily) filter the dataset to a sample of the cars: one of each make/mode
new.cars <- mpg %>%
  mutate(car = paste(manufacturer, model)) %>%  # combine make + model
  distinct(car, .keep_all=TRUE)  %>% # select one of each cars -- lazy filtering!
  slice(1:20)  # only keep 20 cars

# create the plot (you need the `y` mapping since it is not implied by the stat tran
ggplot(new.cars) +
  geom_bar(mapping=aes(x=car, y=hwy), stat="identity") + coord_flip()  # horizontal
```



Additionally, `ggplot2` contains **`stat_`** functions (e.g., `stat_identity` for the "identity" transformation) that can be used to specify a layer in the same way a `geom` does:

```
# generate a "binned" (grouped) display of highway mileage
ggplot(data = mpg) +
  stat_bin(aes(x=hwy, color=hwy), binwidth=4)  # binned into groups of 4 units
```



Notice the above chart is actually a histogram! Indeed, almost every `stat` transformation corresponds to a particular `geom` (and vice versa) by default.

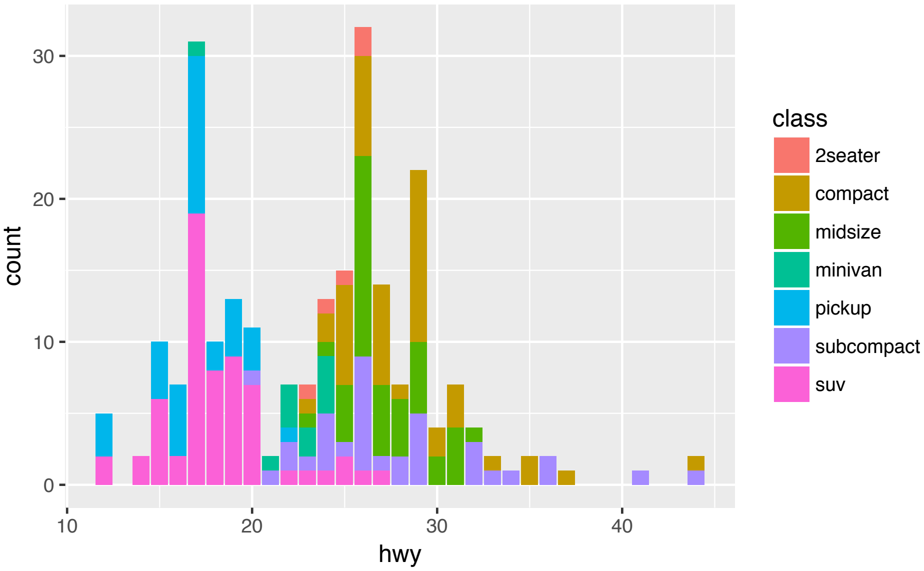
Thus they can often be used interchangeably, depending on how you want to emphasize your layer creation when writing the code.

```
# these two charts are identical
ggplot(data = mpg) +
  geom_bar(mapping = aes(x = class))

ggplot(data = mpg) +
  stat_count(mapping = aes(x = class))
```

13.3.1.2 Position Adjustments

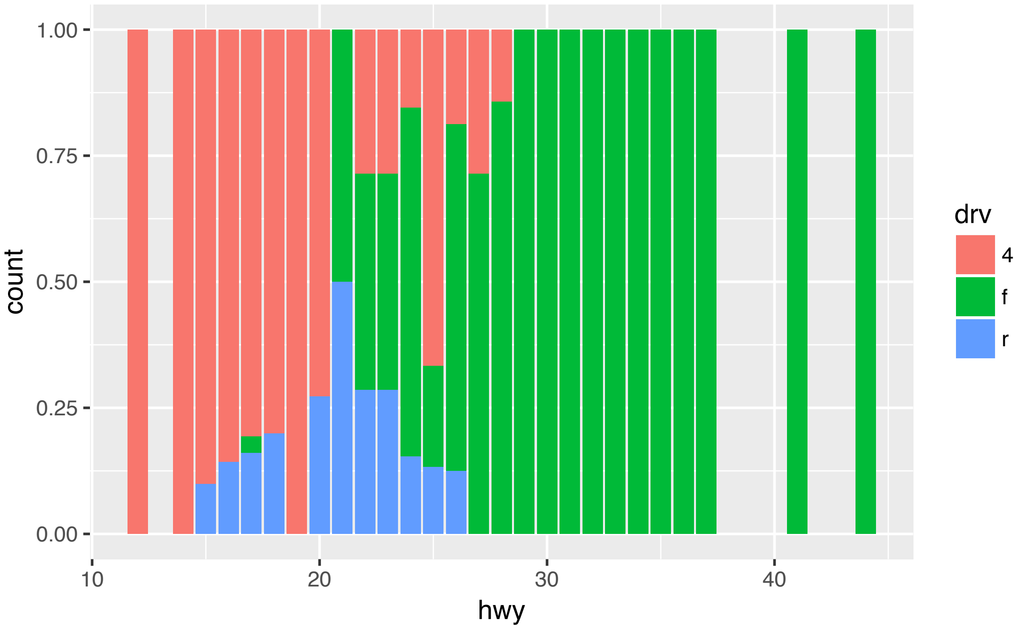
In addition to a default statistical transformation, each `geom` also has a default **position adjustment** which specifies a set of "rules" as to how different components should be positioned relative to each other. This position is noticeable in a `geom_bar` if you map a different variable to the color visual channel:

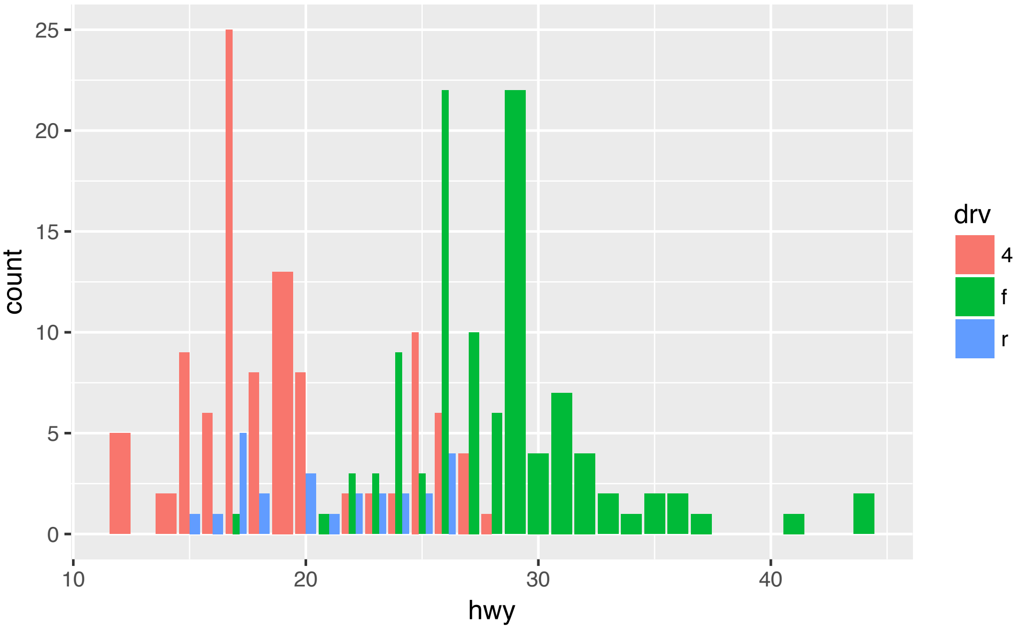
```
# bar chart of mileage, colored by engine type
ggplot(data = mpg) +
  geom_bar(mapping = aes(x = hwy, fill = class))  # fill color, not outline color
```



The `geom_bar` by default uses a position adjustment of `"stack"`, which makes each "bar" a height appropriate to its value and *stacks* them on top of each other. You can use the **`position`** argument to specify what position adjustment rules to follow:

```
# a filled bar chart (fill the vertical height)
ggplot(data = mpg) +
  geom_bar(mapping = aes(x = hwy, fill = drv), position = "fill")

# a dodged (group) bar chart -- values next to each other
# (not great dodging demos in this data set)
ggplot(data = mpg) +
  geom_bar(mapping = aes(x = hwy, fill = drv), position = "dodge")
```





Check the documentation for each particular `geom` to learn more about its possible position adjustments.

13.3.2 Styling with Scales

Whenever you specify an **aesthetic mapping**, `ggplot` uses a particular **scale** to determine the *range of values* that the data should map to. Thus, when you specify

```
# color the data by engine type
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy, color = class))
```

`ggplot` automatically adds a **scale** for each mapping to the plot:

```
# same as above, with explicit scales
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy, color = class)) +
  scale_x_continuous() +
  scale_y_continuous() +
  scale_colour_discrete()
```

Each scale can be represented by a function with the following name: `scale_`, followed by the name of the aesthetic property, followed by an `_` and the name of the scale. A `continuous` scale will handle things like numeric data (where there is a *continuous set* of numbers), whereas a `discrete` scale will handle things like colors (since there is a small list of *distinct* colors).

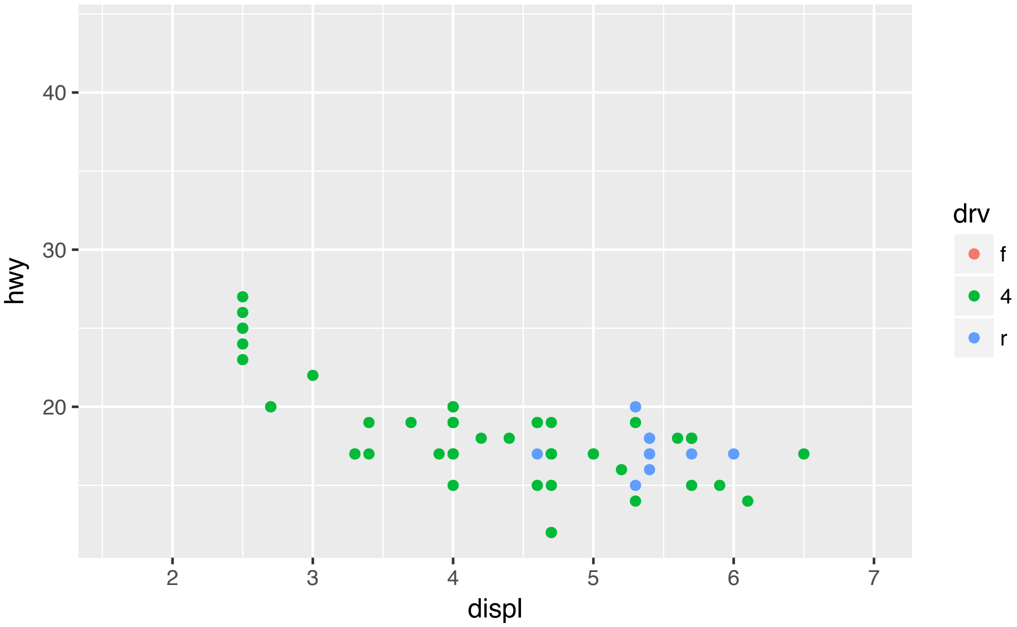
While the default scales will work fine, it is possible to explicitly add different scales to replace the defaults. For example, you can use a scale to change the direction of an axis:

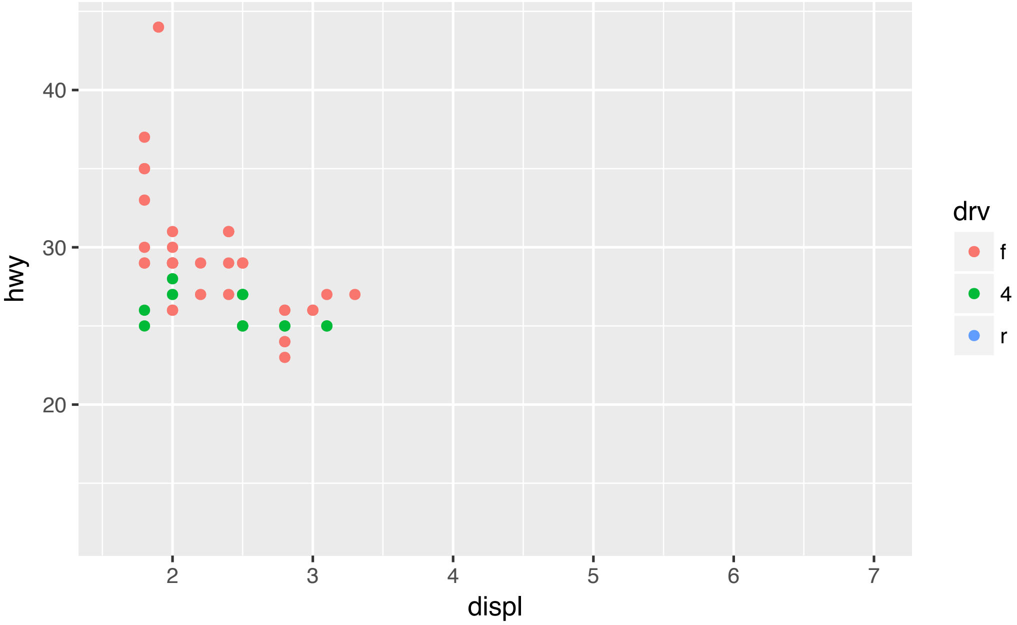
```r
# mileage relationship, ordered in reverse
ggplot(data = mpg) +
  geom_point(mapping = aes(x = cty, y = hwy)) +
  scale_x_reverse()
```

Similarly, you can use `scale_x_log10()` to plot on a logarithmic scale.

You can also use scales to specify the *range* of values on a axis by passing in a `limits` argument. This is useful for making sure that multiple graphs share scales or formats.

```r
# subset data by class
suv = mpg %>% filter(class == "suv")  # suvs
compact = mpg %>% filter(class == "compact")  # compact cars

# scales
x_scale <- scale_x_continuous(limits = range(mpg$displ))
y_scale <- scale_y_continuous(limits = range(mpg$hwy))
col_scale <- scale_colour_discrete(limits = unique(mpg$drv))

ggplot(data = suv) +
  geom_point(mapping = aes(x = displ, y = hwy, color = drv)) +
  x_scale + y_scale + col_scale

ggplot(data = compact) +
  geom_point(mapping = aes(x = displ, y = hwy, color = drv)) +
  x_scale + y_scale + col_scale
```





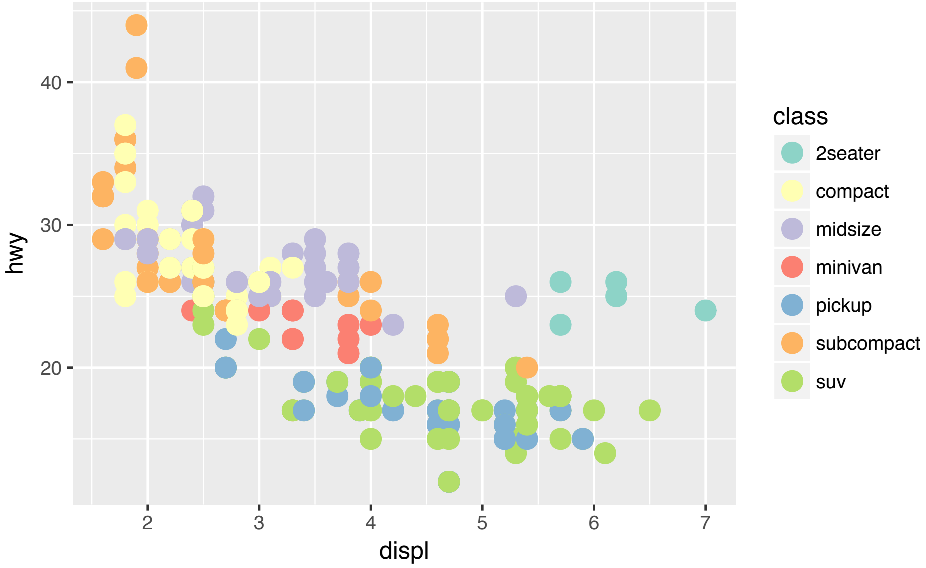
Notice how it is easy to compare the two data sets to each other because the axes and colors match!

These scales can also be used to specify the "tick" marks and labels; see the resources at the end of the chapter for details. And for further ways specifying where the data appears on the graph, see the Coordinate Systems section below.

13.3.2.1 Color Scales

A more common scale to change is which set of colors to use in a plot. While you can use scale functions to specify a list of colors to use, a more common option is to use a pre-defined palette from **colorbrewer.org**. These color sets have been carefully designed to look good and to be viewable to people with certain forms of color blindness. This color scale is specified with the `scale_color_brewer()` function, passing the `palette` as an argument.

