The Statistical Grimoire:
Statistics for the Natural Sciences Using R

Dr. Jeffrey M. Pisklak

University of Alberta

Frontispiece image: Waterhouse, J. M. (1886). *The Magic Circle* [Painting]. The Tate Gallery - London, England

Preface

Since prefaces often go unread, I shall keep this brief. This text was born out of a need to offer my students a robust, open-access introduction to R, specifically for those without any programming experience. Long term it is intended to serve as a practical, open-access manual, guiding complete beginners through both statistics and statistical programming in a thorough and clear a manner. Please consider everything herein a work in progress.

Contents

Chapter 1

An Introduction to R

To begin with ...

1.1 What the f**k is R?[1]

is a **programming language**. A programming language is simply a language humans use to give instructions to computers. Humans are featherless bipedal primates called *Homo sapiens*. Computers are machines that follow fixed rules, with no authority to deviate from those rules (Turing, 1950). Contrary to the belief of many, what constitutes a human and what constitutes a computer are not mutually exclusive. In the past, computation was a biological endeavor, with humans relying on other humans to carry out the laborious task of performing countless mathematical calculations. These human computers required no strict programming language, *per se*, but instead relied on the grunts and scribbles of primates in the upper tiers of their hierarchy to dictate their tasks. In the modern day, human computers are largely a relic of the past. A product of a bygone era where mathematics and other concepts foreign to modern students, such as reading and writing, were commonplace within school curricula.

Today the electronic digital computer reigns supreme. This is a machine, largely silicone based, and similar in many respects to its primate precursor - with the notable exception that it is considerably more logical (some even go so far as to call these devices "smart" - which perhaps says more about the users than the devices themselves). It is therefore no longer deemed necessary to subject children to the cruelties associated with a well rounded education, particularly in the domain of mathematics, where the largest amounts physical and psychological turmoil were often inflicted.

The name "R" was derived from the first initials of its original two programmers, **R**obert Gentleman and **R**oss Ihaka. The decision to name the language using a single English letter is what we might, charitably, call a joke on the part of these two programmers, who saw themselves poking fun at R's parent language, which was given the unimaginative name of "S". In the 1970s the S language had undergone its initial development at the famous Bell Laboratories with the primary aim of enabling and encouraging "GOOD DATA ANALYSIS"- a goal so fundamental to the ethos of S that the authors, Becker and Chambers (1984), felt they had to emphasize it using uppercase lettering inside the preface to the language's inaugural instruction manual (the uppercase lettering has been reproduced here for the readers benefit). The familial correspondence R has with S is present even to this day, to such an extent that Becker and Chambers (1984) original manual could probably function decently well as an introductory manual to R itself.

---

[1]It's not technically swearing if you use asterisks.

1.2 Why the f**k should I use R?

At this point readers might be wondering why it should ever be necessary to learn a programming language to conduct statistics and data analysis more generally. These topics are usually considered difficult enough by many students and educators, what need is their to compound this with a programming language? Why not, for instance, make use of any one of the many pieces of statistical software that already exist and require no requisite knowledge of any kind of programming? In other words, why not use software such as SPSS?[2]

The principle answer to this question has to do with flexibility. Data analysis is not a rigid thing. It is not the case that there is always a single correct way to do things. Different sets of data come with their own unique intricacies and problems which are often not amenable to the "cookie cutter" style of analyses the aforementioned program employs. Which is not to say that these programs are not capable of adapting to these scenarios. They usually are, but this adaptation either requires the user to pay for some additional feature they did not get in their original purchase or it demands a much higher level of expertise with the software than most users are ever likely to acquire - expecting them to learn some obscure programming language, specific to that software, that all but a privileged few actually comprehend. Along these lines, something that non-R users often do not appreciate is how easy R is to learn. At a superficial glance, R can appear intimidating but it is actually fairly intuitive and works, more or less, in the manner of a calculator. Most people who learn R find it to be a rewarding experience that was considerably more user friendly than what its paid counterparts produced (Bro, n.d.). This is thanks in large part to the extensive infrastructure of helpful online resources R users have built over the years - which proprietary equivalents have no equal to. Software like SPSS can give an air of familiarity when a user first encounters them because, superficially, they appear very similar to commonly used spreadsheet software many individuals will already be acquainted with, such as Microsoft's Excel. The user is typically provided with a hefty amount of buttons and menus at the top of the screen followed by a spreadsheet-style grid beneath it. Unfortunately, that is where the similarities end and new users inevitably find themselves overwhelmed by the plethora of bewildering options to perform what should be simple enough tasks (e.g., loading and viewing a data set). The dirty-little secret about these programs is that their learning-curve is considerably steeper than their advertising would have you believe. By contrast there is an inherent logic to R (the logic of mathematics) that new users can often easily grasp and build of off (even if you don't like math). Moreover, R is not beholden to a graphical user interface (i.e., a gui - pronounced "gooey" and often referred to as a "point and click interface."). Consequently, its functionality is only limited by what you, or other people, are able to program and what your computer is capable of handling.

An altogether different answer to the question that opened this section, and one that will appeal to the University students reading this, is simply cost. R is free for the user, with no need to put up with annoying advertising or pay for additional features. The same can not be said of the other aforementioned software which are almost always subscription based, requiring the user to consistently renew an expensive license to use the software. In fact, upon visiting the respective websites for both SPSS and another, slightly less well known, SPSS style software called Minitab, one can see that it is worryingly difficult to find any price listings whatsoever for these programs - evoking the age old wisdom that, if you have to ask the price, you probably can't afford it. But R is not just free in monetary terms, it is also free in philosophical terms. R adopts the Free Software Foundation's GNU General Public License and thus adheres to the philosophy of "free software" (what some might term "open-source"). From the GNU project website (Free Software Foundation, 2022):

A program is free software if the program's users have the four essential freedoms:

- Freedom 0: The freedom to run the program as you wish, for any purpose.

---

[2]SPSS is popular software for conducting statistics that was originally released in the late 1960s and is an acronym for *Statistical Package for the Social Sciences*. At some point it was purchased by IBM and re-branded to mean *Steeply Priced Shitty Software*.

- Freedom 1: The freedom to study how the program works, and change it so it does your computing as you wish. Access to the source code is a precondition for this.

- Freedom 2: The freedom to redistribute copies so you can help others.

- Freedom 3: The freedom to distribute copies of your modified versions to others. By doing this you can give the whole community a chance to benefit from your changes. Access to the source code is a precondition for this.

This is a philosophy that extends as much to the software's various file types and help documentation as it does to the software itself. A major barrier to the dissemination of scientific findings, for many years, has been the tendency of various proprietary tools used in research to rely on exclusive file formats to handle data. For instance, many types of paid software used for building, running, and analyzing computer-based experiments will, by default, output results using restrictive file formats that require the original software with a valid license to be read. This not only makes it difficult to provide free access of the data to other researchers, but makes it that much more difficult to load the data into other types of software (which are also likely to contain their own proprietary formatting schemes) for more in-depth analyses than what the original program affords the researcher. Obviously, in a by-gone pre-internet era, when statistical computing was both a complex and expensive endeavour, and the use of any given computer and its software was restricted to a small coterie of researchers, a company may be forgiven for utilizing proprietary formatting schemes in this manner. However, in an age where the internet is ubiquitous and there is rarely any need to ration computing power, there is little justification for such practices. A generic .CSV (comma-separated values)[3] spreadsheet file will be sufficient as an output for most research endeavors. It is clear to any reasonable person that locking information away in this manner is not done in the best interest of scientific progress; rather, it is done simply to tether researchers to a branded ecosystem of overpriced software. Consequently, we can easily label the continued adoption of these practices for what they are: unscrupulous. In practical terms, what this means is that, if for no other reason, we should use R just to give the middle finger to these companies.

1.3　Installing and Running R on Your Computer

1.3.1　Languages and Environments

R will install and run straightforwardly on Windows and Macintosh operating systems as well as Linux; however, prior to attempting any install it is important to make a simple distinction first. R is a programming language, which means it is nothing more than a language you can use to communicate instructions to your computer. To communicate those instructions, some type of interface is required. This is a basic reality that applies to any language. It is quite difficult to communicate with someone if they have no mouth, eyes, or ears to send and receive communications with. Computers are no different in this respect. Simply "understanding" the language is not sufficient. For this reason, most operating systems come equipped with a basic way of interfacing with the user via a **command console**[4] of some kind. On computers using the Windows operating system, this is referred to as the *Command Prompt* application, on Macintosh computers this is the *Terminal* application. Relying on the basic command console as your primary interface is often a cumbersome and inefficient experience, and definitely not a recommended course of action - unless you are a Linux user who delights in personal suffering. The preferred means of communicating R to your computer is via the use of what, in programming lingo, is commonly termed an "**environment**" or, more garrulously, an "**integrated**

---

[3]A "comma-separated values" spreadsheet is a universal file format for spreadsheets. It is readable by any competent spreadsheet software and, in fact, spreadsheet software is not even necessary to read a .CSV file - a basic text editor will suffice. Moreover, .CSV files can contain large amounts of information with comparatively small file sizes.

[4]A windowed application that allows you to type instructions (a.k.a. "commands") to your computer.

development environment (**IDE**)". This is simply a software application providing the user with a more elegant visual workspace and feature set to make programming a smoother experience.

The standard installation of R will come with an associated environment for the user - provided they are working with either a Windows or Mackintosh operating system. However, while this environment is preferable to the operating system's basic command console, most R users still find it lacking and opt to install a different environment called **RStudio**, which has an open-source (free) version for non commercial use.

1.3.2   Installation

To install R - both the language and the environment simultaneously - simply go to the *R Project for Statistical Computing* website:

https://www.r-project.org/

Somewhere on the front-page of this website should be a link labelled "**CRAN**". This stands for **Comprehensive R Archive Network** and is a set of servers around the world that distribute R alongside packages associated with R. The servers are "mirrored", meaning they all provide the same content. So there is no need to worry about one server providing incomplete, out-of-date, or unofficial versions of R. Technically speaking, the server closest to your home location is the one you should opt to download from; however, the topmost link labeled "0-Cloud" will be sufficient for most users. The install file is only about 80 megabytes large, so unless you live in all but the remotest areas of Earth, download speed, and thus choice of server, is probably not a concern.

Once you have chosen a suitable server, you will need select your operating system and choose the appropriate installation file. If you are using Windows, opt to download the "base" version of R. If you are using a Macintosh operating system, you will need to select the option relevant to your computer. At the time of writing this, Macintosh computers have recently begun being manufactured using their own in-house built processors (i.e., dubbed "Apple silicon"); however, many older Macintosh computers (pre-2023) still contain Intel-made processors. The install file you select will need to be determined by which type of processor your computer is using. Macintosh users can determine this by selecting *About This Mac* via the small little apple logo in the top left corner of the desktop screen. Machines using Apple silicon, will display a row called "Chip" and state something akin to "Apple M1". Machines using Intel processors will display a row reading "Processor" followed by the make and model of the processor.

Downloading and running the install file should prompt you with a installation wizard that walks you through the installation process. Unless you are certain you know what you are doing (which means you probably aren't reading this), you should just accept the wizard's default settings.

Upon installation of R, you can then install the aforementioned RStudio environment at

https://posit.co/products/open-source/rstudio/

Installing RStudio is not strictly necessary to work through this book's content; however, the wealth of features and customization RStudio offers does makes it a worthwhile program to install and is recommended for anyone reading this text.

1.3.3   Upgrading

There are updates made to R about 2-3 times a year and it is generally good practice to upgrade regularly. There are various methods you can use to update R, but the most straightforward method is to just download the latest version of R as though you were installing it for the first time and then re-install commonly used

packages.[5] If you follow the default setup, you do not need to uninstall the previous version. In fact, it is usually preferable not to, as RStudio allows you to easily switch between installed versions on your computer.

At the time of writing, R is on version 4.4.1, nicknamed the "Race for Your Life" version. Each new release of R is given a nickname that, inexplicably, are all obscure references to Peanuts (a.k.a. Charlie Brown and Snoopy) comic strips.

1.3.4   Consoles and Scripts

Upon opening the base R environment you will be shown a pane labelled *R Console*. Opening RStudio environment will show a similar pane simply labelled *Console* alongside a couple of others. The console pane functions as the command console described earlier (see section 1.3.1). Inside it you will see a ">". This symbol denotes the command line's prompt. In other words, it denotes the space in which you type commands, using R code, to your computer. The term "code" here is just a shorthand way of referring to "computer code" which another way of expressing the fact that we are typing commands using a programming language. The presence $>$ also indicates that the computer is awaiting your command.

If you type $1+1$ on this line and the press "enter/return" on your keyboard, you should see a 2 display as an output almost instantaneously beneath it. In this case the expression "$1+1$" is a *line of R code*. Pressing enter/return, *runs* or *executes* this R code. And "2" is the computer's resulting *output*.

Input:

```
1    1 + 1
```

Output:

```
[1] 2
```

If you close R or RStudio, you will find that any history of this calculation is gone when you re-open the environment. Consequently, typing commands into the console offers us a quick way to perform simple tasks that we are not necessarily concerned with preserving. However, in most cases we will be typing R code that we do want to preserve, run, edit, and add to at later date. This is where the concept of a **script** becomes important.

A script is simply a text document on your computer that you can use to type, run, edit, and save your R code. Using the base R environment, selecting the *File* menu at the top left corner and choosing *New Script*, will open a scripting window. In R Studio the process is *File → New File → R Script*.

Once opened, you can type R code into this new window and save it in the conventional manner of most word processing applications (i.e., *File → Save As*). For instance, if you type the following into the script window ...

```
1    1 + 1
2    2 + 2
3    3 + 3
```

You can now place your cursor at a line of your choosing and run that line individually. To do this in the base R environment you select *Edit → Run Line or Selection*. In RStudio you select *Code → Run Selected Line(s)* or click the "run" icon in the upper right of the script window. If you highlight all the lines of code, or just a subset of them, you can then run that highlighted section in a similar manner.

---

[5]Packages (also called "libraries") will be explained later.

1.3.5 Keyboard Shortcuts

It is at this juncture that a handy feature of programming environments be mentioned; specifically, keyboard shortcuts (also called "hotkeys"). All good programming environments will provide their users with the ability to do every conceivable task via their keyboard in some way. For instance, if you are using the Windows operating system, pressing the "control" key simultaneously with the "s" key will save your script file (Ctrl + S). Learning the shortcuts for frequently used features, such as selecting and running lines of code, can make the process of writing code considerably more time efficient and effortless. In theory, a good programmer - using a competently developed coding environment - should never require the use of a mouse. RStudio, in particular, offers a wide range of keyboard shortcuts that can be customized to user preferences. For instance, selecting *Help → Keyboard Shortcuts Help* will display a list of existing shortcuts that users can avail themselves of. Please note, it is not being suggested that you go out of your way to memorize all of these at once. The simple act of trying to use them consistently will be sufficient to learn them in an effortless manner. At the outset, it is to your advantage to merely select a few and attempt to use them consistently while you code. A few of the most useful ones are listed in Table 1.1[6].

	Description	Windows	Macintosh
1.	Run current line/section	Ctrl + Enter	Cmd + Return
2.	Clear Console	Ctrl + L	Ctrl + L
3.	Move to the beginning of a line	Home	Cmd + Left
4.	Move to the end of a line	End	Cmd + Right
5.	Move the cursor one word/block at a time	Ctrl + Left or Right	Option + Left or Right
6.	Highlight all	Ctrl + A	Cmd + A
7.	Highlight sections	Shift + Up, Down, Left, or Right	Shift + Up, Down, Left, or Right
8.	Move cursor to script window	Ctrl + 1	Ctrl + 1
9.	Move cursor to console window	Ctrl + 2	Ctrl + 2
10.	Type the `<-` operator	Alt + - (minus)	Option + - (minus)

Table 1.1: Useful Keyboard Shortcuts

When utilizing these shortcuts, it is worth remembering that standard QWERTY-style keyboards are symmetrically designed. Modifier keys like the *shift* key, *control* key, and *alt* key are located on both the left and right side of the board. This is not by accident. A basic skill primary education tends to neglect is that of touch typing. Consequently, most people - even those who have grown up with unprecedented access to computers and the internet - have never learned to appreciate the utility of this layout or use it appropriately. As an example, to type capital letters you should always depress the shift key on the opposite side of the keyboard to the letter. So, if you desired to type the capital letter Q, you would depress the right shift key with your right hand, and type Q with your left hand. A similar logic applies to the other modifier keys. To use keyboard shortcut #9 in Table 1.1, you would depress the right control key (with your right hand) and use your left hand to press the 2 key. You should not be trying to press both keys with a single hand. Such advice might seem obvious but, given the sheer number of people who contort their wrists and fingers in grotesquely strange and painful ways, it is clearly far from being so.

1.4 How To Code Using R: The Fundamentals

With the formalities of installation, console, and scripting window out of the way, we can now start to learn how to write (i.e. code) using the language called R. Though, it is at this juncture that some advise to novice programmers be offered. Nothing that will be discussed in this section, or any section of this text concerning R

---

[6]Shortcuts 3, 4, and 5 can be combined with shortcut 6 to highlight bigger sections of code.

code, is material you need to go out of your way to memorize. R is a language, and the basic act of trying to use the language consistently will result in a natural and effortless memorization over time. Along these lines, there are some basic recommendations novice programmers can follow to expedite this:

- Do not use your computer's copy and paste functions. Type all code yourself.

- Run all the examples in this textbook and try and produce the same results.

- If you do not know how to do some particular thing, then look up how to do it each time you need to do it.

- Stay organized - this applies to the code you write and the files you save.

- Pledge to do all your stats from this point forward using R. Immerse yourself in the language.

Everything discussed here is done so for the purpose of acquainting you with the R language so that, when you see some R code, you are not compelled into some manner of zombiesque torpor. As you move through the text, you will learn more advanced things and have much of this material repeated and re-explained. Your goal in this chapter is not to become an R expert, but rather to get an intuitive grasp of R's underlying syntax and logic.

1.4.1   Basic Arithmetic

At its core R is really nothing more than a fancy calculator, and we can use it as such. R can be used to add $(+)$, subtract $(-)$, multiply $(\times)$, and divide $(\div)$.

```
1  1 + 1
2  2 - 2
3  3 * 3
4  4 / 4
```
```
[1] 2
[1] 0
[1] 9
[1] 1
```

Exponents can be incorporated as well by using the $\wedge$ ('caret'), symbol. For instance, the expression 5^3 can be written as ...

```
1  5^3
```
```
[1] 125
```

R will also follow the *order of operations* when dealing with more complex expressions. To illustrate, consider the mathematical expression $8 \div 2(2+2)$. Some people mistakenly believe that this expression is equal to 1, some believe it is equal to 4, and others believe that it is improperly written and there is no solution. In fact, it is equal to 16. As many will no doubt have learned in their primary education, according to order of operations (BEDMAS[7]), the order in which you divide and multiply inside the equation is not fixed, sometimes you divide first and sometimes you multiply first. However, what most people never learn is that the order you use is not up to you. You must always calculate from left to right when making a choice between multiplication and division. The same rule applies to addition and subtraction.

---

[7]BEDMAS of course being the famous mnemonic to help memorize the order of operations: Brackets, Exponents, Division, Multiplication, Addition, and Subtraction. Many non-Canadian readers may be more familiar with an inferior variant of this mnemonic known as PEDMAS.

```
1  8/2*(2+2)
```
```
[1] 16
```

If we re-write the equation to be $8 \div (2+2)2$, you will see a corresponding change in the computer's output.

```
1  8/(2+2)*2
```
```
[1] 4
```

R also has the ability to perform *Euclidean Division*, which many simply know from their primary education days as *division with a remainder*. For instance, consider $11 \div 2$. Conventionally, you would want and expect an answer of 5.5, and R will produce that.

```
1  11/2
```
```
[1] 5.5
```

However, if we want to see the result expressed as a quotient and remainder (i.e., if we want to use Euclidean Division), we could obtain the quotient by typing ...

```
1  11 %/% 2
```
```
[1] 5
```

To obtain the remainder we type...

```
1  11 %% 2
```
```
[1] 1
```

Thus, 11 can be split into 2 groups of 5, with 1 left over. More technically, the `%%` is what is known as the **modulo operator** and the remainder value of `1` that results from `11 %% 2` is known as the **modulus**.

Other, more complex, arithmetic operations are available in the R language; however, most of them will require the use of specialized lines of code called *functions*, which are discussed later (see section 1.4.7).

Given that we are on the topic of basic arithmetic, it is perhaps worth considering what happens when you "break the rules" of basic arithmetic. Suppose we divide a positive and negative value by zero, what will happen?

```
1  1/0
2  -1/0
```
```
[1] Inf
[1] -Inf
```

You can see that R produces a result of `Inf` and `-Inf` which is an abbreviated way of referring to **infinity** in the positive and negative directions respectively.[8].

What happens if you take the square root of a negative number?

```
1  (-4)^(1/2)
```
```
[1] NaN
```

---

[8]This will also be generated if a number is too large for a computer to cope with. For example, the code `.Machine$double.xmax` will produce the largest number your computer can handle. Now, R will technically still let you *add* values to this number, but the number won't change from R's perspective because the amount you would have to add to alter what is shown is excessively large. However, if you *multiply* it by 2, you should get `Inf`: `.Machine$double.xmax * 2`

The abbreviation `NaN` here stands for "not a number", and is a fairly sensible output given that the square root of a negative number does not exist as a real number (consequently, it only exists in your imagination).

Finally, since its use crops up from time to time, it can be handy to know that R comes with the number π stored as a constant.[9] To use it, you need only type `pi`.

```
1  pi
```
```
[1] 3.141593
```

1.4.2   Understanding Scientific Notation

On occasion values will either be excessively large or excessively small. In such cases R will often display the values using what is referred to as **scientific notation**. For instance, dividing the number 2 by 100000 will result in scientific notation being employed:

```
1  2 / 100000
```
```
[1] 2e-05
```

Notice the `e-05` in the output. This is how you know R is presenting a number using scientific notation. To interpret this in a conventional manner, imagine there is a decimal point after the 2, like so: `2.0e-05`. Then just move that decimal point five digits to the left. In other words, `2e-05` is the same as writing `0.00002`. Mathematically, `2e-05` translates to 2×10^{-5}

If the output were showing `e+5`, then you would move the decimal five digits to the right. For example, `2e+5` is the same as writing `200000`. Notice there are five 0s; this is because, mathematically, `1e+5` means 2×10^5

Remember that positive powers move the decimal right (in the positive direction), and negative powers move the decimal left (in the negative direction).

1.4.3   Commenting Out Lines

In the course of writing R code, there will be occasions where you would like to run a script you have typed up, but not necessarily run every single line on that script. There might be certain lines that you would, at least tentatively, like to keep for one reason or another. You can accomplish this by "commenting out" your code. If you type a `#` symbol, any code that follows that symbol and is on the same line as that symbol will not be run.

```
1  1 + 1
2  # 2 + 2
3  3 + 3
```
```
[1] 2
[1] 6
```

```
1  1 + 2 + 3 # + 4 + 5
```
```
[1] 6
```

This process is phrased "commenting out" because using the `#` is also frequently employed to write *short* helpful comments to yourself and other readers about your R script.

---

[9]If you find π displayed to seven digits inadequate, you may want to talk to a professionally licensed therapist. Alternatively, you can display more digits by running the code `print(pi, digits = 16)`. Values exceeding 16 digits will be inaccurate given the limitations of 64-bit computers, so it is advisable to not go beyond 16 even though a max of 22 are possible.

1.4.4   Creating Objects

A central feature of R is its ability to call objects in memory. For instance, we can define an object name, `x`, and have that name represent a number by typing a little arrow, `<-`, and following it with a value such as 1.

```
1   x <- 1
```

You will find that running this line of code produces no corresponding output. However, if we now run `x` by itself the computer will display an output of 1

```
1   x
```
```
[1] 1
```

R is technically classified as an object-oriented programming language (OOP). This is because, if you look into how R actually stores what we have done in memory, the "object" here is the number 1. `x` is merely the name we are assigning to that object. However, a lot of R users are under the impression that the reverse is true - i.e., that we have in some sense created an object called `x` and stored something inside of it, but that is not actually the case. `x` is just a name binded to the object 1, and this object 1 is located somewhere inside your computer's memory. Admittedly, unless you are doing some seriously advanced R programming, this is a distinction that will not matter to 99.3% of R users, but it is important because it means that if you do something like this . . .

```
1   x <- 1
2   y <- 1
```

`x` and `y` are technically different objects in the computer's memory. However, if we did this ....

```
1   y <- x
```

they now represent the same object in memory. Moreover, altering one does not affect the other and just ends up creating two separate objects in memory. E.g. ...

```
1   x <- x + 1
2   x
3   y
```
```
[1] 2
[1] 1
```

To assign the names `x` and `y` we typed an arrow, `<-`. Alternatively, we could have assigned the names using an equal sign (`=`) instead.

```
1   y = x + 4
2   y
```
```
[1] 6
```

Both `<-` and `=`, in the manner we are using them here, are what are referred to as **assignment operators** in that, they are used to perform the *operation* of *assigning* a name to an object. For most use cases, there is no practical difference between the two; except insofar as the arrow can be swapped around to assign values to objects like so.

```
1   10 -> z
2   z
```
```
[1] 10
```

The existence of both `=` and `<-` as assignment operators raises an obvious question: which is better to use? This is a question for which there are strong opinions and Appendix A walks through the trivial dispute for those interested.[10]

Object Modes

Thus far all of the objects we have created have been **numeric** objects; though, we can avail ourselves of other types. For instance, another common object is the **character** object which gets defined using quotation marks on each end of the value.

```
1  x <- "SPAM"
2  x
```
```
[1] "SPAM"
```

Both single or double quotation marks can be used to define a character object; however, you are not permitted to mix and match.

```
1  y <- 'SPAM'
2  y
```
```
[1] "SPAM"
```

If you accidentally mix and match quotation styles and attempt to run the code you will find the console stuck in a loop that constantly expects some additional piece of code. If this happens you need only press the escape, (*esc*), key with your cursor inside the console window.

A key consideration about character objects, which will probably seem obvious, is that you cannot perform standard mathematical operations on them.

```
1  y * 5
```
```
Error in y * 5 : non-numeric argument to binary operator
```

```
1  2 + "2"
```
```
Error in 2 + '2' : non-numeric argument to binary operator
```

Another type of object is what is known as a **logical** object. This is an object that contains a value of `TRUE` or `FALSE` and is often referred to as a **boolean** object.

```
1  x <- TRUE
2  y <- FALSE
3  x
4  y
```
```
[1] TRUE
[1] FALSE
```

The values `TRUE` and `FALSE` must be typed completely in uppercase without quotations for R to recognize them as a logical object. Alternatively, R does permit a shorthand version of each. Instead of typing `TRUE` and `FALSE`, you can type `T` and `F` respectively. Though, for ease of reading, using this shorthand version is not advised.

---

[10]TL;DR: While code written using `=` tends to have an intuitive appeal and requires one less key to press, the `<-` has greater functionality and is generally preferred by R's anointed high council (overseers of Tidyverse) for that reason. If you opt to use `<-`, it is worth noting that RStudio contains a keyboard shortcut that offers a more ergonomic means of typing `<-` by pressing the *alt* key followed by a minus (-) sign.

Thus far, we have demonstrated three basic categories of object: *numeric*, *character*, and *logical*. R refers to these various categories as **modes**[11], and as you progress with R, both in this book and more generally, you will encounter other object modes.

Naming Objects

Often times we will run into circumstances where other people are required to read, run, and modify the code we write. Still other times, we may need to look at, and make sense of, code we have written in the past and largely forgotten. These considerations make it of the utmost importance that all of the code we write is intelligible to other people and our future selves. Among the best way to achieve this is by naming objects appropriately. Ideally, the name of an object should be concise and descriptive. Generally, you can name objects almost anything you like, as long as the name begins with a letter, contains no spaces, avoids special characters (except underscores `_`), and does not use any of R's **reserved words** such as `TRUE` , `Inf` , `NaN` , `function` , etc.

Given that spaces are not permitted in the naming of objects, programmers have developed certain conventions to promote readability. One such convention is *snake case*, which separates lowercase lettered words with an underscore:

```
1    snake_case <- 1
```

Another, referred to as *camel case*, denotes separate words by capitalizing the first letter of each:

```
1    camelCase <- 2
```

There is also *period case*:

```
1    period.case <- 3
```

There is *random case* (Wickham et al., 2023):

```
1    Ra.nD0M_CAs.e <- 4
```

Finally, there is of course *angry case* for those moments when you need to communicate your frustration with coding:

```
1    ANGRYCASE <- 5
```

Apart from the last two, R programmers tend to use all of these with seeming abandon. It is worth noting that different style guides for R have been developed and altered over the years with varying degrees of adoption. Presently there is no consensus on which style-guide should act in an official capacity for R; however, the most popular, and widely respected, is the Tidyverse Style Guide[12] (https://style.tidyverse.org/syntax.html) which advocates the strict and concise use of *snake_case* only.

When it comes to naming objects, all of the rules just laid out only apply to what are referred to as **syntactic names**; however, if you are a psychopath, you can ignore all of those rules and create what are called **non-syntactic names** by simply enclosing the name within backticks.

---

[11]You may sometimes hear these referred to as object "classes" as well. The distinction between modes and classes in R is nuanced, with considerable overlap between the two terms, though they are not perfectly equivalent. I have chosen to refer to object modes because it more consistently categorizes objects as numeric, character, or logical, which I believe is helpful for beginners learning R.

