The Magic Circle by J. M. Waterhouse

The Statistical Grimoire: Statistics for the Natural Sciences Using R

Version 4.0.0

Dr. Jeffrey M. Pisklak
University of Alberta

https://statistical-grimoire.neocities.org/
https://github.com/statistical-grimoire/book
https://bsky.app/profile/statgrimoire.bsky.social
statistical-grimoire@proton.me

This is LuaHBTeX, Version 1.18.0 (TeX Live 2024)
Generation Date: 2025-06-30

Header Typeface: IM Fell English
Body Typeface: Computer Modern
Code Typeface: Source Code Pro (Hunt, 2024)

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

This book is dedicated to Satan and the only three good math instructors I ever had. It is written in spite of all the others.

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 (i.e., free) 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 manner. Please consider everything herein a work in progress.

Foolish Assumptions Made by Your Author:

- You will actually read this (yes, I'm an optimist).
- You probably don't like math. You might even fear it, but deep down, you know it's good for you. Math, like confession, is rarely pleasant but often necessary.
- You'll start at the beginning and move forward. Because jumping ahead is like summoning something you don't understand. It will arrive—and you won't like what it brings.
- You won't take the sillier stuff too seriously—especially the parts that sound true. They probably aren't. Probably.

Errata:

- No manuscript, however cursed or consecrated, is free from the creeping taint of error. Should you unearth any horrors—do not consign your discovery to silence. Instead, record your findings within the digital vaults of GitHub, where the keeper of this tome may attempt to contain and exorcise the corruption ... before it spreads further.
  - https://github.com/statistical-grimoire/book/issues

Contents

PART I

R Programming - An Initiation

What lies ahead in this first part is nothing short of a trial by fire. This section contains a wealth of information that, at first glance, should feel overwhelming - like staring into the heart of an inferno. Any reader who believes this content must be memorized is venturing down a perilous path, one that will surely lead to being consumed by the flames they are stepping in to.

The goal here is not to transform the reader into an expert with R, programming, or statistics. Instead, this section is designed to immerse the reader in the R language, offering hands-on experience while building a strong foundational understanding. Think of it as the first incantations in the dark art of programming. The focus should be on grasping the underlying logic of the code rather than committing it to memory.[1] Programming is not merely a list of spells to recite by rote—it is a craft, a skill honed through practice and understanding. Readers would do well to approach it wisely, or risk being scorched by the fire they seek to wield.

[1] Every section in this book has a corresponding bookmark in the PDF file for quick reference.

Chapter 1

Summoning Basics:
An Introduction to R

To begin with ...

1.1 What is R?

R is a programming language. A programming language is a means by which mortals commune with the cold, unyielding logic of machines. Humans—featherless bipedal apes known as *Homo sapiens*—fashion these languages as incantations to command lifeless thralls known as computers. These soulless automatons follow rigid, unforgiving rules, devoid of will, incapable of deviating from their ordained paths (Turing, 1950). Yet, as the ancients knew, the boundary between man and machine is a shadowy one. In a bygone age, computation was not formed of silicon and wires but of flesh and bone. Mortal "computers" laboured in dimly lit chambers, chained to the drudgery of endless calculations, their toil dictated by the cryptic marks and guttural utterances of their superiors. Today, the human computer is a relic, swept away by the inexorable tide of technological sorcery, as mathematics and other arcane arts, such as reading and writing, wither further and further into obscurity, forsaken by the students of this benighted age.

In the modern day, the digital computer reigns as the unchallenged sovereign of computation. Forged from silicon and shrouded in enigmatic circuitry, this mechanical progeny bears a dark resemblance to its primate forerunner—save for one vital distinction: it is unflinchingly logical. Some mortals, perhaps out of desperation or hubris, even dare to call these constructs "smart." Such claims, however, speak volumes about the feeble intellects of their human masters rather than the devices themselves. With these machines now enthroned, society has deemed it unnecessary to burden young minds with the rites of a once-venerated education. Mathematics—the ancient domain of sweat, tears, and torment—has been cast aside, its cruel trials abandoned, and with them, the rich harvest of discipline and understanding that

once emerged from the crucible of suffering.

1.1.1 The Genesis of R

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 might, charitably, be called 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 reader's 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.

1.2 Why a Programming Language?

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 there 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 programming? In other words, why not use software such as SPSS[1] or one of its many malformed, and equally expensive, doppelgängers, Minitab, SAS, and Stata.

The primary answer to this question lies in flexibility. There is rarely a single correct way to analyze data, as different datasets come with their own unique challenges and intricacies. These complexities often resist the rigid, prescriptive approaches employed by many proprietary software programs. This is not to say software like SPSS cannot adapt to such scenarios—it often can. However, this adaptation typically comes at a cost: users may need to pay for additional features not included in the original purchase, or they may face an even steeper learning curve. One that forces mastery of a obscure and enigmatic language. A language that is so specific to the software, that only a select few (if any) even seem to understand it.

In direct contrast to this, R offers an intuitive and empowering experience for users. While it may seem daunting at first, R operates in a straightforward and logical manner, much like a calculator. Many users discover that wielding R is far easier than they initially expected. This

[1] 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.*

is largely due to the vibrant and dedicated R community that exists online, which has cultivated an extensive network of resources over the years. Acolytes of R see it as something worthwhile to support and develop—often at their own personal time and expense. Proprietary software like SPSS has no equivalent to this, nor will it ever. Users are often snared within its ecosystem not out of preference or love for the program, but because it is all they have ever known.

Although programs like SPSS may initially appear familiar to new users due to their resemblance to popular spreadsheet software like Microsoft Excel, this familiarity is purely superficial. To illustrate, SPSS and its equivalents will, in a manner similar to most spreadsheet software, present users with an array of buttons and menus above a friendly spreadsheet-style grid. However, the similarities end there. New users will quickly find themselves overwhelmed by the abundance of strange options needed to perform even simple tasks, such as loading and viewing a dataset. Contrary to what their marketing might lead you to believe, the learning curve for these programs is dangerously steep, and users are unknowingly at risk of being lead off a cliff.

Owing to its nature as a programming language crafted for statistics, R is grounded by the logic of mathematics. This foundation often makes it easier for new users to understand and build upon, even for those who claim to dislike math. Moreover, proficiency with R grants users the ability to work with other statistical software if needed. The reverse, however, is rarely true: mastering SPSS or similar programs does not provide the same level of flexibility or transferable skills.

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 SPSS and other, slightly less talked about, SPSS-style software like Minitab, and Stata, 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, 2024):

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.
- 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 philosophy extends beyond the software itself to include both its file formats and help documentation. For years, the dissemination of scientific findings has been hindered by the reliance on proprietary file formats imposed by commercial research tools. Locking information within these exclusive systems is clearly counterproductive to scientific progress, as it binds researchers to overpriced, branded ecosystems. Such practices prioritize profit over the broader goals of accessibility and collaboration, making their continued adoption ethically questionable. In practical terms, this means that choosing R is not just about its functionality—it is also a statement against the restrictive and exploitative behaviors perpetuated by proprietary software providers. More plainly, and if for no other reason, we should use R just to give the middle finger to these companies.

As if you didn't need any other reasons to start using R immediately, here are some more:

- **R Is Not A Gooey Mess:** Unlike programs tied to a graphical user interface (GUI, often called "gooey"), R is not limited by point-and-click constraints. Its capabilities are as vast as what you and others can program—and your computer can handle.
- **Advanced Statistical Capabilities:** R's packages make it easy to apply best practices in statistics.
- **Enhanced Data Visualization:** With intuitive tools like *ggplot2*, R easily permits sophisticated and customized visualizations, helping you communicate findings with clarity and impact.
- **Reproducible Research:** R is built for reproducible research, aligning with open-science principles. It allows you to create scripts that are easy to share, review, and rerun, by anyone. This helps ensure transparency, accuracy, and reliability.
- **Integration with Other Tools:** R can easily integrate with other software and programming languages, such as Python, SQL, HTML, LaTeX, and even Excel. This makes it a

valuable tool for working in diverse computational environments.

- **Growing Demand in the Job Market:** R is highly valued in the job market, particularly in data science, analytics, and research. Mastering R opens up a wealth of career opportunities.

In summary, while the prospect of learning a programming language like R may seem daunting at first, it ultimately provides a more adaptable, intuitive, ethical, affordable, and rewarding tool for statistical analysis than many of its proprietary counterparts.

1.2.1 Why R?

At this point, it is worth addressing a question that comes up often: "*Why use R instead of something like Python or Julia?*" It is a fair question. After all, Python and Julia—like R—are full-fledged programming languages that are powerful and capable, but the difference comes down to origin story.

R is not a general-purpose language. It is the progeny of S, a language birthed in the depths of statistical practice for one primary goal: GOOD DATA ANALYSIS. Everything about R—from its object types to its default printing behaviour—is tailored to the sorts of things data analysts do every day.

By contrast, Python and Julia are general-purpose languages. They are designed to do many things well: build websites, run simulations, automate tasks, and yes, analyse data. But this means that good data analysis is a goal these languages aspire to, not one they were born to achieve.

To illustrate, suppose you wanted to calculate the mean of the numbers 1 to 666. In R, this is as natural as a reflex:

```r
mean(1:666)
```

```
333.5
```

No need to import packages. No need to loop. No need to define arrays. It just works.

In Python, you will find that a bit more ceremony is required:

```python
x = list(range(1, 667))
print(sum(x) / len(x))
```

```
333.5
```

Additional packages, like Python's excellent *numpy* package can simplify what needs to be written, but it still is not quite so good as what R offers as a baseline user experience.

Julia sits somewhere in the middle. It was built with scientific computing in mind, and its syntax can often be just as clean as R's:

```
using Statistics
mean(1:666)
```

```
333.5
```

However, Julia still expects you to opt in to statistics, with basic functions like `mean()` not being available until you explicitly load them. That is not a flaw—it's a philosophical choice. Julia gives you a lean, high-performance core, and leaves the rest to you.

R, by contrast, assumes you are doing data analysis. You do not have to ask for permission to compute a mean. Like a loyal familiar, it is already waiting for you. Moreover, neither Julia nor Python have the mature ecosystem of statistical tools, diagnostic plots, or nuanced modelling features that R does—at least not yet. R's statistical packages, often written by the very people developing the methods, remain second to none.

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 use to communicate instructions to a 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**[2] of some kind. On computers using the Windows operating system, this is referred to as the *Command Prompt* application, on Macintosh and Linux computers this is the *Terminal* application. Relying on your operating system's basic command console application as a primary interface is often a cumbersome and inefficient experience, and definitely not a recommended course of action - though, for what it is worth, Linux users seem to delight in this sort of thing. 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 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.

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

The standard installation of R will come with an associated environment for the user - provided they are working with either a Windows or Macintosh operating system (see Figure 1.1). 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. An image of the RStudio environment can be seen in Figure 1.2. RStudio is highly customisable in both appearance and function and, consequently, can be tailored to each users personal preferences. Figure 1.2 shows the default appearance upon first installation.

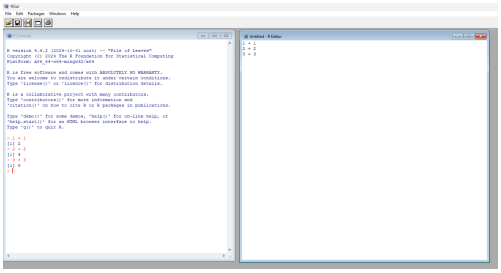


Figure 1.1: Base R installation environment, featuring the default console interface for running R code and viewing outputs (left) and the default scripting window (right) for writing, running, and saving R code.

1.3.2 Installation

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

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

Located 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 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 Linux, then you probably already know what to do. 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. For Macintosh users, when you download the install file for R studio there probably will not be an installation wizard, rather you likely be prompted to "drag" an R studio icon into your applications folder. Once that is done, R studio is installed.

1.3.3 Upgrading

There are updates made to R about two to three 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.[3] 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 4.4.2, nicknamed the "Pile of Leaves" version. New releases of R are given nicknames that, inexplicably, are all obscure references to Peanuts (a.k.a. Charlie Brown and Snoopy) comic strips.

1.3.4 Consoles, Scripts, and Running R Code

The Console

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 (see Figure 1.2). 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 is another way of expressing the fact that we are typing commands using a programming language. The presence of > also indicates that the computer is awaiting your command.

[3] Packages (also commonly referred to as "libraries") will be explained later.