```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy, color = class), size=4) +
  scale_color_brewer(palette = "Set3")
```



You can get the palette name from the *colorbrewer* website by looking at the `scheme` query parameter in the URL. Or see the diagram here and hover the mouse over each palette for its name.

You can also specify *continuous* color values by using a gradient scale, or manually specify the colors you want to use as a *named vector*.

13.3.3 Coordinate Systems

The next term from the *Grammar of Graphics* that can be specified is the **coordinate system**. As with **scales**, coordinate systems are specified with functions (that all start with **`coord_`**) and are added to a `ggplot`. There are a number of different possible coordinate systems to use, including:

- **`coord_cartesian`** the default cartesian coordinate system, where you specify `x` and `y` values.
- **`coord_flip`** a cartesian system with the `x` and `y` flipped
- **`coord_fixed`** a cartesian system with a "fixed" aspect ratio (e.g., 1.78 for a "widescreen" plot)
- **`coord_polar`** a plot using polar coordinates
- **`coord_quickmap`** a coordinate system that approximates a good aspect ratio for maps. See the documentation for more details.

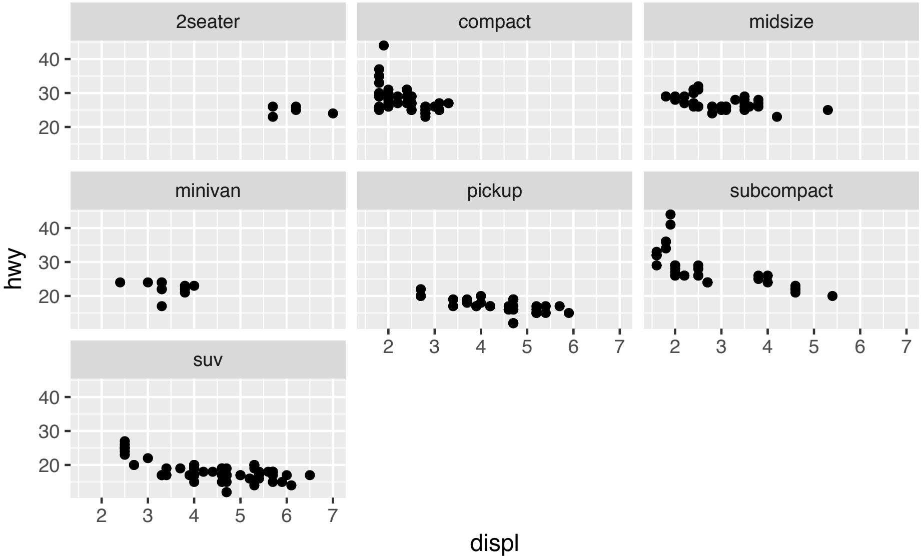
Most of these system support the `xlim` and `ylim` arguments, which specify the *limits* for the coordinate system.

13.3.4 Facets

Facets are ways of *grouping* a data plot into multiple different pieces (*subplots*). This allows you to view a separate plot for each value in a categorical variable. Conceptually, breaking a plot up into facets is similar to using the `group_by()` verb in `dplyr`, with each facet acting like a *level* in an R *factor*.

You can construct a plot with multiple facets by using the **`facet_wrap()`** function. This will produce a "row" of subplots, one for each categorical variable (the number of rows can be specified with an additional argument):

```
# a plot with facets based on vehicle type.
# similar to what we did with `suv` and `compact`!
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy)) +
  facet_wrap(~class)
```



Note that the argument to `facet_wrap()` function is written with a tilde (`~`) in front of it. This specifies that the column name should be treated as a **formula**. A formula is a bit like an "equation" in mathematics; it's like a string representing what set of operations you want to perform (putting the column name in a string also works in this simple case). Formulas are in fact the same structure used with *standard evaluation* in `dplyr`; putting a `~` in front of an expression (such as `~ desc(colname)`) allows SE to work.

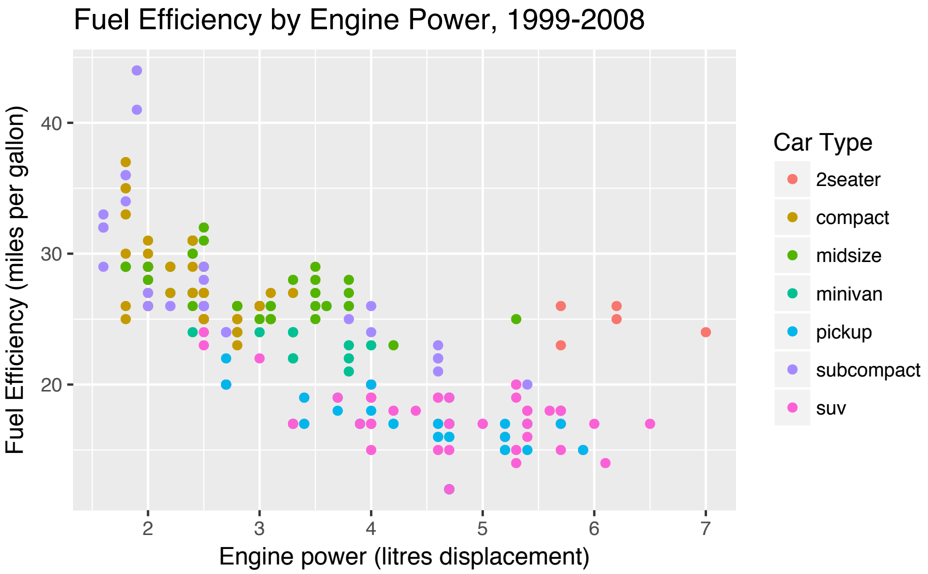
- In short: put a `~` in front of the column name you want to "group" by.

13.3.5 Labels & Annotations

Textual labels and annotations (on the plot, axes, geometry, and legend) are an important part of making a plot understandable and communicating information. Although not an explicit part of the *Grammar of Graphics* (they would be considered a form of geometry), `ggplot` makes it easy to add such annotations.

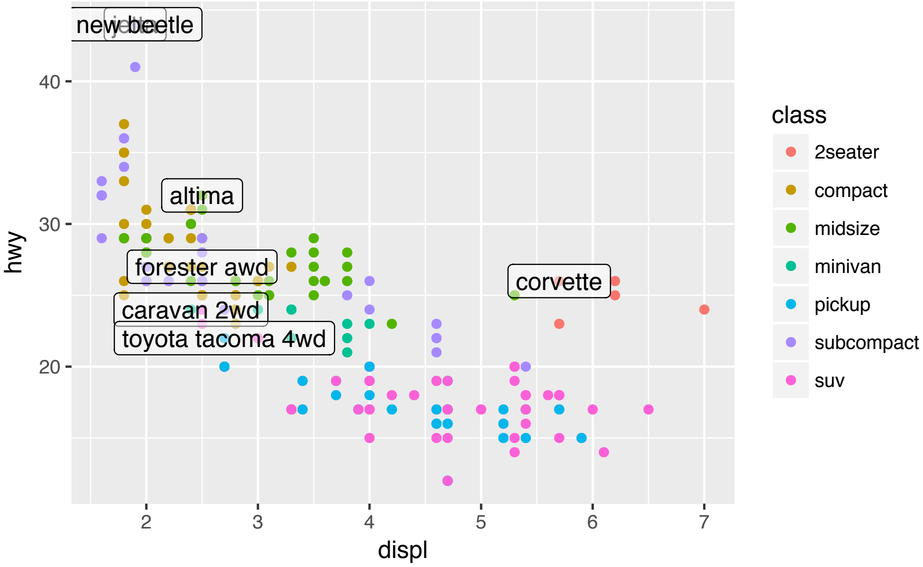
You can add titles and axis labels to a chart using the **`labs()`** function (*not* `labels`, which is a different R function!):

```r
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy, color = class)) +
  labs(title = "Fuel Efficiency by Engine Power, 1999-2008",  # plot title
       x = "Engine power (litres displacement)",  # x-axis label (with units!)
       y = "Fuel Efficiency (miles per gallon)",  # y-axis label (with units!)
       color = "Car Type")  # legend label for the "color" property
```



It is possible to add labels into the plot itself (e.g., to label each point or line) by adding a new `geom_text` or `geom_label` to the plot; effectively, you're plotting an extra set of data which happen to be the variable names:

```
# a data table of each car that has best efficiency of its type
best_in_class <- mpg %>%
  group_by(class) %>%
  filter(row_number(desc(hwy)) == 1)

ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) +  # same mapping for all geom
  geom_point(mapping = aes(color = class)) +
  geom_label(data = best_in_class, mapping = aes(label = model), alpha = 0.5)
```



R for Data Science (linked in the resources below) recommends using the `ggrepel` package to help position labels.

13.4 Other Visualization Libraries

`ggplot2` is easily the most popular library for producing data visualizations in R. That said, `ggplot2` is used to produce **static** visualizations: unchanging "pictures" of plots. Static plots are great for for **explanatory visualizations**: visualizations that are used to communicate some information—or more commonly, an *argument* about that information. All of the above visualizations have been ways to explain and demonstrate an argument about the data (e.g., the relationship between car engines and fuel efficiency).

Data visualizations can also be highly effective for **exploratory analysis**, in which the visualization is used as a way to *ask and answer questions* about the data (rather than to convey an answer or argument). While it is perfectly feasible to do such exploration on a static visualization, many explorations can be better served with **interactive visualizations** in which the user can select and change the *view* and presentation of that data in order to understand it.

While `ggplot2` does not directly support interactive visualizations, there are a number of additional R libraries that provide this functionality, including:

- **`ggvis`** is a library that uses the *Grammar of Graphics* (similar to `ggplot`), but for interactive visualizations. The interactivity is provided through the `shiny` library, which is introduced in a later chapter.

- **Bokeh** is an open-source library for developing interactive visualizations. It automatically provides a number of "standard" interactions (pop-up labels, drag to pan, select to zoom, etc) automatically. It is similar to `ggplot2`, in that you create a figure and then and then add *layers* representing different geometries (points, lines etc). It has detailed and readable documentation, and is also available to other programming languages (such as Python).

- **Plotly** is another libary similar to *Bokeh*, in that it automatically provided standard interactions. It is also possible to take a `ggplot2` plot and wrap it in Plotly in order to make it interactive. Plotly has many examples to learn from, though a less effective set of documentation than other libraries.

- **`rCharts`** provides a way to utilize a number of *JavaScript* interactive visualization libraries. JavaScript is the programming language used to create interactive websites (HTML files), and so is highly specialized for creating interactive experiences.

There are many other libraries as well; searching around for a specific feature you need may lead you to a useful tool!

Resources

- gglot2 Documentation (particularly the function reference)
- ggplot2 Cheat Sheet (see also here)
- Data Visualization (R4DS) - tutorial using `ggplot2`
- Graphics for Communication (R4DS) - "part 2" of tutorial using `ggplot`
- Graphics with ggplot2 - explanation of `qplot()`
- Telling stories with the grammar of graphics
- A Layered Grammar of Graphics (Wickham)

Chapter 14

Git Branches

While `git` is great for uploading and downloading code, its true benefits are its ability to support *reversability* (e.g., undo) and *collaboration* (working with other people). In order to effectively utilize these capabilities, you need to understand git's **branching model**, which is central to how the program manages different versions of code.

This chapter will cover how to work with **branches** with git and GitHub, including using them to work on different features simultaneously and to undo previous changes.

14.1 Git Branches

So far, you've been using git to create a *linear sequence* of commits: they are all in a line, one after another).

Each commit has a message associated with it (that you can see with `git log --oneline`), as well as a unique SHA-1 hash (the random numbers and letters), which can be used to identify that commit as an "id number".

But you can also save commits in a *non-linear* sequence. Perhaps you want to try something new and crazy without breaking code that you've already written. Or you want to work on two different features simultaneously (having separate commits for each). Or you want multiple people to work on the same code without stepping on each other's toes.

To do this, you use a feature of git called **branching** (because you can have commits that "branch off" from a line of development):

In this example, you have a primary branch (called the `master` branch), and decide you want to try an experiment. You *split off* a new branch (called for example `experiment`), which saves some funky changes to your code. But then you decide to make further changes to your main development line, adding more commits to `master` that ignore the changes stored in the `experiment` branch. You can develop `master` and `experiment` simultaneously, making changes to each version of the code. You can even branch off further versions (e.g., a `bugfix` to fix a problem) if you wish. And once you decide you're happy with the code added to both versions, you can **merge** them back together, so that the `master` branch now contains all the changes that were made on the `experiment` branch. If you decided that the `experiment` didn't work out, you can simply delete those set of changes without ever having messed with your "core" `master` branch.

You can view a list of current branches in the repo with the command

```
git branch
```

(The item with the asterisk (`*`) is the "current branch" you're on. The latest commit of the branch you're on is referred to as the **`HEAD`**.

You can use the same command to create a *new* branch:

```
git branch [branch_name]
```

This will create a new branch called `branch_name` (replacing `[branch_name]`, including the brackets, with whatever name you want). Note that if you run `git branch` again you'll see that this *hasn't actually changed what branch you're on.* In fact, all you've done is created a new *reference* (like a new variable!) that refers to the current commit as the given branch name.

- You can think of this like creating a new variable called `branch_name` and assigning the latest commit to that! Almost like you wrote `new_branch <- my_last_commit`.
- If you're familiar with LinkedLists, it's a similar idea to changing a pointer in those.

In order to switch to a different branch, use the command (without the brackets)

```
git checkout [branch_name]
```

Checking out a branch doesn't actually create a new commit! All it does is change the `HEAD` (the "commit I'm currently looking at") so that it now refers to the latest commit of the target branch. You can confirm that the branch has changed with `git branch`.

- You can think of this like assigning a new value (the latest commit of the target branch) to the `HEAD` variable. Almost like you wrote `HEAD <- branch_name_last_commit`.
- Note that you can create *and* checkout a branch in a single step using the `-b` option of `git checkout`:

```
git checkout -b [branch_name]
```

Once you've checked out a particular branch, any *new* commits from that point on will be "attached" to the "HEAD" of that branch, while the "HEAD" of other branches (e.g., `master`) will stay the same. If you use `git checkout` again, you can switch back to the other branch.

- **Important** checking out a branch will "reset" your code to whatever it looked like when you made that commit. Switch back and forth between branches and watch your code change!

Note that you can only check out code if the *current working directory* has no uncommitted changes. This means you'll need to `commit` any changes to the current branch before you `checkout` another. If you want to "save" your changes but don't want to commit to them, you can also use git's ability to temporarily stash changes.

Finally, you can delete a branch using `git branch -d [branch_name]`. Note that this will give you a warning if you might lose work; be sure and read the output message!

14.2 Merging

If you have changes (commits) spread across multiple branches, eventually you'll want to combine those changes back into a single branch. This is a process called **merging**: you "merge" the changes from one branch *into* another. You do this with the (surprise!) `merge` command:

```
git merge [other_branch]
```

This command will merge `other_branch` **into the current branch**. So if you want to end up with the "combined" version of your commits on a particular branch, you'll need to switch to (`checkout`) that branch before you run the merge.

- **IMPORTANT** If something goes wrong, don't panic and try to close your command-line! Come back to this book and look up how to fix the problem you've encountered (e.g., how to exit *vim*). And if you're unsure why something isn't working with git, use **`git status`** to check the current status and for what steps to do next.
- Note that the `rebase` command will perform a similar operation, but without creating a new "merge" commit—it simply takes the commits from one branch and attaches them to the end of the other. This effectively **changes history**, since it is no longer clear where the branching occurred. From an archival and academic view, you never want to "destroy history" and lose a record of changes that were made. History is important: don't

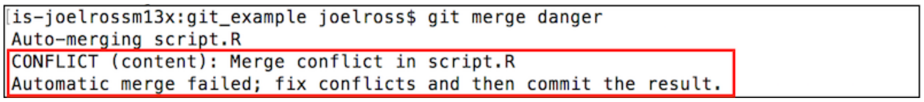
screw with it! Thus we recommend you *avoid* rebasing and stick with merging.

14.2.1 Merge Conflicts

Merging is a regular occurrence when working with branches. But consider the following situation:

1. You're on the `master` branch.
2. You create and `checkout` a new branch called `danger`
3. On the `danger` branch, you change line 12 of the code to be "I like kitties". You then commit this change (with message "Change line 12 of danger").
4. You `checkout` (switch to) the `master` branch again.
5. On the `master` branch, you change to line 12 of the code to be "I like puppies". You then commit this change (with message "Change line 12 of master").
6. You use `git merge danger` to merge the `danger` branch **into** the `master` branch.

In this situation, you are trying to *merge two different changes to the same line of code*, and thus should be shown an error on the command-line:



```
is-joelrossm13x:git_example joelross$ git merge danger
Auto-merging script.R
CONFLICT (content): Merge conflict in script.R
Automatic merge failed; fix conflicts and then commit the result.
```

Figure 14.1: A merge conflict reported on the command-line

This is called a **merge conflict**. A merge conflict occurs when two commits from different branches include different changes to the same code (they conflict). Git is just a simple computer program, and has no way of knowing which version to keep ("Are kitties better than puppies? How should I know?!").

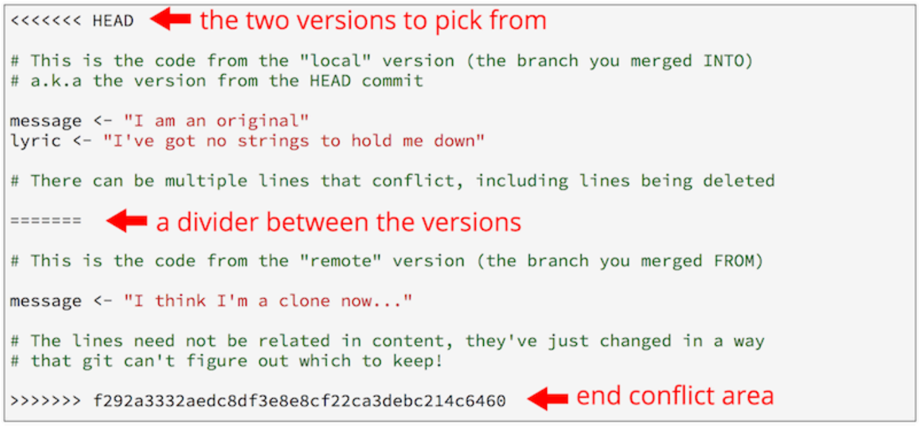
Since git can't determine which version of the code to keep, it ***stops the merge in the middle*** and forces you to choose what code is correct **manually**.

In order to **resolve the merge conflict**, you will need to edit the file (code) so that you pick which version to keep. Git adds "code" to the file to indicate where you need to make a decision about which code is better:

In order to resolve the conflict:

1. Use `git status` to see which files have merge conflicts. Note that files may have more than one conflict!

2. Choose which version of the code to keep (or keep a combination, or replace it with something new entirely!) You do this by **editing the file**



```
<<<<<<< HEAD

# This is the code from the "local" version (the branch you merged INTO)
# a.k.a the version from the HEAD commit

message <- "I am an original"
lyric <- "I've got no strings to hold me down"

# There can be multiple lines that conflict, including lines being deleted

=======

# This is the code from the "remote" version (the branch you merged FROM)

message <- "I think I'm a clone now..."

# The lines need not be related in content, they've just changed in a way
# that git can't figure out which to keep!

>>>>>>> f292a3332aedc8df3e8e8cf22ca3debc214c6460
```

the two versions to pick from

a divider between the versions

end conflict area

Figure 14.2: Code including a merge conflict.

(i.e., open it in Atom or RStudio and change it). Pretend that your cat walked across your keyboard and added a bunch of extra junk; it is now your task to fix your work and restore it to a clean, working state. ***Be sure and test your changes to make sure things work!***

3. Be sure and remove the `<<<<<<<` and `=======` and `>>>>>>>`. These are not legal code in any language.

4. Once you're satisfied that the conflicts are all resolved and everything works as it should, follow the instructions in the error message and `add` and `commit` your changes (the code you "modified" to resolve the conflict):

```
git add .
git commit "Resolve merge conflict"
```

This will complete the merge! Use `git status` to check that everything is clean again.

Merge conflicts are expected. You didn't do something wrong if one occurs! Don't worry about getting merge conflicts or try to avoid them: just resolve the conflict, fix the "bug" that has appeared, and move on with your life.

14.3 Undoing Changes

One of the key benefits of version control systems is **reversibility**: the ability to "undo" a mistake (and we all make lots of mistakes when programming!) Git provides two basic ways that you can go back and fix a mistake you've made previously:

1. You can replace a file (or the entire project directory!) with a version saved as a previous commit.
2. You can have git "reverse" the changes that you made with a previous commit, effectively applying the *opposite* changes and thereby undoing it.

Note that both of these require you to have committed a working version of the code you want to go back to. Git only knows about changes that have been committed—if you don't commit, git can't help you! **Commit early, commit often**.

For both forms of undoing, first recall how each commit has a unique SHA-1 hash (those random numbers) that acted as its "name". You can see these with the `git log --oneline` command.

You can use the `checkout` command to switch not only to the commit named by a branch (e.g., `master` or `experiment`), but to *any* commit in order to "undo" work. You refer to the commit by its hash number in order to check it out:

```
git checkout [commit_number] [filename]
```

This will replace the current version ***of a single file*** with the version saved in `commit_number`. You can also use **`--`** as the commit-number to refer to the `HEAD` (the most recent commit in the branch):

```
git checkout -- [filename]
```

If you're trying to undo changes to lots of files, you can alternatively replace the entire project directory with a version from a previous commit by checking out that commit **as a new branch**:

```
git checkout -b [branch_name] [commit_number]
```

This command treats the commit as if it was the HEAD of a named branch… where the name of that branch is the commit number. You can then make further changes and merge it back into your development or `master` branch.

IMPORTANT NOTE: If you don't create a *new branch* (with **`-b`**) when checking out an old commit, you'll enter **detached HEAD state**. You can't commit from here, because there is no branch for that commit to be attached to! See this tutorial (scroll down) for details and diagrams. If you find yourself in a detached HEAD state, you can use `git checkout master` to get back to the last saved commit (though you will lose any changes you made in that detached state—so just avoid it in the first place!)

But what if you just had one bad commit, and don't want to throw out other good changes you made later? For this, you can use the `git revert` command:

```
git revert [commit_number] --no-edit
```

This will determine what changes that commit made to the files, and then apply the *opposite* changes to effectively "back out" the commit. Note that this **does**

not go back *to* the given commit number (that's what `checkout` is for!), but rather will *reverse the commit you specify.*

- This command does create a new commit (the `--no-edit` option tells git that you don't want to include a custom commit message). This is great from an archival point of view: you never "destroy history" and lose the record of what changes were made and then reverted. History is important: don't screw with it!

  Conversely, the `reset` command will destroy history. **Do not use it**, no matter what StackOverflow tells you to do.

14.4 GitHub and Branches

GitHub is an online service that stores copies of repositories in the cloud. When you `push` and `pull` to GitHub, what you're actually doing is **merging** your commits with the ones on GitHub!

However, remember that you don't edit any files on GitHub's servers, only on your own local machine. And since **resolving a merge conflict** involves editing the files, you have to be careful that conflicts only occur on the local machine, not on GitHub. This plays out in two ways:

1. You will **not** be able to **`push`** to GitHub if merging your commits ***into*** GitHub's repo would cause a merge conflict. Git will instead report an error, telling you that you need to `pull` changes first and make sure that your version is "up to date". Up to date in this case means that you have downloaded and merged all the commits on your local machine, so there is no chance of divergent changes causing a merge conflict when you merge by pushing.

2. Whenever you **`pull`** changes from GitHub, there may be a merge conflict! These are resolved ***in the exact same way*** as when merging local branches: that is, you need to *edit the files* to resolve the conflict, then `add` and `commit` the updated versions.

Thus in practice, when working with GitHub (and especially with multiple people), in order to upload your changes you'll need to do the following:

1. `pull` (download) any changes you don't have
2. *Resolve* any merge conflicts that occurred
3. `push` (upload) your merged set of changes

Additionally, because GitHub repositories are repos just like the ones on your local machine, they can have branches as well! You have access to any *remote* branches when you `clone` a repo; you can see a list of them with `git branch -a` (using the "**a**ll" option).

If you create a new branch on your local machine, it is possible to push *that branch* to GitHub, creating a mirroring branch on the remote repo. You do this by specifying the branch in the `git push` command:

```
git push origin branch_name
```

where `branch_name` is the name of the branch you are currently on (and thus want to push to GitHub).

Note that you often want to associate your local branch with the remote one (make the local branch **track** the remote), so that when you use `git status` you will be able to see whether they are different or not. You can establish this relationship by including the `-u` option in your push:

```
git push -u origin branch_name
```

Tracking will be remembered once set up, so you only need to use the `-u` option *once*.

14.4.1 GitHub Pages

GitHub's use of branches provides a number of additional features, one of which is the ability to **host** web pages (`.html` files, which can be generated from R Markdown) on a publicly accessible web server that can "serve" the page to anyone who requests it. This feature is known as GitHub Pages.

With GitHub pages, GitHub will automatically serve your files to visitors as long as the files are in a branch with a magic name: **`gh-pages`**. Thus in order to **publish** your webpage and make it available online, all you need to do is create that branch, merge your content into it, and then push that branch to GitHub.

You almost always want to create the new `gh-pages` branch off of your `master` branch. This is because you usually want to publish the "finished" version, which is traditionally represented by the `master` branch. This means you'll need to switch over to `master`, and then create a new branch from there:

```
git checkout master
git checkout -b gh-pages
```

Checking out the new branch will create it *with all of the commits of its source* meaning `gh-pages` will start with the exact same content as `master`—if your page is done, then it is ready to go!

You can then upload this new local branch to the `gh-pages` branch on the `origin` remote:

```
git push -u origin gh-pages
```

After the push completes, you will be able to see your web page using the following URL:

`https://GITHUB-USERNAME.github.io/REPO-NAME`

(Replace `GITHUB-USERNAME` with the user name **of the account hosting the repo**, and `REPO-NAME` with your repository name).

- This means that if you're making your homework reports available, the `GITHUB-USERNAME` will be the name of the course organization.

Some important notes:

1. The `gh-pages` branch must be named *exactly* that. If you misspell the name, or use an underscore instead of a dash, it won't work.

2. Only the files and commits in the `gh-pages` branch are visible on the web. All commits in other branches (`experiment`, `master`, etc.) are not visible on the web (other than as source code in the repo). This allows you to work on your site with others before publishing those changes to the web.

3. Any content in the `gh-pages` branch will be publicly accessible, even if your repo is private. You can remove specific files from the `gh-pages` branch that you don't want to be visible on the web, while still keeping them in the `master` branch: use the `git rm` to remove the file and then add, commit, and push the deletion.

   - Be careful not push any passwords or anything to GitHub!

4. The web page will only be initially built when a **repo administrator** pushes a change to the `gh-pages` branch; if someone just has "write access" to the repo (e.g., they are a contributor, but not an "owner"), then the page won't be created. But once an administrator (such as the person who created the repo) pushes that branch and causes the initial page to be created, then any further updates will appear as well.

After you've created your initial `gh-pages` branch, any changes you want to appear online will need to be saved as new commits to that branch and then pushed back up to GitHub. **HOWEVER**, it is best practice to ***not*** make any changes directly to the `gh-pages` branch! Instead, you should switch back to the `master` branch, make your changes there, commit them, then `merge` them back into `gh-pages` before pushing to GitHub:

```
# switch back to master
git checkout master

### UPDATE YOUR CODE (outside of the terminal)

# commit the changes
git add .
git commit -m "YOUR CHANGE MESSAGE"
```

```
# switch back to gh-pages and merge changes from master
git checkout gh-pages
git merge master

# upload to github
git push --all
```

(the `--all` option on `git push` will push all branches that are **tracking** remote branches).

This procedure will keep your code synchronized between the branches, while avoiding a large number of merge conflicts.

Resources

- Git and GitHub in Plain English
- Atlassian Git Branches Tutorial
- Git Branching (Official Documentation)
- Learn Git Branching (interactive tutorial)
- Visualizing Git Concepts (interactive visualization)
- Resolving a merge conflict (GitHub)

Chapter 15

Git Collaboration

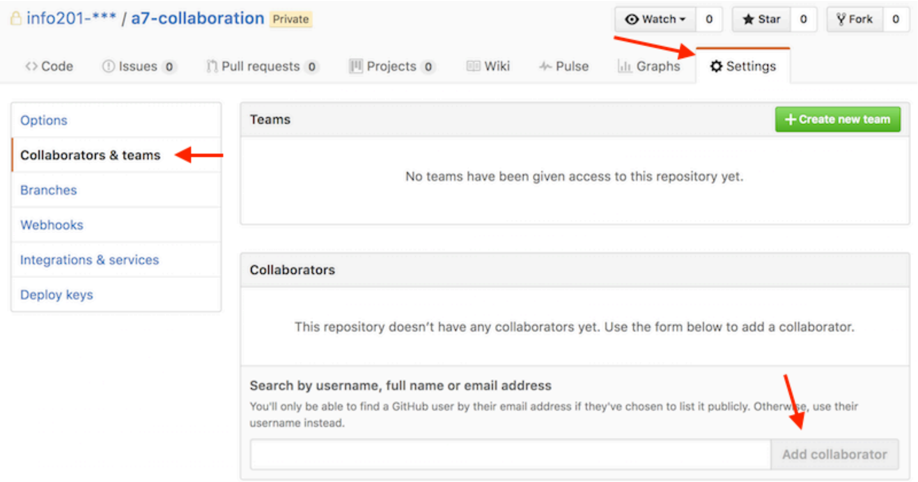
Being able to merge between branches allows you to work **collaboratively**, with multiple people making changes to the same repo and sharing those changes through GitHub. There are a variety of approaches (or **workflows**) that can be used to facilitate collaboration and make sure that people are effectively able to share code. This section describes a variety of different workflows, however we suggest the branch-based workflow called the **Feature Branch Workflow** for this course.

15.1 Centralized Workflow

In order to understand the Feature Branch Workflow, it's important to first understand how to collaborate on a centralized repository. The Feature Branch Workflow uses a **centralized repository** stored on GitHub—that is, every single member of the team will `push` and `pull` to a single GitHub repo. However, since each repository needs to be created under a particular account, this means that a ***single member*** of the team will need to create the repo (such as by accepting a GitHub Classroom assignment, or by clicking the *"New"* button on their "Repositories" tab on the GitHub web portal).

In order to make sure everyone is able to `push` to the repository, whoever creates the repo will need to **add the other team members as collaborators**. You can do this under the **Settings** tab:

Once you've added everyone to the GitHub repository, **each team member** will need to **`clone`** the repository to their local machines to work on the code individually. Collaborators can then `push` any changes they make to the central repository, and `pull` and changes made by others. Because multiple members will be contributing to the *same repositiory*, it's important to ensure that you are working on the most up-to-date version of the code. This means that you



No teams have been given access to this repository yet.

This repository doesn't have any collaborators yet. Use the form below to add a collaborator.

Search by username, full name or email address

You'll only be able to find a GitHub user by their email address if they've chosen to list it publicly. Otherwise, use their username instead.

Figure 15.1: Adding a collaborator to a Github repo (via the web portal).

will regularly have to **pull in changes** from GitHub that your team members may have committed. As a result, we suggest that you have a workflow that looks like this:

```
# Begin your work session by pulling in changes from GitHub
git pull origin master

# If necessary, resolve any merge conflicts and commit them
git add .
git commit -m "Merge in changes from GitHub"

# Do your work, then add, commit and push
git add .
git commit -m "Make progress on feature X"
git push origin master
```

Note, if someone pushes a commit to GitHub *before you push your changes*, you'll need to integrate those into your code (and test them!) before pushing up to GitHub. While working on a single `master` branch in this fashion is possible, you'll encounter fewer conflicts if you use a dedicated **feature branch** for each developer or feature you're working on.

15.2 Feature Branch Workflow

The Feature Branch Workflow is a natural extension of the Centralized Workflow that enhances the model by defining specific *branches* for different pieces of development (still with one centralized repository). The core idea behind the Feature Branch Workflow is that all development should take place on a dedicated **feature branch**, rather than on the `master` branch. This allows for different people to work on different branches without disturbing the main codebase. For example, you might have one branch `visualization` that focuses on adding a complex visualization, or another `experimental-analysis` that tries a bold new approach to processing the data. Each branch is based on a *feature* (capability or part) of the project, not a particular person: a single developer could be working on multiple feature branches.

The idea is that the `master` branch *always* contains "production-level" code: valid, completely working code that you could deploy or publish (read: give to your boss or teacher) at a whim. All feature branches branch off of `master`, and are allowed to contain temporary or even broken code (since they are still in development). This way there is always a "working" (if incomplete) copy of the code (`master`), and development can be kept isolated and considered independent of the whole. This is similar to the example with the `experiment` branch above.

The workflow thus works like this:

1. Ada decides to add a new feature or part to the code. She creates a new feature branch off of `master`:

   ```
   git checkout master
   git checkout -b adas-feature
   ```

2. Ada does some work on this feature

   ```
   # work is done outside of terminal

   git add .
   git commit -m "Add progress on feature"
   ```

3. Ada takes a break, pushing her changes to GitHub

   ```
   git push -u origin adas-feature
   ```

4. After talking to Ada, Bebe decides to help finish up the feature. She checks out the branch and makes some changes, then pushes them back to GitHub

   ```
   # fetch will "download" commits from GitHub, without merging them
   git fetch origin
   git checkout adas-feature
   ```

```
# work is on adas-feature done outside of terminal

git add .
git commit -m "Add more progress on feature"
git push origin adas-feature
```

5. Ada downloads Bebe's changes

   ```
   git pull origin adas-feature
   ```

6. Ada decides the feature is finished, and *merges* it back into `master`. But first, she makes sure she has the latest version of the `master` code to integrate her changes with

   ```
   git checkout master  # switch to master
   git pull origin master  # download any changes

   git merge adas-feature  # merge the feature into the master branch
   # fix any merge conflicts!!

   git push origin master  # upload the updated code to master
   ```

7. And now that the feature has been successfully added to the project, Ada can delete the feature branch (using `git branch -d branch_name`). See also here.

This kind of workflow is very common and effective for supporting collaboration. Note that as projects get large, you may need to start being more organized about how and when you create feature branches. For example, the **Git Flow** model organizes feature branches around product releases, and is often a starting point for large collaborative projects.

15.3 Forking Workflow

The Forking Workflow takes a **fundamentally different approach** to collaboration than the Centralized and Feature Branch workflows. Rather than having a single remote, each developer will have **their own repository** on GitHub that is *forked* from the original repository. As discussed in the introductory GitHub Chapter, a developer can create their own remote repository from an existing project by *forking* it on GitHub. This allows the individual to make changes (and contribute to) the project. However, we have not yet discussed how those changes can be integrated into the original code base. GitHub offers a feature called **pull requests** by which you can merge two remote branches (that is: `merge` two branches that are on GitHub). A **pull request** is a request for the changes from one branch to be pulled (merged) into another.

15.3.1 Pull Requests

Pull requests are primarily used to let teams of developers *collaborate*—one developer can send a request "hey, can you integrate my changes?" to another. The second developer can perform a **code review**: reviewing the proposed changes and making comments or asking for corrections to anything they find problematic. Once the changes are improved, the pull request can be **accepted** and the changes merged into the target branch. This process is how programmers collaborate on *open-source software* (such as R libraries like `dplyr`): a developer can *fork* an existing professional project, make changes to that fork, and then send a pull request back to the original developer asking them to merge in changes ("will you include my changes in your branch/version of the code?").

Pull requests should only be used when doing collaboration using remote branches! Local branches should be `merge`'d locally using the command-line, not GitHub's pull request feature.

In order to issue a pull request, both branches you wish to merge will need to be `pushed` to GitHub (whether they are in the same repo or in forks). To issue the pull request, navigate to your repository on GitHub's web portal and choose the **New Pull Request** button (it is next to the drop-down that lets you view different branches).

In the next page, you will need to specify which branches you wish to merge. The **base** branch is the one you want to merge *into* (often `master`), and the **head** branch (labeled "compare") is the branch with the new changes you want to merge (often a feature branch; see below).

Add a title and description for your pull request. These should follow the format for git commit messages. Finally, click the **Create pull request** button to finish creating the pull request.

Important! The pull request is a request to merge two branches, not to merge a specific set of commits. This means that you can *push more commits* to the head/merge-from branch, and they will automatically be included in the pull request—the request is always "up-to-date" with whatever commits are on the (remote) branch.

You can view all pull requests (including those that have been accepted) through the **Pull Requests** tab at the top of the repo's web portal. This is where you can go to see comments that have been left by the reviewer.

If someone sends you a pull request (e.g., another developer on your team), you can accept that pull request through GitHub's web portal. If the branches can be merged without a conflict, you can do this simply by hitting the **Merge pull request** button. However, if GitHub detects that a conflict may occur, you will need to pull down the branches and merge them locally.

It is best practice to *never* accept your own pull requests! If you don't need any collaboration, just merge the branches locally.

Note that when you merge a pull request via the GitHub web site, the merge is done entirely on the server. Your local repo will not yet have those changes, and so you will need to use `git pull` to download the updates to an appropriate branch.

Resources

- Atlassian Git Workflows Tutorial

Chapter 16

The `shiny` Framework

Adding **interactivity** to a data report is a highly effective way of communicating that information and enabling users to explore a data set. In this chapter, you will learn about the **Shiny** framework for building interactive applications in R. Shiny provides a structure for communicating between a user-interface (i.e., a web-browser) and an R session, allowing users to interactively change the "code" that is run and the data that are output. This not only enables developers to create **interactive graphics**, but provides a way for users to interact directly with an R session (without writing any code!).

16.1 Creating Shiny Apps

Shiny is a **web application framework for R**. As opposed to a simple (static) web page like you've created with R Markdown, a *web application* is an interactive, dynamic web page—the user can click on buttons, check boxes, or input text in order to change the presentation of the data. Shiny is a *framework* in that it provides the "code" for producing and enabling this interaction, while you as the developer simply "fill in the blanks" by providing *variables* or *functions* that the provided code will utilize to create the interactive page.

`shiny` is another external package (like `dplyr` and `ggplot2`), so you will need to install and load it in order to use it:

```r
install.packages("shiny")  # once per machine
library("shiny")
```

This will make all of the framework functions and variables you will need to work with available.

16.1.1 Application Structure

Shiny applications are divided into two parts:

1. The **User Interface (UI)** defines how the application will be *displayed* in the browser. The UI can render R content such as text or graphics just like R Markdown, but it can also include **widgets**, which are interactive controls for your application (think buttons or sliders). The UI can specify a **layout** for these components (e.g., so you can put widgets above, below, or beside one another).

   The UI for a Shiny application is defined as a **value**, usually one returned from calling a **layout function**. For example:

   ```
   # The ui is the result of calling the `fluidPage()` layout function
   my.ui <- fluidPage(
     # A widget: a text input box (save input in the `username` key)
     textInput('username', label="What is your name?"),

     # An output element: a text output (for the `message` key)
     textOutput('message')
   )
   ```

   This UI defines a fluidPage (where the content flows "fluidly" down the page), that contains two *content elements*: a text input box where the user can type their name, and some outputted text based on the `message` variable.

2. The **Server** defines the data that will be displayed through the UI. You can think of this as an interactive R script that the user will be able to "run": the script will take in *inputs* from the user (based on their interactions) and provide *outputs* that the UI will then display. The server uses **reactive expressions**, which are like functions that will automatically be re-run whenever the input changes. This allows the output to be dynamic and interactive.

   The Server for a Shiny application is defined as a **function** (as opposed to the UI which is a *value*). This function takes in two *lists* as argments: an `input` and `output`. It then uses *render functions* and *reactive expressions* that assign values to the `output` list based on the `input` list. For example:

   ```
   # The server is a function that takes `input` and `output` args
   my.server <- function(input, output) {
     # assign a value to the `message` key in `output`
     # argument is a reactive expression for showing text
     output$message <- renderText({
       # use the `username` key from input and and return new value
       # for the `message` key in output
       return(paste("Hello", input$username))
   ```

```
  })
}
```

Combined, this UI and server will allow the user to type their name into an input box, and will then say "hello" to whatever name is typed in.

More details about the UI and server components can be found in the sections below.

16.1.1.1 Combining UI and Server

There are two ways of combining the UI and server:

The first (newer) way is to define a file called **`app.R`**. This file should call the **`shinyApp()`** function, which takes a UI value and Server function as arguments. For example:

```
# pass in the variables defined above
shinyApp(ui = my.ui, server = my.server)
```

Executing the `shinyApp()` function will start the App (you can also click the **"Run App"** button at the top of RStudio).

- Note: if you change the UI or the Server, you do **not** need to stop and start the app; you can simply refresh the browser or viewer window and it will reload with the new UI and server.

- If you need to stop the App, you can hit the "Stop Sign" icon on the RStudio console.

Using this function allows you to define your entire application (UI and Server) in a single file (which **must** be named `app.R`). This approach is good for simple applications that you wish to be able to share with others, since the entire application code can be listed in a single file.

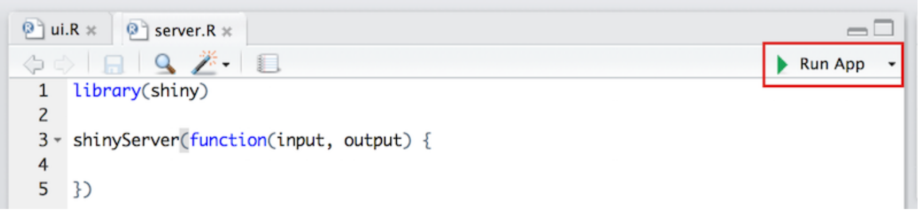
However, it is also possible to define the UI and server as *separate* files. This allows you to keep the presentation (UI) separated from the logic (server) of your application, making it easier to maintain and change in the future. To do this, you define two separate files: **`ui.R`** for the UI and **`server.R`** for the Server (the files **must** be named `ui.R` and `server.R`). In these files, you must call the functions `shinyUI()` and `shinyServer()` respectively to create the UI and server, and then RStudio will automatically combine these files together into an application:

```
# In ui.R file
my.ui <- fluidPage(
  # define widgets
)
```

```
shinyUI(my.ui)
```

```
# In server.R file
my.server <- function(input, output) {
  # define output reactive expressions
}

shinyServer(my.server)
```

You can then run the app by using the **"Run App"** button at the top of RStudio:



```
library(shiny)

shinyServer(function(input, output) {

})
```

Figure 16.1: Use RStudio to run a shiny app defined in separate UI and Server files.

This chapter will primarily use the "single file" approach for compactness and readability, but you are encouraged to break up the UI and server into separate files for your own, larger applications.

- Note that it is also possible to simply define the (e.g.) `my.ui` and `my.server` variables in separate files, and then use `source()` to load them into the `app.R` file and pass them into `shinyApp()`.

16.1.2 The UI

The UI defines how the app will be displayed in the browser. You create a UI by calling a **layout function** such as `fluidPage()`, which will return a UI definition that can be used by the `shinyUI()` or `shinyApp()` functions.

You specify the "content" that you want the layout to contain (and hence the app to show) by passing each **content element** (piece of content) as an *argument* to that function:

```
# a "pseudocode" example, calling a function with arguments
ui <- fluidPage(element1, element2, element3)
```

Content elements are defined by calling specific *functions* that create them: for example `h1()` will create an element that has a first-level heading, `textInput()`

will create an element where the user can enter text, and `textOutput` will create an element that can have dynamic (changing) content. Usually these content elements are defined as *nested* (anonymous) variables, each on its own line:

```
# still just calling a function with arguments!
ui <- fluidPage(
  h1("My App"),  # first argument
  textInput('username', label="What is your name?"),  # second argument
  textOutput('message')  # third argument
)
```

Note that layout functions *themselves return content elements*, meaning it is possible to include a layout inside another layout. This allows you to create complex layouts by combining multiple layout elements together. For example:

```
ui <- fluidPage(   # UI is a fluid page
  titlePanel("My Title"),  # include panel with the title (also sets browser title)

  sidebarLayout(   # layout the page in two columns
    sidebarPanel(  # specify content for the "sidebar" column
      p("sidebar panel content goes here")
    ),
    mainPanel(     # specify content for the "main" column
      p("main panel content goes here")
    )
  )
)
```

See the Shiny documentation and gallery for details and examples of doing complex application layouts.

Fun Fact: much of Shiny's styling and layout structure is based on the Bootstrap web framework.

You can include *static* (unchanging) content in a Shiny UI layout—this is similar to the kinds of content you would write in Markdown (rather than inline R) when using R Markdown. However, you usually don't specify this content using Markdown syntax (though it is possible to include a markdown file's content). Instead, you include content functions that produce HTML, the language that Markdown is converted to when you look at it in the browser. These functions include:

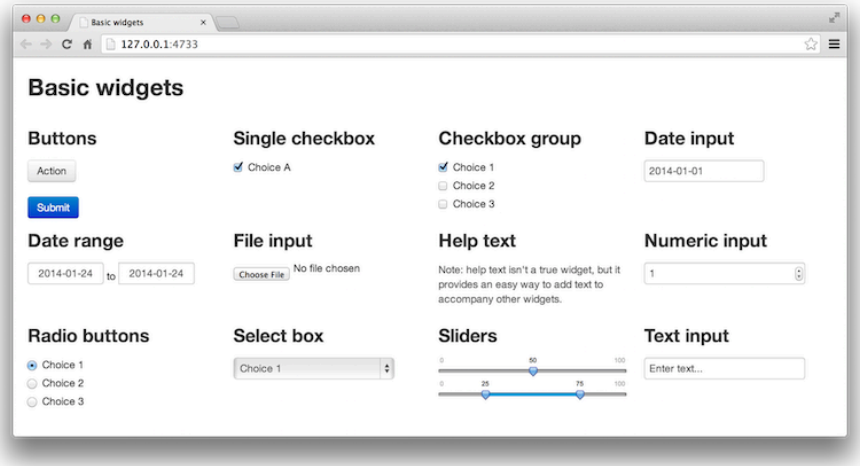
- `p()` for creating paragraphs, the same as plain text in Markdown
- `h1()`, `h2()`, `h3()` etc for creating headings, the same as `# Heading 1`, `## Heading 2`, `### Heading 3` in Markdown
- `em()` for creating *emphasized* (italic) text, the same as `_text_` in Markdown
- `strong()` for creating **strong** (bolded) text, the same as `**text**` in Markdown

- `a(text, href='url')` for creating hyperlinks (anchors), the same as `[text](url)` in Markdown
- `img(text, src='url')` for including images, the same as `![text](url)` in Markdown

There are many other methods as well, see this tutorial lesson for a list. If you are familiar with HTML, then these methods will seem familiar; you can also write content in HTML directly using the `tag()` content function.

16.1.2.1 Control Widgets and Reactive Outputs

It is more common to include **control widgets** as content elements in your UI layout. Widgets are *dynamic* (changing) control elements that the user can interact with. Each stores a **value** that the user has entered, whether by typing into a box, moving a slider, or checking a button. When the user changes their input, the stored *value* automatically changes as well.



Basic widgets

Buttons

Action

Submit

Single checkbox

☑ Choice A

Checkbox group

☑ Choice 1
□ Choice 2
□ Choice 3

Date input

2014-01-01

Date range

2014-01-24 to 2014-01-24

File input

Choose File No file chosen

Help text

Note: help text isn't a true widget, but it provides an easy way to add text to accompany other widgets.

Numeric input

1

Radio buttons

● Choice 1
○ Choice 2
○ Choice 3

Select box

Choice 1

Sliders

0 50 100

0 25 75 100

Text input

Enter text...

Figure 16.2: Examples of control widgets (image from shiny.rstudio.com).

Like other content elements, widgets are created by calling an appropriate function. For example:

- `textInput()` creates a box in which the user can enter text
- `sliderInput()` creates a slider
- `selectInput()` creates a dropdown menu the user can choose from
- `checkboxInput()` creates a box the user can check (using `checkboxGroupInput()` to group them)

- `radioButtons()` creates “radio” buttons (which the user can select only one of at a time)

See the documentation, and this tutorial lesson for a complete list.

All widget functions take at least two arguments:

- A **name** (as a string) for the widget’s value. This will be the **“key”** that will allow the server to be able to access the value the user has input (think: the key in the `input` *list*).
- A **label** (a string or content element described above) that will be shown alongside the widget and tell the user what the value represents. Note that this can be an empty string (`""`) if you don’t want to show anything.

Other arguments may be required by a particular widget—for example, a slider’s `min` and `max` values:

```
# this function would be nested in a layout function (e.g., `fluidPage()`)
sliderInput('age',                # key this value will be assigned to
            "Age of subjects",    # label
            min = 18,             # minimum slider value
            max = 80,             # maximum slider value
            value = 42            # starting value
           )
```

Widgets are used to provide **inputs *to*** the Server; see the below section for how to use these inputs, as well as examples from the gallery.

In order to display **outputs *from*** the Server, you include a **reactive output** element in your UI layout. These are elements similar to the basic content elements, but instead of just displaying *static* (unchanging) content they can display *dynamic* (changing) content produced by the Server.

As with other content elements, reactive outputs are creating by calling an appropriate function. For example:

- `textOutput()` displays output as plain text (note this output can be nested in a content element for formatting)
- `tableOutput()` displays output as a data table (similar to `kable()` in R Markdown). See also `dataTableOutput()` for an interactive version!
- `plotOutput()` displays a graphical plot, such as one created with `ggplot2`

Each of these functions takes as an argument the **name** (as a string) of the value that will be displayed. This is the **“key”** that allows it to access the value the Server is outputting. Note that the functions may take additional arguments as well (e.g., to specify the size of a plot); see the documentation for details.

16.1.3 The Server

The Server defines how the data input by the user will be used to create the output displayed by the app—that is, how the *control widgets* and *reactive outputs* will be connected. You create a Server by *defining a new function* (not calling a provided one):

```
server <- function(input, output) {
  # assign values to `output` here
}
```

Note that this is *just a normal function* that happens two take **lists** as arguments. That means you can include the same kinds of code as you normally would—though that code will only be run once (when the application is first started) unless defined as part of a reactive expression.

The first argument is a list of any values defined by the *control widgets*: each **name** in a control widget will be a **key** in this list. For example, using the above `sliderInput()` example would cause the list to have an `age` key (referenced as `input$age`). This allows the Server to access any data that the user has input, using the key names defined in the UI. Note that the values in this list *will change as the user interacts with the UI's control widgets.*

The purpose of the Server function is to assign new *values* to the `output` argument list (each with an appropriate *key*). These values will then be displayed by the *reactive outputs* defined in the UI. To make it so that the values can actually be displayed by by the UI, the values assigned to this list need to be the results of **Render Functions**. Similar to creating widgets or reactive outputs, different functions are associated with different types of output the server should produce. For example:

- `renderText()` will produce text (character strings) that can be displayed (i.e., by `textOutput()` in the UI)
- `renderTable()` will produce a table that can be displayed (i.e., by `tableOutput()` in the UI)
- `renderPlot()` will produce a graphical plot that can be displayed (i.e., by `plotOutput()` in the UI)

Render functions take as an argument a **Reactive Expression**. This is a lot like a function: it is a **block** of code (in braces **`{}`**) that **returns** the value which should be rendered. For example:

```
output$msg <- renderText({
  # code goes here, just like any other function
  my.greeting <- "Hello"

  # code should always draw upon a key from the `input` variable
  message <- paste(my.greeting, input$username)
  # return the variable that will be rendered
  return(message)
})
```

The only difference between writing a *reactive expression* and a function is that you only include the *block* (the braces and the code inside of them): you don't use the keyword `function` and don't specify a set of arguments.

This technically defines a *closure*, which is a programming concept used to encapsulate functions and the context for those functions.

These *reactive expressions* will be "re-run" **every time** one of the `input` values that it references changes. So if the user interacts with the `username` control widget (and thereby changes the value of the `input` list), the expression in the above `renderText()` will be executed again, returning a new value that will be assigned to `output$msg`. And since `output$msg` has now changed, any *reactive output* in the UI (e.g., a `textOutput()`) will update to show the latest value. This makes the app interactive!

16.1.3.1 Multiple Views

It is quite common in a Shiny app to produce *lots* of output variables, and thus to have multiple reactive expressions. For example:

```
server <- function(input, output) {
  # render a histogram plot
  output$hist <- renderPlot({
    uniform.nums <- runif(input$num, 1, 10)  # random nums between 1 and 10
    return(hist(uniform.nums))  # built-in plotting for simplicity
  })

  # render the counts
  output$counts <- renderPrint({
    uniform.nums <- runif(input$num, 1, 10)  # random nums between 1 and 10
    counts <- factor(cut(uniform.nums, breaks=1:10))  # factor
    return(summary(counts))  # simple vector of counts
  })
}
```

If you look at the above example though, you'll notice that each render function produces a set of random numbers… which means each will produce a *different* set of numbers! The histogram and the table won't match!

This is an example of where you want to share a single piece of data (a single **model**) between multiple different renditions (multiple **views**). Effectively, you want to define a shared variable (the `uniform.nums`) that can be referenced by both render functions. But since you need that shared variable to be able to *update* whenever the `input` changes, you need to make it be a *reactive expression* itself. You can do this by using the **`reactive()`** function:

```
server <- function(input, output) {
  # define a reactive variable
  uniform.nums <- reactive({
    return(runif(input$num, 1, 10))  # just like for a render function
  })

  # render a histogram plot
  output$hist <- renderPlot({
    return(hist(uniform.nums()))  # call the reactive variable AS A FUNCTION
  })

  # render the counts
  output$counts <- renderPrint({
    counts <- factor(cut(uniform.nums(), breaks=1:10))  # call the reactive variable
    return(summary(counts))
  })
}
```

The `reactive()` function lets you define a single "variable" that is a *reactive function* which can be called from within the render functions. Importantly, the value returned by this function (the `uniform.nums()`) only changes **when a referenced `input` changes**. Thus as long as `input$num` stays the same, `uniform.nums()` will return the same value.

This is very powerful for allowing multiple **views** of a single piece of data: you can have a single source of data displayed both graphically and textually, with both views linked off of the same processed data table. Additionally, it can help keep your code more organized and readable, and avoid needing to duplicate any processing.

16.2 Publishing Shiny Apps

Sharing a Shiny App with the world is a bit more involved than simply pushing the code to GitHub. You can't just use GitHub pages to host the code because, in addition to the HTML UI, you need an R interpreter session to run the Server that the UI can connect to (and GitHub does not provide R interpreters)!

While there are a few different ways of "hosting" Shiny Apps, in this course you'll use the simplest one: hosting through **shinyapps.io**. shinyapps.io is a platform for hosting and running Shiny Apps; while large applications cost money, anyone can deploy a simple app (like the ones you'll create in this course) for free.

In order to host your app on shinyapps.io, you'll need to create a free account. Note that you can sign up with GitHub or your Google/UW account. Follow the site's instructions to

1. Select an account name (use something professional, like you used when signing up with GitHub)

2. Install the required `rsconnect` package (may be included with RStudio)

3. Set your authorization token ("password"). Just click the green "Copy to Clipboard" button, and then paste that into the **Console** in RStudio. You should only need to do this once.

   Don't worry about "Step 3 - Deploy"; you'll do that through RStudio directly!

After you've set up an account, you can *Run* your application (as above) and hit the **Publish** button in the upper-right corner:

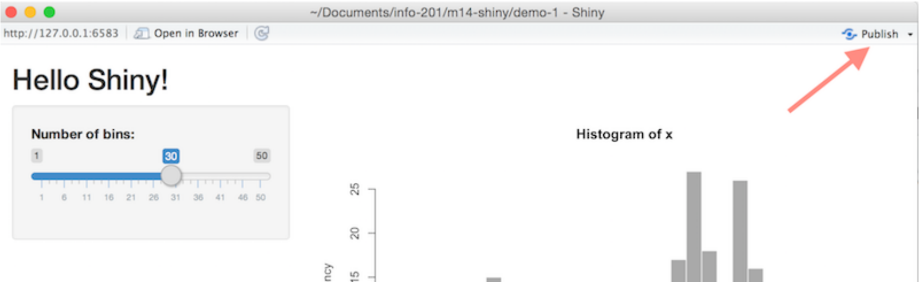


Figure 16.3: How to publish a running Shiny App to shinyapps.io.

This will put your app online, available at

```
https://USERNAME.shinyapps.io/APPNAME/
```

Important Publishing to shinyapps.io is one of the major "pain points" in working with Shiny. For the best experience, be sure to:

1. Always test and debug your app *locally* (e.g., on your own computer, by running the App through RStudio). Make sure it works on your machine before you try to put it online.

2. Use correct folder structures and *relative paths*. All of your app should be in a single folder (usually named after the project). Make sure any `.csv` or `.R` files referenced are inside the app folder, and that you use relative paths to refer to them. Do not include any `setwd()` statements in your code; you should only set the working directory through RStudio (because shinyapps.io will have its own working directory).

3. It is possible to see the logs for your deployed app, which may include errors explaining any problems that arise when you deploy your app.

For more options and details, see the shinyapps.io documentation.

Resources

- Shiny Documentation
- Shiny Basics Article
- Shiny Tutorial (video; links to text at bottom)
- Shiny Cheatsheet
- Shiny Example Gallery
- shinyapps.io User Guide
- Interactive Plots with Shiny (see also here)
- Interactive Docs with Shiny

Appendix A

Plotly

In this module, you'll start building visualizations using the Plotly API. Plotly is a visualization software that recently open-sourced it's API to JavaScript, MatLab, Python, and R, making it quite valuable to learn. Plotly graphs are fairly customizable, and (by default) have a variety of interactive methods with each chart (i.e., hover, brush to zoom, pan, etc.). Many of these events are fairly cumbersome to build programmatically, which makes a library like Plotly quite attractive.

A.1 Getting Started

The Plotly API is an R package that you'll use to build interactive graphics. Like other open-source that we've used in this course, we'll load this API as a R package as follows:

```r
# Install package
install.packages("plotly")

# Load library
library(plotly)
```

Then, the `plot_ly` object will be accessible to you to build graphics on your page.

Note: sometimes RStudio fails to show your plotly graph in a website preview when you use `plotly` in an `RMarkdown` docunment. However, if you click on the **Open in Browser** button, you should be able to interact with your chart as it will show in a web browser. This isn't your fault, and doesn't need to be de-bugged.

A.2 Basic Charts

One of the best ways to start building charts with Plotly is to take a look at a basic example of your choice, and explore the syntax. In general, to build a Plotly object (graph) you'll pass a **dataframe** into the `plot_ly` function, then **adjust the parameters** to specify what you want to visualize. For example, here is the basic example of a scatterplot from the documentation:

```r
# Make a scatterplot of the iris data
plot_ly(data = iris, x = ~Sepal.Length, y = ~Petal.Length, mode = "markers", type =
```

The approach seems pretty straightforward – in fact, if you exclude `type = "scatter"` and `mode = "markers"`, Plotly will make an educated guess about what type of plot you want (and in this case, it will in fact create a scatterplot!). The only syntax that looks a bit strange is the tilde character (~). In R, the tilde designates a variable as a **formula**, which was a design choice of the developers of the API.

A.3 Layout

While the `plot_ly` function controls the data that is being visualized, additional chart options such as *titles and axes* are controlled by the `layout` function. The `layout` function accepts as a parameter **a plotly object**, and *manipulates that object*. Again, I think a great place to start is an example in the documentation:

```r
# From documentation

# Create a plotly object `p`
p <- plot_ly(data = iris, x = ~Sepal.Length, y = ~Petal.Length, type = "scatter", mo
        marker = list(size = 10,
                       color = 'rgba(255, 182, 193, .9)',
                       line = list(color = 'rgba(152, 0, 0, .8)',
                                   width = 2))) %>%
  # Use pipe function to pass the plotly object into the `layout` function
  layout(title = 'Styled Scatter',
         yaxis = list(zeroline = FALSE),
         xaxis = list(zeroline = FALSE))

# Show chart
p
```

This example uses the pipe operator (`%>%`) to pass the plotly object *into* the `layout` function as the first argument. We can then infer the structure of the other parameters, which you can read about more in the API Documentation:

A.4 Hovers

By default, plotly will provide information on each element when you hover over it (which is **awesome**). To manipulate the information in a hover, you can modify the `text` attribute in the `plot_ly` function, and you can **use your data** to populate the information on hover:

```
# From documentation

# Create data
d <- diamonds[sample(nrow(diamonds), 1000), ]

# Create plot, specifying hover text
p <- plot_ly(
  d, x = ~carat, y = ~price, mode = "markers", type = "scatter",
  # Hover text:
  text = ~paste0("Price:$", price, '<br>Cut: ', cut),
  color = ~carat, size = ~carat
)

# Show chart
p
```

Note, plotly allows you to specify HTML syntax in the text formula. In this case, it uses a line break `<br>` to improve the layout.

Resources

- Plotly Website
- Plotly R API
- Getting Started with Plotly for R
- Plotly Cheatsheet
- Plotly book (extensive documentation)

Appendix B

R Language Control Structures

In chapter Functions we briefly described conditional execution using the `if/else` construct. Here we introduce the other main control structures, and cover the if/else statement in more detail.

B.1 Loops

One of the common problems that computers are good at is to execute similar tasks repeatedly many times. This can be achieved through *loops*. R implements three loops: for-loop, while-loop, and repeat-loop, these are amont the most popular loops in many modern programming languages. Loops are often advised agains in case of R. While it is true that loops in R are more demanding than in several other languages, and can often be replaced by more efficient constructs, they nevertheless have a very important role in R. Good usage of loops is easy, fast, and improves the readability of your code.

B.1.1 For Loop

For-loop is a good choice if you want to run some statements repeatedly, each time picking a different value for a variable from a sequence; or just repeat some statements a given number of times. The syntax of for-loop is the following:

```
for(variable in sequence) {
                           # do something many times.  Each time 'variable' has
                           # a different value, taken from 'sequence'
}
```

The statement block inside of the curly braces {} is are repeated through the sequence.

Despite the curly braces, the loop body is still evaluated in the same environment as the rest of the code. This means the variables generated before entering the loop are visible from inside, and the variables created inside will be visible after the loop ends.

A simple and perhaps the most common usage of for loop is just to run some code a given number of times:

```
for(i in 1:5)
   print(i)
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
```

Through the loop the variable `i` takes all the values from the sequence `1:5`. Note: we do not need curly braces if the loop block only contains a single statement.

But the sequence does not have to be a simple sequence of numbers. It may as well be a list:

```
data <- list("Xiaotian", 25, c(2061234567, 2069876543))
for(field in data) {
   cat("field length:", length(field), "\n")
}
```

```
## field length: 1
## field length: 1
## field length: 2
```

As another less common example, we may run different functions over the same data:

```
data <- 1:4
for(fun in c(sqrt, exp, sin)) {
   print(fun(data))
}
```

```
## [1] 1.000000 1.414214 1.732051 2.000000
## [1]  2.718282  7.389056 20.085537 54.598150
## [1]  0.8414710  0.9092974  0.1411200 -0.7568025
```

B.1.2 While loop

While loop let's the programmer to decide explicitly if we want to execute the statements one more time:

```
while(condition) {
                                 # do something if the 'condition' is true
}
```

For instance, we can repeat something a given number of times (although this is more of a task for for-loop):

```
i <- 0
while(i < 5) {
   print(i)
   i <- i + 1
}
```

```
## [1] 0
## [1] 1
## [1] 2
## [1] 3
## [1] 4
```

Note that in case the condition is not true to begin with, the loop never executes and control is immediately passed to the code after the loop:

```
i <- 10
while(i < 5) {
   print(i)
   i <- i + 1
}
cat("done\n")
```

```
## done
```

While loop is great for handling repeated attempts to get input in shape. One can write something like this:

```
data <- getData()
while(!dataIsGood(data)) {
   data <- getData(tryReallyHard = TRUE)
}
```

This will run `getData()` until data is good. If it is not good at the first try, the code tries really hard next time(s).

While can also easily do infinite loops with:

```
while(TRUE) { ... }
```

will execute until something breaks the loop. Another somewhat interesting usage of `while` is to disable execution of part of your code:

```
while(FALSE) {
   ## code you don't want to be executed
}
```

B.1.3 repeat-loop

The R way of implementing repeat-loop does not include any condition and hence it just keeps running infdefinitely, or until something breaks the execution (see below):

```
repeat {
   ## just do something indefinitely many times.
   ## if you want to stop, use `break` statement.
}
```

Repeat is useful in many similar situations as while. For instance, we can write a modified version of getting data using `repeat`:

```
repeat {
   data <- getData()
   if(dataIsGood())
      break
}
```

The main difference between this, and the previous `while`-version is that we a) don't have to write `getData` function twice, and b) on each try we attempt to get data in the same way.

Exactly as in the case of *for* and *while* loop, enclosing the loop body in curly braces is only necessary if it contains more than one statement. For instance, if you want to close all the graphics devices left open by buggy code, you can use the following one-liner from console:

```
repeat dev.off()
```

The loop will be broken by error when attempting to close the last null device.

It should also be noted that the loop does not create a separate environment (scope). All the variables from outside of the loop are visible inside, and variables created inside of the loop remain accessible after the loop terminates.

B.1.4 Leaving Early: `break` and `next`

A straightforward way to leave a loop is the `break` statement. It just leaves the loop and transfers control to the code immediately after the loop body. It breaks all three loops discussed here (*for*, *while*, and *repeat*-loops) but not some of the other types of loops implemented outside of base R. For instance:

```
for(i in 1:10) {
   cat(i, "\n")
   if(i > 4)
      break
}
```

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

```
cat("done\n")
```

```
## done
```

As the loop body will use the same environment (the same variable values) as the rest of the code, the loop variables can be used right afterwards. For instance, we can see that `i = 5`:

```
print(i)
```

```
## [1] 5
```

Opposite to `break`, `next` will throw the execution flow back to the head of the loop without running the commands following `next`. In case of for-loop, this means a new value is picked from the sequence, in case of while-loop the condition is evaluated again; the head of repeat loop does not do any calculations. For instance, we can print only even numbers:

```
for(i in 1:10) {
   if(i %% 2 != 0)
      next
   print(i)
}
```

```
## [1] 2
## [1] 4
## [1] 6
## [1] 8
## [1] 10
```

`next` jumps over printing where modulo of division by 2 is not 0.

B.1.5 When (Not) To Use Loops In R?

In general, use loops if it improves code readability, makes it easier to extend the code, and does not cause too big inefficiencies. In practice, it means use loops for repeating something slow. Don't use loops if you can use vectorized operators instead.

For instance, look at the examples above where we were reading data until it was good. There is probably very little you can achieve by avoiding loops–except making your code messy. First, the input will probably attempted a few times only (otherwise re-consider how you get your data), and second, data input is probably a far slower process than the loop overhead. Just use the loop.

Similar arguments hold if you consider to loop over input files to read, output files to create, figures to plot, webpage to scrape... In all these cases the loop overheads are negligible compared to the task performed inside the loops. And there are no real vectorized alternatives anyway.

The opposite is the case where good alternatives exist. Never do this kind of computations in R:

```r
res <- numeric()
for(i in 1:1000) {
   res <- c(res, sqrt(i))
}
```

This loop introduces three inefficiencies. First, and most importantly, you are growing your when adding new values to it. This involves creating new and longer vectors again and again when the old one is filled up. This is very slow. Second, as the true vectorized alternatives exist here, one can easily speed up the code by a magnitude or more by just switching to the vectorized code. Third, and perhaps most importantly, this code is much harder to read than the pure vectorized expression:

```r
res <- sqrt(1:1000)
```

We can easily demonstrate how much faster the vectorized expressions work. Let's do this using *microbenchmark* library (it provides nanosecond timings). Let's wrap these expressions in functions for easier handling by *microbenchmark*. We also include a middle version where we use a loop but avoid the terribly slow vector re-allocation by creating the right-sized result in advance:

```r
seq <- 1:1000 # global variables are easier to handle with microbenchmark
baad <- function() {
   r <- numeric()
   for(i in seq)
      r <- c(r, sqrt(i))
}
soso <- function() {
   r <- numeric(length(seq))
   for(i in seq)
      r[i] <- sqrt(i)
}
good <- function()
   sqrt(seq)
microbenchmark::microbenchmark(baad(), soso(), good())
```

```
## Unit: microseconds
##    expr     min        lq       mean    median        uq      max neval
##  baad() 934.824 1016.5195 1890.20944 1883.6730 2254.0830 7282.948   100
##  soso()  45.693   48.7990   80.39569   50.5070   55.4955 2250.595   100
##  good()   2.724    3.1115   12.89108    4.2455    5.6015  799.499   100
```

When comparing the median values, we can see that the middle example is 10x slower than the vectorized example, and the first version is almost 300x slower!

The above holds for “true” vectorized operations. R provides many pseudo-vectorizers (the lapply family) that may be handy in many circumstances, but not in case you need speed. Under the hood these functions just use a loop. We can demonstrate this by adding one more function to our family:

```
soso2 <- function() {
   sapply(seq, sqrt)
}
microbenchmark::microbenchmark(baad(), soso(), soso2(), good())
```

```
## Unit: microseconds
##     expr     min        lq       mean    median        uq       max neval
##   baad() 942.480 1037.0090 2099.52239 1633.0095 1956.6935 50294.902   100
##   soso()  45.500   49.0660   54.23969   50.8465   52.7295   100.674   100
##  soso2() 263.889  293.6975  349.25330  303.2855  329.7970  1267.627   100
##   good()   2.849    3.1025    4.18621    3.7955    4.8280    10.257   100
```

As it turns out, `sapply` is much slower than plain for-loop. Lapply-family is not a substitute for true vectorization. However, one may argue that `soso2` function is easier to understand than explicit loop in `soso`. In any case, it is easier to type when working on the interactive console.

Finally, it should be noted that when performing truly vectorized operations on long vectors, the overhead of R interpreter becomes negligible and the resulting speed is almost equalt to that of the corresponding C code. And optimized R code can easily beat non-optimized C.

B.2 More about `if` and `else`

The basics of if-else blocks were covered in Chapter Functions. Here we discuss some more advanced aspects.

B.2.1 Where To Put Else

Normally R wants to understand if a line of code belongs to an earlier statement, or is it beginning of a fresh one. For instance,

```
if(condition) {
   ## do something
}
else {
   ## do something else
}
```

may fail. R finishes the if-block on line 3 and thinks line 4 is beginning of a new statement. And gets confused by the unaccompanied `else` as it has already forgotten about the `if` above. However, this code works if used inside a function, or more generally, inside a {…} block.

But else may always stay on the same line as the the statement of the if-block. You are already familiar with the form

```
if(condition) {
   ## do something
} else {
   ## do something else
}
```

but this applies more generally. For instance:

```
if(condition)
   x <- 1 else {
             ## do something else
          }
```

or

```
if(condition) { ... } else {
             ## do something else
          }
```

or

```
if(condition) { x <- 1; y <- 2 } else z <- 3
```

the last one-liner uses the fact that we can write several statements on a single line with `;`. Arguably, such style is often a bad idea, but it has it's place, for instance when creating anonymous functions for `lapply` and friends.

B.2.2 Return Value

As most commands in R, if-else block has a return value. This is the last value evaluated by the block, and it can be assigned to variables. If the condition is false, and there is no else block, `if` returns `NULL`.

Using return values of if-else statement is often a recipe for hard-to-read code but may be a good choice in other circumstances, for instance where the multy-line form will take too much attention away from more crucial parts of the code. The line

```
n <- 3
parity <- if(n %% 2 == 0) "even" else "odd"
parity
```

```
## [1] "odd"
```

is rather easy to read. This is the closest general construct in R corresponding to C conditional shortcuts `parity = n % 2 ? "even" : "odd"`.

Another good place for such compressed if-else statements is inside anonymous functions in `lapply` and friends. For instance, the following code replaces `NULL`-s and `NA`-s in a list of characters:

```
emails <- list("ott@example.com", "xi@abc.net", NULL, "li@whitehouse.gov", NA)
n <- 0
sapply(emails, function(x) if(is.null(x) || is.na(x)) "-" else x)
```

```
## [1] "ott@example.com"   "xi@abc.net"        "-"
## [4] "li@whitehouse.gov" "-"
```

B.3 switch: Choosing Between Multiple Conditions

If-else is appropriate if we have a small number of conditions, potentially only one. Switch is a related construct that tests a large number of equality conditions. The best way to explain it's syntax is through an example:

```
stock <- "AMZN"
switch(stock,
       "AAPL" = "Apple Inc",
       "AMZN" = "Amazon.com Inc",
       "INTC" = "Intel Corp",
       "unknown")
```

```
## [1] "Amazon.com Inc"
```

Switch expression attempts to match the expression, here `stock = "AMZN"`, to a list of alternatives. If one of the alternative matches the name of the argument, that argument is returned. If none matches, the nameless default argument (here “unknown”) is returned. In case there is not default argument, `switch` returns `NULL`.

Switch allows to specify the same return value for multiple cases:

```
switch(stock,
       "AAPL" = "Apple Inc",
       "AMZN" =, "INTC" = "something else",
       "unknown")
```

```
## [1] "something else"
```

If the matching named argument is empty, the first non-empty named argument is evaluated. In this case this is “something else”, corresponding to the name “INTC”.

As in case of `if`, `switch` only accepts length-1 expressions. Switch also allows to use integers instead of character expressions, in that case the return value list should be without names.

Appendix C

Thinking Big: Data Tables

Data frames are core elements for data handling in R. However, the suffer from several limitations. One of the major issue with data frames is that they are memory hungry and slow. This is not an issue when working with relatively small datasets (say, up to 100,000 rows). However, when your dataset size exceed gigabytes, dataframes may be infeasibly slow and too memory hungry.

C.1 Background: Passing By Value And Passing By Reference

R is (mostly) a pass-by-value language. This means that when you modify the data, at every step a new copy of the complete modified object is created, stored in memory, and the former object is freed (carbage-collected) if not in use any more.

The main advantage of this approach is consistency: we have the guarantee that functions do not modify their inputs. However, in case of large objects, copying may be slow, and even more, it requires at least twice as much memory before the old object is freed. In case of more complex process pipelines, the memory consumption may be even more than twice of the size of the original object.

Data tables implement a number of pass-by-reference functions. In pass-by-reference, the function is not given a fresh copy of the inputs, but is instead told where is the object in memory. Instead of copying gigabytes of data, only a single tiny memory pointer is passed. But this also means the function now is accessing and modifying the original object, not a copy of it. This may sometimes lead to bugs and unexpected behavior, but professional use of pass-by-reference approach may improve the speed and lower the memory footprint substantially.

C.2 Data Tables: Introduction

Data tables and most of the related goodies live in *data.table* library, so you either have to load the library or specify the namespace when using the functions.

C.2.1 Replacement for Data Frames (Sort of)

Data tables are designed to be largely a replacement to data frames. The syntax is similar and they are largely replaceable. For instance, we can create and play with a data table as

```
library(data.table)
```

```
## Warning: package 'data.table' was built under R version 3.4.2
```

```
dt <- data.table(id=1:5, x=rnorm(5), y=runif(5))
dt
```

```
##    id          x         y
## 1:  1 -0.9736743 0.2050983
## 2:  2  1.3961609 0.1886502
## 3:  3  0.5901289 0.5504492
## 4:  4  0.1529862 0.2736286
## 5:  5 -0.1880280 0.6114010
```

The result looks almost identical to a similar data frame (the only difference are the colons after the row numbers). Behind the scenes these objects are almost identical too–both objects are lists of vectors. This structural similarity allows to use data tables as drop-in replacements for dataframes, at least in some circumstances. For instance, we can extract variables with `$`:

```
dt$x
```

```
## [1] -0.9736743  1.3961609  0.5901289  0.1529862 -0.1880280
```

or rows with row indices:

```
dt[c(2,4),]
```

```
##    id         x         y
## 1:  2 1.3961609 0.1886502
## 2:  4 0.1529862 0.2736286
```

However, data tables use unquoted variables names (like *dplyr*) by default:

```
dt[,x]
```

```
## [1] -0.9736743  1.3961609  0.5901289  0.1529862 -0.1880280
```

In case we need to store the variable name into another variable, with have to use the additional argument `with`:

```
var <- "x"
dt[, var, with=FALSE]
```

```
##             x
## 1: -0.9736743
## 2:  1.3961609
## 3:  0.5901289
## 4:  0.1529862
## 5: -0.1880280
```

Note also that instead of getting a vector, now we get a data.table with a single column "x" in the first. This behavior is the main culprit that when replacing data frames with data tables one may need to change quite a bit of code.

C.2.2 Fast Reading and Writing

Many data frame users may appreciate the fact that the data input-output function `fread` and `fwrite` run at least a magnitude faster on large files. These are largely replacement for `read.table` and `write.table`, however they syntax differs noticeably in places. In particular, `fread` accepts either a file name, http-url, or a *shell command that prints output*; it automatically detects the column separator, but it does not automatically open compressed files. The latter is not a big deal when using unix where one can just issue