[12]In case you were wondering, the "Tidyverse" is *"an opinionated collection of R packages designed for data science"* ("Tidyverse", 2024).

```
1   `420 * 69` <- "PARTY TIME!"
2   `The devil made me do it!` <- "Hail Satan"
```

1.4.5   Vectors

It is not the case that an object need only store a single value, as we have been doing above. Particularly when conducting statistical analyses, you are almost always working with variables that contain more than one value (i.e. multiple observations). In view of this, R objects can store as many values as you require.[13] For instance, if we want `x` to be equal to the numbers 1 through 5, we need only type:

```
1   x <- c(1, 2, 3, 4, 5)
2   x
```
```
[1] 1 2 3 4 5
```

The lower case `c` is short for *concatenate*, which is just a fancy word meaning "combine" or "connect". By combining the numbers 1 through 5 in this way we have created what is technically known as a **vector**[14]. We can further use this concatenate function, `c`, to combine vectors with other vectors. In the example below, we create two vectors, `x` and `y`, and concatenate them to create `z`.

```
1   x <- c(1, 2, 3, 4, 5)
2   y <- c(6, 7, 8, 9, 10)
3   z <- c(x, y)
4   z
```
```
[1]  1  2  3  4  5  6  7  8  9 10
```

The concept of a vector is one which will have relevance to people with a fondness of linear and matrix algebra[15] since it amounts to little more than a one-dimensional array/matrix. We can see how R handles vectors for these purposes by simply performing some mathematical operations on them. For instance, if we add a single number to our vector, we can see that R straightforwardly adds that number to each element (i.e. position) in the vector.

---

[13]Obviously, this statement is only true given the memory limitations of your computer's hardware and software.

[14]More specifically, we are speaking of *atomic vectors* here, though most people just call them *vectors.*

[15]While I assume such people must exist, their existence is about as well-confirmed as that of the Sasquatch.

> **Box 1.1: How to Use Your Colon Effectively :**
>
> In the previous examples, we used R's concatenate function to create a basic set of ascending numbers. The need to generate regular sequences of integers is a common occurrence in data analyses, so R provides users with a convenient means to create them using the **colon operator** (`:`).
>
> ```
> 1   x <- 1:5
> 2   x
> ```
> ```
> [1] 1 2 3 4 5
> ```
>
> This can also be used in reverse and with negative values.
>
> ```
> 1   3 : -5
> ```
> ```
> [1] 3  2  1  0 -1 -2 -3 -4 -5
> ```

```
1   x + 2
```
```
[1] 3 4 5 6 7
```

Correspondingly:

```
1   x - 2
2   x * 2
3   x / 2
4   x^2
```
```
[1] -1  0  1  2  3
[1]  2  4  6  8 10
[1] 0.5 1.0 1.5 2.0 2.5
[1]  1  4  9 16 25
```

A similarly logical process is seen when we perform mathematical operations on two or more vectors of the same size. For instance, adding them together results in the the first element of one being added to the first element of the other. The second element of one being added to the second element of the other, and so on.

```
1   x <- c(1,2,3,4,5)
2   y <- c(6,7,8,9,10)
3
4   x + y
```
```
[1]  7  9 11 13 15
```

However, a curious thing will occur if the vectors have an unequal number of elements greater than 1. Suppose, as an example, one vector has four elements and another has five and we want to add them together. In the process of adding the first element with the first element, and the second element with the second, and so on, R will automatically loop back around to the first element in the shorter vector to complete the calculation; though, it does this only after giving you a warning. Needless to say, you should not be performing any arithmetic on vectors of unequal lengths.

```
1   x <- c(1,2,3,4)
2   y <- c(6,7,8,9,10)
3
4   x + y
```
```
Warning in x + y: longer object length is not a multiple
of shorter object length
[1]  7  9 11 13 11
```

Vectors are also not limited to numbers. They can also contain character values and logical values.

```
1   a <- c(1,2,3)
2   b <- c("BREAD", "SPAM", "BREAD")
3   c <- c(TRUE, FALSE, FALSE)
4
5   a
6   b
7   c
```
```
[1] 1 2 3
[1] "BREAD" "SPAM"  "BREAD"
[1]  TRUE FALSE FALSE
```

However, you cannot mix and match. For instance, if you have a character string amongst a set of numeric values, those numeric values will all be converted to character strings as evidenced by the quotation marks in the output.[16]

```
1  d <- c(5, "SPAM", 6, 7, 8)
2  d
```
```
[1] "5"    "SPAM" "6"    "7"    "8"
```

If you have logical values amongst a set of numeric values, those logical values will be transformed such that `TRUE = 1` and `FALSE = 0`, making the entire vector numeric.

```
1  e <- c(666, TRUE, FALSE)
2  e
```
```
[1] 666   1   0
```

In fact if you have an entire vector of logical values you can treat the `TRUE` and `FALSE` values as 1s and 0s respectively. This is a feature of logical vectors that frequently comes in handy.

```
1  x <- c(100)
2  g <- c(TRUE, TRUE, FALSE, FALSE, TRUE)
3  x + g
```
```
[1] 101 101 100 100 101
```

Similar to how R comes with π (`pi`) stored as a constant, it also has constants for a few commonly used character vectors.

```
1  LETTERS
2  letters
3  month.name
4  month.abb
```
```
 [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M"
[14] "N" "O" "P" "Q" "R" "S" "T" "U" "V" "W" "X" "Y" "Z"

 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m"
[14] "n" "o" "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"

 [1] "January"   "February"  "March"     "April"
 [5] "May"       "June"      "July"      "August"
 [9] "September" "October"   "November"  "December"

 [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep"
[10] "Oct" "Nov" "Dec"
```

Indexing Vectors

Notice in the previous example's output that the numbers within brackets indicate the position number of a element in the vector. For example, in the vector `LETTERS`, `"N"` is located in the 14th position. In the vector `month.name`, `"May"` is in the 5th position. Every new line written to the console screen gives the position number of the first element on the line - meaning that the size of your console screen will effect which position numbers get displayed (so you might have different values that what is shown above).

---

[16]You can also check the vector's mode by running `mode(d)`

It is not by accident that these positions are demarcated using square brackets. Square brackets serve a special purpose in R. They allow us to index specific values. For instance, if we want to know what the 17$^{\text{th}}$ letter of the English alphabet is, we need only type...

```
1   LETTERS[17]
```

```
[1] "Q"
```

If we want to list out the first 5 letters we can simply insert a numeric vector...

```
1   LETTERS[c(1,2,3,4,5)]
```

```
[1] "A" "B" "C" "D" "E"
```

By contrast, if we want to list out all of the letters, except the first five (i.e., exclude the first five), we can include a minus sign in front of the concatenate symbol.

```
1   LETTERS[-c(1,2,3,4,5)]
```

```
[1] "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R"
[14] "S" "T" "U" "V" "W" "X" "Y" "Z"
```

The use of vectors inside the indexing brackets allows us to select any position we want. For instance, if we wanted to examine the 2nd, 3rd, 5th, 7th, 11th, 13th, 17th, 19th, and 23rd numbers (all prime numbers), we can create a vector of those values and simply insert it into the index.

```
1   primes <- c(2, 3, 5, 7, 11, 13, 17, 19, 23)
2   LETTERS[primes]
```

```
[1] "B" "C" "E" "G" "K" "M" "Q" "S" "W"
```

1.4.6   Operators And Comparison Statements

Symbols in R such as `<-` , `+` , `-` , and so on are referred to as operators because they are used to perform "operations" such as assigning a name to an object, adding numbers together, etc. Table 1.2 shows a list of some common operators in R that we have seen before and some new ones called **relational operators**. These are operators that evaluate a comparison of some kind. For instance, you can evaluate whether one value is *greater than* or *less than* another value.

```
1   3 > pi
```

```
[1] FALSE
```

In the above example, the statement "three is *greater* than π", is a false statement. In the example below, the statement "three is *less* than π", is a true statement.

```
1   3 < pi
```

```
[1] TRUE
```

In a similar fashion, you can also evaluate whether a value is *greater than or equal to* some other value. For example:

```
1   pi >= pi
```

```
[1] TRUE
```

Alternatively, you might choose to evaluate whether a value is *less than OR equal to* some other value

Type	Operator	Description
Assignment		
	`x <- value`	Assign a value to a name.
	`value -> x`	
	`x <<- value`	(see Box 1.1)
	`value ->> x`	
	`x = value`	
Arithmetic		
	`x + y`	Adds values of objects
	`x - y`	Subtracts values of objects
	`x * y`	Multiplies the value of objects
	`x / y`	Divides the value of objects
	`x^y`	Raises the value of one object to another
	`x %% y`	Returns the quotient of objects
	`x %/% y`	Returns the remainder of objects
Relational		
	`x < y`	Checks if x is less than y
	`x > y`	Checks if x is greater than y
	`x <= y`	Checks if x is less than or equal to y
	`x >= y`	Checks if x is greater than or equal to y
	`x == y`	Checks if x is equal to y
	`x != y`	Checks if x is not equal to y

Table 1.2: Basic R Operators

```r
pi <= 3
```

```
[1] FALSE
```

You can also evaluate whether two values are *equivalent* or *not equivalent*, by using the symbols `==` and `!=` respectively.

```r
pi == pi #testing if equivalent
pi == (22/7)
pi != (22/7) #testing if NOT (!) equivalent
```

```
[1] TRUE
[1] FALSE
[1] TRUE
```

1.4.7 Functions

In conventional mathematics a function is a way of relating an input to an output (Pierce, 2022). Typically this is notated as

$$f(\text{input}) = \text{output} \tag{1.1}$$

When you place something inside the left parentheses, and there is a corresponding output. The use of f here to denote the function is just a formality mathematicians have adopted. A function can be named or symbolized with anything.

As an example of a function's use, we could create one that outputs the square root of a number.

$$f(x) = \sqrt{x} \tag{1.2}$$

In this case, x is just acting as a place holder; thus, swapping the x inside of $f()$ with a real number will give us a corresponding output by taking the square root of that number. For example, if we insert the number 25 into the function:

$$\begin{aligned} f(25) &= \sqrt{25} \\ &= 5 \end{aligned} \tag{1.3}$$

Functions in R work identically to this. For instance, R has a function for finding the square root of a number, except instead of naming the function $f(x)$, it names the function `sqrt(x)`.

```
1   sqrt(25)
```
```
[1] 5
```

And, rather conveniently, R will also store the output of a function as an object if you ask it to.

```
1   x <- sqrt(25)
2   x
```
```
[1] 5
```

As you might expect, given its lineage as a tool for data analysis, R has many such functions. Examples of some of the more common, self-explanatory, ones can be seen below. For each we will insert a vector containing the values one through five.[17]

```
1   x <- c(1, 2, 3, 4, 5)
```

Calculating the *sum* of all the values:

```
1   sum(x)
```
```
[1] 15
```

Calculating the *product* of all the values:

```
1   prod(x)
```
```
[1] 120
```

Calculating the *minimum* and *maximum* of all the values:

```
1   min(x)
2   max(x)
```
```
[1]  1
[1]  5
```

Calculating the *length* (i.e., number of elements) of a vector:

```
1   length(x)
```
```
[1] 5
```

Calculating the *mean* of all the values:

---

[17]It's perhaps worth pointing out that the small `c` we use to combine values into a vector is also a function, which is why it is always followed with parentheses, `c()`

```
1   mean(x)
```
```
[1] 3
```

Calculating the *median* of all the values:

```
1   median(x)
```
```
[1] 3
```

Functions are not limited to just mathematical processes either. For instance, R has a function to tell us what an object's mode is, thus allowing us to determine if the vector consists of numeric, character, or logical values.[18]

```
1   mode(x)
```
```
[1] "numeric"
```

Arguments

The utility of functions in R actually extends far beyond this basic usage because most are easily modified through the use of **arguments**. An "argument" is simply a parameter that allows you to customize how a function operates. A simple example of this is the `round()` function. This is used to round numbers to a specified decimal place. For instance, if we have a vector that contains both the number π and the $\sqrt{2}$

```
1   x <- c(pi, sqrt(2))
2   x
```
```
[1] 3.141593 1.414214
```

We can use the `round()` function and its "digits" argument to round these to 2 digits.

```
1   round(x, digits = 2)
```
```
[1] 3.14 1.41
```

Alternatively, we could round to the nearest integer:

```
1   round(x, digits = 0)
```
```
[1] 3 1
```

The `round()` function only takes one argument but many functions take multiple arguments. A good example of this is the sequence function, `seq()`, which generates regular number sequences. For instance, if you wanted to generate a sequence from 0 to 100, counting by 2's, there are three arguments you will need to set: `from`, `to`, and `by`:

```
1   seq(from = 0, to = 100, by = 2)
```
```
 [1]   0   2   4   6   8  10  12  14  16  18  20  22  24
[14]  26  28  30  32  34  36  38  40  42  44  46  48  50
[27]  52  54  56  58  60  62  64  66  68  70  72  74  76
[40]  78  80  82  84  86  88  90  92  94  96  98 100
```

The sequence function is also illustrative of another feature of functions, often they will have mutually exclusive arguments. Instead of using the `by` argument, we could have used the `length.out` argument to specify how many values we want in our sequence.

---

[18]Do not confuse this with the mathematical concept of a modal value; i.e., the number that appears most often.

```
1  seq(from = 0, to = 100, length.out = 6)
```
```
[1]   0  20  40  60  80 100
```

To save yourself some effort in typing out functions and their corresponding arguments, you can actually just provide the values, without the argument name and equal sign, provided you specify the arguments in the correct order.

```
1  seq(0, 100, 2)
```
```
 [1]   0   2   4   6   8  10  12  14  16  18  20  22  24
[14]  26  28  30  32  34  36  38  40  42  44  46  48  50
[27]  52  54  56  58  60  62  64  66  68  70  72  74  76
[40]  78  80  82  84  86  88  90  92  94  96  98 100
```

To determine the correct ordering of arguments you will need to consult the function's *R documentation*.

1.4.8   R (i.e., Help) Documentation

There are many more functions built into R, some of which do very complex things; consequently, when reading R code you will often encounter functions whose process and use seems mysterious. For this reason, it is often necessary to access R's help documentation. Each function in the base version of R has corresponding documentation that describes its purpose, arguments, and has associated references.[19] Admittedly, the R documentation can often be a bit tricky to decipher for novice R users, but it should always be the first starting point whenever you are confused about how a function should be used or what it is doing. Only after you have consulted it should you branch out to other resources (e.g., an internet search).

To access the R documentation of any function you need only precede the name of the function with a question mark and run it.

```
1  ?sqrt
```

1.4.9   Missing Values

A common hurdle in data analysis are missing values. Values can be missing for any number of reasons; perhaps a participant never showed up for a research session, perhaps an lab animal died, perhaps there was a equipment malfunction, perhaps someone recorded something incorrectly, or maybe you just ran out of time and money. The R language denotes missing values using `NA`, which stands for "not available." In many instances, numerical calculations on a `NA` value will simply result in another `NA` value.

```
1  5 + NA
```
```
[1] NA
```

Intuitively, this behaviour makes a fair amount of sense to most people. We do not know what `NA` is or should be, so the expression `5 + NA` cannot be evaluated. And R, quite logically, extends this principle to functions:

```
1  x <- c(710, 633, 786, NA, 642)
2  mean(x)
```
```
[1] NA
```

---

[19] For common functions in base R, the documented references tend not to be too useful as they usually just reference a guide on the S programming language, which (if you look up the guide) often does little more than show you how the function is used without providing any theoretical background.

However, in this latter case, the logic which seemed so obvious initially seems less so now. Consider that these values might be observations from an experiment. Many researchers will reflexively ignore the `NA` and compute the *mean* of these values as readily as a rat devours a food pellet, and it is to R's credit that it actually prohibits its users from indulging so recklessly.

How missing values should be handled is a matter of great importance and statisticians often disagree on what the best practice should be in any given case. In a situation like this, most people would simply ignore the missing element and treat the vector as containing only four values. However, most data sets are not this simplistic. That `NA` might be *paired* with collected observations of other variables. That is a situation where you might, for the purpose of conducting a certain analysis, require a number to be in that fourth spot. What do you do then? Do you replace `NA` with the mean of the four values, do you replace it with the median, or do you do something else?

There is no one-size-fits-all answer here; however, in those instances where simply ignoring the `NA` is the sensible course of action, many base R functions allow you to specify an additional logical *argument*, `na.rm`, that will remove any `NA` values prior to calculation. You can see this by simply accessing the R documentation (e.g., `?mean`). By default the argument is set to `FALSE` and setting `na.rm = TRUE` will remove the `NA` values accordingly.

```
1   mean(x, na.rm = TRUE)
```
```
[1] 2771
[1] 692.75
```

For situations where a function does not have a `na.rm` argument or equivalent, the function `is.na()` can be easily employed. This function evaluates whether each element of an R object is missing or not and returns a logical (`TRUE` or `FALSE`) value. For example:

```
1   x <- c(710, 633, 786, NA, 642)
2   is.na(x)
```
```
[1] FALSE FALSE FALSE TRUE FALSE
```

Looking at the output, we can see that the fourth value is missing because it has returned a value of `TRUE` (i.e., the function has determined that it *is* a `NA` value). Combing the behaviour of this function with the indexing feature of vectors (see section 1.4.5) and a **logical operator** called the **negation operator** (denoted using `!`), we can easily obtain a version of the vector with missing values excluded.

```
1   x[!is.na(x)]
```
```
[1] 710 633 786 642
```

With the negation operator, the expression `!is.na(x)` can be interpreted as asking, "which values of `x` are *not* missing values?" This is easily seen by comparing the `is.na()` function with and without the negation.

```
1   is.na(x)
2   !is.na(x)
```
```
[1] FALSE FALSE FALSE  TRUE FALSE
[1]  TRUE  TRUE  TRUE FALSE  TRUE
```

Notice that the `!` just provides the logical opposite (i.e., negation) of the original function. Thus, putting all this together, you could write ...

```
1  mean(x[!is.na(x)])
```
```
[1] 692.75
```

...in lieu of using or not having a `na.rm` style argument to remove missing values. To novice users of R, techniques like this may seem cumbersome initially. This is especially the case when you are dealing with so few values and can immediately see what is and is not missing within the data. For instance, noting that the fourth value is missing from `x`, you could simply create a new vector of the form `y <- c(710, 633, 786, 642)` and insert that into your functions. However, many (if not most) data sets are too large to "eyeball" and manually rebuild in this way. Consequently, automated solutions like those shown with the negation operator are not only necessary to save time, but are also less prone to error.

1.4.10   Data Frames

While there are situations where a single vector constitutes the only data that needs to be analyzed, it is more often the case that you are working with "sets" of data. That is to say, typically your data consists of observations across a range of different variables. Consequently, for the purposes of organization, it is helpful to keep all of this data stored as a single object. In R, there are a number of ways you could do this. You could store data as a *table*, a *list*, or a *matrix* which are all unique *classes* of objects R recognizes. However, for most uses cases, a **data frame** is going to be the preferred method of data storage in R.

In its simplest terms a data frame is simply a spreadsheet, where rows represent observations and columns represent variables. Consider a hypothetical experiment with two groups, a control and experimental group, and 10 observations, one of which is missing for some reason. Visually, the data might look like Table 1.3:

Subject	Group	Value
1	Experimental	-0.36
2	Control	0.28
3	Experimental	1.54
4	Control	0.51
5	Experimental	-1.28
6	Experimental	1.15
7	Control	-2.22
8	Experimental	-0.51
9	Control	
10	Control	-1.04

Table 1.3: Example Data Frame

We can easily recreate this in R using the `data.frame()` function. Inside the function, we specify our desired columns as *arguments*.

```
1  df <- data.frame(
2    Subject = 1:10,
3    Group = c("Exp", "Cont", "Exp", "Cont", "Exp", "Exp",
4              "Cont", "Exp", "Cont", "Cont"),
5    Value = c(-0.36,  0.28,  1.54,  0.51, -1.28,  1.15,
6             -2.22, -0.51,  NA, -1.04)
7  )
8
9  df
```
```
   Subject Group Value
1        1   Exp -0.36
2        2  Cont  0.28
3        3   Exp  1.54
```

```
4         4  Cont  0.51
5         5   Exp -1.28
6         6   Exp  1.15
7         7  Cont -2.22
8         8   Exp -0.51
9         9  Cont    NA
10       10  Cont -1.04
```

Alternatively, if you have the variables *Subject*, *Group*, and *Value* already stored as vectors, you could build your data frame in the following way:

```
1  Subject <- 1:10
2  Group <- c("Exp", "Cont", "Exp", "Cont", "Exp", "Exp",
3             "Cont", "Exp", "Cont", "Cont")
4  Value <- c(-0.36,  0.28,  1.54,  0.51, -1.28,  1.15,
5             -2.22, -0.51,  NA, -1.04)
6
7  df <- data.frame(Subject, Group, Value)
8  df
```

Now, strictly speaking, you would almost never input your data into R in the manner we have done here (i.e., by manually typing in the values). However, the basics of constructing a data frame is an essential, and frequently appealed to, piece of knowledge when working with R.

There are two critical features of data frames that separates them from traditional spreadsheets. The first is that each column needs to consist of a single object mode (e.g., numeric, character, or logical; see 1.4.4). For instance, in the data frame above the `Subject` column consists only of *numeric* objects, the `Group` column only consists of *character* objects and the `Value` column, again, only consists of *numeric* objects. We can see this by running the following code:

```
1  sapply(df, FUN = mode)
       Subject       Group       Value
     "numeric" "character"   "numeric"
```

In this example, the `sapply()` function has, quite literally, *applied* the function `mode()` to each of the columns of our data frame, thereby telling us what each column's mode is. This is important because columns behave like vectors insofar as mixing and matching object types will potentially change the entire column. As an example, if we had coded ...

```
1  Value = c("-0.36", 0.28, 1.54, 0.51, -1.28, 1.15, -2.22, -0.51, NA, -1.04)
```

you will find that every single number in that column automatically becomes a character object even though only the first of the nine elements was typed as a character object. This is going to be very irritating if you want to perform mathematical operations on that column and are unaware that all of its elements have been coerced into character objects (notice that printing the data frame does not show character objects with quotes like vectors do).

The second critical feature of data frames is that each column *must* contain the same number of elements as every other column. In our example, *Subject*, *Group*, and *Value* all contain 10 elements (the missing value is counted as an element). In most cases, if you try and build a data frame with columns of unequal lengths, R will produce an error message.

```
1   df_2 <- data.frame(
2     a = 1:4,
3     b = 1:3
4   )
```

```
Error in data.frame(a = 1:4, b = 1:3) :
  arguments imply differing number of rows: 4, 3
```

In other cases, if you have an unequal amount of values in your columns and R determines that it can evenly repeat a sequence, R will automatically recycle that sequence.

```
1   df_3 <- data.frame(
2     a = 1:4,
3     b = 1:2
4   )
5
6   df_3
```

```
  a b
1 1 1
2 2 2
3 3 1
4 4 2
```

Notice in the above example that we assigned four values to the *a* column and two values to the *b* column and instead of producing an error, R simply recycled the values in *b* to fill the empty spots.

Indexing

Similar to how vectors can be indexed using square brackets, data frames can also be indexed. Going back to our original data frame (`df`), suppose we wanted to look at the value found in the fifth row of the third column. This can be easily accomplished in the following way:

```
1   df[5, 3]
```

```
[1] -1.28
```

Notice, the number on the left side of the comma (5) refers to the row, and the number on the right side (3) refers to the column. The easy way to remember this is that the numbers in the brackets represent a x and y coordinate system, with x's being rows, and y's being columns.

In the last example we selected a single element of our data frame, but we can select more than one value and more than one column if need be. For instance, we could isolate rows 1, 3, and 5, from columns 2, and 3 only.

```
1   df[c(1, 3, 5), c(2:3)]
```

```
  Group Value
1   Exp -0.36
3   Exp  1.54
5   Exp -1.28
```

If you wanted to keep all the columns visible while only looking at rows 1, 3 and 5, you need only to leave the left side of the comma blank.

```
1   df[c(1, 3, 5), ]
```

```
  Subject Group Value
1       1   Exp -0.36
3       3   Exp  1.54
5       5   Exp -1.28
```

A similar logic applies to rows:

```
1  df[ , c(2:3)]
```

```
   Group Value
1    Exp -0.36
2   Cont  0.28
3    Exp  1.54
4   Cont  0.51
5    Exp -1.28
6    Exp  1.15
7   Cont -2.22
8    Exp -0.51
9   Cont    NA
10  Cont -1.04
```

Extracting Columns as Vectors

There will also be many circumstances where you need to work with the values of a single column only. For instance, if you want to calculate the mean of the third column (*Value*), you can use one of R's extraction operators, the `$`, to isolate that column. The following code will isolate the *Value* column and output it as a vector:

```
1  df$Value
```

```
[1] -0.36  0.28  1.54  0.51 -1.28  1.15 -2.22 -0.51  NA -1.04
```

You can, therefore, just insert this into the `mean()` function.

```
1  mean(df$Value, na.rm = TRUE)
```

```
[1] -0.2144444
```

Alternatively, instead of using the `$` operator, you can use doubled square brackets to specify the column number you want:

```
1  df[[3]]
```

```
[1] -0.36  0.28  1.54  0.51 -1.28  1.15 -2.22 -0.51  NA -1.04
```

Neither method of extracting a column is intrinsically better than the other. It really boils down to whether you prefer to reference your columns by names or numbers. The former is often easier to read at the expense of writing more code, whereas the latter, while harder to discern at a quick glance, requires less writing and can produce, superficially, a tidier looking script.

If you want to extract a column, but still preserve it's classification as a data frame instead of *dropping* it to a vector you can include the argument `drop = FALSE` inside your indexing brackets.

```
1  df[ , 3, drop = FALSE]
```

```
   Value
1  -0.36
2   0.28
3   1.54
4   0.51
```

```
5  -1.28
6   1.15
7  -2.22
8  -0.51
9     NA
10 -1.04
```

Adding and Removing Columns

Adding new columns to a data frame is very simple. Suppose we wanted to create a column named *Alpha* containing the first 10 letters of the English alphabet.

```r
df$Alpha <- letters[1:10]
df
```

```
   Subject Group Value Alpha
1        1   Exp -0.36     a
2        2  Cont  0.28     b
3        3   Exp  1.54     c
4        4  Cont  0.51     d
5        5   Exp -1.28     e
6        6   Exp  1.15     f
7        7  Cont -2.22     g
8        8   Exp -0.51     h
9        9  Cont    NA     i
10      10  Cont -1.04     j
```

If we wanted to create a column named *new_val* that multiplies all the numbers in the *Value* column by 100, we can easily do that.

```r
df$new_val <- df$Value * 100
df
```

```
   Subject Group Value Alpha new_val
1        1   Exp -0.36     a     -36
2        2  Cont  0.28     b      28
3        3   Exp  1.54     c     154
4        4  Cont  0.51     d      51
5        5   Exp -1.28     e    -128
6        6   Exp  1.15     f     115
7        7  Cont -2.22     g    -222
8        8   Exp -0.51     h     -51
9        9  Cont    NA     i      NA
10      10  Cont -1.04     j    -104
```

To remove a column, there are a few options. Assuming you want to remove the *Alpha* (third) column, you can just set that column equal to a **null value**, which just means that something is undefined and therefore does not exist as an object in the R language.

```r
df$Alpha <- NULL
df
```

```
   Subject Group Value new_val
1        1   Exp -0.36     -36
2        2  Cont  0.28      28
3        3   Exp  1.54     154
4        4  Cont  0.51      51
```

```
5        5   Exp -1.28    -128
6        6   Exp  1.15     115
7        7  Cont -2.22    -222
8        8   Exp -0.51     -51
9        9  Cont    NA      NA
10      10  Cont -1.04    -104
```

If you want to remove multiple columns, a quick way is to simply index the columns you do NOT want to keep, negate them using a minus sign (which means you are now technically indexing the ones you DO want to keep). You can then override your data frame object, which in our case is (df). To illustrate, we will remove column's one and four.

```
1  df <- df[ , -c(1, 4)]
2  df
```

```
    Group Value
1     Exp -0.36
2    Cont  0.28
3     Exp  1.54
4    Cont  0.51
5     Exp -1.28
6     Exp  1.15
7    Cont -2.22
8     Exp -0.51
9    Cont    NA
10   Cont -1.04
```

Adding and Removing Rows

To add a row to an existing data frame, the conventional strategy is to use the rbind() function. "rbind" is short for "row bind" and does more or less what it says one the box: it binds (i.e., combines) objects by rows. For instance, if we create a new dataframe that contains a row (or rows) we want to add, we can then use the rbind() function to append it to the original dataframe.

```
1  new_row <- data.frame(
2    Group = "SPAM",
3    Value = 999
4  )
5
6  df <- rbind(df, new_row)
7  df
```

```
    Group  Value
1     Exp  -0.36
2    Cont   0.28
3     Exp   1.54
4    Cont   0.51
5     Exp  -1.28
6     Exp   1.15
7    Cont  -2.22
8     Exp  -0.51
9    Cont     NA
10   Cont  -1.04
11   SPAM 999.00
```

To remove rows (e.g., 9 and 11), you can follow the same basic process that was outlined for removing columns.

```
df <- df[-c(9, 11), ]
df
```

```
   Group Value
1    Exp -0.36
2   Cont  0.28
3    Exp  1.54
4   Cont  0.51
5    Exp -1.28
6    Exp  1.15
7   Cont -2.22
8    Exp -0.51
10  Cont -1.04
```

Row and Column Names

Notice in the previous example that, by removing row 9 (i.e., the row that contained the `NA` value), the index numbers on the leftmost side of the data frame's output become mislabelled. It counts from 1 to 8, skips 9, and goes straight to 10. The reason it does this is because those numbers on the left are not actually index values, as you might reasonably assume. They are actually *row names* and, when the data frame was initially created, the rows were literally named 1 through 10.