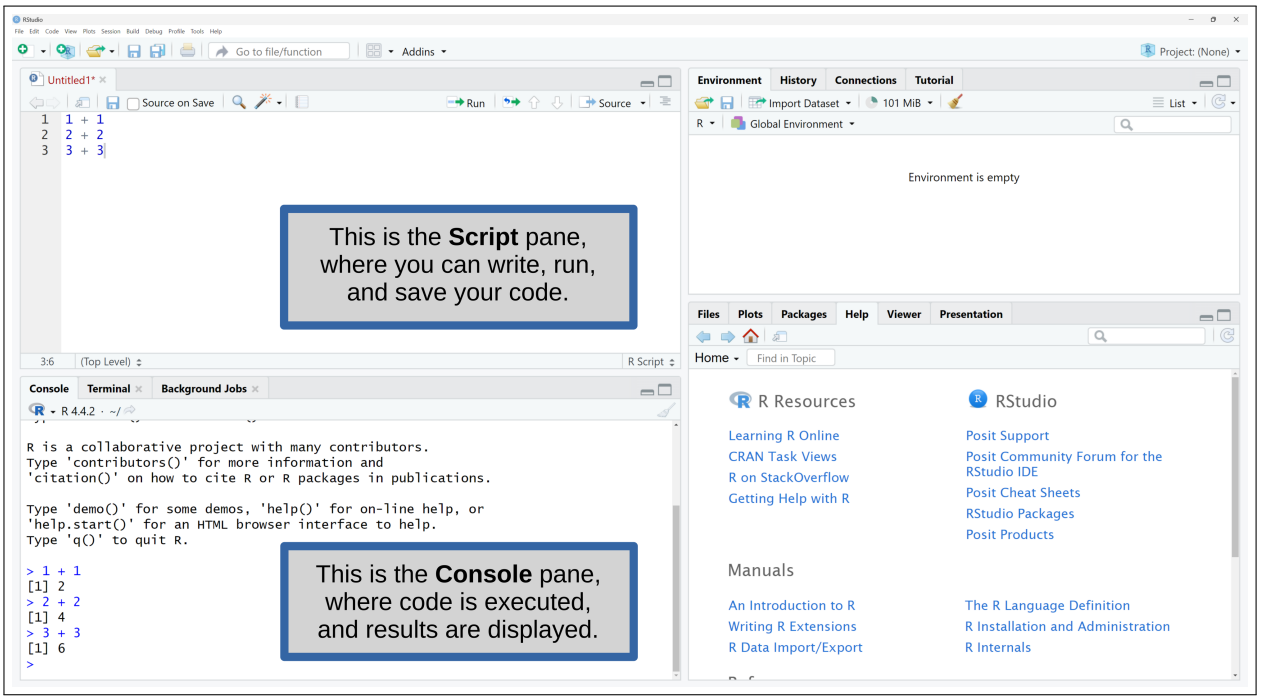


Figure 1.2: Layout of RStudio. Each pane serves a specific purpose in writing, running, and managing R code. By default the scripting pane (top left) is not shown and can be accessed by selecting *File* → *New File* → *R Script*.

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. The "2" is the computer's resulting *output*.

Input:

```
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.

The Script

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 script pane. In R Studio the process is *File* → *New File* → *R Script*.

Once opened, you can type R code into this new pane and save it in the conventional manner of most word processing applications (i.e., *File* → *Save As*). Additionally, this pane permits you to run lines of code selectively or all together. For instance, if you type the following into the script pane ...

```
1 + 1
2 + 2
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.

1.3.5 Keyboard Shortcuts

It is at this juncture that a noteworthy feature of programming environments be mentioned; specifically, keyboard shortcuts (also called "hotkeys"). All robust programming environments equip users with the ability to perform virtually any non-typing task directly from the keyboard, increasing efficiency and comfort. 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, will 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.[4]

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

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

Table 1.1: Useful Keyboard Shortcuts

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 advice to novice programmers be offered. Nothing that will be discussed in this section, or any section of this text concerning R 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. Memorization will happen effortlessly over time.
- 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.

Box 1.1: Are you using your keyboard properly?

When utilizing the keyboard shortcuts mentioned in section 1.3.5, 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 and many 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.

1.4.1 Basic Arithmetic

At its core R is really nothing more than a powerful calculator, and we can use it as such. R can be used to add (+), subtract (−), multiply (×), and divide (÷).

```
666 + 13
13 - 666
9 * 27
666 / 9
```

```
[1] 679
[1] -653
[1] 243
[1] 74
```

Exponents can be incorporated as well by using the ^ ('caret'), symbol. For instance, the expression 9^3 can be written as ...

```
9^3
```

```
[1] 729
```

R will also follow the ritualistic *order of operations* when dealing with more complex expressions. To illustrate, consider the mathematical statement $8 \div 2(2 + 2)$. 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[5]), 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.

```
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.

```
8/(2+2)*2
```

```
[1] 4
```

R also has the ability to perform *Euclidean Division*, which many may recall from their long suffering days in primary education days as simply *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.

```
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 ...

```
11 %/% 2
```

```
[1] 5
```

To obtain the remainder we type...

```
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**.

[5]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 the inferior variants of this mnemonic, PEDMAS and PEMDAS.

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?

```
1/0
-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.[6]

What happens if you take the square root of a negative number?[7]

```
(-4)^(1/2)
```

```
[1] NaN
```

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. To use it, you need only type `pi`.[8]

```
pi
```

```
[1] 3.141593
```

[6] 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. 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`.

[7] Recall that exponents can be used to take the square-root of a number. For example, $\sqrt{4}$ can be expressed as $4^{\frac{1}{2}}$.

[8] 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 not to go beyond 16 even though a max of 22 are possible. If you want R to always display all 16 digits, you can change its default behaviour by running `options(digits = 16)`, though this is not recommended.

1.4.2 Understanding Scientific Notation

On occasion values will be either 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:

```
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 but not necessarily run. 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
# 2 + 2
3 + 3
```

```
[1] 2
[1] 6
```

```
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.

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.

```
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.

```
x
```

```
[1] 1
```

R is technically classified as an object-oriented programming language (an "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 most R users, but it is important because it means that if you do something like this . . .

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

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

```
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. ...

```
x <- x + 1
x
y
```

```
[1] 2
[1] 1
```

The "Environment" pane in R studio will show you the complete list of objects presently loaded in memory. The pane can be displayed by selecting the "View" menu and selecting "Show Environment" or pressing Ctrl + 8 on your keyboard.

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

```
y = x + 4
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.

```
10 -> z
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.[9]

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.

```
x <- "SPAM"
x
```

```
[1] "SPAM"
```

Both single or double quotation marks can be used to define a character object. For instance, running ...

```
y <- 'SPAM'
y
```

```
[1] "SPAM"
```

works just fine, but if you were to mix and match the two types of quotation marks (e.g., try to run `y <- "SPAM'`), you will find that no actual output is produced, and the console just displays the code you tried to run with a small `+` appended to it. The `+` indicates that the line of code

[9] 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.

is incomplete and more is expected before an output can be returned. If this happens you need only press the escape key (*esc*) 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.

```
y * 5
```

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

```
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.

```
x <- TRUE
y <- FALSE
x
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.

Thus far, we have demonstrated three basic categories of object: *numeric*, *character*, and *logical*. R refers to these various categories as **modes**,[10] and as you progress with R, both in this book and more generally, you will encounter other object modes.

Naming Objects

Oftentimes 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

[10] 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.

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:

```
snake_case <- 1
```

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

```
camelCase <- 2
```

There is also *period case*:

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

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

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

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

```
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[11] (https://style.tidyverse.org) 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 villainy is more your style, you can gleefully ignore all of those rules and conjure up what are called **non-syntactic names** by simply enclosing the name within backticks.

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

[11]The *tidyverse* will be explained in the next chapter, just know that all the code written in this book will (do its best) to adhere to its standards.

1.4.5 Vectors

It is not the case that an object need store only 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.[12] For instance, if we want `x` to be equal to the numbers 1 through 5, we need only type:

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

```
[1] 1 2 3 4 5
```

The lower case `c` is short for *combine*. By combining the numbers 1 through 5 in this way we have created what is technically known as a **vector**.[13] We can further use this combine function, `c()`, to combine vectors with other vectors. In the example below, we create two vectors, `x` and `y`, and combine them to create an object called `z`.

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

```
[1]  1  2  3  4  5  6  7  8  9 10
```

Box 1.2: How to Use Your Colon Effectively

In the previous examples, we used R's combine 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** (`:`).

```
x <- 1:5
x
```

```
[1] 1 2 3 4 5
```

This can also be used in reverse and with negative values.

```
3 : -5
```

```
[1] 3  2  1  0 -1 -2 -3 -4 -5
```

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

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

The concept of a vector is one which will have relevance to people with a fondness of linear and matrix algebra[14] 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.

```
x + 2
```

```
[1] 3 4 5 6 7
```

Correspondingly:

```
x - 2
x * 2
x / 2
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 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.

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

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.

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

```r
x <- c(1,2,3,4)
y <- c(6,7,8,9,10)
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.

```r
a <- c(1,2,3)
b <- c("BREAD", "SPAM", "BREAD")
c <- c(TRUE, FALSE, FALSE)

a
b
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.[15]

```r
d <- c(5, "SPAM", 6, 7, 8)
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.

```r
e <- c(666, TRUE, FALSE)
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.

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

```r
x <- c(100)
g <- c(TRUE, TRUE, FALSE, FALSE, TRUE)
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.

```r
LETTERS
letters
month.name
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 14^{th} position. In the vector `month.name`, `"May"` is in the 5^{th} 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).

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 subset values by referencing their position in the vector. For instance, if we want to know what the 17^{th} letter of the English alphabet is, we need only type...

```r
LETTERS[17]
```

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

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

```
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 combine symbol.

```
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.

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

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

Somewhat uniquely, within the R programming language every *single* basic data object—such as a number, character string, or logical value—is inherently a vector. For example, the number `666` is a vector, the character string `"SPAM"` is a vector, and the logical value `FALSE` is also a vector. These are simply vectors with a single element, or a length of 1. As such, all of these can be indexed just like any other vector with a length greater than 1.

```
666[1]
666[2]
"SPAM"[1]
FALSE[1]
```

```
[1] 666
[1] NA
[1] "SPAM"
[1] FALSE
```

Notice in the second line above that when `666` was indexed at position 2, R returned `NA` because no value exists at that position. The moral of the story is that, when the combine function, `c()`, is used, you are not creating a vector, but rather combining vectors.

Type	Operator	Description
Assignment	`x <- value`	Assign a value to a name.
	`value -> x`	Assign a value to a name.
	`x <<- value`	(see Appendix A)
	`value ->> x`	(see Appendix A)
	`x = value`	Assign a value to a name.
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

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.

```
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.

```
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:

```
pi >= pi
```

```
[1] TRUE
```

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

```
pi <= 3
```

```
[1] FALSE
```

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

```
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, 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)`.

```
sqrt(25)
```

```
[1] 5
```

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

```
x <- sqrt(25)
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.[16]

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

Calculating the *sum* of all the values:

```
sum(x)
```

```
[1] 15
```

Calculating the *product* of all the values:

```
prod(x)
```

```
[1] 120
```

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

```
min(x)
max(x)
```

```
[1]  1
[1]  5
```

[16] 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()`

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

```
length(x)
```

```
[1] 5
```

Calculating the *mean* of all the values:

```
mean(x)
```

```
[1] 3
```

Calculating the *median* of all the values:

```
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.[17]

```
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 $\sqrt{2}$

```
x <- c(pi, sqrt(2))
x
```

```
[1] 3.141593 1.414214
```

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

```
round(x, digits = 2)
```

```
[1] 3.14 1.41
```

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

Alternatively, we could round to the nearest integer:

```
round(x, digits = 0)
```

```
[1] 3 1
```

Critically, in the above two examples, we have specified the `digits` argument using an `=` sign. Generally speaking, this is the best practice and original purpose of `=` because, while you are permitted to use the assignment operator `<-` in place of this, doing so will store an object called `digits` unnecessarily, wasting your computers resources and cluttering R's working environment.

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`:

```
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.

```
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.

```
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 (Help) Documentation

R includes a vast array of built-in functions, some of which perform highly complex tasks. Consequently, when reading R code, you will often encounter functions whose purpose and usage seem mysterious. To demystify these functions, it is often necessary to consult R's help documentation. Each function in the base version of R comes with corresponding documentation that outlines its purpose, explains its arguments, and provides references. While R's documentation can often be challenging to interpret for novice users, it should always be your first resource when you are unsure about how a function works or what it does. Only after consulting the help documentation should you turn to additional resources, such as internet searches or forums.