```
data <- fread("bzcat data.csv.bz2")
```

However, the decompression is not that simple on windows and hence it is hard to write platform-independent code that opens compressed files.[1]

If your computer has enough memory and speed is not an issue, your interest for data tables may end here. You can just transform data table into a data frame with `setDF` (and the other way around with `setDT`). Let's transform our data table to data frame:

```
setDF(dt)
dt
```

```
##   id          x         y
## 1  1 -0.9736743 0.2050983
## 2  2  1.3961609 0.1886502
## 3  3  0.5901289 0.5504492
## 4  4  0.1529862 0.2736286
## 5  5 -0.1880280 0.6114010
```

Do you see that the colons after row names are gone? This means `dt` now is a data frame.

[1] Automatic decompression is a feature request for data tables

Note that this function behaves very differently from what we have learned earlier: it modifies the object *in place* (by reference). We do not have to assign the result into a new variable using a construct like `df <- setDF(dt)` (but we still can write like this, handy when using magrittr pipes). This is a manifestation of the power of data.tables: the object is not copied but the same object is modified in memory instead. `setDF` and `setDT` are very efficient, even huge tables are converted instantly with virtually no need for any additional memory.

However, big powers come hand-in-hand with big responsibility: it is easy to forget that `setDF` modifies the function argument.

C.3 Indexing: The Major Powerhorse of Data Tables

Data tables' indexing is much more powerful than that of data frames. The single-bracket indexing is a powerful (albeit confusing) set of functions. It's general syntax is as follows:

```
dt[i, j, by]
```

where `i` specifies what to do with rows (for instance, select certain rows), `j` tells what to do with columns (such as select columns, compute new columns, aggregate columns), and `by` contains the grouping variables.

Let's demonstrate this with the *flights* data from *nycflights13* package. We load the data and transform it into data.table:

```
data(flights, package="nycflights13")
setDT(flights)
head(flights)
```

```
##    year month day dep_time sched_dep_time dep_delay arr_time
## 1: 2013     1   1      517            515         2      830
## 2: 2013     1   1      533            529         4      850
## 3: 2013     1   1      542            540         2      923
## 4: 2013     1   1      544            545        -1     1004
## 5: 2013     1   1      554            600        -6      812
## 6: 2013     1   1      554            558        -4      740
##    sched_arr_time arr_delay carrier flight tailnum origin dest air_time
## 1:            819        11      UA   1545  N14228    EWR  IAH      227
## 2:            830        20      UA   1714  N24211    LGA  IAH      227
## 3:            850        33      AA   1141  N619AA    JFK  MIA      160
## 4:           1022       -18      B6    725  N804JB    JFK  BQN      183
## 5:            837       -25      DL    461  N668DN    LGA  ATL      116
## 6:            728        12      UA   1696  N39463    EWR  ORD      150
##    distance hour minute           time_hour
## 1:     1400    5     15 2013-01-01 05:00:00
## 2:     1416    5     29 2013-01-01 05:00:00
## 3:     1089    5     40 2013-01-01 05:00:00
## 4:     1576    5     45 2013-01-01 05:00:00
## 5:      762    6      0 2013-01-01 06:00:00
## 6:      719    5     58 2013-01-01 05:00:00
```

C.3.1 i: Select Observations

Obviously, we can always just tell which observations we want:

```
flights[c(1:3),]
```

```
##    year month day dep_time sched_dep_time dep_delay arr_time
## 1: 2013     1   1      517            515         2      830
## 2: 2013     1   1      533            529         4      850
## 3: 2013     1   1      542            540         2      923
##    sched_arr_time arr_delay carrier flight tailnum origin dest air_time
## 1:            819        11      UA   1545  N14228    EWR  IAH      227
## 2:            830        20      UA   1714  N24211    LGA  IAH      227
## 3:            850        33      AA   1141  N619AA    JFK  MIA      160
##    distance hour minute           time_hour
## 1:     1400    5     15 2013-01-01 05:00:00
## 2:     1416    5     29 2013-01-01 05:00:00
## 3:     1089    5     40 2013-01-01 05:00:00
```

picks the first three lines from the data. Maybe more interestingly, we can use the special variable `.N` (the number of rows), to get the penultimate row:

```
flights[.N-1,]
```

```
##    year month day dep_time sched_dep_time dep_delay arr_time
## 1: 2013     9  30       NA           1159        NA       NA
##    sched_arr_time arr_delay carrier flight tailnum origin dest air_time
## 1:           1344        NA      MQ   3572  N511MQ    LGA  CLE       NA
##    distance hour minute           time_hour
## 1:      419   11     59 2013-09-30 11:00:00
```

We can select observations with logical index vector in the same way as in data frames:

```
head(flights[origin == "EWR" & dest == "SEA",], 3)
```

```
##    year month day dep_time sched_dep_time dep_delay arr_time
## 1: 2013     1   1      724            725        -1     1020
## 2: 2013     1   1      857            851         6     1157
## 3: 2013     1   1     1418           1419        -1     1726
##    sched_arr_time arr_delay carrier flight tailnum origin dest air_time
## 1:           1030       -10      AS     11  N594AS    EWR  SEA       338
## 2:           1222       -25      UA   1670  N45440    EWR  SEA       343
## 3:           1732        -6      UA     16  N37464    EWR  SEA       348
##    distance hour minute           time_hour
## 1:     2402    7     25 2013-01-01 07:00:00
## 2:     2402    8     51 2013-01-01 08:00:00
## 3:     2402   14     19 2013-01-01 14:00:00
```

will create a new data table including only flights from Newark to Seattle. However, note that we just use `origin`, and not `flights$origin` as were the case with data frames. Data tables evaluate the arguments as if inside `with`-function.

The first, integer indexing corresponds to dplyr's `slice` function while the other one is equivalent to `filter`.

C.3.2 j: Work with Columns

`j` is perhaps the most powerful (and most confusing) of all arguments for data table indexing. It allows both to select and do more complex tasks. Lets start with selection:

```
head(flights[, dest], 3)
```

```
## [1] "IAH" "IAH" "MIA"
```

selects only the `dest` variable from the data. Note this results in a vector, not in a single-variable data table. If you want to get that, you can do

```
head(flights[, .(dest)], 3)
```

```
##    dest
## 1:  IAH
## 2:  IAH
## 3:  MIA
```

`.()` is just an alias for `list()`, encoded differently in data tables to improve readability and make it easier to type. If we want to select more that one variable, we can use the latter syntax:

```
head(flights[, .(origin, dest)], 3)
```

```
##    origin dest
## 1:    EWR  IAH
## 2:    LGA  IAH
## 3:    JFK  MIA
```

Selection supports a number of goodies, such as ranges of variables with `:` (for instance, `dep_time:arr_delay`) and excluding variables with `!` or `-` (for instance, `-year`).

Obviously we can combine both `i` and `j`: let's select origin and departure delay for flights to Seattle:

```
head(flights[dest == "SEA", .(origin, dep_delay)], 3)
```

```
##    origin dep_delay
## 1:    EWR        -1
## 2:    JFK        13
## 3:    EWR         6
```

The example so far broadly correspond to dplyr's `select`.

But `j` is not just for selecting. It is also for computing. Let's find the mean arrival delay for flights to Seattle:

```
flights[dest == "SEA", mean(arr_delay, na.rm=TRUE)]
```

```
## [1] -1.099099
```

Several variables can be returned by wrapping, and optionally named, these in `.()`. For instance, find the average departure and arrival delay for all flights to Seattle, given the flight was delayed on arrival, and name these `dep` and `arr`:

```
flights[dest == "SEA" & arr_delay > 0,
        .(dep = mean(dep_delay, na.rm=TRUE), arr = mean(arr_delay, na.rm=TRUE))]
```

```
##         dep      arr
## 1: 33.98266 39.79984
```

The result is a data table with two variables.

We can use the special variable `.N` to count the rows:

```
flights[dest == "SEA" & arr_delay > 0, .N]
```

```
## [1] 1269
```

will tell us how many flights to Seattle were delayed at arrival.

Handling the case where the variable names are stored in other variables is not that hard, but still adds a layer of complexity. We can specify the variables in `.SDcols` parameter. This parameter determines which columns go into `.SD` (=Subset Data) special variable. Afterwards we make an `lapply` expression in `j`:

```
flights[dest == "SEA" & arr_delay > 0,
        lapply(.SD, function(x) mean(x, na.rm=TRUE)),
        .SDcols = c("arr_delay", "dep_delay")]
```

```
##    arr_delay dep_delay
## 1:  39.79984  33.98266
```

Let's repeat: `.SDcols` determines which variables will go into the special `.SD` list (default: all). `lapply` in `j` computes mean values of each of the variables in the `.SD` list. This procedure feels complex, although it is internally optimized.

These examples correspond to dplyr's `aggregate` function. One can argue, however, that data tables' syntax is more confusing and harder to read. Note also that the functionality data tables offer here is optimized for speed and memory efficiency but still return a new object. Aggregation does not work by reference.

C.3.3 Group in `by`

Finally, all of the above can by computed by groups using `by`. Let's compute the average delays above by carrier and origin:

```
flights[dest == "SEA" & arr_delay > 0,
        .(dep = mean(dep_delay, na.rm=TRUE), arr = mean(arr_delay, na.rm=TRUE)),
        by = .(carrier, origin)]
```

```
##    carrier origin      dep      arr
## 1:      DL    JFK 29.82373 39.49831
## 2:      B6    JFK 28.49767 40.32093
## 3:      UA    EWR 40.24053 41.85078
## 4:      AS    EWR 31.80952 34.36508
## 5:      AA    JFK 34.04132 40.48760
```

We just had to specify the `by` argument that lists the grouping variables. If more than one, these should be wrapped in a list with `.()` function.

We can use the `.N` variable to get the group size. How many flights did each carrier from each origin?

```
flights[, .N, by=.(carrier, origin)] %>%
   head(3)
```

```
##    carrier origin     N
## 1:      UA    EWR 46087
## 2:      UA    LGA  8044
## 3:      AA    JFK 13783
```

Finally, we can also use quoted variables for grouping too just be replacing `.()` with `c()`:

```
flights[, .N, by=c("carrier", "origin")] %>%
   head(3)
```

```
##    carrier origin     N
## 1:      UA    EWR 46087
## 2:      UA    LGA  8044
## 3:      AA    JFK 13783
```

In dplyr context, the examples here include `group_by` and `summarize` verbs.

Read more about the basic usage in data.table the vignette Data analysis using data.table.

C.4 :=–Create variables by reference

While summarizing we compute values in `j`, these will always create a new data.table. Reducing operations are not possible to do in-place. But computing new variables can be done in-place.

In place variable computations (without summarizing) can be done with `:=` assignment operator in `j`. Let's compute a new variable–speed–for each flight. We can do this as follows:

```
flights[, speed := distance/(air_time/60)]
flights %>% head(3)
```

```
##    year month day dep_time sched_dep_time dep_delay arr_time
## 1: 2013     1   1      517            515         2      830
## 2: 2013     1   1      533            529         4      850
## 3: 2013     1   1      542            540         2      923
##    sched_arr_time arr_delay carrier flight tailnum origin dest air_time
## 1:            819        11      UA   1545  N14228    EWR  IAH      227
## 2:            830        20      UA   1714  N24211    LGA  IAH      227
## 3:            850        33      AA   1141  N619AA    JFK  MIA      160
##    distance hour minute           time_hour    speed
## 1:     1400    5     15 2013-01-01 05:00:00 370.0441
## 2:     1416    5     29 2013-01-01 05:00:00 374.2731
## 3:     1089    5     40 2013-01-01 05:00:00 408.3750
```

We see the new variable, speed, included as the last variable in the data. Note we did this operation *by reference*, i.e. we did not assign the result to a new data table. The existing table was modified in place.

The same assignment operator also permits us to remove variables by setting these to `NULL`. Let's remove speed:

```
flights[, speed := NULL]
flights %>% head(3)
```

```
##    year month day dep_time sched_dep_time dep_delay arr_time
## 1: 2013     1   1      517            515         2      830
## 2: 2013     1   1      533            529         4      850
## 3: 2013     1   1      542            540         2      923
##    sched_arr_time arr_delay carrier flight tailnum origin dest air_time
## 1:            819        11      UA   1545  N14228    EWR  IAH      227
## 2:            830        20      UA   1714  N24211    LGA  IAH      227
## 3:            850        33      AA   1141  N619AA    JFK  MIA      160
##    distance hour minute           time_hour
## 1:     1400    5     15 2013-01-01 05:00:00
## 2:     1416    5     29 2013-01-01 05:00:00
## 3:     1089    5     40 2013-01-01 05:00:00
```

Indeed, there is not speed any more.

Assigning more that one variable by reference may feel somewhat more intimidating:

```
flights[, c("speed", "meanDelay") := .(distance/(air_time/60), (arr_delay + dep_dela
flights %>% head(3)
```

```
##    year month day dep_time sched_dep_time dep_delay arr_time
## 1: 2013     1   1      517            515         2      830
## 2: 2013     1   1      533            529         4      850
## 3: 2013     1   1      542            540         2      923
##    sched_arr_time arr_delay carrier flight tailnum origin dest air_time
## 1:            819        11      UA   1545  N14228    EWR  IAH      227
## 2:            830        20      UA   1714  N24211    LGA  IAH      227
## 3:            850        33      AA   1141  N619AA    JFK  MIA      160
##    distance hour minute           time_hour    speed meanDelay
## 1:     1400    5     15 2013-01-01 05:00:00 370.0441       6.5
## 2:     1416    5     29 2013-01-01 05:00:00 374.2731      12.0
## 3:     1089    5     40 2013-01-01 05:00:00 408.3750      17.5
```

Assignment works together with both selection and grouping. For instance, we may want to replace negative delay by zeros:

```
flights[ arr_delay < 0, arr_delay := 0][, arr_delay] %>%
   head(20)
```

```
##  [1] 11 20 33  0  0 12 19  0  0  8  0  0  7  0 31  0  0  0 12  0
```

Indeed, we only see positive numbers and zeros. But be careful: now we have overwritten the `arr_delay` in the original data. We cannot restore the previous state any more without re-loading the dataset.

As an example of groupings, let's compute the maximum departure delay by origin:

```
flights[, maxDelay := max(dep_delay, na.rm=TRUE), by=origin] %>%
   head(4)
```

```
##    year month day dep_time sched_dep_time dep_delay arr_time
## 1: 2013     1   1      517            515         2      830
## 2: 2013     1   1      533            529         4      850
## 3: 2013     1   1      542            540         2      923
## 4: 2013     1   1      544            545        -1     1004
##    sched_arr_time arr_delay carrier flight tailnum origin dest air_time
## 1:            819        11      UA   1545  N14228    EWR  IAH      227
## 2:            830        20      UA   1714  N24211    LGA  IAH      227
## 3:            850        33      AA   1141  N619AA    JFK  MIA      160
## 4:           1022         0      B6    725  N804JB    JFK  BQN      183
##    distance hour minute           time_hour    speed meanDelay maxDelay
## 1:     1400    5     15 2013-01-01 05:00:00 370.0441       6.5     1126
## 2:     1416    5     29 2013-01-01 05:00:00 374.2731      12.0      911
## 3:     1089    5     40 2013-01-01 05:00:00 408.3750      17.5     1301
## 4:     1576    5     45 2013-01-01 05:00:00 516.7213      -9.5     1301
```

We can see that `by` caused the delay to be computed for each group, however, the data is not summarized, just the max delay is added to every single row.

Finally, if you *do not* want to modify the original data, you should use `copy` function. This makes a deep copy of the data, and you can modify the copy afterwards:

```
fl <- copy(flights)
fl <- fl[, .(origin, dest)]
head(fl, 3)
```

```
##    origin dest
## 1:    EWR  IAH
## 2:    LGA  IAH
## 3:    JFK  MIA
```

```
head(flights, 3)
```

```
##    year month day dep_time sched_dep_time dep_delay arr_time
## 1: 2013     1   1      517            515         2      830
## 2: 2013     1   1      533            529         4      850
## 3: 2013     1   1      542            540         2      923
##    sched_arr_time arr_delay carrier flight tailnum origin dest air_time
## 1:            819        11      UA   1545  N14228    EWR  IAH      227
## 2:            830        20      UA   1714  N24211    LGA  IAH      227
## 3:            850        33      AA   1141  N619AA    JFK  MIA      160
##    distance hour minute           time_hour    speed meanDelay maxDelay
## 1:     1400    5     15 2013-01-01 05:00:00 370.0441       6.5     1126
## 2:     1416    5     29 2013-01-01 05:00:00 374.2731      12.0      911
## 3:     1089    5     40 2013-01-01 05:00:00 408.3750      17.5     1301
```

As you see, the `flights` data has not changed.

These operations correspond to the dplyr's `mutate` verb. However, `mutate` always makes a copy of the original dataset, something that may well make your analysis slow and sluggish with large data.

Read more in vignette Data.table reference semantics

C.5 keys

Data tables allow fast lookup based on *key*. In it's simplest version, a key is a column (or several columns) which is used to pre-sort the data table. Pre-sorting makes it much faster to look up certain values, perform grouping operations and merges. As data can only be sorted according to one rule at a time, there can only be one key on data.table (but a key may include several variables).

Let's set origin and destination as keys for the data table:

```
data(flights, pkg="nycflights13")
```

```
## Warning in data(flights, pkg = "nycflights13"): data set 'flights' not
## found

## Warning in data(flights, pkg = "nycflights13"): data set 'nycflights13' not
## found
```

```
setDT(flights, key=c("origin", "dest"))
fl <- flights[,.(origin, dest, arr_delay)]
                           # focus on a few variables only
head(fl, 3)
```

```
##    origin dest arr_delay
## 1:    EWR  ALB         0
## 2:    EWR  ALB        40
## 3:    EWR  ALB        44
```

We see that both origin and destination are alphabetically ordered. Note that when selecting variables, the resulting data table `fl` will have the same keys as the original one.

When set, we can easily subset by key by just feeding the key values in `i`:

```
fl["LGA"] %>%
   head(5)
```

```
##    origin dest arr_delay
## 1:    LGA  ATL         0
## 2:    LGA  ATL        12
## 3:    LGA  ATL         5
## 4:    LGA  ATL         0
## 5:    LGA  ATL        17
```

will extract all LaGuardia-originating flights. In terms of output, this is equivalent to `fl[origin == "LGA"]`, just much more efficient. When you want to extract flights based on origin-destination pair, you can just add both key columns:

```
fl[.("EWR", "SEA")] %>%
   head(4)
```

```
##    origin dest arr_delay
## 1:    EWR  SEA         0
## 2:    EWR  SEA         0
## 3:    EWR  SEA         0
## 4:    EWR  SEA         0
```

Again, this can be achieved in other ways, just keys are more efficient. Finally, if we want to extract based on the second key, the syntax is more confusing:

```
fl[.(unique(origin), "SEA")] %>%
   head(4)
```

```
##    origin dest arr_delay
## 1:    EWR  SEA         0
## 2:    EWR  SEA         0
## 3:    EWR  SEA         0
## 4:    EWR  SEA         0
```

We have to tell the `[` that we want to extract all observations where the first key is everything, and the second one is “SEA”.

Read more in the vignette Keys and fast binary search based subset.

C.6 Resources

- Data Table Cheatsheet
- Data Table CRAN page. Vignettes are a very valuable source of information.

Appendix D

Using Remote Server

Sooner-or-later you are in a situation where you have to work on a **distant networked computer**. There are many reasons for this, either your laptop is to weak for certain tasks, or certain data is not allowed to be taken out from where it is, or you are expected to use the same computer as your teammates. The server may be a standalone box located in a rack in your employer's server room, or it may be a virtual machine in a cloud like Amazon EC2. You may also want to set up your own server.

D.1 Server Setup

There are many ways one can set up a distant machine. It may be Windows or linux (or any of the other unixes). It may or may not have graphical user interface (GUI) installed or otherwise accessible (many unix programs can display nice windows on your laptop while still running on the server). It may or may not have RStudio made available over web browser. Here we discuss the most barebone setup with no access to GUI and no web access to RStudio.

This is a fairly common setup, for instance when dealing with sensitive data, in organizations where computer skills and sysadmin's time is limited, or when you rent your own tiny but cheap server.

D.2 Connecting to the Remote Server

Given the server is already running, your first task is to connect to it. Here it means that you will enter commands on your laptop, but those command are actually run on the server.

The most common way to connect to remote server is via *ssh*. ssh stands for "secure shell" and means that both all network communication is encrypted. You connect to the server as

```
ssh myserver.somewhere.com
```

The remote server asks for your password and opens remote shell connection. Note: when entering your password, it usually does not print anything in response, not even asterisks. It feels as if your keyboard is not working. It will offer you a similar bash shell environment as you are using on your computer but most likely you see a different prompt, one that contains the server's name. You may also see some login messages. Now all the commands you are issuing are running on the remote machine. So `pwd` shows your working directory on the server, which in general is not the same as on the local machine, and `ls` shows the files on the server, not on your laptop. Now you can use `mkdir` to create the project folder on the server.

By default, ssh attempts to login with your local username. If your username on the server differs from that on your laptop, you want to add it to the ssh command:

```
ssh username@myserver.somewhere.com
```

Finally, when done, you want to get out. The polite way to close the connection is with command

```
exit
```

that waits until all open connections are safely closed. But usually you can as well just close the terminal.

D.3 Copying Files

Before you can run your R scripts on the server, you have to get these copied over. There are several possibilities.

D.3.1 scp

The most straightforward approach is `scp`, **s**ecure **c**o**p**y. It works in many ways in the same way as `cp` for the local files, just `scp` can copy files between your machine and a remote computer. Under the hood it uses ssh connection, just like `ssh` command itself. It syntax is rather similar to that of `cp`:

```
scp user1@host1:file1 user2@host2:file2
```

This copies "file1" from the server "host1" under username "user1" to the other server. Passwords are asked for as needed. The "host" part of the file must be understood as the full hostname including dots, such as “hyak.washington.edu”. “file” is the full path to file, relative to home directory, such as Desktop/info201/myscript.R. When accessing local files, you may omit the ”user@host:” part. So, for instance, in order to copy your myscript.R from folder info201 on your laptop’s Desktop to the folder scripts in your home folder on the server, you may issue

```
scp Desktop/info201/myscript.R myusername@server.ischool.edu:scripts/
```

(here we assume that the working directory of your laptop is the one above Desktop.) Note that exactly as with cp, you may omit the destination file name if the destination is a directory: it simply copies the file into that directory while preserving its name.

After running your script, you may want to copy your results back to your laptop. For instance, if you need to get the file figure.png out of the server, you can do

```
scp myusername@server.ischool.edu:scripts/figure.png Desktop/info201/
```

As above, this copies a file from the given directory, and drops it into the info201 folder on your Desktop.

D.3.2 rsync

rsync is a more advanced approach to scp. It works in many ways like scp, just it is smart enough to understand which files are updated, and copy the updated parts of the files only. It is the recommended way for working with small updates in large files. Its syntax is rather similar to that of scp. To copy file to the remote server as file2 (in the home directory), we do

```
rsync file user2@host2:file2
```

and in order to copy a file1 from server as local file (in the current working directory):

```
rsync file user1@host1:file1 file
```

I also recommend to explore some of its many options, for instance -v (verbose) reports what it’s doing. The example above with your code and figure might now look like that:

```
rsync -v Desktop/info201/myscript.R myusername@server.ischool.edu:scripts/
# now run the script on the remote machine
rsync -v myusername@server.ischool.edu:scripts/figure.pdf Desktop/info201/
```

Maybe the easiest way to copy your files is to copy (or rather update) the whole directories. For instance, instead of the code above, you can do

```
# copy all files to server:
rsync -v Desktop/info201/* myusername@server.ischool.edu:scripts/
# now run the script on the remote machine
# ... and copy the results back:
rsync -v myusername@server.ischool.edu:scripts/* Desktop/info201/
```

Here * means *all files in this directory*. Hence, instead of copying the files individually between the computers, we just copy all of them. Even better, we actually do not copy but just update. Huge files that do not change do not take any bandwidth.

D.3.3 Graphical Frontends

Instead on relying on command line tools, one can also use graphical front-ends. For instance, "WinSCP" is a nice Norton Commander-Style frontend for copying files between the local and a remote machine over scp for Windows. It provides a split window representing files on the local and the remote end, and one can move, copy-and-paste and interact with the mouse on these panes. On Mac you may take a look at "Cyberduck".

D.3.4 Remote Editing

Besides copying your files, many text editors also offer a "remote editing" option. From the user perspective this looks as if directly working on the remote server's hard disk. Under the hood, the files are copied back and forth with scp, rsync or one of their friends. Emacs and vi do it out-of-the box, VSCode, Atom and sublime require a plugin. AFAIK it is not possible with RStudio.

It is also possible to mount (attach) the harddisk of the remote server to your laptop as if it were a local disk. Look yourself for more information if you are interested.

D.4 R and Rscript

When your code has been transferred to the server, your next task is to run it. But before you can do it, you may want to install the packages you need. For instance, you may want to install the whole "tidyverse" bundle. This must be done from R console using `install.packages()`. You start R interactively by the command

```
R
```

It opens an R session, not unlike what you see inside of RStudio, just here you have no RStudio to handrail you through the session. Now all loading, saving, inspecting files, etc must be done through R commands.

The first time you do it, R complains about non-writeable system-wide library and proposes to install and create your personal libary. You should answer "yes" to these prompts. As Linux systems typically compile the packages during installations, installation is slow and you see many messages (including warnings) in the process. But it works, given that the necessary system libraries are available.

Now you can finally run your R code. I strongly recommend to do it from the directory where you intend to run the project before starting R (`cd scripts` if you follow the example directory setup above). There are two options: either start R interactively, or run it as a script. If you do it from an interactive R session, you have to *source* your script:

```
source("myscript.R")
```

The script will run, and the first attempt most likely ends with an error message. You have to correct the error either on your laptop and copy the file over to the server again, or directly on the server, and re-run it again. Note that you don't have to exit from the R session when copying the files between your laptop and the server. Edit it, copy it over from your laptop (using `scp` or other tools), and just re-source the file from within the R session.

Opening a separate R session may be useful for installing packages. For running your scripts, I recommend you to run it entirely from command line, either as

```
R CMD BATCH myscript.R
```

or

```
Rscript myscript.R
```

The first version produces a little more informative error messages, the other one handles the environment in a little more consistent and efficient manner.

D.4.1 Graphics Output with No GUI

If the server does not have any graphics capabilities, you have to save your figures as files. For instance, to save the image in a pdf file, you may use the following code in your R program:

```
pdf(file="figure1.pdf", width=12, height=8)
    # width and height in inches
    # check also out jpeg() and png() devices.
# do your plotting here
plot(1:10, rnorm(10))
```

```
# done plotting
dev.off()
    # saves the image to disk and closes the file.
```

Afterwards you will have to copy the image file to your laptop for inspection, or addition in your final project report.

D.5 Life on Server

The servers operate the same in many ways as the command line on your own computer. However, there are a number of differences.

D.5.1 Be Social!

While you laptop is yours, and you are free to exploit all its resources for your own good, this is not true for the server. The server is a multiuser system, potentially doing good work for many people at the same time. So the first rule is: **Don't take more resources than what you need!**

This that means don't let the system run, grab memory, or occupy disk space just for fun. Try to keep your R workspace clean (check out `rm()` function) and close R as soon as it has finished (this happens automatically if you run your script through `Rscript` from command line). Don't copy the dataset without a good reason, and keep your copies in a compressed form. R can open gzip and bzip2 files on the fly, so usually you don't even need to decompress these. Avoid costly recalculations of something you already calculated. All this is even more important the last days before the deadline when many people are running using the server.

Servers are typically well configured to tame misbehaving programs. You may sometimes see your script stopping with a message "killed". This most likely means that it occupied too much memory, and the system just killed it. Deal with this.

D.5.2 Useful Things to Do

There are several useful commands you can experiment with while on the server.

```
htop
```

(press `q` to quit) tells you which programs run on the server, how much memory and cpu do these take, and who are their owners (the corresponding users). It also permits you to kill your misbehaving processes (press `k` and select `SIGKILL`). Read more with `man htop`.

```
w
```

(**w**ho) prints the current logged-in users of the server.

```
df -h
```

(**d**isplay **f**ree in **h**uman-readable units) shows the free and occupied disk space. You are mainly influenced by what is going on in the file system `/home`.

D.5.3 Permissions and ownership

Unix systems are very strict about ownership and permissions. You are a normal user with limited privileges. In particular, you cannot modify or delete files that you don't own. In a similar fashion, you cannot kill processes you did not start. Feel free to attempt. It won't work.

In case you need to do something with elevated privileges (as "superuser"), you have to contact the system administrator. In practice, their responsiveness and willingness to accommodate your requests will vary.

D.5.4 More than One Connection

It perfectly possible to log onto the server through multiple terminals at the same time. You just open several terminals and log onto the server from each of these. You can use one terminal to observe how your script is doing (with `htop`), the other one to run the script, and the third one to inspect output. If you find such approach useful, I recommend you to familiarize yourself with gnu screen (command `screen` that includes many related goodies.)

D.6 Advanced Usage

D.6.1 ssh keys, .ssh/config

Without further configuration, every time you open a ssh connection, you have to type your password. Instead of re-entering it over and over again (this may not be particularly secure), you can configure your ssh keys and copy it to the server. Next time, you will be automatically authenticated with the key and you don't have to type the password any more. Note: this is the same ssh key that is used by GitHub if you use ssh connection to GitHub.

As the first step, you have to create the key with `ssh-keygen` (you may choose an empty passphrase) unless you already have created one. Thereafter copy your public key to the server with `ssh-copy-id`. Next time you log onto the server, no password is needed. A good source for help with creating and managing ssh keys is GitHub help.

You can also configure your ssh to recognize abbreviated server names and your corresponding user names. This allows you to connect to server with a simple command like `ssh info201`. This information is stored in the file `~/.ssh/config`, and should contain lines like

```
Host info201
    User <your username>
    Hostname info201.ischool.uw.edu
```

The `Host` keyword is followed by the abbreviated name of the server, the following lines contain your username and the publicly visible hostname for the server. Seek out more information if you are interested.

D.6.2 Running RScript in ssh Session

Passwordless ssh connection gives you new wonderful possibilities. First, you don't even have to log into the server explicitly. You can run a one-command ssh session on the server directly from your laptop. Namely, ssh accepts commands to be run on the remote machine. If invoked as

```
ssh myusername@server.ischool.edu "Rscript myscript.R"
```

It does not open a remote shell but runs `Rscript script.R` instead. Your command sequence for the whole process will accordingly look something like:

```
rsync -v Desktop/info201/* myusername@server.ischool.edu:scripts/
ssh myusername@server.ischool.edu "Rscript scripts/myscript.R"
rsync -v myusername@server.ischool.edu:scripts/* Desktop/info201/
```

All these command are issued on your laptop. You can also save these to a text file and run all three together as a single **shell script**!

Further, you don't even need the shell. Instead, you may explain R on your laptop how to start R on the remote server over ssh. In this way you can turn your laptop and server combination into a high-performance-computing cluster! This allows you to copy the script and run it on the server directly from within your R program that runs on your laptop. Cluster computing is out of scope of this chapter, but if you are interested, look up the **makePSOCKcluster()** function in **parallel** package.