R users tend to be on the fence as to whether this is a useful feature or not. It does provide a nice visual confirmation that specific rows have been removed, but it makes future indexing potentially more confusing since the row named 10 is actually the 9th row. Thus, its often helpful to rename the rows after you have subset or removed certain values. You can do this using the `rownames()` function.

```
rownames(df) <- 1:nrow(df)
df
```

```
   Group Value
1    Exp -0.36
2   Cont  0.28
3    Exp  1.54
4   Cont  0.51
5    Exp -1.28
6    Exp  1.15
7   Cont -2.22
8    Exp -0.51
9   Cont -1.04
```

Note that we used the function `nrow()` to create the sequence of numbers. This function simply counts how many rows are in a data frame.

```
nrow(df)
```

```
[1] 9
```

An alternative way of defining the row names would have been to type `rownames(df) <- 1:9`; however, this is STRONGLY discouraged. The reasons being that 1) if you are working with a large data frame, you often do not know how many rows there are and 2) if some aspect about your data frame changes in the future (maybe because you have updated your data set or indexed different values), the `1:9` is no longer going to be

accurate and will produce errors that you may or may not notice, unless you have remembered to change it. Using the code `1:nrow(df)` ensures that your row names will always be correct.

Here we have named our rows using numbers, but you can technically name rows anything you want.

```
1  rownames(df) <- month.name[1:nrow(df)]
2  df
```

```
           Group Value
January      Exp -0.36
February    Cont  0.28
March        Exp  1.54
April       Cont  0.51
May          Exp -1.28
June         Exp  1.15
July        Cont -2.22
August       Exp -0.51
September   Cont -1.04
```

Generally speaking though, this is not something you should be doing. If you wanted to label each row with a name of the month, you would be better off creating a new column called *Month*, and keeping the row names as ascending integers.

Column names can be renamed in a similar fashion using the `colnames()` function. Though, for *syntactic* column names you are not permitted to name them solely with numeric values, nor can you include spaces or any special characters other than an underscore.

```
1  colnames(df) <- c("1st_Col", "2nd_Col")
2  df
```

```
           1st_Col 2nd_Col
January        Exp   -0.36
February      Cont    0.28
March          Exp    1.54
April         Cont    0.51
May            Exp   -1.28
June           Exp    1.15
July          Cont   -2.22
August         Exp   -0.51
September     Cont   -1.04
```

If you do use a number, space, or special character to name your column, it becomes a *non-syntactic* name (see section 1.4.4) and backticks become necessary to isolate it.

```
1  colnames(df) <- c(1, "Col 2")
2  df
```

```
              1 Col 2
January     Exp -0.36
February   Cont  0.28
March       Exp  1.54
April      Cont  0.51
May         Exp -1.28
June        Exp  1.15
July       Cont -2.22
August      Exp -0.51
September  Cont -1.04
```

```
1  df$1
```
```
Error: unexpected numeric constant in "df$1"
```

```
1  df$Col 2
```
```
Error: unexpected numeric constant in "df$Col 2"
```

```
1  df$`1`
```
```
[1] "Exp"  "Cont" "Exp"  "Cont" "Exp"  "Exp"  "Cont" "Exp"   "Cont"
```

```
1  df$`Col 2`
```
```
[1] -0.36  0.28  1.54  0.51 -1.28  1.15 -2.22 -0.51 -1.04
```

1.5   Packages

As a standalone piece of software, (base) R has an excellent toolbox of functions and operations for most data analysis/science scenarios; however, it is by no means a complete toolbox. Like any statistical software, there are scenarios for which it is simply not equipped to handle on its own. But R being a language means it is adaptable to these scenarios. R users can program their own sets of functions to suit a specific purpose and **package** these functions with appropriate documentation and data for other R users to install into their own personal library of packages.

The packages R users make publicly available are downloaded from online *repositories* (often called "repos"). The *Comprehensive R Archive Network* (CRAN) discussed in section 1.3.2 is one such repository, another well known one would be *GitHub*.[20] The CRAN repository is easily the most frequented by R users and is likely to be the only R repository you will ever need. It is special in that the packages it provides are curated by the *The R Project for Statistical Computing*.

To install a package from the CRAN repository you simply run the function `install.packages(" ")` with the package name inside the quotation marks. As an example, we shall install the "cowsay" package.

```
1  install.packages("cowsay")
```

Running the above line of code should prompt a variety of interesting things to occur inside the console window. This is the package installing into the *library* of packages stored on your computer. Upon successful completion of the install should be, among other things, a statement reading something to the effect of `package 'cowsay' successfully unpacked`. What this means is we can now access the various functions contained within the package, but before we do we should install another package called "praise".

```
1  install.packages("praise")
```

In order to access the functions contained in these packages we need only execute the line `library()` with the with the package name inside the parentheses (quotation marks are not necessary here).

```
1  library(cowsay)
2  library(praise)
```

We can now run the functions `say()` and `praise()` in the following way:

```
1  say(praise())
```

---

[20]This textbook actually has its own GitHub repo: https://github.com/statistical-grimoire/book

It should be noted that when you close your R environment, you will not have access to these two functions the next time you open R. However, you can easily regain access to them by re-running the `library()` functions above (meaning these lines should be saved in the scripts you write). You do NOT need to reinstall the packages unless you have updated to a new release of R itself (e.g., you have moved from version 4.4.1 to version 4.5.0).

Each package downloaded from the CRAN repository has documentation associated for both it and the functions it provides. This documentation can be accessed through the usual route of typing a `?` followed by the package name or function name. Since it is easy to miss, it should be noted that the top left corner of R documentation specifies what package a function belongs too (see section 2.1 for details on handling conflicting packages). Insofar as learning about a package is concerned, R Documentation is quite useful, but often times a better option is to seek out its accompanying .PDF reference manual. A basic internet search is usually the simplest way to find these for any given package; however, the R project has links to the manuals of all its packages in the package's description page. The following web address will take you to a complete list of all the current CRAN packages available to download and provide you with a link to each package's description page.

https://cran.r-project.org/web/packages/available_packages_by_name.html

1.6   File Extensions

Most users are familiar with the fact that computers store a multitude of files, each serving different purposes. We encounter various types of files daily: image files, text files, audio files, and much more. Within these broad categories lie even more specific file types, each with unique characteristics and uses. For example, image files can be distinguished into formats such as GIF, PNG, and TIFF, each catering to different needs in terms of quality, compression, and usage.

Historically, the way in which users could distinguish different file types was by looking at the **file extension** appended to the file's name. For instance, when looking at an image file, you might see a `.png` at the end of the name (e.g., `grandma.png`) indicating that it is a portable network graphics file. The file extension dictates which programs can read the file and how they read them.[21] This is in contrast to *directories* which have no extension (directories will be discussed next in section 1.7).

Unfortunately, most modern operating systems are configured in such a way that they do NOT display file extensions and, if a (conventional) user needs to identify a file type, they are expected to determine it on the basis of how the file's icon looks, which is often unreliable. Microsoft's Windows operating system began adopting this practice of hiding extensions around 2015 with Windows 10, and Macintosh computers had been doing it even longer than that.

The reasons why this change took place are not altogether clear, but the main justification seems to be that there is an inherent danger in users accidentally deleting or altering an extension when renaming a file. At face value this makes a certain amount of sense, but not when you consider the problems that it creates. In particular, this compromises a computer's (and by extension network's) security much more. Seeing an unfamiliar file extension and knowing not to click on it (because it is unfamiliar) is one of the most effective ways of preventing malicious software from attacking your computer. Seeing unfamiliar file extensions also means the user is less likely to mess about with file types on their system they do not understand and are integral for the running of their system and its applications. However, with no file extension displayed there is no obvious way of distinguishing familiar file types from unfamiliar ones.

Hiding extensions also creates the problem of a wolf in sheep's clothing. Seeing `grandma.png.exe` on a system that is configured to hide extensions will display for the user as `grandma.png`, leading someone (a child

---

[21] I apologize if this is obvious to many of you reading this, but experience teaching has taught me that this is no longer common knowledge and needs to be explained to modern audiences.

perhaps) to believe they are clicking an innocent image of their grandma, when in fact their computer is about to be devoured by grandma.[22]

Figure 1.1: From the National Gallery of Victoria, Melbourne: Gustave Doré's illustration of the "*penultimate moment, just before the triumphant, and satiated, wolf bites off Little Red Riding Hood's head*" in Charles Perrault's version of the classic fairy tale (Doré, 1862).

For both security and everyday use, it is important for users to understand that different types of files exist and they can easily identify them. The relatively modern practice of hiding file extensions prevents new users from gaining the essential experience needed to learn this and tends to make programming a more cumbersome process than it needs to be. The reality is that file extensions are essential pieces of information for any programmer working with or creating files. Fortunately, operating systems still make it possible to display extensions and it is highly recommend that readers of this book enable that feature on their respective system:

- **Windows 11:**

    1. In the Windows search bar type "File Explorer Options"
    2. Open the *File Explorer Options* menu.
    3. Select the *View* tab.
    4. In the *Advanced Settings* scroll area, uncheck the box labelled *Hide extensions for known file types*.

- **Macintosh:**

    1. In a Finder window on your Mac

---

[22]Once upon a time users were expected to be the "smart" ones, not their devices.

2. Select *Finder* at the top of the screen.

3. Open *Settings* ("*Preferences*" on older Macs)

4. Select *Advanced.*

5. Choose select *Show all filename extensions.*

1.7   Directories

Something often overlooked in introductions to programming languages is the concept of directories. Particularly in the context modern operating systems, directories have fallen into the background of basic computing knowledge users are expected to have. It is very much something that modern operating systems do not want their general user base to think or even know about, but they are an essential piece of knowledge for programming in any language.

A **directory** is what most people refer to as a file folder on their computer - but this is a misnomer because the literal image of a folder you see on your desktop is actually just your operating system's way of visually representing what is more technically called a directory. Speaking more accurately, a directory is an address that *directs* you to a file. Thus, in the same way that people have an address indicating where they live, files that are stored on your computer also have addresses.

As an example, if you right click the icon of a file on your desktop (control-click on a Mac) and select "properties" (or "get info" on a Mac), among the various pieces of information it lists is "Location" (or "Where") information. For instance, on your computer you might see something similar to these:

- *Location:* C:\Users\Your Name\Desktop

- *Where:* Macintosh HD > Users > Your Name > Desktop

1.7.1   The Working Directory

Any time R needs to grab or create a file, it needs to grab or create that file somewhere and if you do not tell R where that somewhere is, it will default to what is known as the **working directory**.

To see where your current working directory is set to you can just run the function `getwd()`.

```
1  getwd()
```
```
[1] "C:/Users/Acheron/Documents"
```

the R output in this case will likely vary between different computers, so you should not expect to see the exact same output, but it should be relatively similar.

The way to interpret what we are seeing here is as as a path, or route to get to the directory called `Documents`. `C:/` represents the hard drive and many computers will have more than one of these, so it is vital to know which one you are working in. Within the hard drive is the directory called `Users`. We can tell that `Users` is a directory here and not a file because it is bounded by forward slashes, `/`, and has no file extension. Then we have a subdirectory of that called (on my computer) `Acheron`. From this subdirectory `Acheron`, we have another subdirectory, which is called `Documents`.

To change working directory you can simply use the function `setwd()` and specify the full address. As an example, to change the working directory to the desktop you would type something akin to ...

```
1  setwd("C:/Users/Acheron/Desktop")
2  getwd() # Run to confirm wd
```

```
[1] "C:/Users/Acheron/Desktop"
```

Generally speaking, the default behaviour of RStudio is to set the working directory as the computer's main "Documents" folder. This default behaviour of RStudio can be changed by selecting *Tools > Global Options > General*. Alternatively, if you open a script file in R studio by clicking on it with your mouse, RStudio will automatically set the working directory to the location of that script file.

To illustrate how directories work and how you can easily navigate them, we are going to create a simple data frame and save it as a spreadsheet file that we can open on our computer.

```
1  # Create the data frame
2  df <- data.frame(Alphabet = letters)
```

To save this as a spreadsheet file, we can use the function `write.csv()`. This function will save our data frame as something called a .CSV file, which is just a universal type of spreadsheet file that any spreadsheet software can open. To use this function, we just need to give it our data frame and tell it what we want our file name to be.

```
1  write.csv(df, file = "letters_1.csv")
```

Running this function will save a file on our computer called `"letters_1.csv"`, but where has it saved it? As you have hopefully realized, it has saved it to our working directory. Thus, if your working directory is set to your desktop, you should see the file `"letters_1.csv"` located there. Alternatively, you can have R list the files (and subdirectories) in your working directory by running

```
1  list.files(path = ".")
```

```
[1] letters_1.csv
```

Alternatively we could have saved the file by specifying the complete file path followed by the file name we want our spreadsheet to have.

```
1  write.csv(df, file = "C:/Users/Acheron/Desktop/letters_1.csv")
```

This method, while much more annoying to type, is valuable because it allows us to save the file in any location we want on our computer. For instance, we could have saved the file the Documents folder, even though the working directory is set to the Desktop.

```
1  write.csv(df, file = "C:/Users/Acheron/Documents/letters_2.csv")
```

1.7.2   Navigating Directories

When it comes to navigating directories, it is quite cumbersome to type the full address of a location on your computer. Additionally, writing a fixed address into your code makes it difficult for other people run that same code on their computers since directories vary from computer to computer. Consequently, it is usually beneficial to specify a path relative to the working directory. To illustrate we are going to use the function `dir.create()`.

```
1  dir.create(path = "./Directory A")
```

This will create a directory (i.e., visually you will see a folder) called `Directory A` inside your working directory. The period (`.`) in front of the forward slash (`/`) is a shorthand way of referring to the current working directory. Thus, you can view the path here as equivalent to typing `"C:/Users/Acheron/Desktop/Directory A"`.

Next we will nest another new directory, B, inside A, and then nest a directory, C, inside B, such that the path structure ends up like this:

```
Directory A
└──Directory B
    └──Directory C
```

```r
1  dir.create(path = "./Directory A/Directory B")
2  dir.create(path = "./Directory A/Directory B/Directory C")
```

When a directory is nested within another directory, we refer to that as a **subdirectory**.

Now suppose we wanted to save our spreadsheet inside `Directory A`. One way of doing this would be to specify the full path, but an easier way is to specify the path relative to our working directory using the period notation.

```r
1  write.csv(df, file = "./Directory A/letters_3.csv")
```

```
Directory A
├──Directory B
│   └──Directory C
└──letters_3.csv
```

Moving further down a directory is a straightforward matter, but what if you wanted to move up the file path? For instance, suppose the working directory is located in `Directory C`.

```r
1  setwd("./Directory A/Directory B/Directory C")
```

Further suppose we wanted to save the spreadsheet in `Directory B`. To do this we would just represent moving "up" one directory with two periods (`..`).

```r
1  write.csv(df, file = "../letters_4.csv")
```

```
Directory A
├──Directory B
│   ├──Directory C
│   └──letters_4.csv
└──letters_3.csv
```

If you wanted to save the file two directories up, you just carry forward the logic.

```r
1  write.csv(df, file = "../../letters_5.csv")
```

```
Directory A
├──Directory B
│   ├──Directory C
│   └──letters_4.csv
├──letters_3.csv
└──letters_5.csv
```

And you can use the same logic reset the working directory back to the Desktop.

```
1  setwd("../../..")
```

It has to be said that, even if you are specifying a locations relative to the working directory, path addresses can still get quite long, for this reason it is often helpful to store directories as character strings that are easier to type and combine as needed. If we run ...

```
1  wd <- getwd()
2  dir_A <- "Directory A"
3  dir_B <- "Directory B"
4  dir_C <- "Directory C"
```

We can then use the `file.path()` function to, for instance, to produce a complete path directly to directory A, B, or C with minimal code that is easier to read.

```
1  file.path(wd, dir_A, dir_B, dir_C)
```

```
"C:/Users/Acheron/Desktop/Directory A/Directory B/Directory C"
```

So if we wanted to save our spreadsheet in `Directory C` using the full file path we could run ...

```
1  name <- file.path(wd, dir_A, dir_B, dir_C, "letters_6.csv")
2  write.csv(df, file = name)
```

```
Directory A
├──Directory B
│  ├──Directory C
│  │  └──letters_6.csv
│  └──letters_4.csv
├──letters_3.csv
└──letters_5.csv
```

Chapter 2

The tidyverse and the Basics of Plotting Data with R

T**HE** history of R can be split into two epochs. There was the time before the tidyverse, a period of primordial chaos that involved much personal sacrifice and necessary violence. Then there was the time of the tidyverse. The tidyverse is a set of mystical, yet cohesive, R packages brought forth by the sorcery of Hadley Wickham and his coven of arcane programmers (Wickham et al., 2019).

The tidyverse offers much order to the world of R, allowing common folk to bend and visualize data to their will in ways not previously possible to all but the most privileged. Though once viewed as complex and heretical (Muenchen, 2017), through collaboration and shared learning, the dark art of *tidy data* has continued to grow and flourish. While some sacrifice is inevitable (see Figure 2.1), there is no denying that the tidyverse offers an unparalleled path to power, efficiency, and dark beauty in the seemingly purposeless world of data analysis. For this reason we must begin our journey into the basics of data plotting with a brief discussion of it.

2.1 Worshiping at the alter of the tidyverse

As described by its website (https://www.tidyverse.org/), the **tidyverse** is an opinionated collection of R packages that share an underlying design philosophy. Each package can be installed individually, though most find it easiest to install every package within the scope of the tidyverse all at once.

Figure 2.1: An engraving depicting acolytes of the tidyverse burning live sacrifices, captive within a large wicker effigy, to appease their deities (Pennant, 1784).

```r
install.packages("tidyverse")
library(tidyverse)
```
```
-- Attaching core tidyverse packages -- tidyverse 2.0.0 --
 dplyr     1.1.4      readr     2.1.5
 forcats   1.0.0      stringr   1.5.1
 ggplot2   3.5.1      tibble    3.2.1
 lubridate 1.9.3      tidyr     1.3.1
 purrr     1.0.2
-- Conflicts ------------------- tidyverse_conflicts() --
 dplyr::filter() masks stats::filter()
```

```
dplyr::lag()    masks stats::lag()
Use the conflicted package to force all conflicts to
become errors
```

While the above code installs all the packages, running `library(tidyverse)` only loads the the nine "core" packages: *ggplot2*, *dplyr*, *tidyr*, *readr*, *purr*, *tibble*, *stringr*, *forcats*. Other tidyverse packages, such as *readxl*, will need to be loaded separately using the `library()` function.

Speaking for the beginner, it will be noticed that when the tidyverse is loaded, not only is there a confirmation of what packages (and their versions) have been loaded, but there is also a list of "conflicts" displayed in the output.[1] For instance, two functions from the *dplyr* package, `filter()` and `lag()`, have the same name as pre-existing functions within R and, when you load a package with a conflict like this, precedence is always given to the most recently loaded package. This means, when you use the `filter()` function for example, R is going to use the version belonging to *dplyr*, not the original version that was a part of base R's *stats* package (which is pre-loaded each time you use R). Though you can still use that original version in the following manner: `package_name::function_name()`. For example, `stats::filter()`.

As a whole, the tidyverse will not solve all your problems, but it will come damn close. Admittedly, and this is particularly true for beginners, much of what the tidyverse offers will not be needed in your daily programming rituals, but will come in handy when least expected.

2.2   Plotting with R

A core component of any GOOD DATA ANALYSIS obviously involves visualizing your data. As you progress through the various topics in this book, specific types of plots and their uses will be discussed in detail; however, for the time being, it will be helpful to get an intuitive sense of how plotting works with R generally. Thus, what follows in this section is intended to help you understand the logic of plotting with R. The goal at this point is not to make you an expert; rather, it is to provide beginners with a base level of knowledge that can be built off of.

By itself, base R comes with a stock set of functions for plotting data. To illustrate we can run the following code to produce a nice looking histogram ...

```
1   x <- rnorm(10000)
2   hist(x)
```

In the case of the above code, the function `rnorm()` is just generating 10,000 random values.[2] The function `hist(x)`, is simply plotting those values as a histogram. Running the code should generate an output similar to what you see below.

---

[1]Most packages will not display this information for you quite so nicely as the tidyverse does, so pay attention to any messages you receive using the `library()` function.

[2]The random values are technically coming from a "standard normal" distribution (hence the "norm" in `rnorm`), but don't worry about that for now.
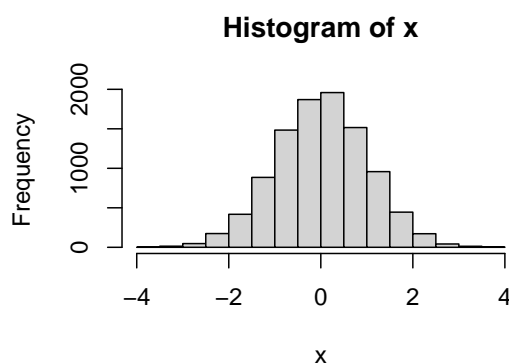
Figure 2.2: An example of base R's plotting functions.

R's base plotting functions offer a convenient means of producing simple quality plots and can be very efficient when working with **univariate data** or **bivariate data**. That is, data which consists of only one (uni) or two (bi) variables; however, most research concerned with analyzing **multivariate data**. That is data with more than two variables. Each additional variable adds ever increasing amounts of complexity and nuance to your data and, by extension, the plots you use to visualizing those data. The stock set of plotting functions R offers can accommodate these more complex scenarios; however, that level of accommodation is heavily dependent on the users proficiency with R. For this reason, this book will adopt the practice of ignoring R's base plotting functions, and instead rely on well-known R package called *ggplot2* which is among the most venerated portions of the tidyverse.

The *"gg"* in *ggplot2* stands for "grammar of graphics" and provides users with a logical framework for the construction of plots within R. The term "grammar" here is likely to conjure up long forgotten traumas of boring English and Language Arts lessons, but do not fear, the use of the term grammar is really just to emphasize that *ggplot2* is constructed in a way that allows users to build plots of various kinds in a consistent and efficient manner that is easily tailored to their specific needs. This is in contrast to how plotting in software works generally, where you are frequently stuck trying to fit a square peg (your data) into a circular hole (the software's narrow conception of how data should be presented).

The easiest way to understand how *ggplot2* works is to simply dive in an use it. Along the way, we will also learn a little bit more about R and data manipulation. The first thing to do will be to ensure that *ggplot2* has been installed into our computer's library of packages and loaded so we can access its functions. As mentioned in section 2.1, if you have installed and loaded the tidyverse, this is already done, but if you chose not to do that,[3] *ggplot2* can be installed and loaded as a standalone package as well.

```
1   install.packages("ggplot2")
2   library(ggplot2)
```

2.2.1   An example data set: msleep

Before we can plot anything, we need something to plot. In addition to its large set of plotting functions, the *ggplot2* package also provides a few illustrative data sets.[4] We will work with the `msleep` data set, which provides a variety of measurements relevant to the sleep behaviour of a wide range of mammals. To access the

---

[3]Shame on you.

[4]Base R comes with a nice collection of data sets as well. To obtain a list you need only run the function `data()`. To obtain the list of data sets for `ggplot2` you need only include the package name as an argument in this function: `data(package = "ggplot2")`

data you need only run the code `msleep`, which will output a 83 × 11 data frame.[5] Given the limited space available in the console window, the data frame is going to be truncated substantially. Thus, if you would like to view the entire data set, you can utilize R's `View()` function, which will display the data in a separate spreadsheet style window.

```
1  msleep # print data to console
2  View(msleep) # view the data in a spreadsheet-style window
```

```
       name                      genus       vore    order        conservation  sleep_total
1      Cheetah                   Acinonyx    carni   Carnivora    lc                   12.10
2      Owl monkey                Aotus       omni    Primates                          17.00
3      Mountain beaver           Aplodontia  herbi   Rodentia     nt                   14.40
4      Greater short-tailed shrew Blarina    omni    Soricomorpha lc                   14.90
5      Cow                       Bos         herbi   Artiodactyla domesticated          4.00
6      Three-toed sloth          Bradypus    herbi   Pilosa                            14.40
7      Northern fur seal         Callorhinus carni   Carnivora    vu                    8.70
8      Vesper mouse              Calomys             Rodentia                           7.00
9      Dog                       Canis       carni   Carnivora    domesticated         10.10
10     Roe deer                  Capreolus   herbi   Artiodactyla lc                    3.00
```

Table 2.1: First 10 rows and 6 columns of the `msleep` data

Table 2.1 shows the first 10 rows and 6 columns of `msleep` data. Looking more closely at the data, we can see a variety of variables (the column names) that are, for the most part, self explanatory. In this case, the column names represent distinct variables that have been measured and, particularly with larger data frames that cannot be adequately printed to the console, it is often useful to have R list out the name of each column. We can do this quite easily using the `names()` function.

```
1  names(msleep)
```

```
 [1] "name"         "genus"         "vore"
 [4] "order"        "conservation"  "sleep_total"
 [7] "sleep_rem"    "sleep_cycle"   "awake"
[10] "brainwt"      "bodywt"
```

Now, while the names of each column are self-explanatory, the elements of each column are perhaps less so. For instance, in the `$sleep_total` column, are we looking at values in minutes, hours, or days? In the `$conservation` column we can see a number of abbreviations such as `lc`, `nt`, `vu`, and so on. What do we make of those? A good starting point for answering these questions is to check the documentation associated with the data set, which all CRAN packages required to include. This can be accessed in the usual way with a `?`

```
1  ?msleep
```

Inspecting the documentation, we can see that `$sleep_total` is given in hours and that the column `$conservation` indicates "the conservation status of the animal." Admittedly, concerning this latter column, that does not tell us too much, but it does at least give us a starting point for understanding what those values might represent. In all likelihood, we are seeing abbreviations for the IUCN's (International Union for Conservation of Nature) species ranking.

- `lc` = Least Concern
- `nt` = Near Threatened
- `vu` = Vulnerable
- `en` = Endangered
- `cd` = Conservation Dependent

---

[5]Technically we are looking at a "tibble", which is the "tidyverse's" own take on a data frame. For our present purposes though, this is a distinction without a difference.

Using a **scatter plot** as a basic starting point, we will plot the relationship between the variables body weight (kg) and sleep total (hours). These are represented by the columns `$bodywt` and `$sleep_total` respectively.

2.3   Adding layers

ggplot2 constructs plots by adding visual layers on top of one another. The first layer is the grid upon which our scatter plot's points will appear. To generate this first layer we can simply type ...

```
1   ggplot(data = msleep, aes(x = bodywt, y = sleep_total))
```
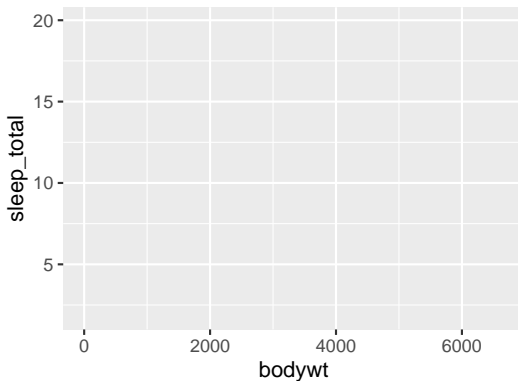

Looking at the `ggplot()` function we typed, we can see that the argument `data` tells *ggplot2* where the data is coming from - in this case it is coming from the `msleep` data frame. The `x` and `y` arguments are telling *ggplot2* what variables/columns should be mapped to the x and y axis respectively. Notice that, not only has *ggplot2* labelled the axis accordingly, but it has also given them scales that correspond to size of the values found in both columns.

Next we will, quite literally, add (`+`) a layer of points on top of this by typing `+ geom_point()`. The term "geom" here is just an abbreviation for "geometric object", and points are one of many different types of geometric object *ggplot2* recognizes.

```
1   ggplot(data = msleep, aes(x = bodywt, y = sleep_total)) +
2     geom_point()
```
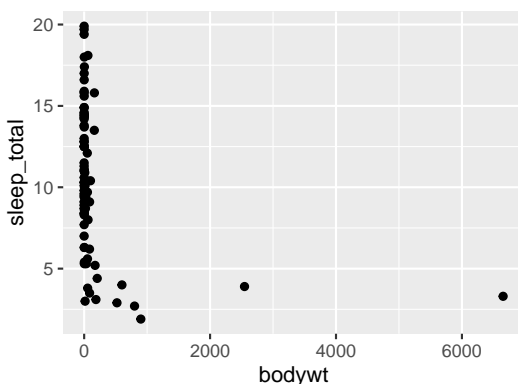

At this juncture, it is worth taking a moment to talk about how this code we have written has been organized. Here we placed `geom_point()` on a new line and indented it. This was not something we strictly had to do. We could have put everything on a single line like so ...

```
1  ggplot(data = msleep, aes(x = bodywt, y = sleep_total)) + geom_point()
```

But, particularly as we add more customization to the plot, this style of writing becomes hard to read. The (tidyverse's) R style guide recommends that no line of code exceed 80 characters, which is the advice most of the R community adheres to. In fact, Rstudio can be configured to display a margin representing the 80 character limit: (*Tools → Global Options → Code → Display*). To ensure that you do not exceed limit with larger blocks of code, it is worth remembering that you can always move portions of code to a new line after a comma, operator, or unclosed parentheses. The indentation we used is purely to guide the eye in recognizing that `geom_point()` belongs to a larger block of code.[6]

2.3.1 Inspecting potential outliers

At present, the plot does not look like much. There are numerous points scattered between 0 and 1000, and a couple of very extreme points beyond which are skewing the x-axis scale and making the majority of the data difficult to visualize. Given how rare and extreme these two values appear, we should inspect them to ensure that they are not errors within the data set (i.e., ensure that there is not a 2500 kg mouse, bird, or other such abomination in our data set). To accomplish this, most people will instinctively try to scan the data frame's 83 rows one by one with their eyes. Obviously, that strategy will be slow, inefficient, and highly prone to error. A better strategy is to have R isolate these values using the `filter()` function which is part of the tidyverse's *dplyr* package.[7] We simply give the function our data frame, and then specify a logical rule to subset by. In this case we will tell the function to show us all the rows that have a body weight greater than 2000.

```
1  filter(msleep, bodywt > 2000)
```

```
# A tibble: 2 × 11
  name             genus     vore  order        conservation sleep_total
  <chr>            <chr>     <chr> <chr>        <chr>               <dbl>
1 Asian elephant   Elephas   herbi Proboscidea  en                    3.9
2 African elephant Loxodonta herbi Proboscidea  vu                    3.3
# 5 more variables: sleep_rem <dbl>, sleep_cycle <dbl>, awake <dbl>,
# brainwt <dbl>, bodywt <dbl>
```

A quick glance at the output reveals that these two points represent the Asian and African elephant respectively. Thus, while these values are quite extreme and do not seem to be terribly representative of the data as a whole, they are not mistakes and therefore should remain in the data set. However, this begs the question, how do we visualize this data adequately with such odd scaling?