To access the documentation for any function in R, simply precede the function name with a question mark and run it in the R console. For example, running `?mean` will bring up the documentation for the arithmetic mean function.

```
?mean
```

If you are using RStudio, the documentation will likely appear in the lower right quadrant of RStudio's display. If you are using the base R environment, you can expect the documentation to appear in your default web browser.

All R documentation follows a consistent structure, designed to provide users with a comprehensive understanding of each function. At the top, you will find the name of the function along with the name of the package it belongs to, enclosed in braces. For example, consulting the documentation for the `mean()` function displays "*mean {base}*" at the top, indicating that this function is part of base R. Similarly, for other common functions like `sd()`, you might see "*{stats}*" listed. The *stats* package, included with R, contains functions for statistical calculations and random number generation, provided by the R Core Team alongside base R functions.

Beneath the function name and package, you will find a brief *Description* section outlining the function's purpose. This is typically followed by a *Usage* section, which includes a code block demonstrating how the function is used and detailing its arguments. For instance, documentation for the `mean()` function includes the following usage:

```
mean(x, ...)

## Default S3 method:
mean(x, trim = 0, na.rm = FALSE, ...)
```

The topmost line of the code block, `mean(x, ...)`, represents the minimal working example for the function. This indicates that, at a minimum, the argument `x` must be provided for the function to work. The *Arguments* section below the code block provides further details about `x`. Specifically, it states that `x` is "*an R object. Currently, there are methods for numeric/logical*

vectors and date, date-time, and time interval objects..." In simpler terms, this is saying that `x` should be a numeric or logical object and not, for instance, a character object. For example:

```
nums <- 0:666
mean(x = nums)
```

```
[1] 333
```

In this case, `nums`, a numeric vector, is the R object provided to the argument `x`.

Beneath the minimal working example in the *Usage* block is a line of code displaying the various additional arguments the function has: `trim` and `na.rm`. These arguments are optional because they come with default values, meaning they do not need to be explicitly set by the user, unlike the required argument `x`.

Further down, the documentation includes a *Value* section, which describes the output of the function based on the arguments provided and the data types used. Finally, the documentation concludes with additional details references, supplemental links, and practical examples to demonstrate the function's usage in real-world scenarios.

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.

```
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:

```
x <- c(710, 633, 786, NA, 642)
mean(x)
```

```
[1] NA
```

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.

```
mean(x, na.rm = TRUE)
```

```
[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:

```
x <- c(710, 633, 786, NA, 642)
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.

```
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.

```
is.na(x)
!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 ...

```
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. 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	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	3.78
8	Exp	-0.51
9	Cont	NA
10	Cont	-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*.

```
df <- data.frame(
  Subject = 1:10,
  Group = c("Exp", "Cont", "Exp", "Cont", "Exp", "Exp",
            "Cont", "Exp", "Cont", "Cont"),
  Value = c(-0.36,  0.28,  1.54,  0.51, -1.28,  1.15,
            -2.22, -0.51,  NA, -1.04)
)
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 individual vectors, you could build your data frame in the following way:

```
df <- data.frame(Subject, Group, Value)
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:

```
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. The argument `FUN` is just short for "function" and is telling `sapply()` what function you want to *apply* to the columns. In this case we are applying the `mode()` function.

Knowing the mode of a column is very important because columns behave like vectors insofar as trying to mix and match different object types within a single column will potentially change that entire column. As an example, if we had coded ...

```
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.

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

```
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.

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

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:

```
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.

```
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.

```
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:

```
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:

```
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.

```
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:

```
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 its classification as a data frame instead of *dropping* it to a vector you can include the argument `drop = FALSE` inside your indexing

brackets. This is useful for situations where you want to preserve the name of the column you have indexed.

```
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.

```
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* (fourth) 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.

```r
df <- df[ , -c(1, 4)]
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 on 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.

```r
new_row <- data.frame(
  Group = "SPAM",
  Value = 999
)

df <- rbind(df, new_row)
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.[18] 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
```

[18] If you, like the *tidyverse* high council, see this as a mild heresy, fear not—the *tibble* (covered in Chapter 3) was made with you in mind.

```
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.

```
rownames(df) <- month.name[1:nrow(df)]
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, R's syntax does not permit you to name them solely with numeric values, nor are you allowed to include spaces or any type of special characters other than an underscore.

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

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

```r
df$1
```

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

```r
df$Col 2
```

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

```r
df$`1`
```

```
[1] "Exp"  "Cont" "Exp"  "Cont" "Exp"  "Exp"  "Cont" "Exp"  "Cont"
```

```r
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, 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*.[19] 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 package name inside the parentheses (quotation marks are not used here).

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

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

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

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

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

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`, `.jpg`, `.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.[20] 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.

[20] 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 younger audiences.

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, thereby causing it not to run. 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 a 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 move, delete, or open 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 perhaps) to believe they are clicking an innocent image of their grandma, when in fact their computer is about to be devoured by grandma.[21]

For both security and everyday use, it is important for users to understand that different types of files exist and that 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*.

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

Figure 1.3: 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).

- **Macintosh:**

  1. In a Finder window on your Mac
  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`

This indicates that the file is located within the `Desktop` directory; which itself is located within the `Your Name` directory, which is located within the `Users` directory; which is located on the hard drive named `C` or `Macintosh HD`.

1.7.1 The Working Directory

Any time R needs to access or create a file, it needs to access 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()`.

```
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 on your computer, 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.[22] 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 ...

```
setwd("C:/Users/Acheron/Desktop")
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.

```
# Create the data frame
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.

[22] When it comes to directory paths, it is not uncommon to also see them written using backslashes (\), particularly on Windows. The reasons for this difference in convention boil down to the development history of various types of software. All you need to know is that R will always use a forward slash /.

```
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. You can have R list the files (and subdirectories) in your working directory by running

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

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

A word of warning: If your working directory contains many files, this command may produce a long list of them, with `letters_1.csv` being just one among many.

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

```
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.

```
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()`.

```
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
```

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

```
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`.

```
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 (`".."`).

```
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.

```
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`.

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

We end up back at the Desktop because ...

- `".."` moves us from `Directory C` up to `Directory B`.
- `"../.."` moves us from `Directory C` up to `Directory A`.
- `"../../.."` moves us from `Directory C` up to the `Desktop`, which is where `Directory A` is located in this case.

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 ...

```
wd <- getwd()
dir_A <- "Directory A"
dir_B <- "Directory B"
dir_C <- "Directory C"
```

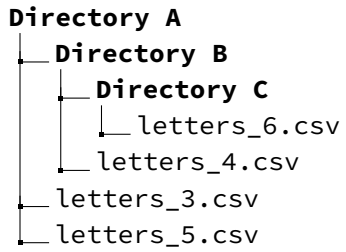
We can then use the `file.path()` function to, for instance, to produce a complete path directly to directory A, B, or C with minimal code that is easier to read.

```
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 ...

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



Chapter 2

Harnessing Sacred Rites of the tidyverse:
The Basics of Plotting Data with R

THE history of R can be split into two epochs. First, there was the age before the tidyverse—a time of primordial chaos, where data analysts toiled in the shadows, their efforts marred by inefficiency and hardship. It was a brutal time, full of necessary violence, that forged the tools for what was to come.

Then came the tidyverse. A revelation. A collection of mystical and cohesive R packages, summoned into the light by Hadley Wickham and his coven of arcane programmers (Wickham et al., 2019). Their sorcery brought order to the chaos, shaping the wild, unruly cosmos of R into something usable—something powerful.

The tidyverse became a gateway for common folk, allowing them to twist, shape, and visualize data with an ease once reserved for only the most privileged elite. Though once viewed as complex and heretical (Muenchen, 2017), the tidyverse has become an essential craft, passed from hand to hand, spreading like whispers in the dark. The art of tidy data flourished, growing stronger through collaboration, trial, and sacrifice.

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).

Make no mistake: sacrifice is inevitable (see Figure 2.1). There will be frustration, moments of despair, and the occasional bout of madness. Yet the tidyverse offers an unparalleled path to power, a means to harness the dark beauty of data in ways that defy the purposeless void.

And so, we begin our journey here—with the basics of data plotting. Like any great practitioner of forbidden arts, you must first master

the grimoires and sigils of power. The tidyverse is no mere toolset; it is a pact, a ritual binding you to its magic. To wield such dark sorcery without understanding is to invite chaos—but with mastery, you will bend the data to your will, carving order from the void itself.

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.

```
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()`.

[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.

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 and experience.

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.

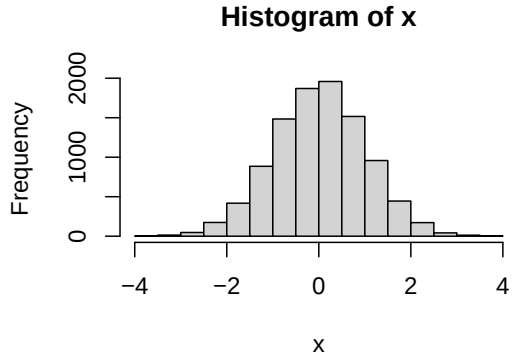


Figure 2.2: An example of base R's plotting functions.

R's base plotting functions provide a convenient way to produce simple, high-quality plots, and they can be quite efficient when working with basic **univariate data** involving only one or two variables. However, modern research often demands the handling of far more complex datasets—ones that may include multiple **response variables** alongside numerous explanatory variables. Each additional variable adds layers of complexity and nuance to your data and, by

[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.

extension, to the plots used to visualize them. While R's built-in plotting tools can accommodate these more elaborate scenarios, doing so often requires a high level of fluency with R's syntax and customization options. For this reason, this book will forgo R's base plotting system in favour of the widely respected *ggplot2* package, a core component of the *tidyverse*. *ggplot2* provides a more consistent and powerful framework for building complex plots, making it the preferred tool for data visualization throughout this text.

The "*gg*" in *ggplot2* stands for "grammar of graphics," a term borrowed from Leland Wilkinson's influential book of the same name (Wilkinson, 2005). Now, the word "grammar" might dredge up long-buried memories of dull English classes—but fear not. In this context, the term simply emphasizes that *ggplot2* is built on a coherent set of rules for assembling visualizations. This structure allows users to build a wide variety of plots in a consistent, modular way that can be easily tailored to their data and needs. This is a major improvement over many traditional plotting systems, which often require you to awkwardly cram your data into rigid, predefined formats—like trying to fit a square peg (your beautifully weird data) into a round hole (the software's narrow expectations).

The easiest way to understand how *ggplot2* works is to simply dive in and use it. Along the way, we will also learn a little bit more about R and data manipulation. However, a disclaimer is perhaps useful here:

This chapter contains a large variety of functions and strategies for plotting data with *ggplot2*. The reader would do well to head the advice provided on page 1.

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.

```
install.packages("ggplot2")
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 data you need only run the code `msleep`, which will

[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")`

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.

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

name	genus	vore	order	conservation	sleep_total	sleep_rem	sleep_cycle	awake	brainwt	bodywt
Cheetah	Acinonyx	carni	Carnivora	lc	12.1	NA	NA	11.9	NA	50.000
Owl monkey	Aotus	omni	Primates	NA	17.0	1.8	NA	7.0	0.016	0.480
Mountain beaver	Aplodontia	herbi	Rodentia	nt	14.4	2.4	NA	9.6	NA	1.350
Greater short-tailed shrew	Blarina	omni	Soricomorpha	lc	14.9	2.3	0.133	9.1	0.000	0.019
Cow	Bos	herbi	Artiodactyla	domesticated	4.0	0.7	0.667	20.0	0.423	600.000
Three-toed sloth	Bradypus	herbi	Pilosa	NA	14.4	2.2	0.767	9.6	NA	3.850
Northern fur seal	Callorhinus	carni	Carnivora	vu	8.7	1.4	0.383	15.3	NA	20.490
Vesper mouse	Calomys	NA	Rodentia	NA	7.0	NA	NA	17.0	NA	0.045
Dog	Canis	carni	Carnivora	domesticated	10.1	2.9	0.333	13.9	0.070	14.000
Roe deer	Capreolus	herbi	Artiodactyla	lc	3.0	NA	NA	21.0	0.098	14.800

Table 2.1: First 10 rows 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.

```
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 are required to include. This can be accessed in the usual way with a `?`

```
?msleep
```

[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.

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.