2.3.2 Logarithms

A common strategy in cases like this where larger values tend to become more and more extreme (i.e., exhibit some kind of exponential growth) is to plot the logarithm of the values. As a refresher of high school mathematics, logarithms are essentially exponents in reverse. For example:

$$10^3 = 10 \times 10 \times 10 = 1000$$

A *base-10* logarithm simply undoes this process by stating how many 10s it takes to create 1000.

$$\log_{10}(1000) = 3$$

A *base-2* logarithm asks: how many 2s are required to create 1000?

---

[6]While the R programming language allows users to indent code with reckless abandon, some programming languages, such as *Python*, require it to be used in very specific ways.

[7]Base R has a (more or less) equivalent function `subset()` that we could use as well. There are reasons for preferring `filter()`, but in this context there is no advantage to using either.

$$\log_2(1000) \approx 9.966$$

Thus, $2^{9.966} \approx 1000$.

A *natural* logarithm uses a base denoted as e (Euler's Number), which is approximately 2.71828.

$$\log_e(1000) \approx 6.908$$

Base-10, base-2, and natural logarithms represent the most widely used types of logarithms,[8] but you can technically use any base you desire. As seen below, the use of logarithms in R is very straightforward.

```
1   log10(1000) # Base-10 function
2   log2(1000) # Base-2 function
3   log(1000) # Natural log
4   log(1000, base = 666) # Pick your own base
```

```
[1] 3
[1] 9.965784
[1] 6.907755
[1] 1.062521
```

A base-10 logarithm is generally considered the most intuitive so we will use that. There are various ways to incorporate a logarithmic scale on our plot's axis, but perhaps the safest way is to simply add a new column of $\log_{10}$ values to our dataframe and plot that instead of the standard `$bodywt` column.

```
1   # Add new column of log bodywt values.
2   msleep$bodywt_log10 <- log10(msleep$bodywt)
3
4   # Re-plot the data
5   ggplot(msleep, aes(x = bodywt_log10, y = sleep_total)) +
6     geom_point()
```
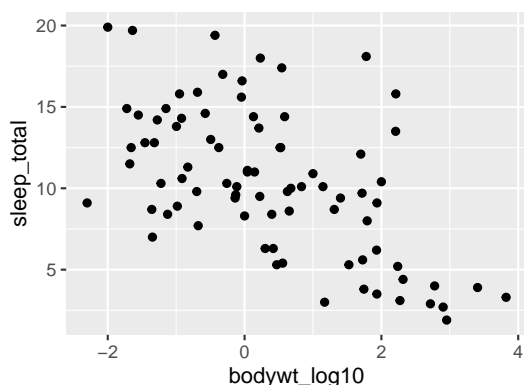

2.4   Aesthetics

Geometric objects in *ggplot2*, like the point geom, all have various traits, like their size, shape, and colour that can be customized. In the language of *ggplot2*, these are referred to as **aesthetics**. For example, we can customize the points in the following way ...

---

[8]For clarity and consistency the natural logarithm of 1000 has been written $\log_e(1000)$, but it is common practice to identify natural logarithms using "ln". E.g., $\ln(1000) \approx 6.908$.

Box 2.1: An alternative way to scale

In the previous example, the logarithm was applied by creating a new column of x-axis values and plotting that. However, this means that, if you want to interpret the numbers in their original units, you need to calculate 10^x, which can be annoying.

An alternative strategy would be to keep the `$bodywt` column as is and just scale the plot's axis itself to increment logarithmically which `ggplot2` will do straightforwardly.

```
1  ggplot(msleep, aes(x = bodywt, y = sleep_total)) +
2    geom_point() +
3    scale_x_continuous(trans = "log10")
```
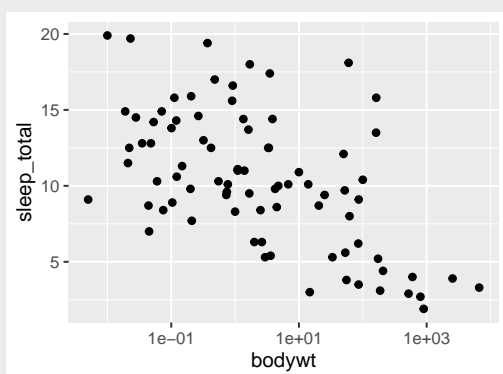
The advantage of this method is you can look at a point's value on the x-axis and know immediately that it corresponds to a weight of x kg. The drawback is you may end up with excessively small or large values on the axis, hence the *scientific notation* you see in the plot.

```
1  ggplot(msleep, aes(x = bodywt_log10, y = sleep_total)) +
2    geom_point(size = 3,
3               shape = 4,
4               colour = "blue",
5               stroke = 1.5)
```
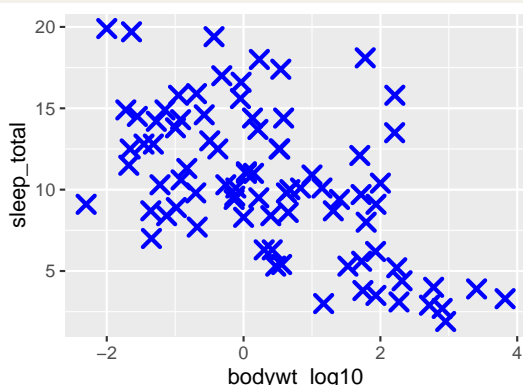
With a bit of experimentation, it should be apparent how the arguments `size` and `stroke` work in the above example; however, the `shape` and `colour` arguments are slightly less intuitive.[9] R comes with a variety of point shapes (technically called "plotting characters" or "**pch**" symbols for short) that are denoted by numbers. The various possibilities are depicted in Figure 2.3. In this case, number 4 is an ×. Notably, the

---

[9]If you accidentally omit the "u" when typing "colour," *ggplot2* will still understand what you mean, even though it isn't correct English.

last five plotting characters (21 through 25) incorporate both a `colour` aesthetic for their edges and a `fill` aesthetic. All the other symbols only require a `colour` aesthetic to be specified.

```
ggplot(msleep, aes(x = bodywt_log10, y = sleep_total)) +
  geom_point(
    size = 3,
    shape = 25,
    colour = "black",
    stroke = 1.5,
    fill = "red"
  )
```


The plotting characters shown in Figure 2.3 are just a few of the options available. For instance, by using values ranging between 32 and 127, you can display a variety of ASCII characters. Additionally, you can specify a particular character instead of providing a numeric value, e.g., `shape = "&"`.
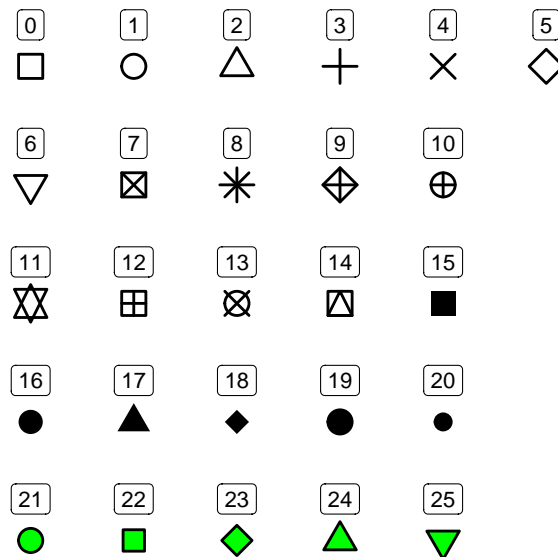

Figure 2.3: R Plotting Characters

In the above examples we specified a desired colour by typing the name of a primary colour, but we are not limited to just using primary colours. R comes with a built in set of 657 differently named colours. You can obtain the full list of colour names by running `colors()`. R also has a built-in demo of these colours you can run to get a visual representation of each. Simply run the command `demo("colors")`.

Alternatively, instead of typing a colour name, you can use a hexadecimal value (also referred to as a

Box 2.2: Hexadecimal Notation for Colours

Hexadecimal values are simply numbers that use a base-16 counting method. In other words, in the world of hexadecimals, there are 16 different numbers that are used to count with, instead of the typical 10 numbers (0:9), you were probably raised to use. These are

Decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Hexadecimal	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Because of their larger base, a single hexadecimal digit can store more information than a conventional base-10 digit can. For instance, if a computer stores various gradations of the colour red using just two digits, that only allows for 100 (10×10) different reds. Using hexadecimals you can have 256 (16×16) reds, with just two digits. Thus, if a colour is some combination of red, green, and blue, and each is stored using two hexadecimal digits that gives you $256^3 = 16,777,216$ colours as opposed to the meagre $100^3 = 1,000,000$ you would have using the inferior base-10 counting method.

To use hexadecimals to represent colour, two digits are assigned to red (RR), green (GG) and blue (BB), in that order like so `"#RRGGBB"`. Smaller values are brighter, and larger values are darker. Consequently, black is represented as `"#000000"` and white is represented as `"#FFFFFF"`. Thus, if you want the "purest" red, you would input `"#FF0000"`, the purest green would be `"#00FF00"`, and the purest blue would be `"#0000FF"`.

"hex code" or "hex value") that represents a specific colour. For example, the hex value `"#FFC0CB"` represents the colour pink. Hexadecimal values offer the user a lot of nuance when it comes to colour selection and, in most cases, the simplest way of finding an appropriate hex value is to consult one of the many websites devoted to colour codes and colour theory (i.e., do an internet search). However, if you would like to understand the theory behind hex codes and why they are used, see Box 2.2.

2.4.1   Aesthetics by variable

In the above example the aesthetic changes we made to plot effected all of the points. In the language of *ggplot2*, the aesthetics were mapped to all of the points. However, it is often necessary to visually break up the points according to one of the other variables in your data. For instance, we could colour the points in our plot according to the categories in the data's `$vore` column.

```
ggplot(msleep, aes(x = bodywt_log10, y = sleep_total)) +
    geom_point(size = 3, aes(colour = vore))
```
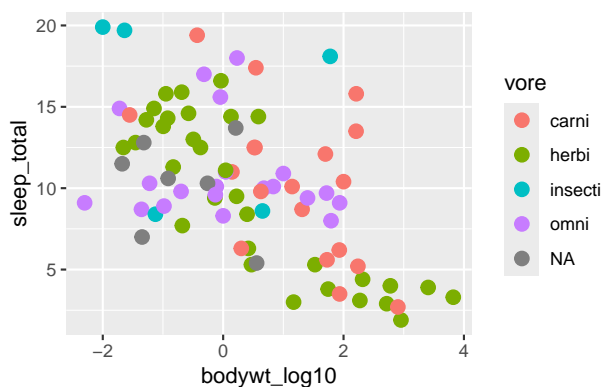

Notice that the plot's legend shows an "NA" category. This is because there are `NA` values found within the `$vore` column (run `msleep$vore` to see them). Thus, the legend's "NA" category represents values that

we have body weight and sleep total information for, but we do not know what those animals diet consists of and therefore cannot categorize them properly.[10] So instead of referring to this category as "NA", we could refer to these as "unknown." All we need to do is change the `NA` values in the data frame's `$vore` column to character values that read `"unknown"`. This can be done simply by using the `ifelse()` function, which tests a statement you write. If that statement is true, it produces a value you have specified, if it false, then it produces an alternative value you have specified. In other words, it works like this:

`ifelse(test, true result, false result)`.

In this case, we want to test *if* the value in each row *is* an `NA` value or not. Recall that the function `is.na()` tells us whether the value of a vector is an `NA` value or not.

```
is.na(msleep$vore)
```
```
 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE
 [8]  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
[15] ...
```

Thus, we can use that as the "test" in the `ifelse()` function.

```
ifelse(is.na(msleep$vore), "unknown", msleep$vore)
```
```
 [1] "carni"   "omni"    "herbi"   "omni"    "herbi"   "herbi"   "carni"
 [8] "unknown" "carni"   "herbi"   "herbi"   "herbi"   "omni"    "herbi"
[15] "omni"    "omni"    "omni"    "carni"   "herbi"   "omni"    "herbi"
[22] "insecti" "herbi"   "herbi"   "omni"    "omni"    "herbi"   "carni"
[29] "omni"    "herbi"   "carni"   "carni"   "herbi"   "omni"    "herbi"
[36] "herbi"   "carni"   "omni"    "herbi"   "herbi"   "herbi"   "herbi"
[43] "insecti" "herbi"   "carni"   "herbi"   "carni"   "herbi"   "herbi"
[50] "omni"    "carni"   "carni"   "carni"   "omni"    "unknown" "omni"
[57] "unknown" "unknown" "carni"   "carni"   "herbi"   "insecti" "unknown"
[64] "herbi"   "omni"    "omni"    "insecti" "herbi"   "unknown" "herbi"
[71] "herbi"   "herbi"   "unknown" "omni"    "insecti" "herbi"   "herbi"
[78] "omni"    "omni"    "carni"   "carni"   "carni"   "carni"
```

When you run the above code, the `ifelse()` function scans each row of the `$vore` column and evaluates whether `is.na(msleep$vore)` is `TRUE`. If it is true, it replaces the existing `NA` value with `"unknown"`. However, if it `FALSE`, it leaves it as the original value (this is why we wrote `msleep$vore` after the second comma). The end result is a vector of values that we can use to replace the existing `$vore` column with.
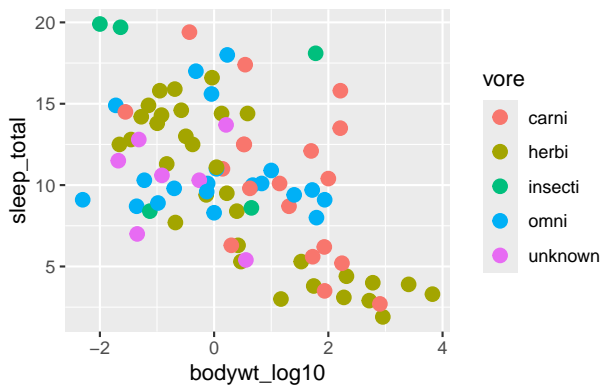
```
msleep$vore <- ifelse(is.na(msleep$vore), "unknown", msleep$vore)
```

Now, when we re-plot the graph, we get something much more sensible ....

```
ggplot(msleep, aes(x = bodywt_log10, y = sleep_total)) +
    geom_point(size = 3, aes(colour = vore))
```

---

[10]To see the full list of animals who have a missing `$vore` value, you can run `filter(msleep, is.na(vore))`. This will show all the rows for which `is.na(vore)` evaluates to `TRUE`.

When plotting, it is usually inadvisable to *only* adjust the colour of your points because a sizeable portion of the population has some form of colour vision deficiency (a.k.a., colour blindness). And while there are "colourblind friendly" palettes we can use, there is no universal palette that works optimally for all cases of colour deficiency. Consequently, the best practice is to have each category be represented by a distinct shape.

```
ggplot(msleep, aes(x = bodywt_log10, y = sleep_total)) +
    geom_point(size = 3, aes(colour = vore, shape = vore))
```


2.5   Displaying trends

Notice that the data points appear to trend downward as you move from left to right on the x-axis. In other words, as body weight increases, you tend to see decreases in sleep total. By simply adding a second geom, called `geom_smooth()`, we can use a line of best fit to represent (i.e., model) this trend.
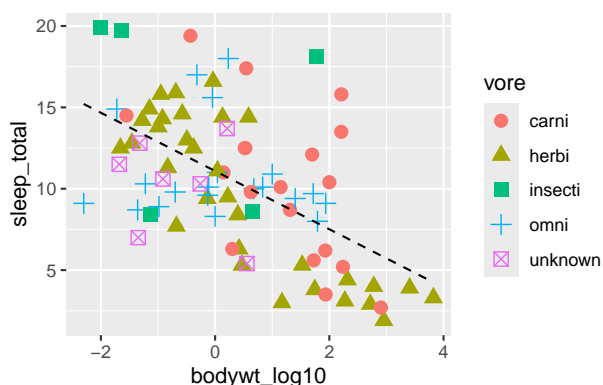
```
ggplot(msleep, aes(x = bodywt_log10, y = sleep_total)) +
    geom_point(size = 3, aes(colour = vore, shape = vore)) +
    geom_smooth()
```
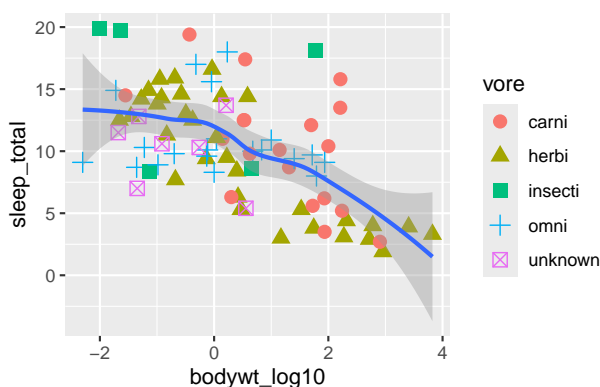
The shaded grey area represents the *standard error* and the line was drawn using a fancy smoothing method called *Local Polynomial Regression Fitting*, but we can use a more common regression line as well and modify various aspects of it just like we had done earlier using `geom_point()`.

```
ggplot(msleep, aes(x = bodywt_log10, y = sleep_total)) +
  geom_point(size = 3, aes(colour = vore, shape = vore)) +
  geom_smooth(
    method = "lm", se = FALSE,
    linetype = 2,
    linewidth = 0.5,
    colour = "black"
  )
```


This is where the versatility of the *ggplot2* begins to really shine. For instance, if we wanted to create a separate regression line for each category of `$vore` we can accomplish that by once again making use of the `aes()` function and "grouping" by `$vore`.

```
ggplot(msleep, aes(x = bodywt_log10, y = sleep_total)) +
  geom_point(size = 3, aes(colour = vore, shape = vore)) +
  geom_smooth(
    method = "lm",
    se = FALSE,
    colour = "black",
    linewidth = 0.5,
    aes(group = vore)
  )
```
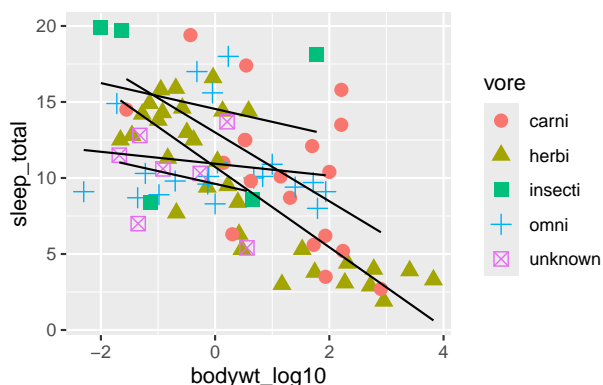

At present it is not clear which line applies to which category, but we could also have each regression line correspond to the colour mapped to `$vore`, and (in consideration of colour blindness) give each line a separate `linetype`.

```
1  ggplot(msleep, aes(x = bodywt_log10, y = sleep_total)) +
2    geom_point(size = 3, aes(colour = vore, shape = vore)) +
3    geom_smooth(
4      method = "lm",
5      se = FALSE,
6      linewidth = 0.5,
7      aes(colour = vore, linetype = vore)
8      )
```
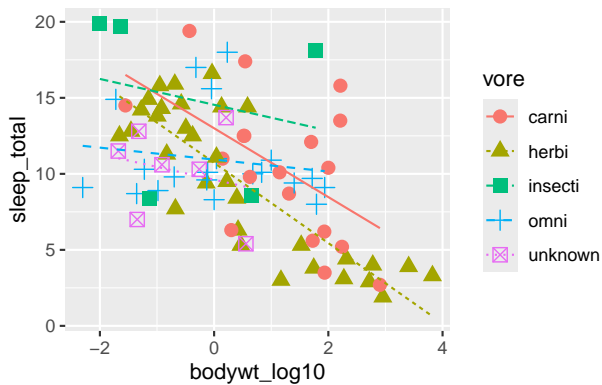

2.6 Facets

As interesting as our plot looks, it is becoming rather cluttered and difficult to visually parse. In situations like this, it is often helpful to split the plot up into separate facets (i.e., give each category its own graph). *ggplot2* makes this very easy with its `facet_wrap()` function.

```
1  ggplot(msleep, aes(x = bodywt_log10, y = sleep_total)) +
2    geom_point(size = 3, aes(colour = vore, shape = vore)) +
3    geom_smooth(
4      method = "lm",
5      se = FALSE,
6      linewidth = 0.5,
7      aes(colour = vore)
8    ) +
9      facet_wrap(~ vore)
```

You can interpret the small formula we wrote (`~vore`) as meaning "*plot as a function of vore.*"

Notice that now, the colour and shape aesthetics are providing redundant information with the facet labels. As a general rule, you want to avoid redundancy in your plots because additional visual elements might bias the viewer's eye in unpredictable ways. We can easily fix this by removing some of the aesthetics we added earlier, and we can also adjust the facets so that they are all on a single row by adding the argument, `nrow = 1` to our `facet_wrap()` function.

```
1  ggplot(msleep, aes(x = bodywt_log10, y = sleep_total)) +
2    geom_point(size = 3) +
3    geom_smooth(method = "lm", se = FALSE, linewidth = 0.5) +
4    facet_wrap(~ vore, nrow = 1)
```
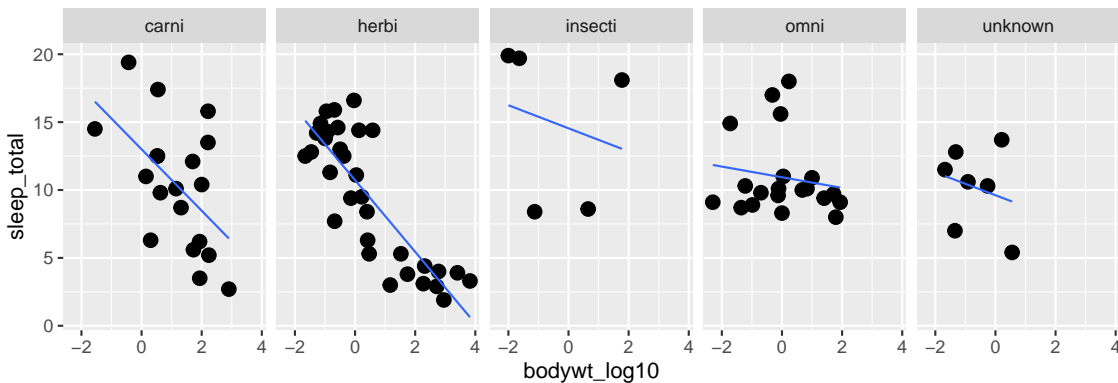

The default behaviour of `facet_wrap()` preserves the x and y axis scales across the facets, making them easy to compare. In most cases, this is a feature you do not want to override but it can be done (see the R documentation: `?facet_wrap`).

Particularly for beginners with R, it is difficult to impress how useful *ggplot2* is here. Using base R plotting functions to produce a comparable graph would be a considerably more complex process and require a heftier amount of code to be written, whereas ggplot does it all for us in four short lines.

To finish up the plot, we should adjust some of the labelling, save it, and then take a look at some other more advanced features of *ggplot2*.

2.7   Labels

To adjust the x and y axis titles we can simply use the functions `xlab()` and `ylab()`.

```
1   ggplot(msleep, aes(x = bodywt_log10, y = sleep_total)) +
2     geom_point(size = 3) +
3     geom_smooth(
4       method = "lm",
5       se = FALSE,
6       linewidth = 0.5
7     ) +
8     facet_wrap(~vore, nrow = 1) +
9     xlab("Log10(Body Weight kg)") +
10    ylab("Sleep Total (hrs)")
```
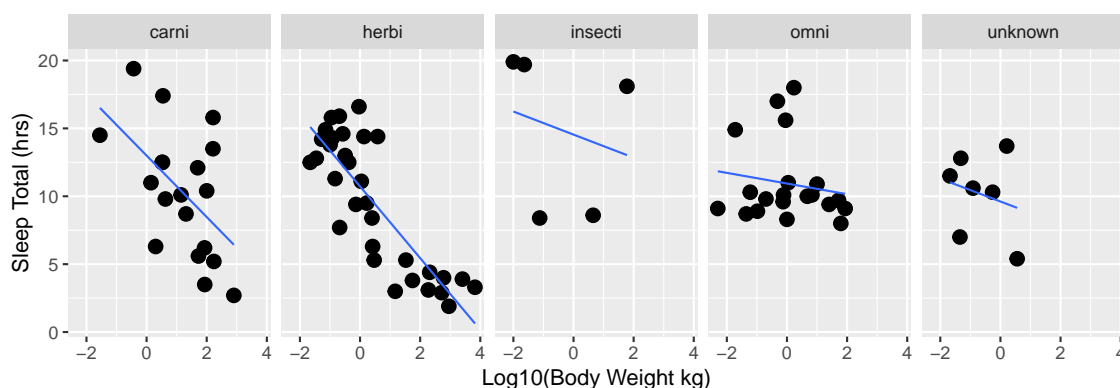
2.8   Saving the plot

Users of R studio will notice that in the *Plots* pane there is a button that can be used to "export" your plot. However, it is usually more efficient and useful to save the plot via written code, and there are different methods you could use to go about this. Since we are using *ggplot2* to create our graphs, the optimal strategy is to use the `ggsave()` function, which will save the last generated plot unless you tell it otherwise.

```
ggsave("msleep_plot.png", dpi = 300, units = "cm", width = 20, height = 7)
```

Running this code as is will save the plot to your *working directory* (see section 1.7 for more info about directories and saving files). Within the function, we have chosen to name our image file `"msleep_plot.png"`. The file extension you specify at the end of the file name here will dictate what type of image the plot is saved as. In this case, it will save as a .PNG (Portable Network Graphics) image file, which is a very standard type of image that most people and software are used to handling, though you could save it as other common formats as well (e.g., .JPG, .GIF, .TIFF, etc.). The argument `dpi` stands for "dots per inch" and specifies the resolution of the image. For publication quality plots it is generally recommended that you have a minimum resolution of 300 dpi. Anything less than that will likely produce very noticeable artifacting or fuzziness, particularly if the image has been resized or magnified. The last three arguments `units`, `width` and `height` allow you to specify the dimensions of your plot and should be relatively self-explanatory. If you wanted to, for instance, give the dimensions of your plot in millimeters you would specify `"mm"`, inches would be `"in"`, and so on.

2.8.1   Vector graphics vs. Raster graphics

The above code saved the plot as a .PNG which is a type of "raster" image, meaning it is an image composed of tiny coloured squares called pixels. The more pixels an image has, the more detail it can provide (i.e., the higher its resolution). The problem with using raster images though, is that resizing, stretching, and magnification has deleterious effects on their quality. For instance, the image below shows a small section of our 300 dpi graph magnified substantially.
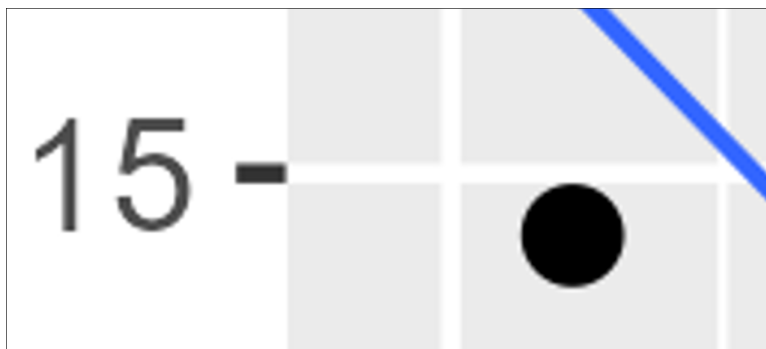
Figure 2.4: Artificating present on our 300 dpi raster image when magnified.

A close inspection reveals jaggedness on the blue line and general blurriness around the rest of the image's elements. In academic publications, manuscripts, and presentations, this is something you want to avoid because, while these problems may not be immediately noticeable at first glance, they can impact a person's sensation of the image and, by extension, their opinion of its creator. Moreover, imperfections like these can be exacerbated in the printing and publishing process.

Now you might think that a simple remedy would be to increase the dpi to a much higher value, but this is generally a strategy you want to avoid. There tends to be diminishing returns with resolution increases and anything beyond 300 dpi is not going to do much for you apart from ballooning the image's file size. The optimal strategy is to make use of something called a vector graphic.

Vector graphics are not really images in the traditional sense; rather, they are more akin to a set of instructions your computer uses to draw the image. Consequently, a vector-based image can be resized and magnified as much as you would like and it will never lose its quality. The drawback to vector graphics is that they do not work too well for highly detailed photographs (e.g., a forested landscape) and they are not always recognized by software. For instance, the most common types of vector you will encounter are .PDF, .SVG, and .EPS. Recent versions of Microsoft Word and PowerPoint will happily accommodate .SVG files, but if you are wanting to use a .PDF or .EPS, you will be out of luck. Correspondingly, Google Docs and Google Slides will not accept any type of vector graphic, which is doubly frustrating because these apps will also downscale the resolution of raster graphics you import. Libre Office's Writer and Impress applications will accept a .PDF image, but it converts it to a lower resolution raster graphic when it is imported. Despite these types of compatibility limitations, if you are able to use vector graphics then you should, because they will give your work a level polish other people are not likely to have.

To save a file as a vector graphic, the process is the same as before, we just need to modify the file extension and remove the `dpi` argument (because dpi has no meaning for vector graphics).

```
ggsave("msleep_plot.svg", units = "cm", width = 20, height = 7)
```

The above code saved the image as a .SVG (Scalable Vector Graphic) file. This is a commonly used image file in web design, meaning it will, by default, be most likely displayed within a web-browser when you open it.
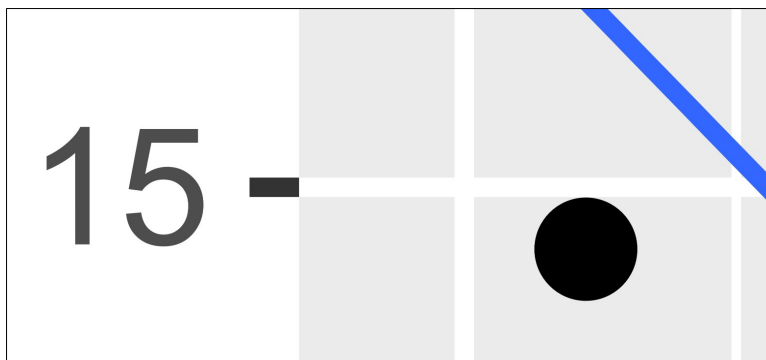
Figure 2.5: Magnification of a vector graphic.

2.9 Scales

A key concept in the "grammar" of *ggplot2* is that of scales. Scales control how data is mapped to different aesthetics. For instance, there are scales for position, colour, size, shape, linetype, and so on. When you map an aesthetic to a variable like we did above where we had mapped both colour and shape to the `$vore` column - e.g.,

```
...
geom_point(aes(colour = vore, shape = vore))
...
```
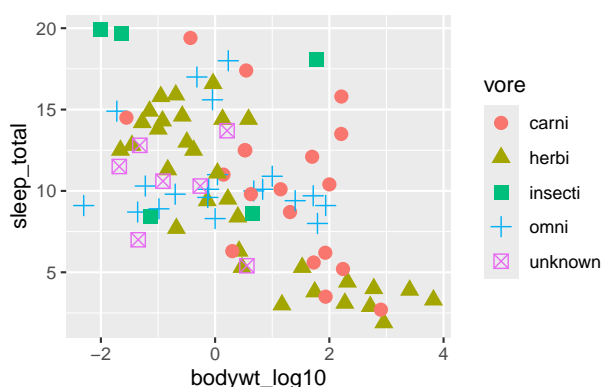

ggplot2 automatically chose which colours and shapes got applied to each category, but you can use functions to override these automatic mappings.

The function you use to make these overrides is going to be dictated by the aesthetic you are working with. For instance, to adjust the colour (or edge colour) of a point you could use `scale_colour_manual()`. If you wanted to adjust the shapes, you could use `scale_shape_manual()`. If you wanted to adjust the fill colour of something (e.g., the fill colour of points or the fill colour of bars on a graph), you could use `scale_fill_manual()`. Hopefully you can see the pattern.

2.9.1 Position Scales: Modifying the Axis Breaks

When we first created the grid on which we drew our points we actually mapped some aesthetics to do this. Specifically, we mapped the x and y aesthetics to the `$bodywt` and `$sleep_total` columns respectively. In other words, we had written:

```
1   ggplot(data = msleep, aes(x = bodywt, y = sleep_total))
```

When first mapping the x and y axes of a plot, *ggplot2* typically selects an appropriate sequence of values to display for each. These are what are referred to as axis *breaks* and, most of the time, *ggplot2*'s default scaling for the breaks is excellent. However, there are occasions where more customized scaling is necessary. In these situations, the following four functions are useful:

1. `scale_x_continuous()`
2. `scale_y_continuous()`
3. `scale_x_discrete()`
4. `scale_y_discrete()`

The above four functions allow you to easily modify what values appear on your axis; though, which one you use depends on whether your axis has a *continuous* or *discrete* **position scale**. Position scales control the location mappings of a plots visual elements.

In the case of the mammal sleep data we plotted, both the x-axis scale (body weight) and y-axis scale (sleep total) are *continuous* in nature. In other words, the axis values represent measured numeric values as opposed to categories. Another way of conceptualizing this continuous vs discrete distinction is to approach it from R's perspective. In this case, both axis represent *numeric* objects as opposed to *character* objects. Thus, for the purpose of plotting, they are treated as a continuous scale.

```
1   mode(msleep$bodywt_log10) # x-axis
2   mode(msleep$sleep_total) # y-axis
```

```
[1] "numeric"
[1] "numeric"
```

If we had, for instance, plotted a categorical variable on the x-axis (e.g., the conservation status of the animal) then the x-axis would be discrete while the y-axis remains continuous (we will see an example of this later on).

To customize the breaks on our axis, we simply need to add one of the aforementioned functions to our plot's code and provide a vector of values we want to see displayed using the argument `breaks`. For instance, if we want the x-axis to only display the numbers 1, 2, and 3, we would add `scale_x_continuous(breaks = c(1,2,3))` to our code.
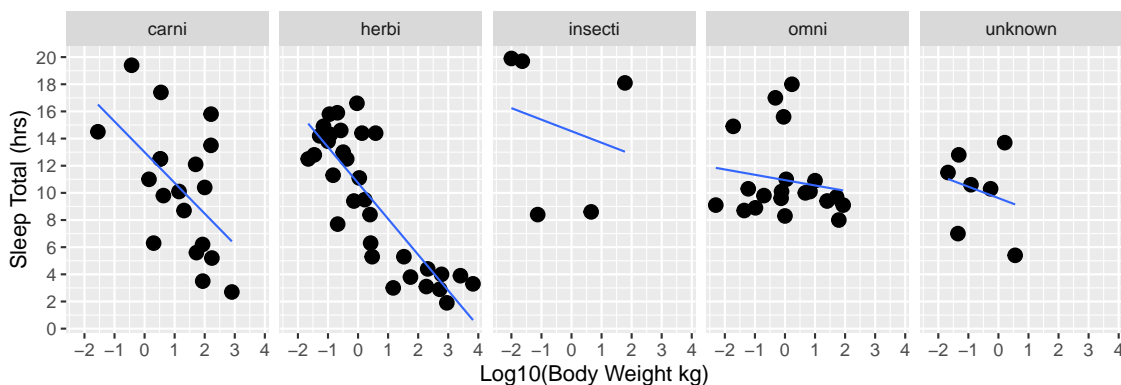
```
1    ggplot(msleep, aes(x = bodywt_log10, y = sleep_total)) +
2      geom_point(size = 3) +
3      geom_smooth(
4        method = "lm",
5        se = FALSE,
6        linewidth = 0.5
7      ) +
8      facet_wrap(~vore, nrow = 1) +
9      xlab("Log10(Body Weight kg)") +
10     ylab("Sleep Total (hrs)") +
11     scale_x_continuous(breaks = c(1,2,3))
```
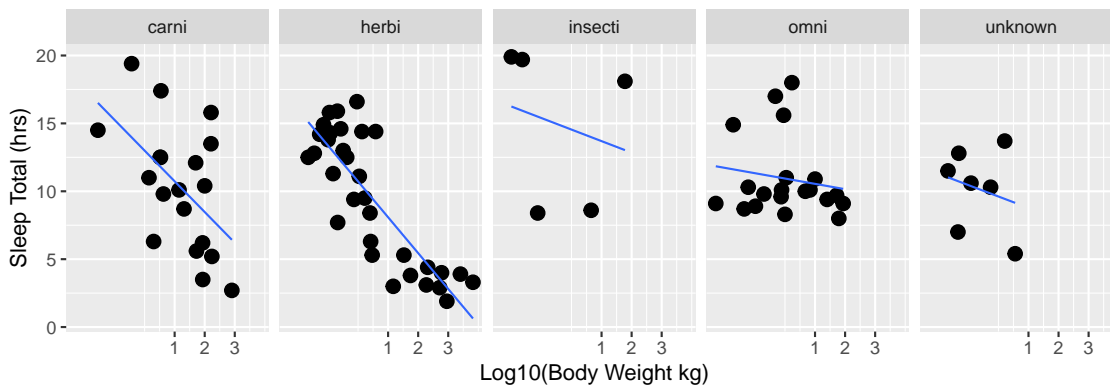
In general, the best practice is not to specify values individually, but rather specify a sequence using the `seq()` function we learned about in Chapter 1 (see section 1.4.7). For instance, we could have the x-axis increment by 1s and the y-axis increment by 2s.

```r
ggplot(msleep, aes(x = bodywt_log10, y = sleep_total)) +
  geom_point(size = 3) +
  geom_smooth(
    method = "lm",
    se = FALSE,
    linewidth = 0.5
  ) +
  facet_wrap(~vore, nrow = 1) +
  xlab("Log10(Body Weight kg)") +
  ylab("Sleep Total (hrs)") +
  scale_x_continuous(breaks = seq(-2, 4, 1)) +
  scale_y_continuous(breaks = seq(0, 20, 2))
```


The four scale functions above can achieve a lot more than what is being shown here, but for most use cases, this basic adjustment of the axis breaks will be their primary purpose.

2.9.2   Modifying the Axis Range

In addition to axis break adjustment, the range of the axis will often require customization as well. To achieve this, the best practice is usually to use the function `coord_cartesian()`. To illustrate with some absurd values, we could have the x-axis span between -2 and +1 and have the y-axis span between -5 and $+10$.
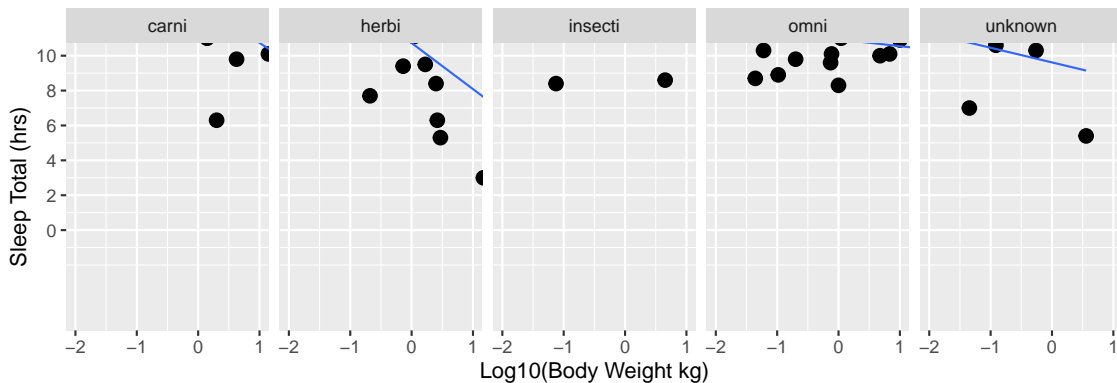
```r
ggplot(msleep, aes(x = bodywt_log10, y = sleep_total)) +
  geom_point(size = 3) +
  geom_smooth(
```

```
4        method = "lm",
5        se = FALSE,
6        linewidth = 0.5
7      ) +
8      facet_wrap(~vore, nrow = 1) +
9      xlab("Log10(Body Weight kg)") +
10     ylab("Sleep Total (hrs)") +
11     scale_x_continuous(breaks = seq(-2, 4, 1)) +
12     scale_y_continuous(breaks = seq(0, 20, 2)) +
13     coord_cartesian(xlim = c(-2, 1), ylim = c(-5, 10))
```


Note that, while the y-axis goes as low as -5, it does not show breaks below 0 because of how the `breaks` argument in `scale_y_continuous()` were set.

At this point it is worth offering a disclaimer. Within the position scale functions mentioned earlier (i.e., `scale_x_continuous()` and `scale_y_continuous()`, there is an argument called `limits` that will allow you to set the range of the scale in a manner similar to the `coord_cartesian()` function. Additionally, *ggplot2* also has two other functions, `xlim()` and `ylim()`, that will do the same. However, setting the limits of your plot with these these arguments and functions is best avoided because they will remove data falling outside of those specified limits. This can result in problems if your plot's code is performing some type of statistical calculation. For instance, if you remove lines 12 and 13 in the above script and add `ylim(-2, 1)` you will be confronted with a very nasty error message, telling you (among other things) that ...

```
Warning messages:
1: Removed 83 rows containing non-finite outside the scale range
(`stat_smooth()`).
```

This occurs because values in our data falling outside of -2 and $+1$ are not recognized anymore, but *ggplot2* needed those values to calculate that blue regression line using the `geom_smooth()` function. Thus, the moral of the story is, if you need to "zoom-in" or "zoom-out" on a plot, use `coord_cartesian()`. Do not be tempted by those other options.[11]

2.9.3   Colour Scales: Modifying Colour Mappings

Similar to how *ggplot2* automatically selected a scaling for the breaks on the x and y axis, it also automatically selected various colours to use when we mapped colour to the *vore* column. Moreover, the distinction between *continuous* and *discrete* scaling applies just as much to colour as it does position. As illustrated in Figure 2.6 and 2.7, discrete colour scales are usually represented with a colour gradient and discrete scales are represented

---

[11]Readers are probably wondering "*what use does removing data outside of the limits serve? It seems like it would only ever cause more problems than it solves (especially if you are unaware it is happening).*" And to that I say, yes.

by distinct colours (like in a box of crayons). While it is possible to do this, you usually do not want to use a gradient to represent distinct categories because it makes the categories difficult to discriminate visually. For instance, the `$vore` column we mapped to the colour aesthetic earlier contained distinct non-numeric categories (e.g., carni, herbi, insecti, and so on); thus, a colour palette such as that seen in Figure 2.7 would be much more appropriate than Figure 2.6.

Figure 2.6: Example of a continuous colour scale (i.e., a colour gradient).

Figure 2.7: Example of a discrete colour scale (a.k.a. a qualitative palette.)

2.9.4   Discrete Colour Scales

To illustrate the use of discrete colour scales lets create a simple plot we can experiment with.

```
1  ggplot(msleep, aes(x = bodywt_log10, y = sleep_total)) +
2    geom_point(size = 3, shape = 21, stroke = 2)
```
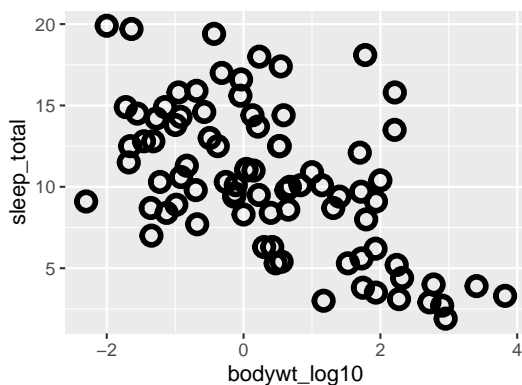

First we will map the edge colour to the column `$vore`.

```
1  ggplot(msleep, aes(x = bodywt_log10, y = sleep_total)) +
2    geom_point(
3      size = 3, shape = 21, stroke = 2,
4      aes(colour = vore)
5    )
```

Then, to override these colours we can simply use `scale_colour_discrete()` and input a vector of colours.

```
1   ggplot(msleep, aes(x = bodywt_log10, y = sleep_total)) +
2     geom_point(
3       size = 3, shape = 21, stroke = 2,
4       aes(colour = vore)
5     ) +
6     scale_colour_discrete(type = c("red", "blue", "green", "purple", "orange"))
```


The same effect can be achieved by using `scale_colour_manual()` instead.
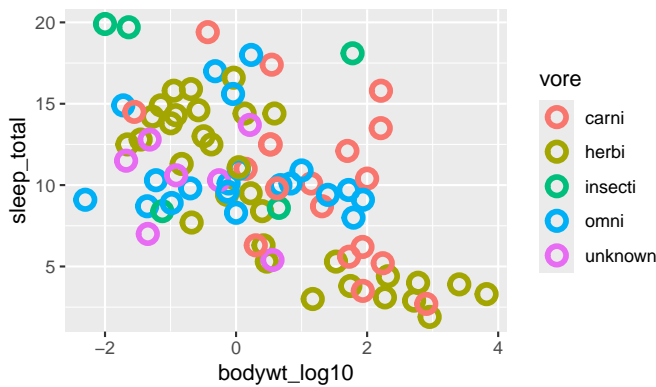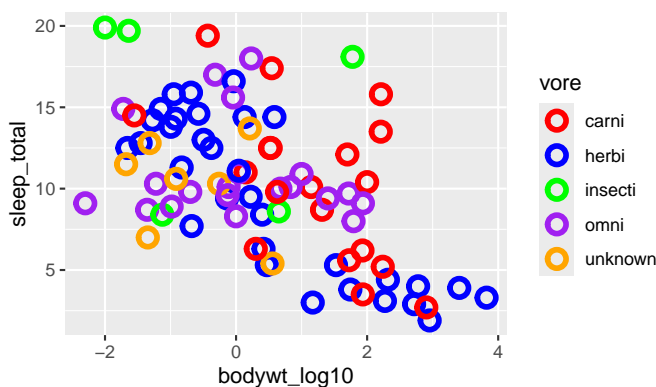
```
1   ...
2   scale_colour_manual(values = c("red", "blue", "green", "purple", "orange"))
```

However, the advantage to using `scale_colour_discrete()` is you are not limited by the number of categories in your palette. This means you can create a bigger colour palette and *ggplot2* will only use as many colours as needed. By contrast, if you use `scale_colour_manual()`, you have to ensure that you specify the same amount of colours as there are categories. To illustrate, we can create a palette with eight colours, but *ggplot2* will only use the first six.

```
1   # Create a colour palette
2   palette <- c(
3     "#000000", "#DF536B", "#61D04F", "#2297E6", "#28E2E5", "#CD0BBC", "#F5C710",
4     "#9E9E9E"
5   )
6
7   # Use that palette in your plot
8   ggplot(msleep, aes(x = bodywt_log10, y = sleep_total)) +
9     geom_point(
10      size = 3, shape = 21, stroke = 2,
```

```
11      aes(colour = vore)
12    ) +
13    scale_colour_discrete(type = palette)
```
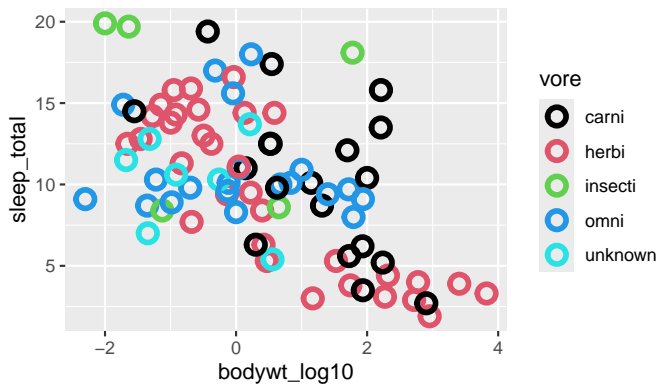

Notice that we are using pch 21 as our shape. Recall that this shape takes both an edge and fill colour (see Figure 2.3). At present, we have not specified a fill colour, so the points are hollow. However, instead of modifying the edge colour like we have been doing, we could modify the fill colour of the points and just keep the edges black.

```
1   ggplot(msleep, aes(x = bodywt_log10, y = sleep_total)) +
2     geom_point(
3       size = 3, shape = 21, stroke = 1, colour = "black",
4       aes(fill = vore)
5     ) +
6     scale_fill_discrete(type = palette)
```
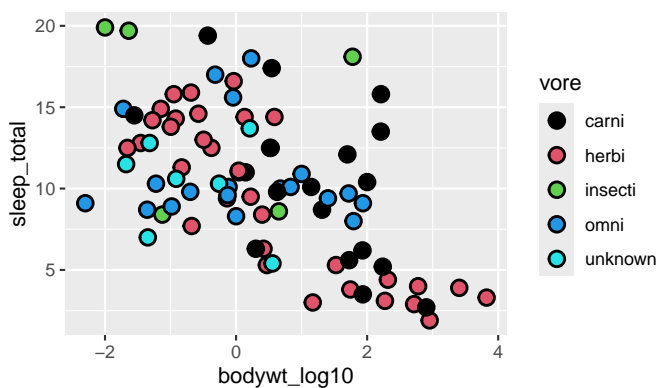

Notice where the important changes have taken place in the code. We have moved the `colour` aesthetic outside of the `aes()` function. This means a single `colour` (black) will now be mapped to all the points. We have also mapped the `$vore` column to the `fill` aesthetic inside of `aes()` and, for that reason, now specify `scale_fill_discrete()` to modify the colour options. In other words, we are now adjusting the *fill* colour, not the point/edge colour.

Pre-Existing Discrete Colour Palettes

Until now, we have been specifying our own custom colour palettes; however, base R contains a variety of pre-existing palettes we can make use of. To obtain the list you can simply run the following:

```
1   palette.pals()
```

```
 [1] "R3"            "R4"             "ggplot2"        "Okabe-Ito"
 [5] "Accent"        "Dark 2"         "Paired"         "Pastel 1"
 [9] "Pastel 2"      "Set 1"          "Set 2"          "Set 3"
[13] "Tableau 10"       "Classic Tableau" "Polychrome 36"   "Alphabet"
```

Of note, palettes `"R4"`, `"Okabe-Ito"`, `"Dark 2"`, `"Paired"`, and `"Set 2"`, are all decently robust under conditions of colour vision deficiency. To obtain a vector of the hex codes used for a specific palette, you can just run `palette.colors(n = NULL, "Dark 2")`, but it is usually more convenient to insert this function directly into *ggplot2*. Figure 2.8 illustrates the colours employed in each palette - only eight colours are shown but some do contain more.

```
1  ggplot(msleep, aes(x = bodywt_log10, y = sleep_total)) +
2    geom_point(
3      size = 3, shape = 21, stroke = 1, colour = "black",
4      aes(fill = vore)
5    ) +
6    scale_fill_discrete(type = palette.colors(n = NULL, "Dark2"))
```
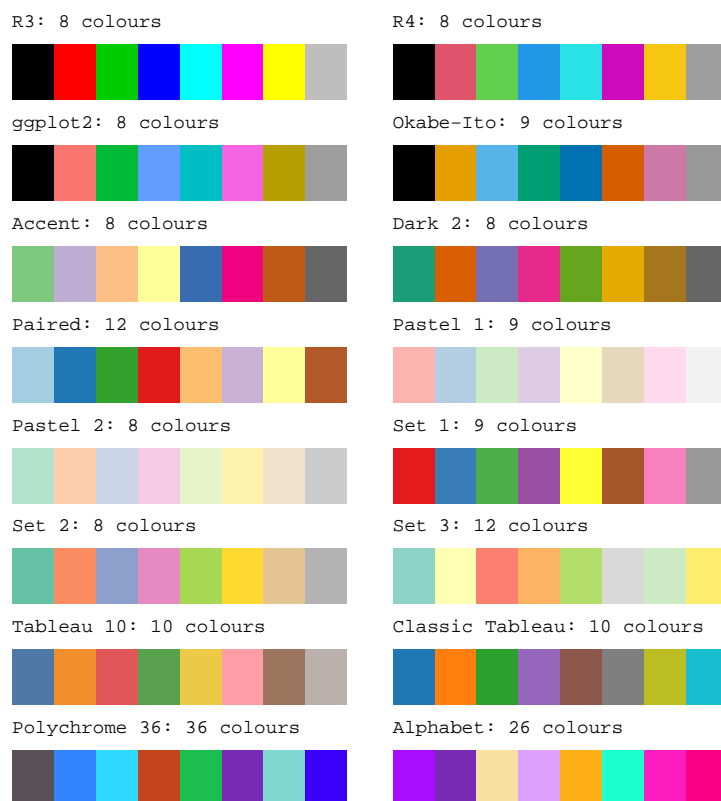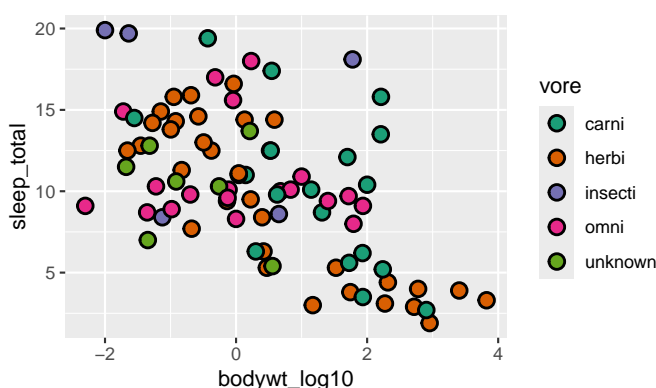

Figure 2.8: Examples of the various discrete colour palettes in base R.

2.9.5 Continuous Colour Scales

Continuous colour scales operate more or less in the same manner as discrete ones; however, to illustrate them, we need to map colour to a continuous variable. In the `msleep` data, there is a column called `$brainwt` which, similar to `$bodywt`, is a continuous measure; though, to visualize it adequately we will need to log transform it as well. For simplicity we will do this directly in the plot's code:

```r
ggplot(msleep, aes(x = bodywt_log10, y = sleep_total)) +
  geom_point(
    size = 3, shape = 21, stroke = 1, colour = "black",
    aes(fill = log10(brainwt))
  )
```
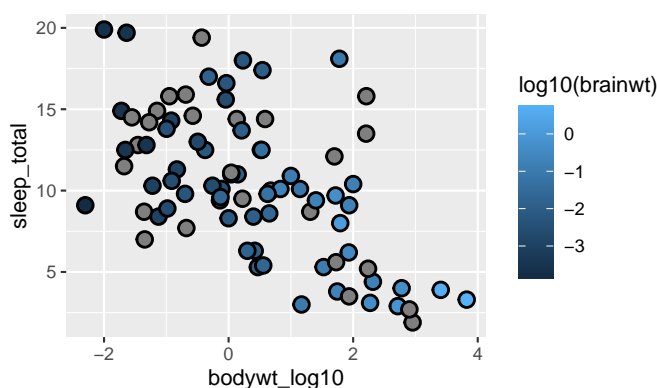

Immediately you can see we are now presented with a *colourbar* instead of a set of fixed colours. This is because the nature of the variable `$brainwt` is such that it does not fall neatly into distinct categories. Between any two brain weights there is a theoretically infinite amount of values and the colourbar's gradient offers a means of representing that. As you move from black to blue, lighter shades of blue are indicative of a heavier brain weight. Looking at the graph, increases in body weight also seem to correspond to increases in brain weight, but notice the grey points in the graph. Those are indicative of missing values in the `$brainwt` column and with a bit of R code, we can filter the data to see what values these are specifically.

```r
filter(msleep, is.na(brainwt))
```

In case it is not obvious, this code works by using the `is.na()` function to check whether each row in the `msleep` data frame's `$brainwt` column contains an `NA` value. Rows which result as `TRUE` are displayed and everything else is ignored. This leaves us with a data frame of 27 different animals, all of which have a `NA` value in the `$brainwt` column.

If you are left unsatisfied by the default black to blue gradient, *ggplot2* makes it easy to produce custom colour gradients using the `scale_fill_gradient()` and `scale_fill_gradient2()` functions, and of course there are colour aesthetic variants of this for situations where you want to modify the edge and point colours.[12] Both functions simply require you to specify a `low` colour argument that represents the bottom of the colourbar and a `high` colour argument that represents the top of the colour bar. However, `scale_colour_gradient2()` also requires you to specify the argument `mid`, which indicates a third midpoint colour. You can even specify the location of this midpoint with the argument `midpoint`. More succinctly `scale_colour_gradient()` creates *sequential* colour palettes, and `scale_colour_gradient2()` creates *diverging* colour palettes.