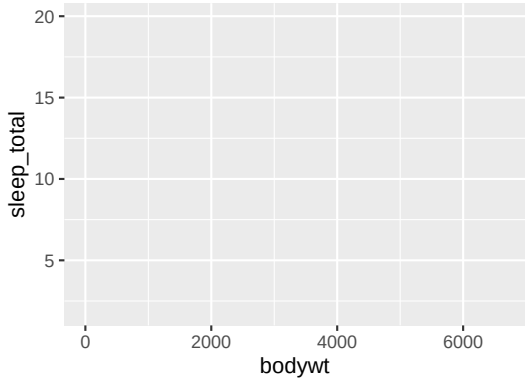
- `lc` = Least Concern
- `nt` = Near Threatened
- `vu` = Vulnerable
- `en` = Endangered
- `cd` = Conservation Dependent

Using a **scatter plot** as a basic starting point, we will graph 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:

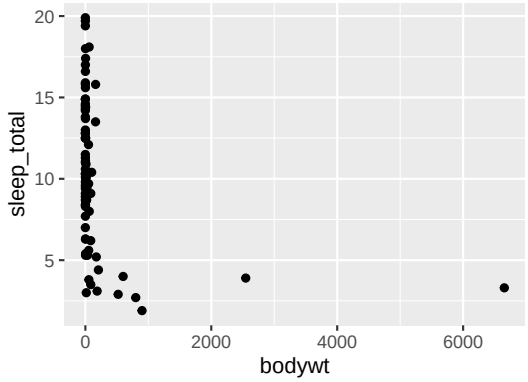
```
1 ggplot(data = msleep, aes(x = bodywt, y = sleep_total))
```



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.

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

```
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

[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.

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.

```
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?

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

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

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

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

[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.

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.

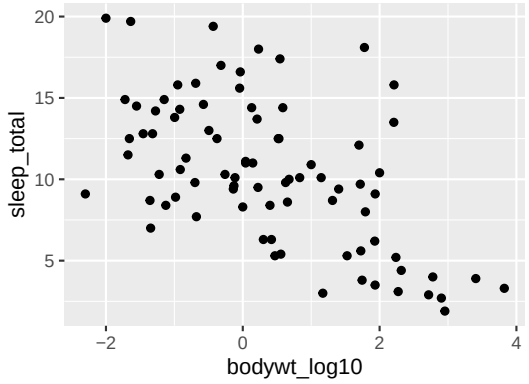
```
log10(1000) # Base-10 function
log2(1000) # Base-2 function
log(1000) # Natural log
log(1000, base = 666) # Pick your own base
```

```
[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 (see Box 2.1 for a alternative method of scaling the axis).

```
# Add new column of log bodywt values.
msleep$bodywt_log10 <- log10(msleep$bodywt)

# Re-plot the data
ggplot(msleep, aes(x = bodywt_log10, y = sleep_total)) +
  geom_point()
```



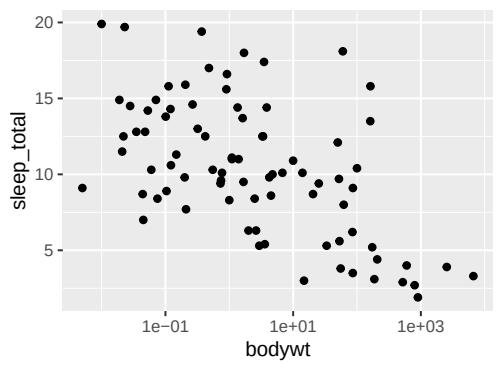
[8] For clarity and consistency the natural logarithm of 1000 has been written $\log_e(1000)$, but it is common practice to identify natural logarithms using "ln". E.g., $\ln(1000) \approx 6.908$.

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.

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

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 ...

```
ggplot(msleep, aes(x = bodywt_log10, y = sleep_total)) +
  geom_point(size = 3,
             shape = 4,
             colour = "blue",
             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 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"
  )
```

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

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 = "&"`.

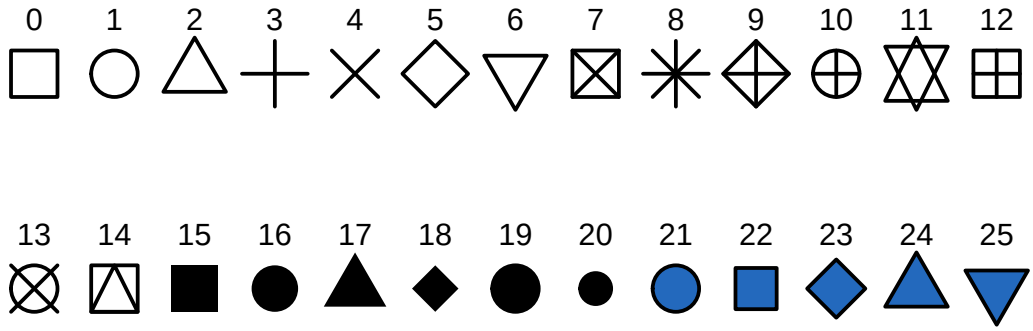


Figure 2.3: R Plotting Characters

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")`.[10]

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

[10]For some reason R spells "colour" incorrectly in these functions. This error has persisted for years, and the developers show no signs of correcting it.

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

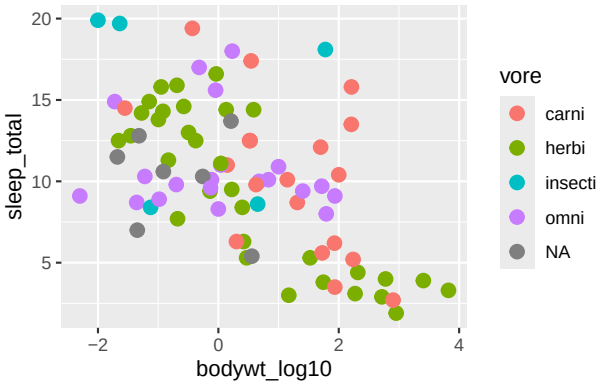
Because of their larger base, a single hexadecimal digit can store more information than a conventional base-10 digit can. For instance, if a computer stores various gradations of the colour red using just two digits, that only allows for 100 (10×10) different reds. Using hexadecimals you can have 256 (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 darker, and larger values are brighter. 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"`.

2.4.1 Aesthetics by variable

In the above examples, the aesthetic changes we made to the plots affected all of the points. In the language of *ggplot2*, we would say that the aesthetics were mapped to all 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.[11] 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] ...
```

[11] 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`.

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

```
ifelse(is.na(msleep$vore), "unknown", msleep$vore)
```

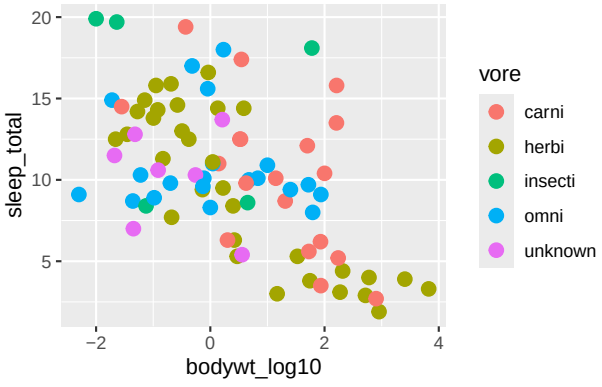
```
 [1] "carni"   "omni"    "herbi"   "omni"    "herbi"   "herbi"   "carni"
 [8] "unknown" "carni"   "herbi"   "herbi"   "herbi"   "omni"    "herbi"
[15] "omni"    "omni"    "omni"    "carni"   "herbi"   "omni"    "herbi"
[22] "insecti" "herbi"   "herbi"   "omni"    "omni"    "herbi"   "carni"
[29] "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))
```



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

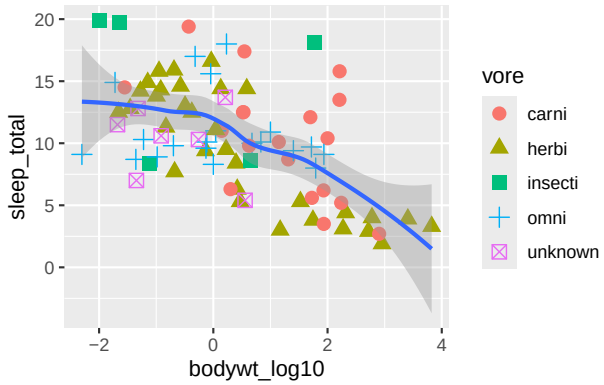
to have each category be represented by a distinct shape.

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

2.5 Displaying trends

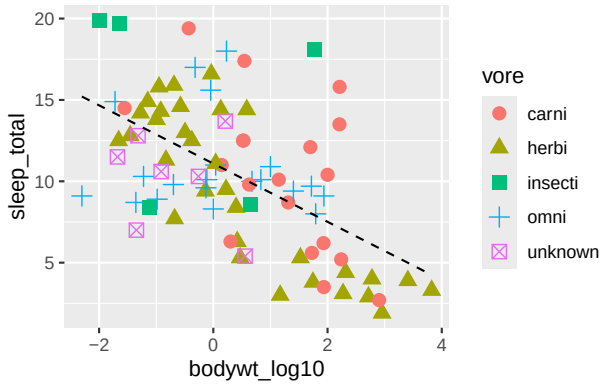
Notice that the data points appear to trend downward as you move from left to right on the x-axis. In other words, as body weight increases, you tend to see decreases in sleep total. By simply adding a second geom, called `geom_smooth()`, we can use a line of best fit to represent (i.e., model) this 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 a statistic called 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()`.

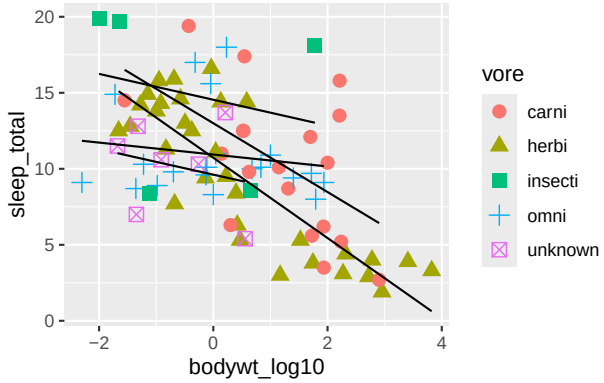
```
1 ggplot(msleep, aes(x = bodywt_log10, y = sleep_total)) +
2   geom_point(size = 3, aes(colour = vore, shape = vore)) +
3   geom_smooth(
4     method = "lm", se = FALSE,
5     linetype = 2,
6     linewidth = 0.5,
7     colour = "black"
8   )
```



By setting `method = "lm"` on line 4, we are instructing *ggplot2* to draw a linear model. While the concepts of standard error, polynomial regression, and linear models are more advanced topics, their value in displaying trends should be clear enough, even if the underlying mathematics is not yet fully understood.

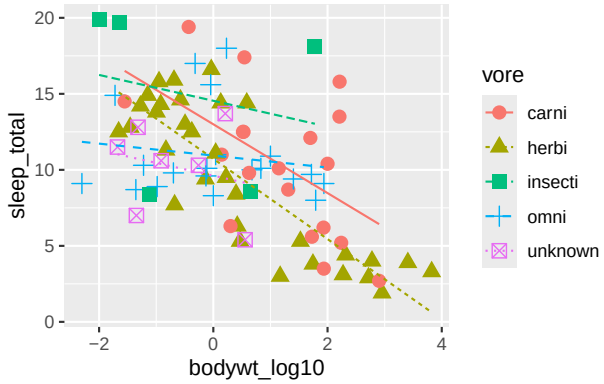
It is at this point where the versatility of the *ggplot2* really begins to shine. For instance, if we wanted to create a separate regression line for each category of `$vore` we can accomplish that by once again making use of the `aes()` function and "grouping" by `$vore`.

```
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     colour = "black",
7     linewidth = 0.5,
8     aes(group = vore)
9   )
```



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`.

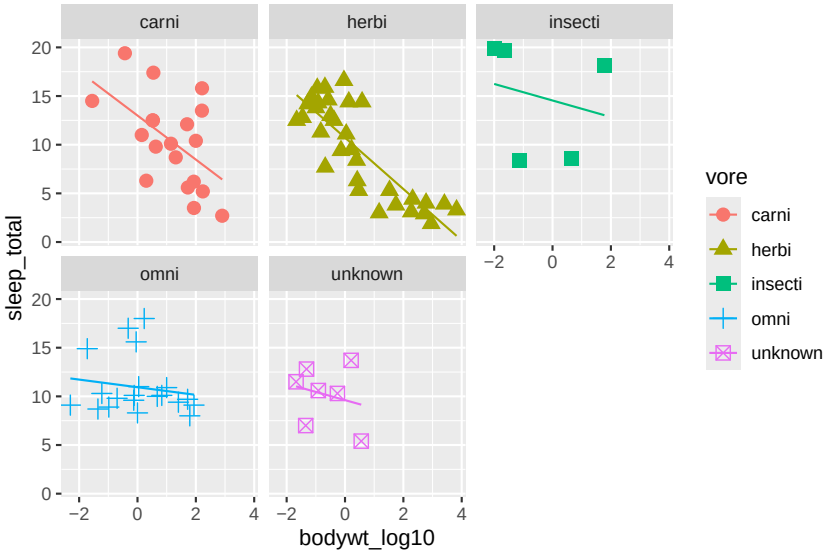
```
ggplot(msleep, aes(x = bodywt_log10, y = sleep_total)) +
  geom_point(size = 3, aes(colour = vore, shape = vore)) +
  geom_smooth(
    method = "lm",
    se = FALSE,
    linewidth = 0.5,
    aes(colour = vore, linetype = vore)
    )
```



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.

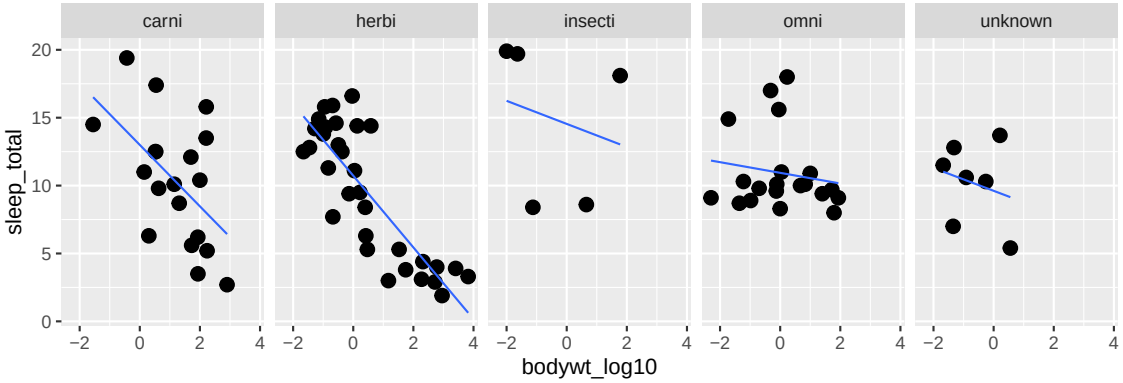
```
ggplot(msleep, aes(x = bodywt_log10, y = sleep_total)) +
  geom_point(size = 3, aes(colour = vore, shape = vore)) +
  geom_smooth(
    method = "lm",
    se = FALSE,
    linewidth = 0.5,
    aes(colour = vore)
  ) +
    facet_wrap(~ vore)
```



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.

```
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)
```



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`).

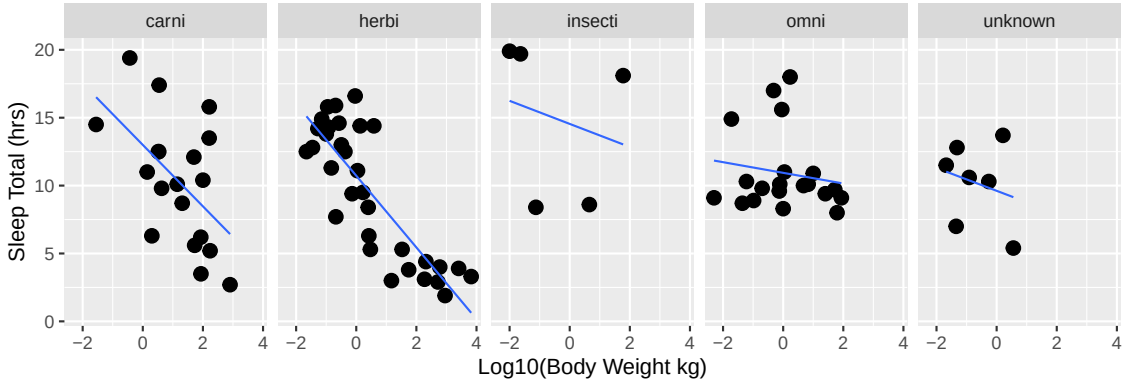
Particularly for beginners with R, it is difficult to impress how useful *ggplot2* is here. Using base R plotting functions to produce a comparable graph would be a considerably more complex process and require a heftier amount of code to be written, whereas *ggplot2* does it all for us in four short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

```
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) +
  labs(
    x = "Log10(Body Weight kg)",
    y = "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.

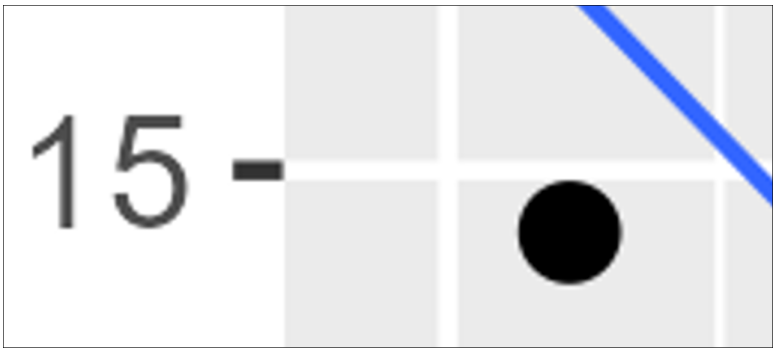


Figure 2.4: Artificating present on our 300 dpi raster image when magnified.

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.

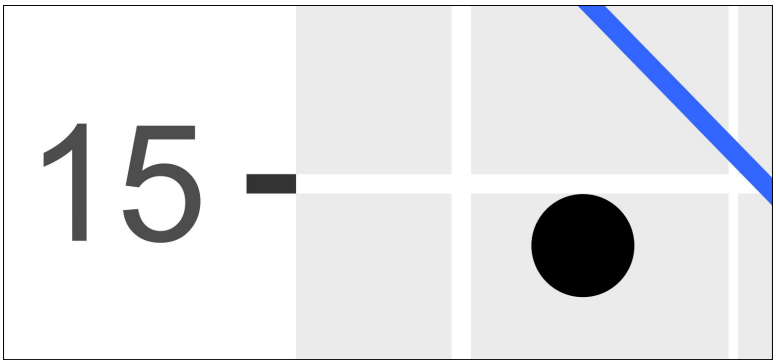
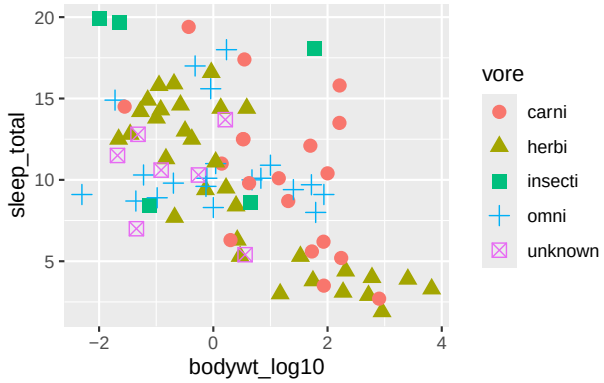


Figure 2.5: Magnification of a vector graphic.

2.9 Scales

A core 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 we mapped an aesthetic to a variable—like we did above where we had mapped both colour and shape to the `$vore` column:

```
...
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 want to modify. For instance, to adjust the colour (or edge colour) of a point you could use the `scale_colour_discrete()` function or the `scale_colour_continuous()` function. If you wanted to adjust the shapes of the points, you could use `scale_shape_discrete()` function. 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_discrete()` function or the `scale_fill_continuous()` function.

There are a large amount of functions like these and, at this point, you do not need to concern yourself with all their varieties and how they work. What is important to recognize here is that each scale function specifies, inside its name, what aesthetic (e.g., colour, shape, fill, etc.) it is modifying:

`scale_<aesthetic name>_<transformation>()`

The "transformation" part of the function's name is intended to describe how the function modifies the aesthetic which will hopefully become more apparent as we move through some examples.

2.9.1 Position Scales: Modifying the Axis Breaks

When we first created the grid on to which we drew our points, we had 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:

```
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 (i.e., x and y) mappings of a plot's 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 axes represent *numeric* objects as opposed to *character* objects.

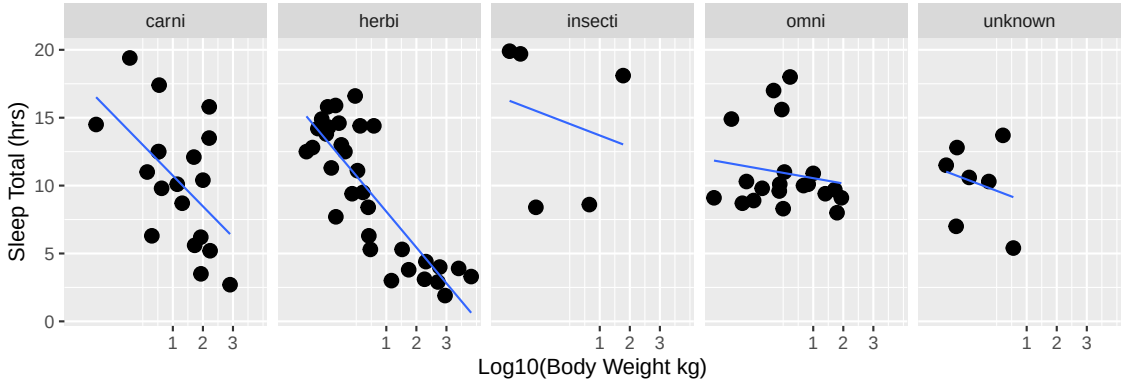
```
mode(msleep$bodywt_log10) # x-axis
mode(msleep$sleep_total) # y-axis
```

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

Thus, for the purpose of plotting, they are treated as a continuous scale and you would need to use `scale_x_continuous()` and `scale_y_continuous()` respectively. 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 in Chapter 3 when we build a bar graph of skulls 🤘).

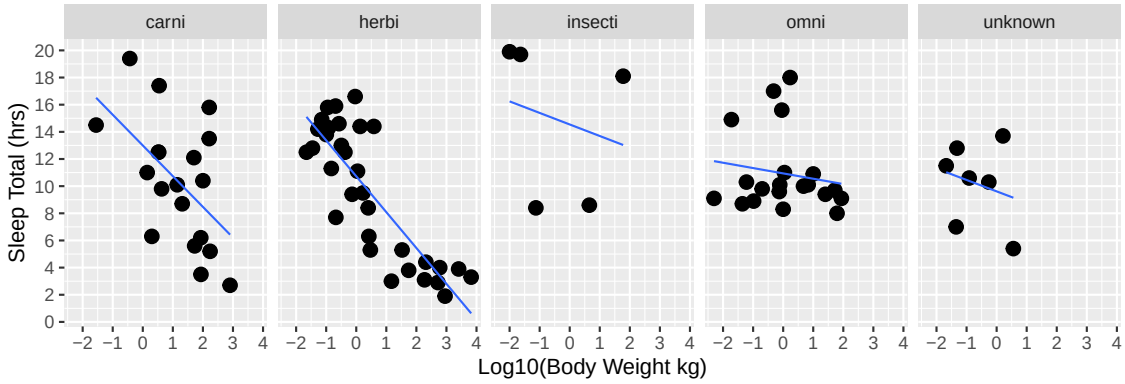
To customize the breaks on our axis, we simply need to add one of the aforementioned functions to our plot's code and provide a vector of values we want to see displayed using the argument `breaks`. For instance, if we want the x-axis to only display the numbers 1, 2, and 3, we would add
`scale_x_continuous(breaks = c(1,2,3))` to our code (see line 13).