In addition to those main arguments, you can also specify what colour you would like `NA` values to be represented by and set the the breaks that appear on the colourbar. These are given by the arguments

---

[12]These are `scale_colour_gradient()` and `scale_colour_gradient2()`.

na.value and breaks respectively.

```
1   # scale_colour_gradient example
2   ggplot(msleep, aes(x = bodywt_log10, y = sleep_total)) +
3     geom_point(
4       size = 3, shape = 21, stroke = 1, colour = "black",
5       aes(fill = log10(brainwt))
6     ) +
7     scale_fill_gradient(
8       low = "blue",
9       high = "red",
10      na.value = "green",
11      breaks = seq(-4, 1, 1)
12    )
```
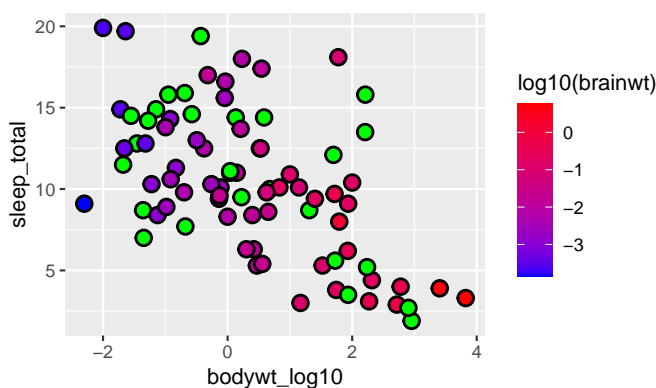

With the mammal sleep data, there is no logical reason to plot a midpoint colour using scale_colour_gradient2() but to illustrate its use we will depict a midpoint using the colour "grey" and we will place it at a $\log_{10}(\text{brain weight}) = -1.5$.

```
1   # scale_colour_gradient2 example
2   ggplot(msleep, aes(x = bodywt_log10, y = sleep_total)) +
3     geom_point(
4       size = 3, shape = 21, stroke = 1, colour = "black",
5       aes(fill = log10(brainwt))
6     ) +
7     scale_fill_gradient2(
8       low = "blue",
9       mid = "grey",
10      high = "red",
11      midpoint = -1.5,
12      na.value = "green",
13      breaks = seq(-4, 1, 1)
14    )
```
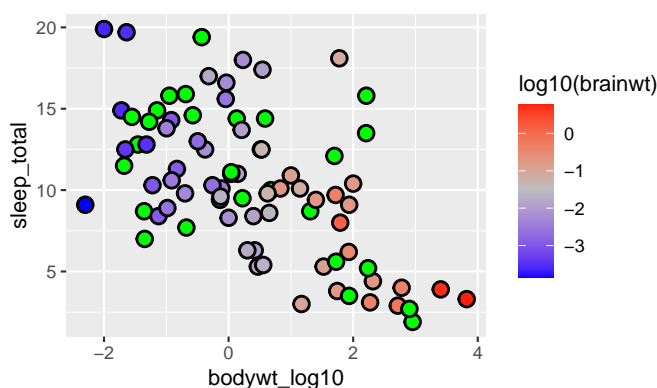
Pre-Existing Continuous Colour Palettes

Similar to what we saw with discrete colour scales, R comes with a set of continuous colour palettes we can use, some of which are sequential and some of which are diverging. For those interested, these palettes are based around an HCL (hue-chroma-luminance) colour space model which confers some advantages over the HSV (hue-saturation-value) colour space model computers have traditionally employed (Zeileis & Murrell, 2019).

To obtain a list of these HCL palettes you can simply run any of the following lines for sequential, diverging, and qualitative palettes respectively.

```
1   hcl.pals(type = "sequential")
2   hcl.pals(type = "diverging")
3   hcl.pals(type = "qualitative")
```

The qualitative palettes work best for discrete scales (i.e., identifying distinct categories) where you want each category to have equal perceptual weight. These are not much use for our present purposes but are notable because they are based on a HCL colour space model. That means we are not limited by the amount of colours in the palette like we were with R's standard discrete colour palettes (see section 2.9.4). Though, anecdotally, when you go beyond 6 categories the HCL qualitative palettes' colours start to become more and more difficult to discriminate between (even with standard colour vision). Interestingly, *ggplot2*'s default discrete colour selection relies on a similar underlying theory.

To obtain the hex codes for any given palette (e.g., `"Inferno"`) you will, in addition to providing the palette name, need to specify how many hex codes you want to see using the argument `n`. Visual examples of the three HCL palette types are provided in Appendix B.

```
1   hcl.colors(n = 8, palette = "Inferno")
    [1] "#040404" "#341348" "#701069" "#AB1E75" "#DC4962" "#F58426" "#F8C149" "#FFFE9E"
```

To use one of base R's HCL colour palettes in our plot we can use the function `scale_fill_gradientn()` to set our palette. The function just takes a vector of colours and extrapolates a gradient from that.

```
1   ggplot(msleep, aes(x = bodywt_log10, y = sleep_total)) +
2     geom_point(
3       size = 3, shape = 21, stroke = 1, colour = "black",
4       aes(fill = log10(brainwt))
5     ) +
6     scale_fill_gradientn(
7       colours = hcl.colors(n = 50, palette = "Inferno"),
8       na.value = "grey",
```

```
9        breaks = seq(-4, 1, 1)
10    )
```
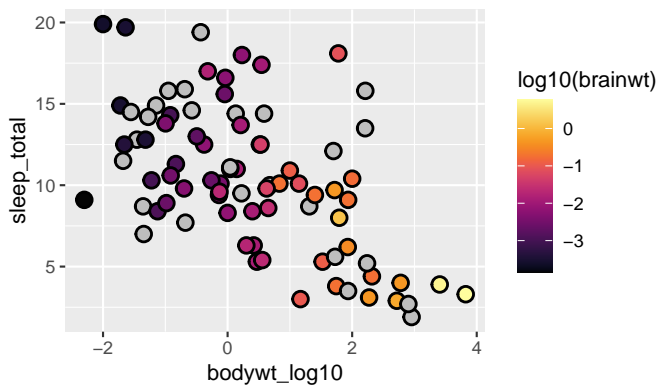

2.9.6   Shape Scales

We know that relying soly on colour to visually discriminate categories is inadvisable due to colour vision deficiencies people may have; thus, in addition to adjusting the colour scales, we can also adjust the shape scale simultaneously by mapping `$vore` to both `shape` and `fill` within the `aes()` function. For the shapes we will use the pch symbols 21 - 24 and also have the category "unknown" be represented by pch 13 (see Figure 2.3) - recall that these particular symbols (21 - 24) take both a colour and fill aesthetic. We will keep the edges (i.e., colour aesthetic) black but, for the fill aesthetic, we will use the `"R4"` colour palette (see Figure 2.8).

```
1   ggplot(msleep, aes(x = bodywt_log10, y = sleep_total)) +
2     geom_point(
3       size = 3, stroke = 1, colour = "black",
4       aes(fill = vore, shape = vore)
5     ) +
6     scale_shape_manual(values = c(21:24, 13)) +
7     scale_fill_discrete(type = palette.colors(n = NULL, "R4"))
```
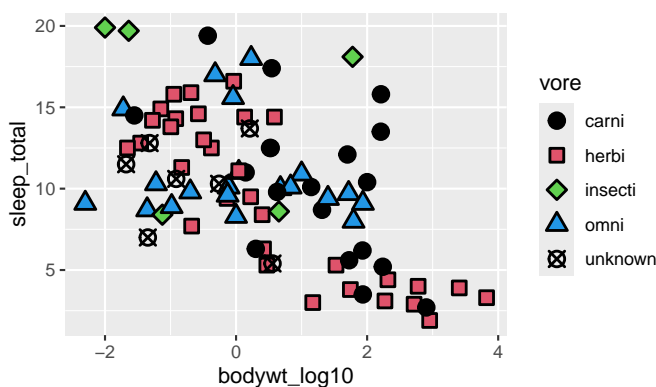

2.9.7   Legend Titles

In all the examples above, the legend that *ggplot2* produced has always been titled with the name of the column it is representing. For instance, when we mapped the categories in the `$vore` column it was titled "vore." When we mapped `log10(brainwt`, it was titled "log10(brainwt)." To adjust the name of the legend, each `scale` function we have used also takes a `name` argument which will dictate how the legend is titled. For instance, keeping with the above example, we could adjust the legend title to read "Diet".

```
1   ggplot(msleep, aes(x = bodywt_log10, y = sleep_total)) +
2     geom_point(
3       size = 3, stroke = 1, colour = "black",
4       aes(fill = vore, shape = vore)
5     ) +
6     scale_shape_manual(values = c(21:24, 13), name = "Diet") +
7     scale_fill_discrete(type = palette.colors(n = NULL, "R4"), name = "Diet")
```
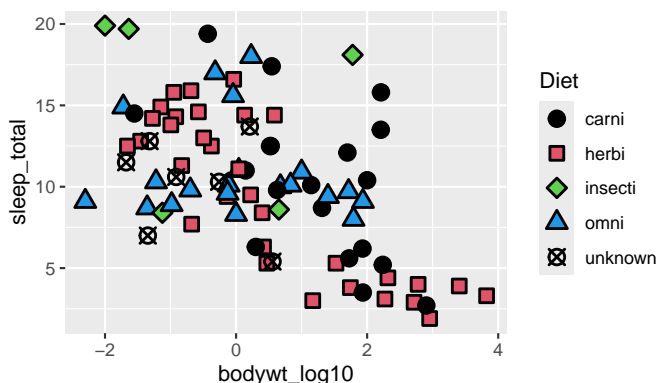

In this example, we have two scales in our legend, the `shape` scale and the `fill` scale. If you do not specify an identical name for each, they will be treated as separate legends. For instance, try giving `scale_shape_manual()` a different name than `scale_fill_discrete()` and see what happens.

An alternative method for renaming your legend is to add the function `labs()` to your plot's code and specify the name of each scale as a separate argument.

```
1    ggplot(msleep, aes(x = bodywt_log10, y = sleep_total)) +
2      geom_point(
3        size = 3, stroke = 1, colour = "black",
4        aes(fill = vore, shape = vore)
5      ) +
6      scale_shape_manual(values = c(21:24, 13)) +
7      scale_fill_discrete(type = palette.colors(n = NULL, "R4")) +
8      labs(
9        shape = "Diet",
10       fill = "Diet"
11     )
```

2.9.8   Other Scales

In the sections above, we have only considered the position, colour, fill, and shape scales, which are among the most frequently appealed to when graphing, but similar functions exist for other scales. For instance, there are scale functions to modify the size, linewidth, and linetype aesthetics if needed. To learn more about these and other features, an excellent resource is the tidyverse's official *ggplot2* website, which contains a learning section that will direct you to various excellent resources (https://ggplot2.tidyverse.org/), the best and most comprehensive of which is the official manual for *ggplot2* titled "ggplot2: Elegant Graphics for Data Analysis." Keeping with the ethos of "free software", this is available to read online for free at

https://ggplot2-book.org/

2.10   Modifying Other ``Non-data'' Components

One thing that will be apparent is that *ggplot2* has a very specific "look" to it, and that look is not arbitrary. It was crafted meticulously on the basis of expert advice. In the language of *ggplot2*, this look is what is referred to as a *theme*. Specifically, we are seeing `theme_grey()` and in the dark master's own words:

> The theme is designed to put the data forward while supporting comparisons, following the advice of (Tufte 2006; Brewer 1994; Carr 2002, 1994; Carr and Sun 1999). We can still see the gridlines to aid in the judgement of position (Cleveland, 1993), but they have little visual impact and we can easily `tune' them out. The grey background gives the plot a similar typographic colour to the text, ensuring that the graphics fit in with the flow of a document without jumping out with a bright white background. Finally, the grey background creates a continuous field of colour which ensures that the plot is perceived as a single visual entity.
>
> - Wickham et al., 2024

To sum up, the grey theme is immaculate in its conception and cannot be improved upon. In fact, once one has borne witness to the majesty of `theme_grey()`, even small departures from it can have drastic effects on a person's physical and mental well being. That being said, *ggplot2* still offers its users the ability to modify any aspect of the plot they wish - just be careful what you wish for.

2.10.1   Built-in Themes

Once the scaling and other main visual elements related to data presentation are complete, it is often helpful to set your plot's code as a variable you can append other elements too. Meaning that, in the same way a number in R is an object that you can name and add things too - e.g.,

```
1   x <- 1
2   x + 2
  [1] 3
```

Your plot is also an object (just a very complex one) that you can *add* things to. For instance, on the first line of our plot's code, right before the function `ggplot()`, we could give our plot the name `my_plot`.

```
1   my_plot <- ggplot(msleep, aes(x = bodywt_log10, y = sleep_total)) +
2     geom_point(
3       size = 3, colour = "black",
4       aes(fill = vore, shape = vore)
5     ) +
6     scale_shape_manual(values = c(21:24, 13)) +
7     scale_fill_discrete(type = palette.colors(n = NULL, "R4")) +
8     labs(
9       shape = "Diet",
10      fill = "Diet"
11    ) +
12    xlab("Log10(Body Weight kg)") + ylab("Sleep Total (hrs)")
```

Now, when you run `my_plot` you can see it output to the plot window.

```
1   my_plot
```
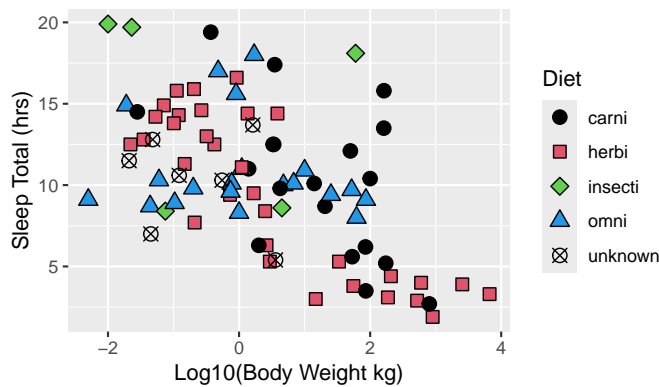
The quickest way to modify the overall appearance of your plot - which works well as a starting point for other modifications you want to make - is to use one of ggplot2's built in themes shown in Figure 2.9. Simply add the theme's function to your plot's code. For instance, if you wanted to use the black and white theme, `theme_bw()`, you would run ...

```
1   my_plot + theme_bw()
```

Additional pre-built themes can be accessed via other R packages, such as `ggthemes`.
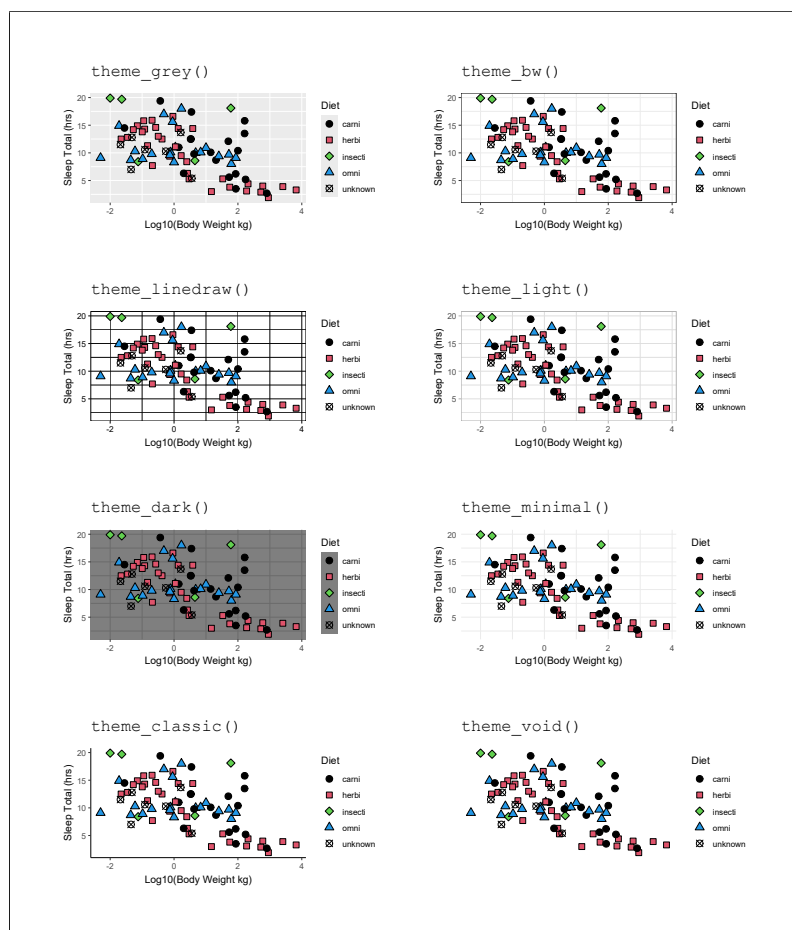
Figure 2.9: Visual examples of the eight built-in themes *ggplot2* provides.

2.10.2   Customizing Themes

Obtaining a more fine-grained control over the visual elements will require the use of *ggplot2*'s `theme()` function. Admittedly, there is so much customization possible here that an exhaustive explanation would require at least an additional chapter's worth of content. For simplicity, we will restrict the discussion to axis text modifications. This should illustrate the overall process well-enough and generalize nicely across the plot's numerous other elements. That being said, readers looking to adjust these other elements will still need consult documentation of some kind for specifics. The official *ggplot2* manual is unquestionably the best resource in this respect:

https://ggplot2-book.org/themes.html#sec-theme-elements

To modify the axis text, we first need to specify, within the `theme()` function, the name of the element we want to modify. In this case, since we want to modify *both* the x and y axis, we will specify `axis.text`. Then we need to specify a function to modify this element we have chosen. In this case, since we want to modify text, we will use the function `element_text()`. Within that, we can specify numerous arguments related to the text. For a full list of arguments, it is highly recommended that the reader consult the R documentation: `?element_text()`

```
1  my_plot + theme_bw() +
2    theme(
3      axis.text = element_text(size = 18, face = "bold", colour = "red", angle = 45)
4    )
```
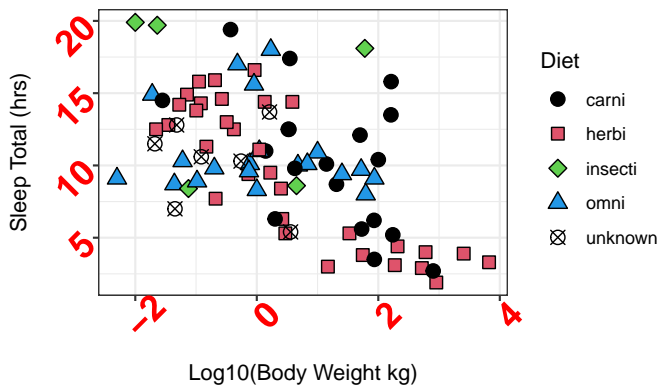

Notice that the code effected both axis; however, if we want to affect a change for only one axis (e.g., the x-axis) we just specify the element as `axis.text.x`. This will also allow us to include a `margin` argument to affect the spacing around the text.

```
1  my_plot + theme_bw() +
2    theme(
3      axis.text.x = element_text(
4        size = 18, face = "bold", colour = "red", angle = 45,
5        margin = margin(t = 1, r = 0, b = 0, l = 0, unit = "cm")
6      )
7    )
```
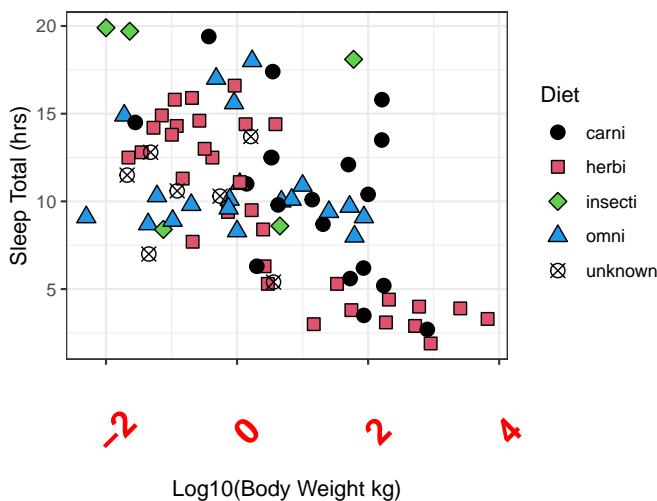
A similar logic applies to the axis title. In that case we would modify the `axis.title` element. And again, if we wanted to modify the x-axis title specifically, we would use `axis.title.x`. The y-axis title would of course be `axis.title.y`.

```r
my_plot + theme_bw() +
  theme(
    axis.text.x = element_text(
      size = 18, face = "bold", colour = "red", angle = 45,
      margin = margin(t = 1, r = 0, b = 0, l = 0, unit = "cm")
    ),
    axis.title.y = element_text(
      size = 18, face = "italic", colour = "deepskyblue3", angle = 90
    )
  )
```
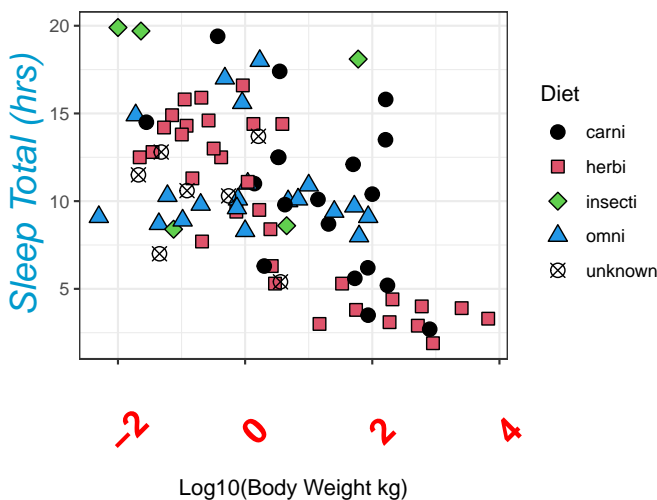

2.11   A Final Note

In the plots created above, we have gone through how to adjust a wide variety of elements but there are two adjustments that have not been discussed:

1. How do you change the order of the categories? For instance, suppose we wanted "herbi" to be at the top of the "Diet" legend. Or suppose we wanted it to come first in our sequence of faceted plots we created

in section 2.6. How can we make that happen?

2. How do we adjust the names of the categories? Each category of vore/diet has had its name shortened, but what if we wanted to write out each category in its entirety. E.g., display "carnivore" instead of "carni", and "herbivore" instead of "herbi", and so on.

   The answer to both these questions requires first understanding "factors," which will be explained later in the next chapter.

Chapter 3

The Basics of Loading and Manipulating Data

CHAPTER 1 stated that data frames are essential for keeping a host of related information stored in a well organized manner that is easy to manipulate. When printed to the console, data frames present as a familiar spreadsheet-like structure that can be created, subset, and altered in various ways (see section 1.4.10 for details). Moreover, in chapter 1, we saw how a data frame can be constructed by manually entering values with R code. And, as should be apparent, for all but the smallest of data sets, this method is time-consuming and highly prone to error. A better strategy is to take an existing file of information and import that directly into R as a data frame or, depending on the nature of the data, as a list, matrix, array, or table. Though, a data frame is usually going to be the optimal choice and will be the primary focus of this chapter.

Data can come in all kinds of different layouts and file formats and, in this respect, R has the ability to handle pretty much any scenario that might arise. This chapter will, for the most part, work under the assumption that the kind of data that you need to work with is in a conventional "spreadsheet-style" of format. That is to say, like the `msleep` data used in Chapter 2, you have a bunch of rows and a bunch of columns, and each cell contains just a single value.

3.1 Spreadsheet Software

When it comes to spreadsheets, there are many different file formats they could be saved as. Pretty much every spreadsheet application has its own specific file type that is tailored to its unique purpose and platform. For instance, *Microsoft's Excel* spreadsheet application has its own proprietary format called the `.XLSX` file format. The stock spreadsheet application on Macintosh computers, called *Numbers*, use the `.NUMBERS` file format. And if you use an open-source spreadsheet software like *Libre Office's Calc* application, you may be familiar with the `.ODS` file format.

As everyone who is reading this doubtlessly appreciates, spreadsheet applications like, Microsoft's Excel, Numbers, Libre Office's Calc, etc., do more than just structure your data in a big table (which is all a spreadsheet is). They allow you to do things like perform calculations, adjust cell colours, add images, insert comments, etc. And all of this extra stuff is saved, in one form or another, inside the specific file associated with that software. These features make applications like Microsoft's Excel, for instance, a great tool for basic tasks like balancing the household budget. However, for serious data analysis that requires the use of large data sets and complex or heavy calculations, this kind of software is going to be more of a hindrance than a help. Adding in all of those layers of additional functionality is going to increase your file sizes, inflate your load times, create restrictions on how much information can be contained within your spreadsheet, and will increase the chance of a glitch

occurring. Additionally, and most importantly, both the analyses and the data are all contained within the same file, which makes it very easy to irrevocably fuck up your original data set, often without even realizing it. The fact is, we should care about analyzing our data efficiently and safely, not making it look pretty in what amounts to a fancy table, and this is one of the key benefits of using R.

From the point of view of R, a spreadsheet is just a way of displaying the raw information it is analysing, and nothing more. The analysis of that information is what R does. Technically then, we should not be referring to something as a "spreadsheet file," but rather a "data file." The spreadsheet aspect of all of this is more about how the data is structured for our viewing. However, data does not necessarily need to be viewed as a spreadsheet - it can be viewed in all kinds of different ways. It is just that a spreadsheet is usually the most convenient and intuitive way to view it and talk about it.

3.2   Using an Ethical File Format

As noted above, there are a variety of different spreadsheet file types data could be formatted as (`.XLSX`, `.ODS`, etc). To remain consistent with open-science principles (UNESCO, 2021), best practice dictates that you work with your data in a file format that is both universally recognized across applications and will also stand the test of time in terms of compatibility. In other words, we want to (ideally) work with a file format that has no immediate risk of becoming obsolete and can be read by multiple computers on multiple platforms without forcing the user to pay for some proprietary application. Along these lines, the most widely used and recognized format is the `.CSV` file format.

3.3   The .CSV Format

"CSV" stands for "comma separated values." It gets its name from the fact that it is, quite literally, nothing more than a generic text document that uses commas to denote a tabular (spreadsheet structure) in the data.[1] This is easiest to see with an example. The GitHub repository for this book contains a file called `MM_Madison_wide.csv`. The GitHub repo can be accessed at the following URL:

https://github.com/statistical-grimoire/book/

The file is located within the `data` directory at `./data/ch-3/MM-candy/`. It contains measurements of the colour distribution of M&M Milk Chocolate candies, collected by Josh Madison for his blog (Madison, 2007). Josh purchased a case of Milk Chocolate M&M's (which is 48 separate packages of M&M's) and counted how many of each M&M colour (blue, brown, green, orange, red, yellow) were contained in each pack. He did this, ostensibly, to evaluate a theory he had about the manufacturing process of the candies.[2]

Upon opening the file within GitHub,[3] you will see that it displays the file's contents in a fairly typical spreadsheet layout (see Table 3.1 for an example displaying the first 6 rows).

---

[1] "Tabular" and "spreadsheet" mean the same thing here.

[2] Or, more realistically, he just wanted a excuse to purchase and eat Milk Chocolate M&M's guilt free.

[3] https://github.com/statistical-grimoire/book/blob/main/data/ch-3/MM-candy/MM__Madison__wide.csv

	pkg	weight_oz	year	blue	brown	green	orange	red	yellow
1	1.00	1.69	2007	13	7	12	9	7	8
2	2.00	1.69	2007	8	3	13	13	10	6
3	3.00	1.69	2007	8	10	11	10	5	10
4	4.00	1.69	2007	14	4	6	14	7	9
5	5.00	1.69	2007	6	8	10	12	8	8
6	6.00	1.69	2007	7	11	13	7	4	13

Table 3.1: First 6 rows of the `MM_Madison_wide.csv` data displayed in a spreadsheet structure.

However, this is just how GitHub presents `.CSV` files. The actual raw data is a basic text document that separates individual values with a comma. We can see this more clearly if we click the button labelled "Raw" which will present the file in its unaltered (i.e., raw) text format. The first 6 rows can be seen below

```
pkg,weight_oz,year,blue,brown,green,orange,red,yellow
1,1.69,2007,13,7,12,9,7,8
2,1.69,2007,8,3,13,13,10,6
3,1.69,2007,8,10,11,10,5,10
4,1.69,2007,14,4,6,14,7,9
5,1.69,2007,6,8,10,12,8,8
6,1.69,2007,7,11,13,7,4,13
...
```

Example of the `MM_Madison_wide.csv` data file displayed in its raw text format. Only the first six rows are shown.

Comparing the two versions it can readily be seen how the commas are functioning. They separate individual cells/columns and each new line represents a new row in the spreadsheet. This not only makes it easy to read `.CSV` files within a basic text editor, but create them as well. Just save (or rename) the text document with a `.CSV` file extension (which you may need to configure your computer to display). Alternatively, if you have a good spreadsheet software on your computer, they will always have the ability to *"Save As"* a `.CSV` file or *"Export"* to one. For instance, the save menu of Microsoft Excel will present the user with a drop down list of potential file types it can save as and (as of writing this) has four different versions of `.CSV` files (the best option is the one labelled "UTF-8 (Comma delimited)"). By contrast the Numbers application on a Mac will not permit a spreadsheet to save as anything other than a `.NUMBERS` file, but will allow you to export your saved spreadsheet as a `.CSV`. Just select *File → Export To → CSV...*

3.4   Delimiters

In the case of the `MM_Madison_wide.csv` file, the comma is functioning as a **delimiter**; which is to say it is a character that defines the limits of (i.e., it "delimits") individual values. Commas are not the only characters that can be used to delimit, any character can technically be used. Other common delimiters include semicolons (;) and tab-key spaces. Semicolons are often used when the data is logged with commas representing decimal points instead of periods (e.g., 13.666 = 13,666), which is a frequent practice in many countries. Oddly, when a delimited file uses semicolons, it is still often given a .CSV file extension despite it being a completely different character. In R, to avoid confusion, the convention is to refer to these semicolon delimited files as `CSV2` files in function names (e.g., `write_csv()` would use a comma to delimit whereas `write_csv2()` would use a semicolon).

Tab spaces (i.e., pressing "tab" on your keyboard), are also frequently employed as a delimiter, but these

are usually denoted as .TSV files (i.e., tab separated values). In fact, the name for the keyboard key "tab" comes from the the verb "tabulate" because the key facilitated easier generation of tables when working on a typewriter. Prior to the tab key's development, the space bar had to be repeatedly pressed to advance the typewriter's carriage to align columns appropriately.

If you were to save the `MM_Madison_wide` data set as a `.TSV` file and open it within a generic text editor, you would see something very similar to the following ...

pkg	weight_oz	year	blue	brown	green	orange	red	yellow
1	1.69	2007	13	7	12	9	7	8
2	1.69	2007	8	3	13	13	10	6
3	1.69	2007	8	10	11	10	5	10
4	1.69	2007	14	4	6	14	7	9
5	1.69	2007	6	8	10	12	8	8
6	1.69	2007	7	11	13	7	4	13
...								

Example of the `MM_Madison_wide.csv` data file displayed in its raw text format if it were a .TSV file. Only the first six rows are shown.

Notice that the tabular separation gives the file a much more grid-like aesthetic that is easier to read. Incorporating spaces into the text file can be used to further refine the alignment.

3.5   Reading a CSV File into R

Now that we have a good sense of what a `.CSV` file is we should discuss how to load it into R as a data frame object so we can conduct our analyses. To begin with, you should download `MM_Madison_wide.csv` from the aforementioned GitHub repo by simply clicking the "down arrow" icon labelled "Download raw file." Once downloaded, simply place the file inside your working directory.[4]

With the file in its appropriate location you can simply run the function `read_csv()` and give it the full name (with extension) of your file. This will create a data frame object in R. However, `read_csv()` is a function that belongs to the *readr* package which is part of the *tidyverse*, so if you do not have the *tidyverse* loaded, this will not work. In order to easily call our loaded data, we will assign it the name `mm_df`.

```r
library(tidyverse)
mm_df <- read_csv("MM_Madison_wide.csv")
```

```
Rows: 48 Columns: 9
Column specification

Delimiter: ","
dbl (9): pkg, weight_oz, year, blue, brown, green, orange, red, yellow

Use `spec()` to retrieve the full column specification for this data.
Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

Running the above code presents us with some useful information about the data set we have loaded. We can see that it has 48 rows and 9 columns, uses a `,` as a delimiter, and the 9 columns all consist of `dbl` values, which is a shorthand way of referring to *double-precision number*. To simplify a complex story, R has multiple types of *numeric* objects; i.e., it has multiple ways of representing a number. A *double*, as its often

---

[4]If you are unsure what a "working directory" is see section 1.7

referred to, is one such representation. If that is confusing, do not worry, what is important to take away from the output is that `dbl` means the 9 columns all contain numeric values.

Running `mm_df` will print the data frame to the console.

```
mm_df
```

```
# A tibble: 48 × 9
      pkg weight_oz  year  blue brown green orange   red yellow
    <dbl>     <dbl> <dbl> <dbl> <dbl> <dbl>  <dbl> <dbl>  <dbl>
 1      1      1.69  2007    13     7    12      9     7      8
 2      2      1.69  2007     8     3    13     13    10      6
 3      3      1.69  2007     8    10    11     10     5     10
 4      4      1.69  2007    14     4     6     14     7      9
 5      5      1.69  2007     6     8    10     12     8      8
 6      6      1.69  2007     7    11    13      7     4     13
 7      7      1.69  2007     8     7    13      8     7      9
 8      8      1.69  2007    12    10     6      8     8      9
 9      9      1.69  2007     6     9    12     14     8      6
10     10      1.69  2007     7     8    12     10    11      7
# 38 more rows
# Use `print(n = ...)` to see more rows
```

You can now subset and manipulate `mm_df` like we did in Chapter 1 when we first discussed data frames (see section 1.4.10). For instance, if we wanted to look at the mean number of green M&M's across all the packages we could simply run:

```
mean(mm_df$green)
```

```
[1] 10.0625
```

read.csv() vs. read_csv()

To load the M&M data frame above, we used the function `read_csv()`, which is part of the *tidyverse*. However, base R as similar function, `read.csv()`, that will do essentially the same thing - it will read a `.csv` file in to R. For most use cases there is little advantage to adopting one function over the other, but if you have the *tidyverse* loaded, you may as well use `read_csv()` because it does have some big advantages. First, it offers excellent customization options, which are particularly useful when loading very large datasets or merging multiple datasets. Second, it alerts you to any issues encountered during the loading process. Third, it performs much faster under heavy loads than its base R counterpart, even providing a progress bar when reasonable to do so. Finally, it stores the data as a *tibble* which will be discussed later.

3.5.1   Reading Other File Types into R

If your data is delimited by some character other than a comma (e.g., a semicolon, tab, backslash, etc.), there is a more general function that can be employed called `read_delim()` which allows you to specify any delimiter (i.e., separator) using the argument `delim`. For instance, we could have loaded the M&M data in the following way:

```
mm_df <- read_delim("MM_Madison_wide.csv", delim = ",")
```

If your text document was separated by semicolons you would just include `delim = ";"`, if it was separated using tabs you would just `delim = "\t"`, and so on.

One thing that is worth appreciating about delimited files is that their file extension (e.g., the `.CSV` or `.TSV` at the end of the file name) is irrelevant to how R reads the file. As has been previously emphasized, `.CSV` files and `.TSV` files for instance, are just generic text documents, nothing more. This means you may see them with the file extension `.TXT`, but that will not impact how any of the above functions operate.

Now, what would you do if you wanted to load a Microsoft Excel spreadsheet file (i.e., a `.XLSX` file) into R directly? Well as per the discussion on spreadsheets and ethical file formats (see section 3.1 and 3.2), the best practice is to save it as a `.CSV` using Excel and load that new file directly into R. However, should you wish to eschew this advice, the *tidyverse* does have a package called *readxl* with functions that will allow you to do this. This is not part of the nine core packages, so it will need to be loaded using the `library()` function. A word of warning is in order though. As well made as the *readxl* package is, reading `.XLSX` files directly will, almost certainly, cause more problems than it solves. These files are not intended to be read by anything other than Excel and Microsoft does not want them read by anything other than Excel. Thus, by loading the `.XLSX` file directly into R, you are (computationally speaking) picking an unnecessary fight with Microsoft. 9 times out of 10 you will win that fight, thanks to *readxl*, but you will still probably end up with some nasty bruises and scars.

3.6   Tibbles vs. Data Frames

In the output for `mm_df` (and the `msleep` data from chapter 2) you can see that the output printed to the console specifies that we are looking at something called a **tibble**: `# A tibble: 48 x 9`. The output also helpfully displays the dataset's dimensions and the class of object contained within each column. This is in contrast to the data frame created in chapter 1, which did not do any of that for us. In the *tidyverse's* own words, a tibble is a ...

> modern reimagining of the data.frame, keeping what time has proven to be effective, and throwing out what is not. Tibbles are data.frames that are lazy and surly: they do less (i.e. they don't change variable names or types, and don't do partial matching) and complain more (e.g. when a variable does not exist). This forces you to confront problems earlier, typically leading to cleaner, more expressive code. Tibbles also have an enhanced print() method which makes them easier to use with large datasets containing complex objects.
>
> *- https://tibble.tidyverse.org/*
> *(2024/07/28)*

In terms of basic usage, tibbles function almost identically to the classic data frame discussed in chapter 1. For instance, we can re-create chapter 1's data frame as a tibble using an identical syntax.

```
1   df <- tibble(
2     Subject = 1:10,
3     Group = c("Exp", "Cont", "Exp", "Cont", "Exp", "Exp",
4               "Cont", "Exp", "Cont", "Cont"),
5     Value = c(-0.36,  0.28,  1.54,  0.51, -1.28,  1.15,
6               -2.22, -0.51,  NA, -1.04)
7   )
8
9   df
```
```
# A tibble: 10 × 3
   Subject Group Value
     <int> <chr> <dbl>
 1       1 Exp   -0.36
 2       2 Cont   0.28
```

```
3         3 Exp    1.54
4         4 Cont   0.51
5         5 Exp   -1.28
6         6 Exp    1.15
7         7 Cont  -2.22
8         8 Exp   -0.51
9         9 Cont    NA
10       10 Cont  -1.04
```

There are a number of interesting differences between tibbles and data frames, but nothing that merits any in depth discussion for a beginner with R. What is perhaps worth noting is that it is easy to switch between the two should the need arise. For example, to convert our tibble `df` to a data frame, we can simply use the `as.data.frame()` function in R.

```
1  # tibble to data frame
2  df <- as.data.frame(df)
3  df
     Subject Group Value
1          1   Exp -0.36
2          2  Cont  0.28
3          3   Exp  1.54
4          4  Cont  0.51
5          5   Exp -1.28
6          6   Exp  1.15
7          7  Cont -2.22
8          8   Exp -0.51
9          9  Cont    NA
10        10  Cont -1.04
```

To convert it back to a tibble ...

```
1  # data frame to tibble
2  df <- as_tibble(df)
3  df
# A tibble: 10 × 3
    Subject Group Value
      <int> <chr> <dbl>
1         1 Exp   -0.36
2         2 Cont   0.28
3         3 Exp    1.54
4         4 Cont   0.51
5         5 Exp   -1.28
6         6 Exp    1.15
7         7 Cont  -2.22
8         8 Exp   -0.51
9         9 Cont     NA
10       10 Cont  -1.04
```

3.6.1   Displaying Tibbles in the Console

Tibble Dimensions

Given the limited screen space and the large size of most datasets, tibbles are designed to display only the first 10 rows when printed to the console, making it easier for users to work with their data.

Generally, if you want to view an entire data set, the best practice is not to display it in the console but rather use R's `View()` function which opens it in a spreadsheet style viewer. That being said, many people will still find the amount of rows dispyed by a tibble within the console lacking, particularly if you are working on anything other than a small laptop. For this reason, the 10 row limit is a behaviour which can be circumvented in various ways. One simple way is to make use of the `print()` function. For instance, if we want to display the first 20 rows we can simply run ...

```
1    print(mm_df, n = 20)
```

An alternative method is to change R's default display behaviour by setting the amount using the `options()` function. You can set both a minimum and maximum number of rows to display.

```
1    options(
2      pillar.print_min = 30,
3      pillar.print_max = 30
4    )
5
6    mm_df
```

If that method is your preference, then it is usually advisable to place the `options()` code at the top of your R script because it only needs to be run once.

What if you wanted to display every single row each time you print a tibble? Well, recall that R represents infinity in the positive direction as (`Inf`). We can use that to our advantage here:

```
1    options(pillar.print_min = Inf)
2
3    mm_df
```

What about columns though? Well, interestingly tibbles will actually conform to the size of your console screen. So if you can only fit five columns on screen, the tibble will only display those five and notify you of the others not displayed beneath the output. This is done to preserve the "rectangleness" of the data so it can be visualized appropriately. This also stands in stark contrast to how base R's data frames behave which will stack columns on top of each other, with no consideration of column or row space. Admittedly, its nice to have all that information displayed, but it comes at the cost of being difficult for a human to visually parse. That being said, if you wanted your tibbles to behave like this and always display all columns, you can just add an additional argument, `pillar.width = Inf`, to the `options()` function:

```
1    options(
2      pillar.print_min = 30,
3      pillar.print_max = 30,
4      pillar.width = Inf
5    )
6
7    mm_df
```

However, if you wanted a more temporary solution, you can just add a `width` argument to the `print()` function. E.g.,

```
1    print(mm_df, n = 30, width = Inf)
```

Understanding Significant Digits

To save space and facilitate easier reading, both tibbles and data frames will round values with many decimal values. Though, in the case of tibbles, they do not just simply round to a preset number of digits. To illustrate what tibbles are doing in this respect, recall that R has a built-in constant for π.

```
pi
```
```
[1] 3.141593
```

Lets create a simple data frame that repeats π four times within a single column.

```
pi_df <- tibble(pie = rep(pi, 4))
pi_df
```
```
# A tibble: 10 × 1
      pie
    <dbl>
  1  3.14
  2  3.14
  3  3.14
  4  3.14
```

One thing that will be noticed is that the tibble is only displaying π to two decimal places. However, all of the digits still exist in R's memory and any calculations you do will take those unseen digits into account. For instance, if we isolate the first row's value you can see that all the digits of π are displayed.

```
pi_df$pie[1]
```
```
[1] 3.141593
```

It is important to understand that tibbles do not strictly control the handling of decimals. Instead, they work with *significant digits*. This allows the tibble to preserve its desired "rectangleness" by giving it cleaner looking columns. Since tibbles default to three significant digits, π will only display as 3.14.

As a refresher of primary school math, with significant digits, everything in front of the decimal point is always displayed, but each number in front of that decimal point uses up a significant digit (a.k.a. a "sig dig"). For instance, if you had a number like 666.13. Displaying that to two sig digs would give you 666. Displayed to three sig digs would again be 666. Displayed to four sig digs would be 666.1. Five sig digs would be 666.13. Six sig digs would be 666.130. Seven would be 666.1300, and so on.

To increase the amount of sig digs shown within a tibble we can simply add another argument to the `options()` function:

```
options(pillar.sigfig = 16)
pi_df
```
```
# A tibble: 10 × 1
                 pie
               <dbl>
  1 3.141592653589793e0
  2 3.141592653589793e0
  3 3.141592653589793e0
  4 3.141592653589793e0
```

Because of limitations of 64-bit computing, a tibble is not going let you exceed 16 sig digs and in certain cases will display results in scientific notation. In this case we see some scientific notation, but it is to the power of 0, so it can be ignored.

3.7 Wide Data vs. Tidy Data

3.7.1 Wide Data

Looking at the `MM_Madison_wide.csv` loaded at the outset of this chapter, you can see that the data is laid out in a fairly logical manner. Each row corresponds to a different package of M&Ms and we can clearly see how many M&Ms of a particular colour are in each package by examining the columns named, blue, brown, green, etc.

```
1   mm_df
```

```
# A tibble: 48 × 9
      pkg weight_oz  year  blue brown green orange   red yellow
    <dbl>     <dbl> <dbl> <dbl> <dbl> <dbl>  <dbl> <dbl>  <dbl>
 1      1      1.69  2007    13     7    12      9     7      8
 2      2      1.69  2007     8     3    13     13    10      6
 3      3      1.69  2007     8    10    11     10     5     10
 4      4      1.69  2007    14     4     6     14     7      9
 5      5      1.69  2007     6     8    10     12     8      8
 6      6      1.69  2007     7    11    13      7     4     13
 7      7      1.69  2007     8     7    13      8     7      9
 8      8      1.69  2007    12    10     6      8     8      9
 9      9      1.69  2007     6     9    12     14     8      6
10     10      1.69  2007     7     8    12     10    11      7
# 38 more rows
# Use `print(n = ...)` to see more rows
```

This style of layout makes it easy to do certain things with the data. For instance, if we wanted to know the mean number of red M&Ms we could just run

```
1   mean(mm_df$red)
```

```
[1] 7.75
```

If we wanted to obtain the mean of every column, we could use the `apply()` function. This function literally *applies* a function of your choosing to either the columns or rows. So we could, for instance, apply the `mean()` function to each *column*.

```
1   apply(mm_df, MARGIN = 2, FUN = mean)
```

```
       pkg    weight_oz         year         blue        brown        green
 24.500000     1.690000  2007.000000    10.020833     7.729167    10.062500
    orange          red       yellow
 11.333333     7.750000     7.687500
```

The argument `FUN` specifies what function is applied. The argument `MARGIN` specifies whether that function is applied to the rows (`1`) or columns (`2`). In this case we are applying it to columns, so we specified `2` .

If you wanted to count how many M&Ms are in each individual package, you could *apply* the `sum()` function to the *rows* - though, keep in mind that we would want to ignore `pkg` , `weight_oz` , and `year` values when doing this. Luckily we can use what we learned about indexing in chapter 1 to ignore these columns (see section 1.4.10).

```
1   apply(mm_df[, 4:9], MARGIN = 1, FUN = sum)
```

```
 [1] 56 53 54 54 52 55 52 53 55 55 54 54 52 55 55 53 55 56 54 57 55 54 55 57
[25] 53 54 54 57 56 57 55 54 54 57 56 56 55 55 55 53 55 54 54 53 54 54 55 55
```

At a superficial glance, you can see that R makes it fairly easy to work with data in this kind of layout. However, we have not attempted anything fairly complicated here. The reality is that this layout, which we will refer to in this book as **wide data**, is *not* optimal for most types of analyses and plotting. Additionally, until you are actually in a position to do those kinds of analyses, it will be difficult to convince you of the truth of this - so you will just have to take that point on faith. Wide data spreads variables across multiple columns. In this case, you can treat of the colour of the M&Ms here (blue, brown, green etc.) as a variable in its own right. We might call this variable simply "type." That is to say, there is a "blue" *type* of M&M, a "brown" *type* of M&M, and so on.[5] There is also a second variable, which we could refer to as the "amount" of M&M's, within each of the six colour type columns.

When organizing or arranging data, best practices dictate that you **restrict a single variable to a single column**. In this case, the variable "type" is being spread across six column headers. And the variable "amount" is spread within those six columns. To fix this, we can use the *tidyverse* function `pivot_longer()`.

```
mm_tidy <- pivot_longer(mm_df,
  cols = blue:yellow,
  names_to = "type", values_to = "amount"
)

mm_tidy
```

```
# A tibble: 288 × 5
     pkg weight_oz  year type    amount
   <dbl>     <dbl> <dbl> <chr>    <dbl>
 1     1      1.69  2007 blue        13
 2     1      1.69  2007 brown        7
 3     1      1.69  2007 green       12
 4     1      1.69  2007 orange       9
 5     1      1.69  2007 red          7
 6     1      1.69  2007 yellow       8
 7     2      1.69  2007 blue         8
 8     2      1.69  2007 brown        3
 9     2      1.69  2007 green       13
10     2      1.69  2007 orange      13
11     2      1.69  2007 red         10
12     2      1.69  2007 yellow       6
# 276 more rows
```

Notice a few differences with this pivoted data. There are now considerably more rows in the data, 288 vs. 48, and there are two new columns, `$type` and `$amount` which have replaced the six different colour type columns. In terms of the `pivot_longer()` function we used, the most important argument we specified is the argument `cols`, as this defines which columns are going to be collapsed into a single column. Using the `:`, we specified a range of columns (i.e., from blue to yellow), but we could have also given it a vector of column names like so: `cols = c(blue, brown, green, orange, red, yellow)`.[6]

The arguments `names_to` and `values_to` just specify the name of the new columns and are not strictly required, but are good practice to include.

---

[5]A better variable name would be "colour," but colour is also a plotting aesthetic and, later in the chapter, we will be plotting this data. So to keep the plot's code a bit more intelligible, I'm refraining from using "colour" as the variable name.

[6]The *tidyverse* has numerous methods for selecting multiple columns that don't exist in base R. For a rundown of each see the R documentation: `?tidyr_tidy_select`

3.7.2   Tidy data

By collapsing the six colour columns into one, the data has ascended into a sacred arrangement known as **tidy data** (or, as some heretics call it, "the long format"). Tidy data is a cornerstone of the *tidyverse's* thaumaturgy and all tidy data adheres to three basic precepts:

    I. Each variable is a column; each column is a variable.
    II. Each observation is a row; each row is an observation.
    III. Each value is a cell; each cell is a single value.

It can be seen that the M&M data now satisfies these three standards, as did the `msleep` data used in chapter 2. As we progress through the remainder of this chapter, it will become apparent that having your data in this tidy form will greatly facilitate both plotting and analysis.

3.8   Laying Pipe (The |> and %>% Operators)

One of the biggest contributions of the *tidyverse* to R has been its ability to utilize what is known as a "piping" syntax with an almost otherworldly efficiency. This is not to say that the concept of piping was something conjured by the *tidyverse* out of nothing. It is more that the *tidyverse's* relentless invocation of it unearthed its abilities to such a degree that the R community has, for the most part, adopted it as common usage. There is no better truth of this than the fact that, as of version 4.1.0 released in 2021, piping has been integrated into base R.[7] But what is this curious thing called "piping?"

    The essence of piping is that you are transferring the output of one thing to another. For instance, suppose we wanted to know the mean amount of different M&M colours. We could of course insert the `$amount` column into the mean function like so ...

```
mean(mm_tidy$amount)
```
```
[1] 9.097222
```

    Alternatively, we could "pipe" (i.e., transfer) the `$amount` column in our tidy data to the `mean()` function using R's pipe operator `|>` .

```
mm_tidy$amount |> mean()
```
```
[1] 9.097222
```

    As another example, suppose we wanted a tidy data frame that only contained the red M&M type. The standard methodology would be to specify the data frame within the `filter()` function, like so ...

```
filter(mm_tidy, type == "red")
```
```
# A tibble: 48 × 5
     pkg weight_oz  year type  amount
   <dbl>     <dbl> <dbl> <chr>  <dbl>
 1     1      1.69  2007 red        7
 2     2      1.69  2007 red       10
 3     3      1.69  2007 red        5
 4     4      1.69  2007 red        7
 5     5      1.69  2007 red        8
 6     6      1.69  2007 red        4
 7     7      1.69  2007 red        7
```

---

[7]Officially, I have no direct evidence that the pipe update to base R was motivated by the influence of the *tidyverse*, but given the ubiquity of `%>%` and the *dplyr* package, it seems unreasonable to think otherwise.

```
 8      8      1.69   2007 red          8
 9      9      1.69   2007 red          8
10     10      1.69   2007 red         11
# 38 more rows
```

Alternatively, we could pipe the data into the `filter()` function.

```
1  mm_tidy |> filter(type == "red")
```
```
# A tibble: 48 × 5
      pkg weight_oz  year type  amount
    <dbl>     <dbl> <dbl> <chr>  <dbl>
 1      1      1.69  2007 red        7
 2      2      1.69  2007 red       10
 3      3      1.69  2007 red        5
 4      4      1.69  2007 red        7
 5      5      1.69  2007 red        8
 6      6      1.69  2007 red        4
 7      7      1.69  2007 red        7
 8      8      1.69  2007 red        8
 9      9      1.69  2007 red        8
10     10      1.69  2007 red       11
# 38 more rows
```

In addition to this, suppose you did not want the `$weight_oz` or `$year` column in the output. To achieve this, this output could be further piped into the *tidyverse's* `select()` function which allows you to grab specific columns.

```
1  mm_tidy |>
2    filter(type == "red") |>
3    select(pkg, type, amount)
```
```
# A tibble: 48 × 3
      pkg type  amount
    <dbl> <chr>  <dbl>
 1      1 red        7
 2      2 red       10
 3      3 red        5
 4      4 red        7
 5      5 red        8
 6      6 red        4
 7      7 red        7
 8      8 red        8
 9      9 red        8
10     10 red       11
# 38 more rows
```

Now that the logic of piping is clear, it is worth reiterating that the `|>` operator is a relatively new arrival in base R. Prior to this, the convention would be to use the *tidyverse's* pipe operator `%>%` instead. This comes from a package called *magrittr*, which contains a variety of pipes for different purposes, but the most significant of these is `%>%`. Before R version 4.1.0, `%>%` was the de facto pipe used by the R community at large. However, the recommend wisdom now (even by the keepers of the *tidyverse*) is to use base R's pipe and not *magrittr's*. That being said, many are unaware of this update to base R and much of the help documentation on websites like stack overflow still use *magrittr's* `%>%`.

In terms of functionality, there is little meaningful difference between `|>` and `%>%` and all of the above

code could have been written using `%>%` .

The above examples nicely show how the pipe operator works, but we should consider a more realistic use case to illustrate its versatility.

3.8.1 Data Manipulation Example

Summarising the Data

The M&M data we loaded earlier was of course in the wide format originally, which is rarely needed. So what we could have done instead is loaded that data in to R using `read_csv` , then pipe it to the `pivot_longer()` function and, if we wanted to, pipe that to the `select()` function to avoid keeping irrelevant columns.

```
1  mm_data <- read_csv("MM_Madison_wide.csv") |>
2    pivot_longer(cols = blue:yellow, names_to = "type", values_to = "amount") |>
3    select(pkg, type, amount)
4
5  mm_data
```

```
# A tibble: 288 × 3
     pkg type    amount
   <dbl> <chr>    <dbl>
 1     1 blue        13
 2     1 brown        7
 3     1 green       12
 4     1 orange       9
 5     1 red          7
 6     1 yellow       8
 7     2 blue         8
 8     2 brown        3
 9     2 green       13
10     2 orange      13
# 278 more rows
```

Its worth emphasizing the utility of the pipe operator here: It allowed us to get our data into the form we wanted without creating and calling multiple different objects in memory. Only one object was created, `mm_data` . Moreover, the "arrow-like" notation of the pipe `|>` nicely shows the workflow, i.e., logic, of our code.

Now suppose we wanted to compute some summary statistics for this data set. For instance, maybe (amoung other things) we want to know the mean amount of each type of M&M. This is where the *tidyverse's* functions `group_by()` and `summarise()` become extremely useful. Both of these functions, as well as the `filter()` and `select()` functions we have been using, come from the *dplyr* package in the *tidyverse*.

We will begin with the `summarise()` function which is used to create a data frame based on columns/variables in your data.

```
1  mm_data |>
2    summarise(m = mean(amount))
```

```
# A tibble: 1 × 1
       m
   <dbl>
1   9.10
```

> **Box 3.1: Why is it called *dplyr*?**
>
> Generally, the names of R packages are relatively intuitive or are based on an initialism of some kind. The *dplyr* package is an exception to that. The package's strange name is a reference to both pliers (the tool) and a family of functions based around the `apply()` function that we briefly used in section 3.7.1. The "d" refers to data frames. i.e., its as if you are taking a pair of pliers to data frames.
>
> A common go-to strategy of programmers generally is to use for-loops to do much of the computational grunt work. For-loops just repeatedly execute a set of code until some condition has been satisfied. While for-loops can be used in R, its users often prefer to take a different, more efficient, "vectorized" approach. i.e., The goal is to use what are called **functionals**. These are functions that accept another function as an input and produce a vector as output. That is precisely what the `apply()` function and its relatives like `lapply`, `sapply`, `vapply` do. R is incredibly adept at working with vectors, matrices, and arrays, and *dplyr's* functions are all based around a strategy of using functionals for data manipulation.

The code we have written is telling the `summarise()` function to apply the `mean()` function to the `$amount` column. When it did this, it also created a new data frame and stored that calculation as a column called `$m` (though, we could have named the column whatever we wanted).[8]

At present, none of this may not seem terribly useful; however, we can make it more useful by including the `group_by()` function which will tell R to literally "group by" categories found in a different column or set of columns. Specifically, we can tell it to group by `$type` and then summarise the data.

```
mm_data |>
  group_by(type) |>
  summarise(m = mean(amount))
```
```
# A tibble: 6 × 2
  type       m
  <chr>   <dbl>
1 blue    10.0
2 brown    7.73
3 green   10.1
4 orange  11.3
5 red      7.75
6 yellow   7.69
```

We can now see the mean of each type in the data set and if we wanted, we could create another column showing how many M&M's of each type there are in total by taking the sum of all the blue M&M's, the sum of all the brown M&M's and so on.

```
mm_data |>
  group_by(type) |>
  summarise(
    m = mean(amount),
    tot_type = sum(amount)
  )
```

---

[8]If you are obtaining a giant value like `9.09722222222222e0` you probably forgot to reset the significant digits back to something reasonable. See section 3.6.1.

```
# A tibble: 6 × 3
  type        m tot_type
  <chr>   <dbl>    <dbl>
1 blue    10.0      481
2 brown    7.73     371
3 green   10.1      483
4 orange  11.3      544
5 red      7.75     372
6 yellow   7.69     369
```

If we wanted to add in a column that represented the total amount of M&M's across all the packages (ignoring type) we could take the sum of the entire amount column ...

```
1  sum(mm_data$amount)
```
```
[1] 2620
```

and include that in the `summarise()` function.

```
1  mm_data |>
2    group_by(type) |>
3    summarise(
4      m = mean(amount),
5      tot_type = sum(amount),
6      tot_overall = sum(mm_data$amount)
7    )
```
```
# A tibble: 6 × 4
  type        m tot_type tot_overall
  <chr>   <dbl>    <dbl>       <dbl>
1 blue    10.0      481        2620
2 brown    7.73     371        2620
3 green   10.1      483        2620
4 orange  11.3      544        2620
5 red      7.75     372        2620
6 yellow   7.69     369        2620
```

With the columns `$tot_type` and `$tot_overall` we could determine the percentage of each M&M type by just doing the math inside the `summarise()` function.

```
1  mm_data |>
2    group_by(type) |>
3    summarise(
4      m = mean(amount),
5      tot_type = sum(amount),
6      tot_overall = sum(mm_data$amount),
7      percent = tot_type / tot_overall * 100
8    )
```
```
# A tibble: 6 × 5
  type        m tot_type tot_overall percent
  <chr>   <dbl>    <dbl>       <dbl>   <dbl>
1 blue    10.0      481        2620    18.4
2 brown    7.73     371        2620    14.2
3 green   10.1      483        2620    18.4
4 orange  11.3      544        2620    20.8
5 red      7.75     372        2620    14.2
6 yellow   7.69     369        2620    14.1
```

To finish up, lets also include the maximum and minimum number of each M&M type and store this data frame as an object called `mm_summary`.

```
1   mm_summary <- mm_data |>
2     group_by(type) |>
3     summarise(
4       m = mean(amount),
5       tot_type = sum(amount),
6       tot_overall = sum(mm_data$amount),
7       percent = tot_type / tot_overall * 100,
8       min = min(amount),
9       max = max(amount)
10    )
11
12  mm_summary
```

```
# A tibble: 6 × 7
  type        m tot_type tot_overall percent   min   max
  <chr>   <dbl>    <dbl>       <dbl>   <dbl> <dbl> <dbl>
1 blue    10.0      481        2620    18.4     5    16
2 brown    7.73     371        2620    14.2     3    12
3 green   10.1      483        2620    18.4     5    17
4 orange  11.3      544        2620    20.8     7    17
5 red      7.75     372        2620    14.2     2    12
6 yellow   7.69     369        2620    14.1     2    14
```

## Plotting the Summarised Data

Now that we have all of these summary statistics in a nice convent data frame, we can plot them. In our case, the M&M data contains six discrete categories in the `$type` column. This lends itself nicely to a bar plot, so that is what we shall make.

The basic logic of plotting has been discussed at length in chapter 2, and this discussion will follow from that.[9] The first step will be to give *ggplot2* the data and tell it which columns to map to the x and y axis respectively. Then we will add the `geom_bar()` function to this. In this case, we are going to display the mean (i.e., column `$m`) on the y-axis because that is a fairly standard practice many people will be familiar with.[10] Though, it is worth remembering that any of the other numeric columns could be used as well.