```
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) +
  labs(
    x = "Log10(Body Weight kg)",
    y = "Sleep Total (hrs)"
  ) +
  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 (see lines 13 and 14).

```
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    labs(
10     x = "Log10(Body Weight kg)",
11     y = "Sleep Total (hrs)"
12   ) +
13   scale_x_continuous(breaks = seq(-2, 4, 1)) +
14   scale_y_continuous(breaks = seq(0, 20, 2))
```

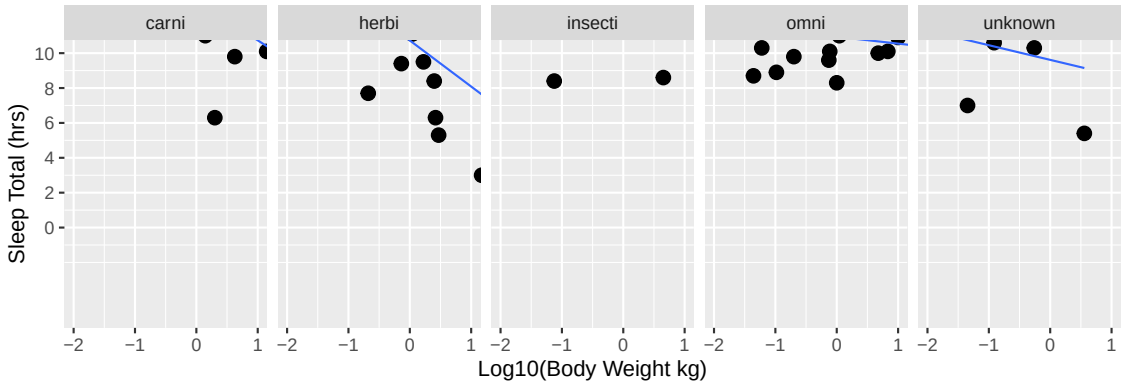


The four scale functions above can achieve a lot more than what is being shown here, but for most uses, this basic adjustment of the axis breaks will be their primary purpose.

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.

```
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) +
  labs(
    x = "Log10(Body Weight kg)",
    y = "Sleep Total (hrs)"
  ) +
  scale_x_continuous(breaks = seq(-2, 4, 1)) +
  scale_y_continuous(breaks = seq(0, 20, 2)) +
  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 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

that is dependent on those values. For instance, if you replace lines 13-15 in the above script with `xlim(-2, 1)` you will be confronted with a very nasty error message, telling you (among other things) that ...

```
Warning messages:
1: Removed 27 rows containing non-finite outside the scale range
(`stat_smooth()`).
2: Removed 27 rows containing missing values or values outside the scale
range (`geom_point()`).
```

This occurs because values in our data falling outside of -2 and $+1$ are not recognized any more, which, by extension means they are not being used by `geom_smooth()` and `geom_point()`. Particularly in the case of `geom_smooth()`, that will impact how the trend line is calculated. 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.[12]

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 `$vore` column. Moreover, the distinction between *continuous* and *discrete* scaling applies just as much to colour as it does position. As illustrated in Figure 2.6 and 2.7, continuous colour scales are usually represented with a colour gradient and discrete scales are represented by distinct colours (like in a box of crayons). While it is possible to do this, you usually do not want to use a gradient to represent distinct categories because it makes the categories difficult to discriminate visually. For instance, the `$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.

[12] 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; however, *ggplot2* has its reasons. Functions like `xlim()`, `ylim()`, and arguments like `limits =` can be useful when you want to restrict the data used in calculations, not just what's shown. This can help speed things up or focus analyses on a particular range—especially when letting *ggplot2* handle things like smoothing or density estimation. Just be warned: this is a power best wielded carefully.

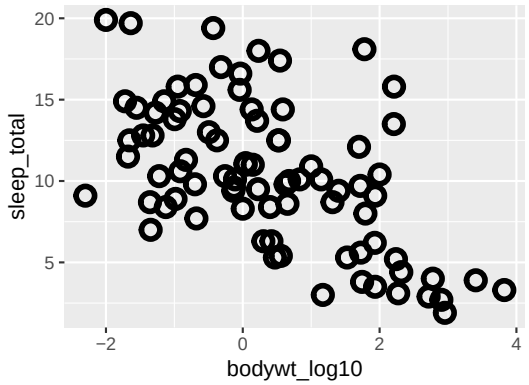
Figure 2.6: Example of a continuous colour scale (i.e., a colour gradient)

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

2.9.4 Discrete Colour Scales

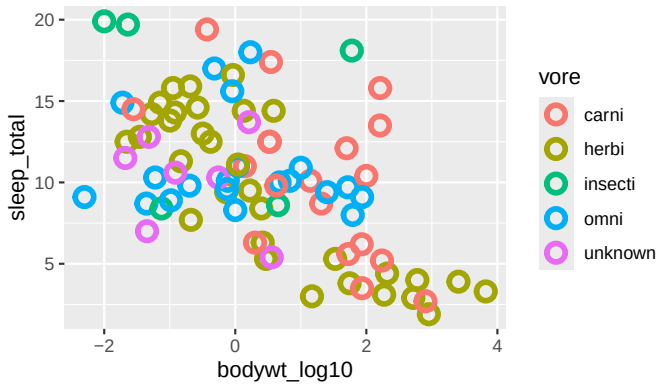
To illustrate the use of discrete colour scales lets create a simple plot we can experiment with.

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



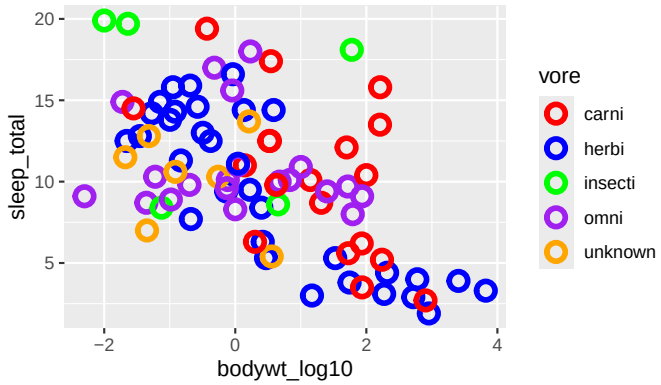
First we will map the edge colour to the column `$vore`.

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



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

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



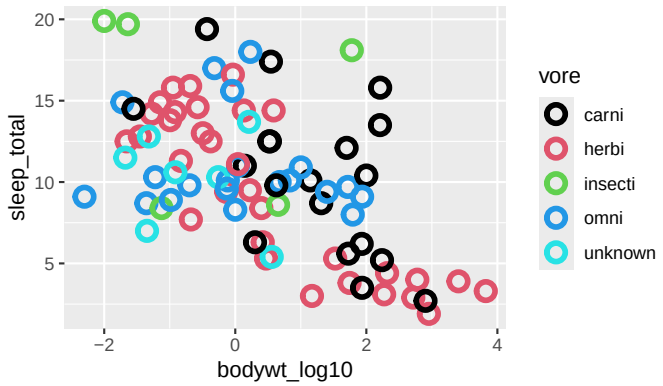
The same effect can be achieved by using `scale_colour_manual()` instead.

```
...
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.

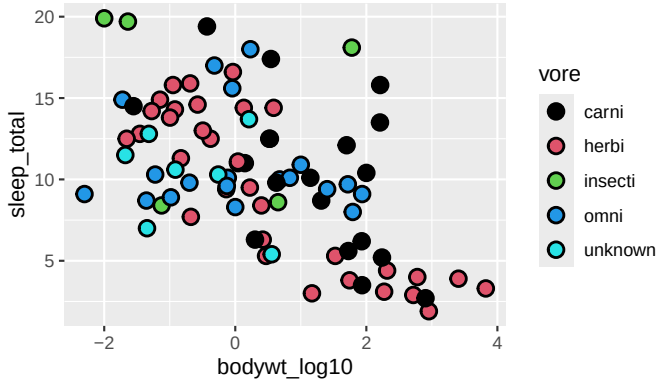
```
# Create a colour palette
palette <- c(
  "#000000", "#DF536B", "#61D04F", "#2297E6", "#28E2E5", "#CD0BBC", "#F5C710",
  "#9E9E9E"
)

# Use that palette in your plot
ggplot(msleep, aes(x = bodywt_log10, y = sleep_total)) +
  geom_point(
    size = 3, shape = 21, stroke = 2,
    aes(colour = vore)
  ) +
  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.

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

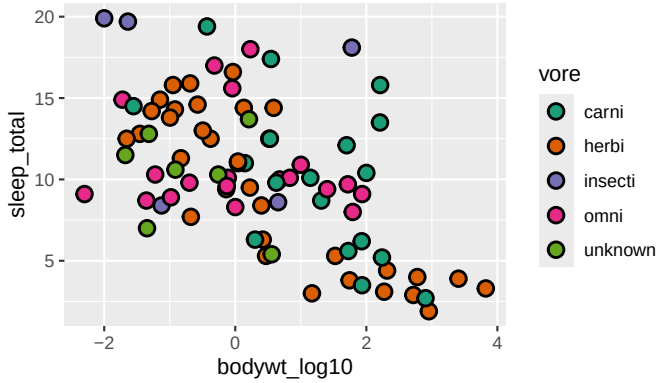
Until now, we have been specifying our own custom colour palettes; however, base R contains a variety of pre-existing palettes we can make use of. To obtain the list you can simply run the following:

```
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.

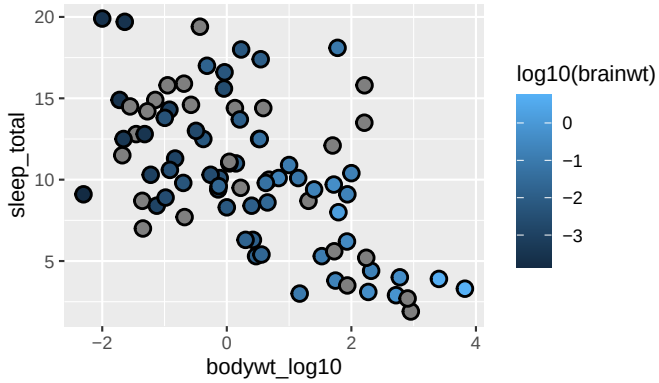
```
ggplot(msleep, aes(x = bodywt_log10, y = sleep_total)) +
  geom_point(
    size = 3, shape = 21, stroke = 1, colour = "black",
    aes(fill = vore)
  ) +
  scale_fill_discrete(type = palette.colors(n = NULL, "Dark2"))
```



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. 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:

```
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 that it is continuous. Thus, 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

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

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.

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

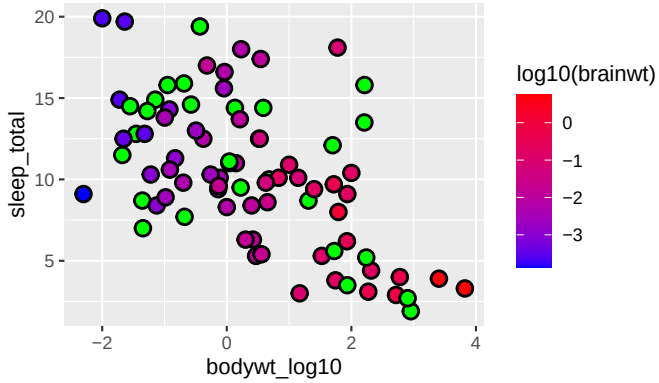
In case it is not obvious, this code works by using the `is.na()` function to check whether each row in the `msleep` data frame's `$brainwt` column contains an `NA` value. Rows which result as `TRUE` are displayed and everything else is ignored. This leaves us with a data frame of 27 different animals, all of which have a `NA` value in the `$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.[13] 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 `na.value` and `breaks` respectively.

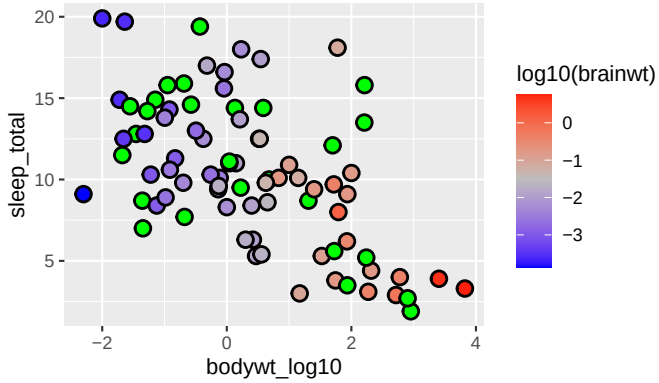
```
# scale_colour_gradient example
ggplot(msleep, aes(x = bodywt_log10, y = sleep_total)) +
  geom_point(
    size = 3, shape = 21, stroke = 1, colour = "black",
    aes(fill = log10(brainwt))
  ) +
  scale_fill_gradient(
    low = "blue",
    high = "red",
    na.value = "green",
    breaks = seq(-4,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

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



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 $\log_{10}(\text{brain weight}) = -1.5$.

```
# scale_colour_gradient2 example
ggplot(msleep, aes(x = bodywt_log10, y = sleep_total)) +
  geom_point(
    size = 3, shape = 21, stroke = 1, colour = "black",
    aes(fill = log10(brainwt))
  ) +
  scale_fill_gradient2(
    low = "blue",
    mid = "grey",
    high = "red",
    midpoint = -1.5,
    na.value = "green",
    breaks = seq(-4, 1, 1)
  )
```



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.

```
hcl.pals(type = "sequential")
hcl.pals(type = "diverging")
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.

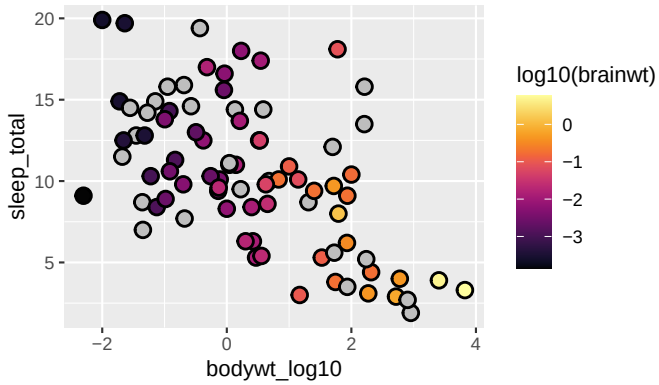
To retrieve the hex codes for a given palette (e.g., `"Inferno"`), you will need to specify not only the palette name but also the number of colours you want using the `n` argument. This determines the granularity of the palette—higher values yield a more finely graded colour spectrum. That said, for most practical purposes, distinctions beyond 4 or 5 colours tend to become visually negligible with the sequential and diverging palettes. Visual examples of all three HCL palette types can be found in Appendix B.