```
1   ggplot(mm_summary, aes(x = type, y = m)) +
2     geom_bar(stat = "identity")
```

---

[9]In other words, if you haven't read chapter 2, go back and do that.

[10]However, I should emphasize that just because everyone jumps off of a bridge, it doesn't mean you should.
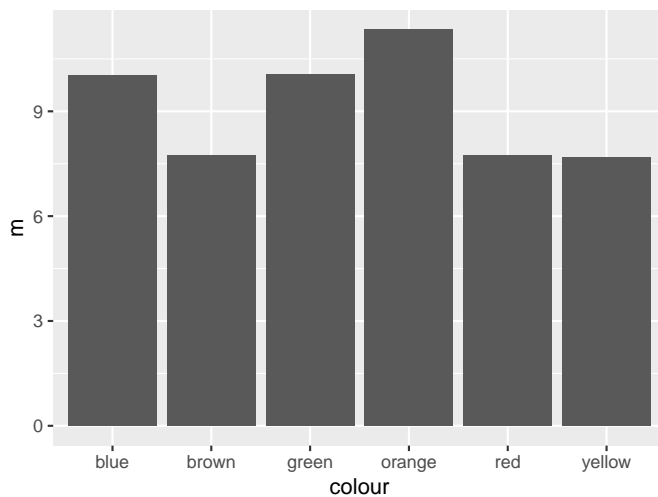
The argument `stat = "identity"` is simply telling *ggplot2* to use the values within the `mm_summary` data frame to create the bars. We needed to specify this because *ggplot2* has the ability to take the raw data directly (e.g., `mm_data`) and perform its own summary calculations. However, we do not need it to do that in this particular case, hence why we included this argument.

You can see that what results is a bar graph representing the mean of each M&M type. To make this look a bit nicer, we could change the fill colour of the bars to correspond to the respective colour types.[11] When we do this, we have to be mindful of the fact that the x-axis contains a *discrete* scale, not a *continuous* one like we saw in chapter 2 (for more information on discrete vs. continuous scales see section 2.9.1).

First we will define our colour palette. To do this we could just specify the names of primary colours: blue, brown, green and so on. However, the actual colour of M&M candies are slightly different than their naming scheme would suggest. The hex codes used below are much more colour accurate.

```
1  mm_palette <- c("#2f9fd7", "#603a34", "#31ac55", "#f26f22", "#b11224", "#fff200")
```

Once `my_palette` has been created we can use it to adjust the colour of our bar graph.

```
1  ggplot(mm_summary, aes(x = type, y = m)) +
2    geom_bar(
3      stat = "identity",
4      colour = "black",
5      aes(fill = type)
6    ) +
7    scale_fill_discrete(type = mm_palette)
```

---

[11]Technically, this is something we should NOT do because, for the sake of comparison, its better to give all the bars the same "visual weight." Keeping all the bars the same colour does precisely that. Moreover, with the x-axis labels, there is no reason to add additional elements that could be distracting. That being said, if you are collaborating on a project, your collaborators will probably demand to see colourful bars irrespective this rationale (experience has taught me this). And if they outnumber you, they can probably beat you in a fight - it doesn't matter if you have the moral or logical high ground.
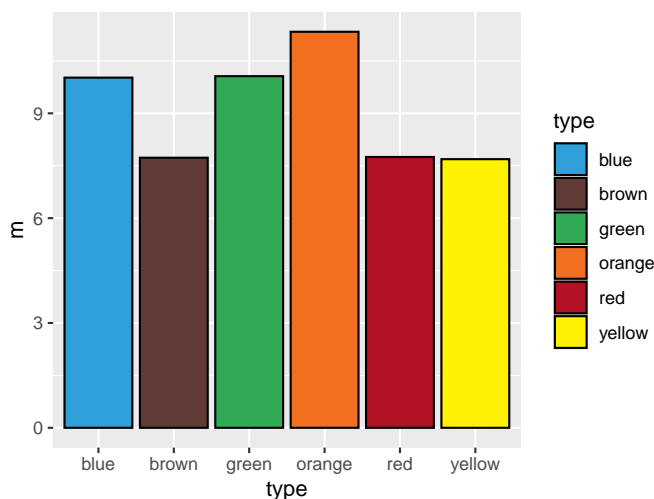
Now, in addition to the mean of each M&M type, our data frame also has information pertaining to the minimum and maximum number of M&M's (these are columns `$min` and `$max` respectively). We could incorporate that information in our graph with the use of *errorbars*. Errorbars are a visual representation of our data's *spread*, and the difference between the minimum and maximum represent a measure of spread called the *range*.[12]

To create errorbars, we can simply use *ggplot2's* `geom_errorbar()` function. We just need to tell it which column corresponds to the bottom of the error bar (`ymin`) and which column corresponds to the top of the errorbar (`ymax`).

```
1   ggplot(mm_summary, aes(x = type, y = m)) +
2     geom_bar(
3       stat = "identity",
4       colour = "black",
5       aes(fill = type)
6     ) +
7     scale_fill_discrete(type = mm_palette) +
8     geom_errorbar(aes(ymin = min, ymax = max), width = 0.25)
```
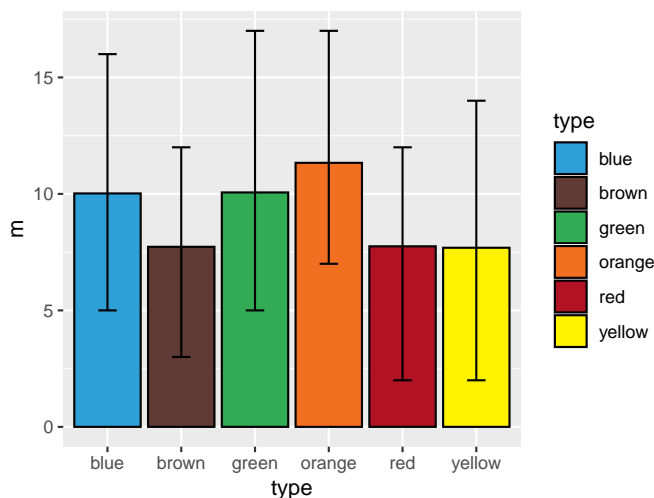

All that remains is to give the plot some new labelling. In other words, we should add a better x and y axis title and we should also put the x-axis labels in Title Case. The legend can be removed as well since it is

---

[12]If that isn't entirely clear, don't worry. The concept of spread as a statistical term will be explained in more detail in later chapters.

redundant with the labels on the x-axis.

To change the current labels "blue", "brown", "green" and so on to Title Case, we can use the function `scale_x_discrete()` and use its `labels` argument. We just have to give it a character vector containing the new labelling (FYI: make sure you specify them in the correct order).

To remove the legend, there are different methods you could employ. In this case, since we only have the fill aesthetic mapped, it is easy enough to just add `guide = "none"` to the `scale_fill_discrete()` function.

```r
1   ggplot(mm_summary, aes(x = type, y = m)) +
2     geom_bar(
3       stat = "identity",
4       colour = "black",
5       aes(fill = type)
6     ) +
7     scale_fill_discrete(type = mm_palette, guide = "none") +
8     geom_errorbar(aes(ymin = min, ymax = max), width = 0.25) +
9     scale_x_discrete(
10      labels = c("Blue", "Brown", "Green", "Orange", "Red", "Yellow")
11    ) +
12    xlab("M&M Colour") +
13    ylab("M&M Amount")
```
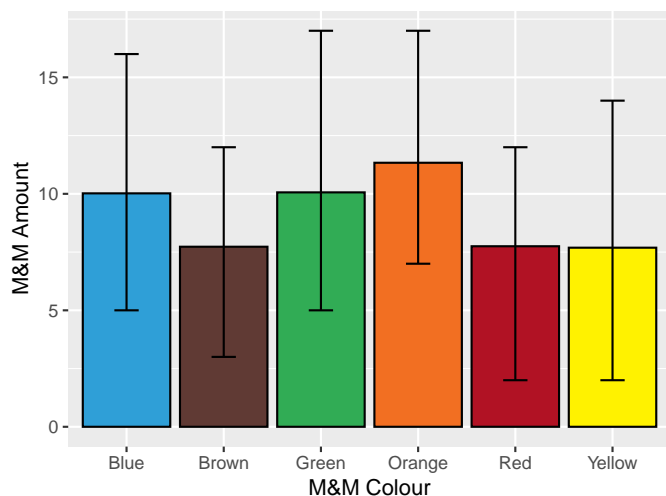

Having read both chapter 2 and now this current chapter, there is one key aspect of plotting that has not been dealt with yet. Specifically, how do you adjust the order of the categories? Suppose we wanted the the colours, going from left to right, to be "red", "orange", "yellow", "green", "blue", and "brown". How would we make that happen? That is where the concept of factors becomes important.

3.9   Factors

In statistics you often speak of a variable as something called a **factor**, and factors have different **levels**. For instance, in our tidy data (`mm_data`) the variable `$type` is what is known as a factor. Each specified colour in that column is a level of that factor: blue is its own level, brown is its own level, green is its own level, and so on. In other words, in the M&M data, the "type" factor has 6 levels.

To summarise, you can treat the term "factor" as synonymous with the terms "column" or "variable."

And you can treat the term "level" as synonymous with the term "category." Though, this only applies to tidy data, not wide data.

- Factor = column / variable
- Level = category within a column / variable

If we examine `mm_data` :

```
1  mm_data
```

```
# A tibble: 288 × 3
      pkg type    amount
    <dbl> <chr>    <dbl>
 1      1 blue        13
 2      1 brown        7
 3      1 green       12
 4      1 orange       9
 5      1 red          7
 6      1 yellow       8
 7      2 blue         8
 8      2 brown        3
 9      2 green       13
10      2 orange      13
# 278 more rows
```

You can see that the output is telling us that the `$type` column is a character vector (notice the `<chr>`). In other words, R does not know that "blue", "brown", "green", etc. are categories. It just sees 278 individual character values in that particular column.

3.9.1   Factoring a Column

For the purpose of plotting and analyses, it is important that R understands "blue", "brown", "green", etc. are levels of a factor (i.e., it is important that it treats these as categories). We can easily tell R that a particular column is a factor using the function `factor()` .[13]

```
1  mm_data$type <- factor(mm_data$type)
2
3  mm_data
```

```
# A tibble: 288 × 3
      pkg type    amount
    <dbl> <fct>    <dbl>
 1      1 blue        13
 2      1 brown        7
 3      1 green       12
 4      1 orange       9
 5      1 red          7
 6      1 yellow       8
 7      2 blue         8
 8      2 brown        3
 9      2 green       13
10      2 orange      13
# 278 more rows
```

---

[13]Technically, when we use this function we are replacing an existing column with a new column that happens to be a factor. We are not really "telling" R it is a factor, we are "creating" a factor - but that's just a nitpicky semantic issue.

Notice that now the column `$type` is now listed as `<fct>`, which stands for "factor." Moreover, if we isolate the column after doing this ...

```
1   mm_data$type
```

```
...
[261] green  orange red    yellow blue   brown  green  orange red     yellow
[271] blue   brown  green  orange red    yellow blue   brown  green   orange
[281] red    yellow blue   brown  green  orange red    yellow
Levels: blue brown green orange red yellow
```

You can see at the bottom of the output, the six levels of our factor have been specified. Generally speaking, to view the levels of a factor, a better practice is to use the `levels()` function.

```
1   levels(mm_data$type)
```

```
[1] "blue"   "brown"  "green"  "orange" "red"     "yellow"
```

3.9.2   Ordering Levels

Discerning readers will have noticed that the order of the levels here (from left to right) matches the order of the bars on the plot we made earlier. This is because anytime you plot or summarise categories using *ggplot2* and *dplyr* functions respectively, these packages silently factor the data behind the scenes and R's default behaviour is to put factors in alphabetical order (which is why we saw the order we did). But we can change the order by specifying it inside the factor function.

```
1   mm_data$type <- factor(mm_data$type,
2     levels = c("red", "orange", "yellow", "green", "blue", "brown")
3   )
4
5   levels(mm_data$type)
```

```
[1] "red"    "orange" "yellow" "green"  "blue"    "brown"
```

It is important to emphasize that this does not change anything about how the data is laid out in our data frame. All those values are still in the same order. All we are doing here is telling R that, when it does it any analyses or plotting, that "red" comes before "orange" which comes before "yellow", and so on. For instance, when we re-run our earlier code that created the summary statistic data, you can see that the `$type` column now follows this new order we have specified.

```
1    mm_summary <- mm_data |>
2      group_by(type) |>
3      summarise(
4        m = mean(amount),
5        tot_type = sum(amount),
6        tot_overall = sum(mm_data$amount),
7        percent = tot_type / tot_overall * 100,
8        min = min(amount),
9        max = max(amount)
10     )
11
12   mm_summary
```

```
# A tibble: 6 × 7
  type       m tot_type tot_overall percent   min   max
  <fct>  <dbl>    <dbl>       <dbl>   <dbl> <dbl> <dbl>
1 red     7.75      372        2620    14.2     2    12
```

```
2 orange 11.3        544         2620      20.8     7     17
3 yellow  7.69       369         2620      14.1     2     14
4 green   10.1       483         2620      18.4     5     17
5 blue    10.0       481         2620      18.4     5     16
6 brown   7.73       371         2620      14.2     3     12
```

Moreover, when we now plot the data the bars will also have shifted their position accordingly.

```
1  ggplot(mm_summary, aes(x = type, y = m)) +
2     geom_bar(stat = "identity")
```
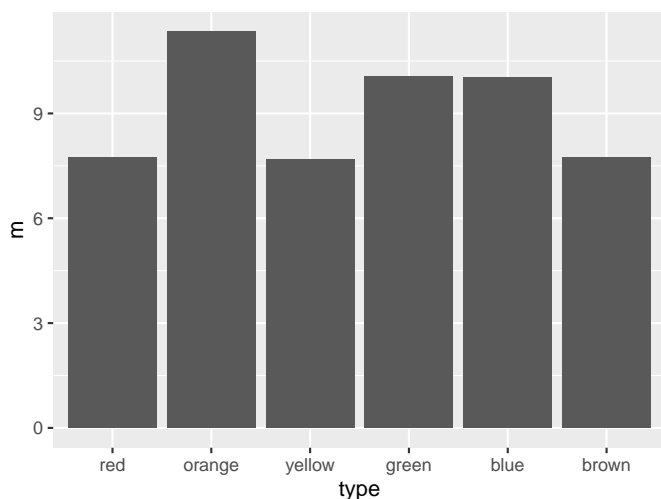

3.9.3 Naming Levels

On occasion, it will be useful to rename the levels of a factor. For instance, all of our levels are currently lower case, but to make them title case we could use the `levels()` function from earlier.

```
1  levels(mm_summary$type) <- c("Red", "Orange", "Yellow", "Green", "Blue", "Brown")
2
3  mm_summary
# A tibble: 6 × 7
   type        m tot_type tot_overall percent    min    max
   <fct>  <dbl>    <dbl>        <dbl>   <dbl>  <dbl>  <dbl>
1 Red     7.75      372         2620    14.2      2     12
2 Orange 11.3       544         2620    20.8      7     17
3 Yellow  7.69      369         2620    14.1      2     14
4 Green  10.1       483         2620    18.4      5     17
5 Blue   10.0       481         2620    18.4      5     16
6 Brown   7.73      371         2620    14.2      3     12
```

A corresponding change will be seen on the plot's x-axis labels as well when that is generated.

A word of warning is needed here. DO NOT confuse the `levels` *argument* inside `factor()` function with the `levels()` *function*. The `levels` argument is used for ordering levels. The `levels()` function is for re-naming levels.[14]

- Ordering levels: `factor(df$column, levels = c(new order))`

---

[14]At the risk of confusing readers, I feel obligated to mention that the `factor()` function has an additional argument `labels` that will allow you to change the level names. See R documentation: `?factor`

- Naming levels: `levels(df$column) <- c(new names)`

Particularly for beginners, factors are annoying to contend with, but they are vital for so many things within R and therefore a necessary evil. Consequently, it is recommended to new users that they submit and wholeheartedly embrace this wickedness. Only then will they find inner peace with factoring.

Glossary

aesthetics The modifiable visual elements of a *ggplot2* graph. E.g., point shapes, fill colours, edge colours, etc.

argument Modifiable parameters of a function that alters how it operates.

assignment operator A symbol (e.g., `<-`) that assigns a name to an object in R so it can be easily sourced by the user from the computer's memory. R contains three different assignment operators. R Documentation: `?assignOps`

bivariate data Data consisting of a two variables.

boolean A term used to denote **logical** (true or false) statements and objects. Named after the English mathematician and logician George Boole.

character A type of storage mode in R for character strings.

colon operator A symbol, `:`, used to create regular sequences of integers. R Documentation: `?colon`

command console An interface used for communicating instructions to a computer and (usually) viewing outputs. On modern digital computers it typically takes the form of a software application but, in ancient times, was a physical console of buttons and dials that you "commanded" the computer from.

Comprehensive R Archive Network A set of mirrored servers around the world that distribute R and its associated packages.

CRAN Comprehensive R Archive Network

data frame A object class in R with rows and columns resembling a spreadsheet structure. R Documentation: `?data.frame`

delimiter A character within a data file used to delimit (i.e., define the limits of) individual values.

directory An address that *directs* you to a file

factor A vector that stores categorical data. Each category within a factor is called a **level**. R Documentation: `?factor`

file extension An identifier appended to the end of a file name that dictates how a file should be read by an application. The extension is indicated by a period followed by one to three characters. E.g., `my_script.R` or `cat.png`

function A line of code that takes inputs (objects and **arguments**) and produces a corresponding output.

functional A function that accepts another function as an input and produces a vector as output. E.g., `apply()`

IDE integrated development environment

infinity Trying to define this is way above my pay grade (which for this textbook is literally nothing). Just see the "Math is Fun" website:

https://www.mathsisfun.com/numbers/infinity.html

integrated development environment A software application that aims to give programmers a nice visual workspace and comprehensive feature set with which to do their programming.

level A category belonging to a **factor** class of object.

logical A type of storage mode in R for logical (i.e., true and false) values (also referred to as **boolean** values).

logical operator A symbol used to refine logical statements. R Documentation: `?Logic`

mode A classification (e.g., numeric, character, logical) of how an object is stored in R.

modulo operator A mathematical operator that returns the remainder of a *dividend* and *divisor*.

modulus The value returned using a modulo operation.

multivariate data Data consisting of a more than two variables.

negation operator Symbolized using a exclamation mark (`!`), this is a type of **logical operator** that indicates the negation of an object's values. For example, `!x` is read as "not x."

non-syntactic name A object name enclosed by backticks. E.g. `` `fav num` <- 666 ``.

null value Represented as `NULL` in the R language, this is used to represent undefined objects. R Documentation: `?NULL`

numeric A type of storage mode in R for numbers.

package A collection of functions, associated documentation, and data compiled for users to install via a online repository.

pch R's abbreviation for "plotting character". An integer or character value that specifies what symbol gets plotted as a point on a graph. R Documentation: `?points`

position scale In *ggplot2*, this refers to a type of scale that controls the location mapping of a plot's visual elements.

programming language A language humans use to communicate instructions to a computer.

relational operator A symbol (e.g., `==`) that is used to determine whether a specific comparison between two values is true or false. R Documentation: `?Comparison`

reserved words Words that have specific functions and meanings within the R language and cannot be used as syntactic names. R Documentation: `?Reserved`

RStudio An **integrated development environment** for R.

scatter plot A type of graph that is used to visualize the relationship between two paired variables. The observations of one variable are plotted on the x-axis, while the observations of the other variable are plotted on the y-axis. The intersection of the x-y pairs are plotted as points on a Cartesian plane (i.e., a grid). For further details see https://www.mathsisfun.com/data/scatter-xy-plots.html

scientific notation A method of writing very large or small numbers in a compact way. E.g., 66613000 can be written as 666.13×10^5 or `666.13e+5`

script A text document (e.g., .R or .txt) for storing computer code that can be run or modified by a user. Integrated development environments usually provide a separate window for typing and saving scripts.

subdirectory A directory nested within another directory.

syntactic name A object name consisting of letters, numbers and the dot or underline characters and starts with a letter or the dot not followed by a number. R Documentation: `?make.names`

tibble The tidyverse's modern reimagining of the data frame.

tidy data A sacred formation of data, guided by three precepts, that form the bedrock of the **tidyverse's** magik. Also referred to as the "long format" data by unbelievers.

tidyverse A powerful set of R magick, with an underlying philosophy, that allows those devoted to it to weave, transform, and manipulate data with a dark mystical ease that some call unnatural.

univariate data Data consisting of a single variable.

vector In R, a (atomic) vector is an object with at least one value and a single **mode**. R Documentation: `?vector`
In computer programming more generally, a vector is a one-dimensional array of values.

wide data Data which spreads variables across multiple columns.

working directory The default address on a computer where R saves and pulls files.

Appendix A

<- vs. =

The original assignment operator of the S programming language was `<-` . The use of `=` to assign names to objects was a more recent development in S's history. This was doubtlessly motivated by 1) the intuitive appeal of `=` (you are setting something *equal* to something else), 2) its cleaner look, 3) its correspondence with other modern programming languages, and 4) the basic fact that it requires one less key to type. It also has the added benefit of not resulting in confusion when dealing with inequalities. For instance, something like `x<-1` could be read as either assigning a value of 1 to `x` or could be evaluating whether `x` is *less* than −1. As written here, the statement will result in the former unless appropriate spacing is applied; i.e., `x < -1` .

Despite the obvious benefits of using `=` , much of R's core user-base has held as steadfastly to `<-` as a child would to a teddy bear. To understand the reluctance towards using `=` , it is helpful to know that, prior to its use as an assignment operator, the `=` was used to designate values to *arguments* inside a *function* (see section 1.4.7) and, to this day, it still serves this purpose. Consequently, when it was granted the coveted position of "assignment operator" it now had dual syntactic roles within the language but with a particular limitation. Specifically, you cannot use it to assign a name to an object within an R function's argument. i.e., you cannot use `=` to set an argument and assign an name simultaneously. However, you can do this using the `<-` .

For example, if we use R's `sum()` function to calculate the sum of the numbers one through five using `=` to set the function's main *argument*. We can see that, while the function works as intended (producing a value of 15), there is no new variable generated that stored the values one through five:

```
1  sum(x = 1:5)
2  x
```
```
[1] 15
Error: object 'x' not found
```

However, if we run the same code, but use the `<-` to set the argument, we can see that the numbers 1 through 5 are stored.

```
1  sum(x <- 1:5)
2  x
```
```
[1] 15
[1] 1 2 3 4 5
```

The `<-` also has an advantage in that a simple variant of it, `<<-` , allows you to create variables within your own custom-made functions that are executable outside the scope of that function. Admittedly, this is a

more advanced usage than readers of this text are likely to need, but it is an useful feature to know about as skills with R develop.

As a basic illustration, suppose we created a function, `rational_pi()`, that rounds π to 3 like so...

```
1  rational_pi = function() {
2    rat_pi <- round(pi)
3    return(rat_pi)
4  }
```

When we run the function, it straightforwardly spits out a 3

```
1  rational_pi()
```
```
[1] 3
```

But when we run object `rat_pi` we get an error message saying the object cannot be found:

```
1  rat_pi
```
```
Error: object 'rat_pi' not found
```

At face value this is odd behaviour because, to be able to run the line `return(rat_pi)`, the object `rat_pi` must have been stored at some point. And it was stored, but only *within the scope of the function*. To make `rat_pi` available outside the function's scope, we can employ `<<-` when we define the function:

```
1  rational_pi = function() {
2    rat_pi <<- round(pi)
3    return(rat_pi)
4  }
5
6  rational_pi()
7  rat_pi
```
```
[1] 3
[1] 3
```

Now we have a "rational" version of π stored as `rat_pi`. However, one other intriguing feature of `<<-` needs to be mentioned in this context: `<<-` only assigns a value within the function's scope, *if* the object you are creating does not already exist inside the function. However, the value will still get assigned globally (i.e., outside of the function's scope). This is easiest to comprehend with a simple example:

```
1  rational_pi = function() {
2    rat_pi <- 10
3    rat_pi <<- round(pi)
4    return(rat_pi)
5  }
6
7  rational_pi() #Notice the function produces 10
8  rat_pi #However, the object stores 3
```
```
[1] 10
[1] 3
```

A couple of other final points in favour of `<-` is its reversibility (i.e., being able to write it as `->` and `->>`) and the fact that most of the example code inside R's help documentation is written using `<-`. Thus, in theory, using `<-` consistently is likely to make this documentation more intelligible at a quick glance for a user than constantly using `=` would.

Appendix B

HCL Colour Palettes

B.1 Sequential Palettes

Grays

Light Grays

Blues 2

Blues 3

Purples 2

Purples 3

Reds 2

Reds 3

Greens 2

Greens 3

Oslo

Purple-Blue

Red-Purple

Red-Blue

Purple-Orange

Purple-Yellow

Blue-Yellow

Green-Yellow

Red-Yellow

Heat

Heat 2

Terrain

Terrain 2

Viridis

Plasma

Inferno

Rocket

Mako

Dark Mint

Mint

BluGrn

Teal

TealGrn

Emrld

BluYl

ag_GrnYl

Peach

PinkYl

Burg

BurgYl

RedOr

OrYel

Purp

PurpOr

Sunset

Magenta

SunsetDark

ag_Sunset

BrwnYl

YlOrRd

YlOrBr

OrRd

Oranges

YlGn

YlGnBu

Reds

RdPu

PuRd

Purples

PuBuGn

PuBu

Greens

BuGn

GnBu

BuPu

Blues

Lajolla

Turku

Hawaii

Batlow

B.2 Diverging Palettes

Blue-Red

Red-Green

Green-Brown

Green-Orange

Broc

Berlin

Blue-Red 2

Purple-Green

Blue-Yellow 2

Cyan-Magenta

Cork

Lisbon

Blue-Red 3

Purple-Brown

Blue-Yellow 3

Tropic

Vik

Tofino

B.3 Qualitative Palettes

Pastel 1

Set 2

Cold

Dark 2

Set 3

Harmonic

Dark 3

Warm

Dynamic

References

Becker, R. A., & Chambers, J. M. (1984). *S: An interactive environment for data analysis and graphics.* Wadsworth.

Brewer, C. A. (1994). Color use guidelines for mapping and visualization. In A. M. MacEachren & D. R. F. Taylor (Eds.), *Visualization in modern cartography* (pp. 123–147, Vol. 2). https://doi.org/10.1016/B978-0-08-042415-6.50014-4

Bro. (n.d.). *Dude, trust me.*

Carr, D. (1994). Using gray in plots. *ASA Statistical Computing and Graphics Newsletter*, *2*(5), 11–14.

Carr, D. (2002). Graphical displays. In A. H. El-Shaarawi & W. W. Piegorsch (Eds.), *Encyclopedia of environmetrics* (pp. 933–960, Vol. 2). John Wiley & Sons.

Carr, D., & Sun, R. (1999). Using layering and perceptual grouping in statistical graphics. *ASA Statistical Computing and Graphics Newsletter*, *10*(1), 25–31.

Cleveland, W. S. (1993). A model for studying display methods of statistical graphics. *Journal of Computational and Graphical Statistics*, *2*(4), 323–343. https://doi.org/10.2307/1390686

Doré, G. (1862). *Little red riding hood* [Painting]. National Gallery of Victoria, Melbourne. https://www.ngv.vic.gov.au/explore/collection/work/3918/

Free Software Foundation. (2022). *What is free software?* Retrieved June 27, 2022, from https://www.gnu.org/philosophy/free-sw.html

Madison, J. (2007). *M & M's color distribution analysis.* Retrieved July 24, 2024, from https://joshmadison.com/2007/12/02/mms-color-distribution-analysis/

Muenchen, B. (2017). *R-bloggers: The tidyverse curse.* Retrieved July 18, 2024, from https://www.r-bloggers.com/2017/03/the-tidyverse-curse/

Pennant, T. (1784). *A tour in wales* (Vol. 4). http://hdl.handle.net/10107/4691510

Pierce, R. (2022). *Math is fun: What is a function.* Retrieved July 9, 2022, from http://www.mathsisfun.com/sets/function.html

Tidyverse. (2024). Retrieved July 10, 2024, from https://www.tidyverse.org/

Tufte, E. R. (2006). *Beautiful evidence.* Graphics Press.

Turing, A. M. (1950). Computing machinery and intelligence. *Mind*, *59*(236), 433–460. https://doi.org/10.1093/mind/LIX.236.433

UNESCO. (2021). UNESCO recommendation on open science. https://doi.org/10.54677/MNMH8546

Wickham, H., Çetinkaya-Rundel, M., & Grolemund, G. (2023). *R for data science: Import, tidy, transform, visualize, and model data* (Second). O'Reilly Media. https://r4ds.hadley.nz/

Wickham, H., Navarro, D., & Pedersen, T. L. (2024). *ggplot2: Elegant graphics for data analysis (3e).* Retrieved July 18, 2024, from https://ggplot2-book.org/

Wickham, H., Averick, M., Bryan, J., Chang, W., McGowan, L. D., François, R., Grolemund, G., Hayes, A., Henry, L., Hester, J., Kuhn, M., Pedersen, T. L., Miller, E., Bache, S. M., Müller, K., Ooms, J., Robinson, D., Seidel, D. P., Spinu, V., … Yutani, H. (2019). Welcome to the tidyverse. *Journal of Open Source Software*, *4*(43), 1686. https://doi.org/10.21105/joss.01686

Zeileis, A., & Murrell, P. (2019). *HCL-based color palettes in grDevices.* Retrieved July 21, 2024, from https://developer.r-project.org/Blog/public/2019/04/01/hcl-based-color-palettes-in-grdevices/