```
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.

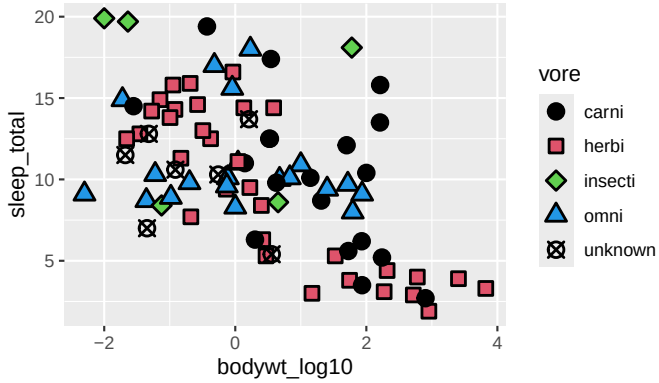
```
ggplot(msleep, aes(x = bodywt_log10, y = sleep_total)) +
  geom_point(
    size = 3, shape = 21, stroke = 1, colour = "black",
    aes(fill = log10(brainwt))
  ) +
  scale_fill_gradientn(
    colours = hcl.colors(n = 50, palette = "Inferno"),
    na.value = "grey",
    breaks = seq(-4, 1, 1)
  )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



2.9.6 Shape Scales

We know that relying solely 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).

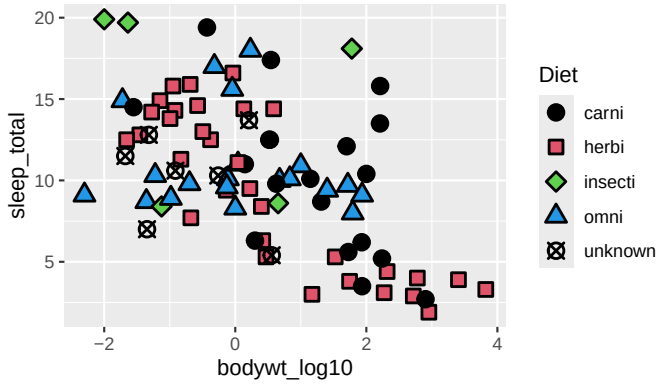
```
ggplot(msleep, aes(x = bodywt_log10, y = sleep_total)) +
  geom_point(
    size = 3, stroke = 1, colour = "black",
    aes(fill = vore, shape = vore)
  ) +
  scale_shape_manual(values = c(21:24), na.value = 13) +
  scale_fill_discrete(type = palette.colors(n = NULL, "R4"))
```



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".

```
ggplot(msleep, aes(x = bodywt_log10, y = sleep_total)) +
  geom_point(
    size = 3, stroke = 1, colour = "black",
    aes(fill = vore, shape = vore)
  ) +
  scale_shape_manual(values = c(21:24), na.value = 13, name = "Diet") +
  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.

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

2.9.8 Other Scales

In the sections above, we have only considered the position, colour, fill, and shape scales, which are among the features 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 to. Meaning that, in the same way a number in R is an object that you can name and add things to - e.g.,

```
x <- 1
x + 2
```

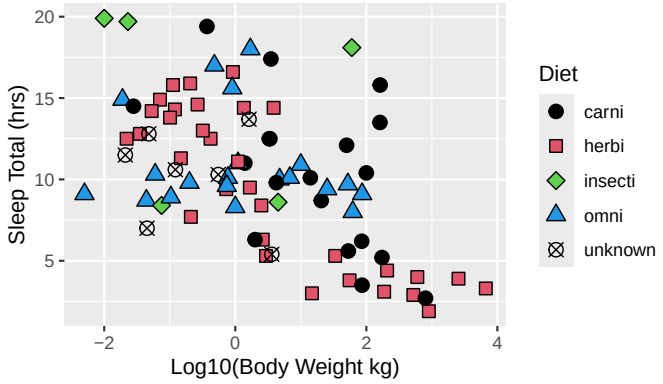
```
[1] 3
```

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`.

```
my_plot <- ggplot(msleep, aes(x = bodywt_log10, y = sleep_total)) +
  geom_point(
    size = 3, colour = "black",
    aes(fill = vore, shape = vore)
  ) +
  scale_shape_manual(values = c(21:24, 13)) +
  scale_fill_discrete(type = palette.colors(n = NULL, "R4")) +
  labs(
    x = "Log10(Body Weight kg)",
    y = "Sleep Total (hrs)",
    shape = "Diet",
    fill = "Diet"
  )
```

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

```
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*.

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()`

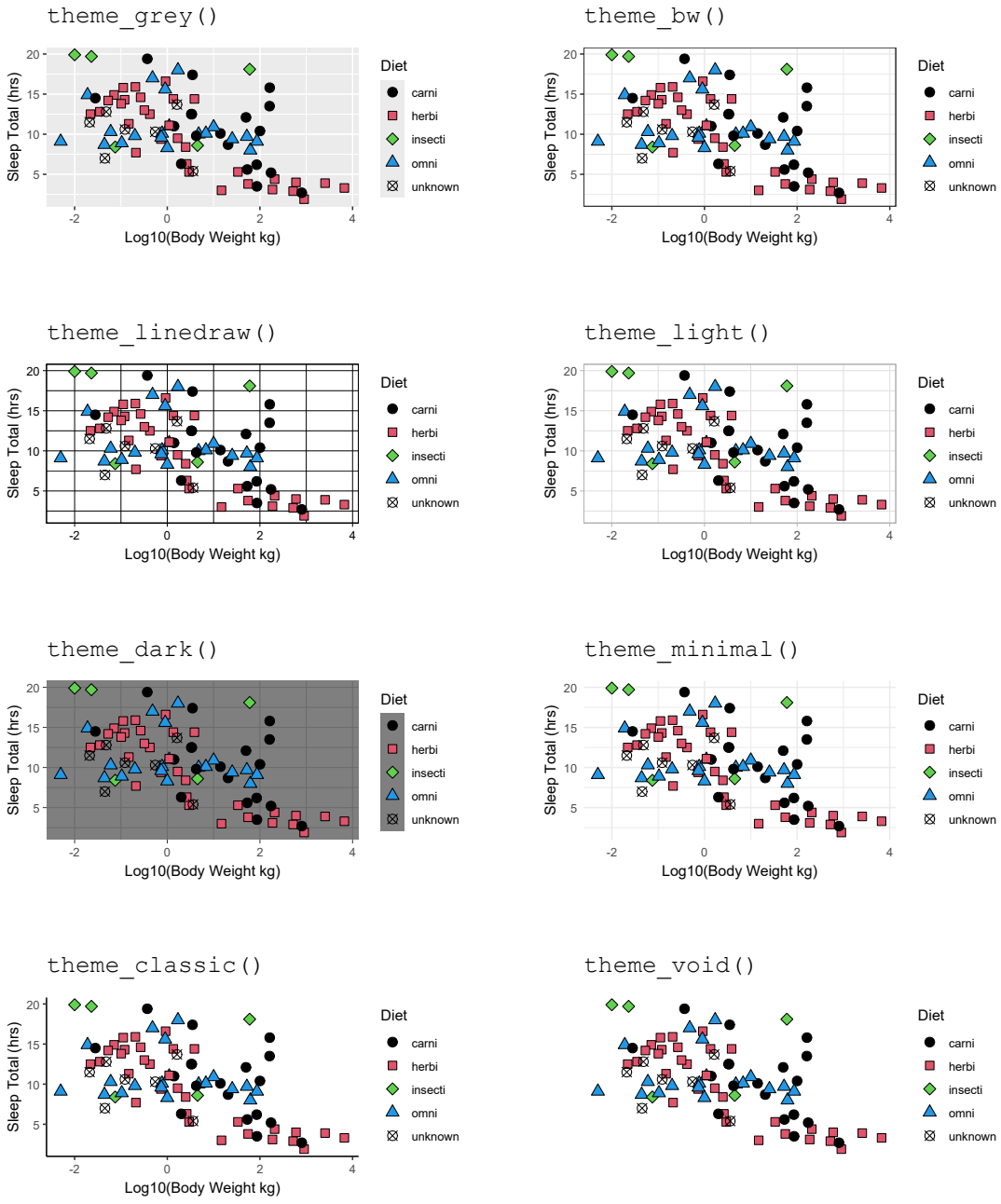
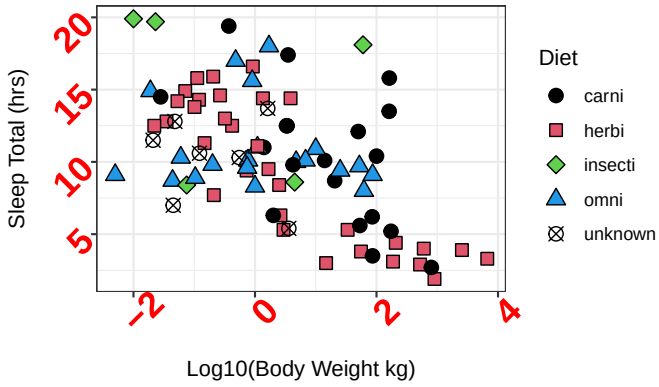


Figure 2.9: Visual examples of the eight built-in themes *ggplot2* provides.

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



Notice that the code affected both axes; 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.

```
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")
    )
  )
```

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`.

```
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 was labelled using the shortened version contained in the data set, 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 in the next chapter.

Chapter 3

The Invocation and Metamorphosis of Data

KNOWLEDGE is power as they say, but data—data is something else entirely. It is the ghost in the machine, the thing lurking beneath the surface, waiting for you to look too close. Heed this warning: The data frame, and its accursed successor the tibble, are your most loyal servants ... and your most treacherous foes. Treat them with reverence, for a single misstep may awaken errors best left entombed.

Chapter 1 had 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 information in 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, for all but the smallest of data sets, this method, while simple, is both 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 and what needs to be done with it, 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 manner 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 be working 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, there are sets of rows and columns, with each cell containing just a single value.

3.1 Spreadsheet Software

Given the ubiquity of spreadsheet software, it is important to discuss its use and why R offers a preferable alternative for data analysis. Most spreadsheet applications have their 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 awful stock spreadsheet application on Macintosh computers, called *Numbers*, uses 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. They allow you to do things like perform calculations, adjust cell colours, add images, insert comments, etc. And all of this is saved, in one form or another, as information 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. Incorporating all those layers of additional functionality is going to boost file sizes, inflate load times, limit the amount of information the spreadsheet can hold, and 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 damage your original data set, often without even realizing it. The fact is, we should care about analysing 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 to be analysed 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 as humans. 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`, `.wks`, 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 `skull_cap_partial_wide.csv`. It is located in the `./data` directory at the following URL:

https://github.com/statistical-grimoire/thomson-randallmaciver-1905

This data represents a subset of a much larger dataset,[2] containing estimated cranial capacities in cubic centimetres for 1,449 ancient Egyptian skulls.[3] These skulls span numerous historical periods, ranging from Egypt's early predynastic era to its Roman occupation.

Upon opening the file on GitHub, the contents appear in a standard tabular format, resembling a typical spreadsheet (see Table 3.1 for an example displaying the first ten rows). With the exception of the columns labelled `sex` and `predynastic`, each column header corresponds to the starting year of an approximate date range, as reported by the original authors. The prefix *c* denotes *circa* (meaning "approximately"), followed by a year and the abbreviation *BC* ("Before Christ"), reflecting the historical dating conventions employed by the original authors. A more contemporary and inclusive alternative would be *BCE* ("Before Common Era"). The term *predynastic* refers to periods preceding the earliest recorded Egyptian dynasties, which, at the time of Thomson and Randall-MacIver's (1905) research, were not yet clearly established or reliably dated.

While GitHub nicely formats the file as a spreadsheet for viewing, the actual raw data consists of nothing more than a basic text document that separates individual values with a comma. We can see this more clearly if we click the button on GitHub labelled "Raw" which will present the file in its unaltered (i.e., raw) text format. The first 10 rows can be seen below:

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

[2] `Thomson_Randall-MacIver_1905.csv`

[3] This is not necessarily a statistic anyone should care about, but Ancient Egypt is really cool and skulls are metal AF. Also, for any Americans reading this, 1 centimetre is equal to 1.181 barleycorns.

sex	predynastic	c4800BC	c4200BC	c4000BC	c3700BC	c3500BC	c2780BC	c1590BC	c378BC	c331BC
Male	1370	1410	1320	1445	NA	1395	1425	1440	1310	1450
Male	1250	1445	1565	1540	NA	1420	1505	1355	1395	1460
Male	1430	1440	1600	1565	NA	1380	1360	1490	1360	1360
Male	1350	1340	1460	1710	NA	1260	1385	1425	1485	1410
Male	1130	1460	1520	1690	NA	1285	1350	1380	1365	1215
Male	1670	1290	1440	1775	NA	1505	1440	1490	1220	1320
Male	1195	1290	1740	1390	NA	1230	1400	1385	1195	1550
Male	1500	1385	1410	1620	NA	1250	1255	1270	1410	1320
Male	1325	1290	1510	1500	NA	1315	1450	1585	1370	1460
Male	1480	1565	1550	1255	NA	1360	1310	1330	1365	1560

Table 3.1: First ten rows of `skull_cap_partial_wide.csv`, displaying estimated cranial capacity (cm^3).

```
sex,predynastic,c4800BC,c4200BC,c4000BC,c3700BC,c3500BC,c2780BC,c1590BC,c378BC,c331BC
Male,1370,1410,1320,1445,,1395,1425,1440,1310,1450
Male,1250,1445,1565,1540,,1420,1505,1355,1395,1460
Male,1430,1440,1600,1565,,1380,1360,1490,1360,1360
Male,1350,1340,1460,1710,,1260,1385,1425,1485,1410
Male,1130,1460,1520,1690,,1285,1350,1380,1365,1215
Male,1670,1290,1440,1775,,1505,1440,1490,1220,1320
Male,1195,1290,1740,1390,,1230,1400,1385,1195,1550
Male,1500,1385,1410,1620,,1250,1255,1270,1410,1320
Male,1325,1290,1510,1500,,1315,1450,1585,1370,1460
Male,1480,1565,1550,1255,,1360,1310,1330,1365,1560
```

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

Comparing the two versions it can readily be seen how the commas are functioning. They separate individual 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, it will 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 `skull_cap_partial_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., `read_csv()` would use a comma to delimit whereas `read_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 `skull_cap_partial_wide.csv` data set as a `.tsv` file and open it within a generic text editor, you would see something very similar to the following ...

sex	predynastic	c4800BC		c4200BC	c4000BC		c3700BC	c3500BC	c2780BC	
Male	1370	1410	1320	1445	**NA**	1395	1425	1440	1310	1450
Male	1250	1445	1565	1540	**NA**	1420	1505	1355	1395	1460
Male	1430	1440	1600	1565	**NA**	1380	1360	1490	1360	1360
Male	1350	1340	1460	1710	**NA**	1260	1385	1425	1485	1410
Male	1130	1460	1520	1690	**NA**	1285	1350	1380	1365	1215
Male	1670	1290	1440	1775	**NA**	1505	1440	1490	1220	1320
Male	1195	1290	1740	1390	**NA**	1230	1400	1385	1195	1550
Male	1500	1385	1410	1620	**NA**	1250	1255	1270	1410	1320
Male	1325	1290	1510	1500	**NA**	1315	1450	1585	1370	1460
Male	1480	1565	1550	1255	**NA**	1360	1310	1330	1365	1560

Excerpt of the `skull_cap_partial_wide.csv` file displayed in raw text format as if it were a `.tsv`. Only the first 10 rows are shown; the last three column headers (`c1590BC`, `c378BC`, and `c331BC`) are omitted for space.

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 `skull_cap_partial_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] Depending on the browser you are using you may have to hunt around for the download option. For instance, if you are using Safari, you may have to select "more file actions."

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 as a character string. 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 `skulls`.

```
1 library(tidyverse)
2 skulls <- read_csv("skull_cap_partial_wide.csv")
```

```
Rows: 343 Columns: 11
── Column specification ────────────────────────────────────────
Delimiter: ","
chr  (1): sex
dbl (10): predynastic, c4800BC, c4200BC, c4000BC, c3700BC, c35...

i Use `spec()` to retrieve the full column specification for this data.
i 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 343 rows and 11 columns, uses a `,` as a delimiter. One column, `sex`, consists of character (`chr`) values and the remaining 10 columns 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 referred to, is one such representation. If that is confusing, don't worry, what is important to take away from the output is that `dbl` means the column contains numeric values (i.e., we can use them to do mathematics).

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

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

```
skulls
```

```
# A tibble: 343 × 11
   sex   predynastic c4800BC c4200BC c4000BC c3700BC c3500BC c2780BC
   <chr>       <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
 1 Male         1370    1410    1320    1445      NA    1395    1425
 2 Male         1250    1445    1565    1540      NA    1420    1505
 3 Male         1430    1440    1600    1565      NA    1380    1360
 4 Male         1350    1340    1460    1710      NA    1260    1385
 5 Male         1130    1460    1520    1690      NA    1285    1350
 6 Male         1670    1290    1440    1775      NA    1505    1440
 7 Male         1195    1290    1740    1390      NA    1230    1400
 8 Male         1500    1385    1410    1620      NA    1250    1255
 9 Male         1325    1290    1510    1500      NA    1315    1450
10 Male         1480    1565    1550    1255      NA    1360    1310
# i 333 more rows
# i 3 more variables: c1590BC <dbl>, c378BC <dbl>, c331BC <dbl>
# i Use `print(n = ...)` to see more rows
```

You can now subset and manipulate the data frame `skulls` like was done in Chapter 1 when we first discussing data frames (see section 1.4.10). For instance, if we wanted to look at the mean breadth of skulls from the predynastic period, we could simply run:

```
mean(skulls$predynastic, na.rm = TRUE)
```

```
[1] 1320.142
```

One thing that is not apparent from the ten row output is that *all* of the numeric columns contain some missing, `NA`, values—hence the need to specify `na.rm = TRUE`.

`read.csv()` vs. `read_csv()`

To load the `skulls` data frame above, we used the function `read_csv()`, which is part of the *tidyverse*. However, base R has a similar function, `read.csv()`, that will do essentially the same thing—it will read a `.csv` file into 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 advantages over its predecessor. 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, instead of a data frame, 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 `skulls` data in the following way:

```
skulls <- read_delim("Max_Breadth_TRM_1905.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. Nine times out of ten, 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 `skulls` (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**. 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, with the *tidyverse* loaded, we can re-create chapter 1's data frame as a tibble using an identical syntax.

```
df <- tibble(
  Subject = 1:10,
  Group = c("Exp", "Cont", "Exp", "Cont", "Exp", "Exp",
            "Cont", "Exp", "Cont", "Cont"),
  Value = c(-0.36,  0.28,  1.54,  0.51, -1.28,  1.15,
            -2.22, -0.51,  NA, -1.04)
)

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. For the most part they behave identically. However, there is one difference worth mentioning: for tibbles, indexing a single column by specifying row and column values outputs a tibble. For example, suppose we use our index brackets, `[ ]`, to isolate the first 5 rows of column 3 in the `skulls` data.

```
skulls[1:5, 3]
```

```
# A tibble: 5 × 1
  c4800BC
    <dbl>
1    1410
2    1445
3    1440
4    1340
5    1460
```

This seems sensible enough behaviour, but is in contrast to the traditional behaviour of R's data frame which will output a vector unless more than one column is selected.

```
# Using read.csv to load the data as a data frame
skulls_df <- read.csv("skull_cap_partial_wide.csv")

skulls_df[1:5, 3]
```

```
[1] 1410 1445 1440 1340 1460
```

This may seem to be a trivial distinction; however, operations such as computing the mean of a column are quite common and often require inserting a numeric vector. Consequently, when the output is a tibble rather than a numeric or logical vector, attempting such operations results in an error.

```
mean(skulls[1:5, 3])
```

```
[1] NA
Warning message:
In mean.default(skulls[1:5, 3]) :
  argument is not numeric or logical: returning NA
```

However, you can set the argument `drop = TRUE` inside the indexing brackets to coerce the output into a vector.

```
skulls[1:5, 3, drop = TRUE]
mean(skulls[1:5, 3, drop = TRUE])
```

```
[1] 1410 1445 1440 1340 1460
[1] 1419
```

Should the need arise, switching between tibbles and data frames is a simple matter. For example, to convert our tibble `skulls` to a data frame, we can simply use the `as.data.frame()` function in R.

```
# tibble to data frame
skulls <- as.data.frame(skulls)
skulls
```

```
   sex  predynastic c4800BC c4200BC c4000BC c3700BC c3500BC c2780BC
1  Male        1370    1410    1320    1445      NA    1395    1425
2  Male        1250    1445    1565    1540      NA    1420    1505
3  Male        1430    1440    1600    1565      NA    1380    1360
4  Male        1350    1340    1460    1710      NA    1260    1385
5  Male        1130    1460    1520    1690      NA    1285    1350
6  Male        1670    1290    1440    1775      NA    1505    1440
7  Male        1195    1290    1740    1390      NA    1230    1400
8  Male        1500    1385    1410    1620      NA    1250    1255
9  Male        1325    1290    1510    1500      NA    1315    1450
10 Male        1480    1565    1550    1255      NA    1360    1310
...
```

To convert it back to a tibble:

```
# data frame to tibble
skulls <- as_tibble(skulls)
skulls
```

```
# A tibble: 343 × 11
   sex   predynastic c4800BC c4200BC c4000BC c3700BC c3500BC c2780BC
   <chr>       <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
 1 Male         1370    1410    1320    1445      NA    1395    1425
 2 Male         1250    1445    1565    1540      NA    1420    1505
 3 Male         1430    1440    1600    1565      NA    1380    1360
 4 Male         1350    1340    1460    1710      NA    1260    1385
 5 Male         1130    1460    1520    1690      NA    1285    1350
 6 Male         1670    1290    1440    1775      NA    1505    1440
 7 Male         1195    1290    1740    1390      NA    1230    1400
 8 Male         1500    1385    1410    1620      NA    1250    1255
 9 Male         1325    1290    1510    1500      NA    1315    1450
10 Male         1480    1565    1550    1255      NA    1360    1310
# i 333 more rows
# i 3 more variables: c1590BC <dbl>, c378BC <dbl>, c331BC <dbl>
# i Use `print(n = ...)` to see more rows
```

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 window. That being said, many people will still find the number of rows displayed 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 ...

```
print(skulls, n = 20)
```

An alternative method is to change R's default display behaviour by setting the minimum number of rows to output using the `options()` function.

```
options(pillar.print_min = 20)
skulls
```

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.[5]

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:

```
options(pillar.print_min = Inf)
skulls
```

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, seemingly 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

[5] If you are wondering why we specify `pillar...` to set rows, it's because *pillar* is a package in the *tidyverse*.

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:

```
options(
  pillar.print_min = 20,
  pillar.width = Inf
)

skulls
```

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

```
print(skulls, n = 20, width = Inf)
```

The Precision and Display of Decimals in Tibbles

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
```

Using that, we will create a simple tibble 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.

```
print(pi_df$pie[1], digits = 16)
```

```
[1] 3.141592653589793
```

It is important to understand that tibbles do not limit the actual numeric precision of your data. Rather, they format numbers using **significant figures** (also known as *significant digits*) when printing to the console. This helps maintain the clean, rectangular structure of the tibble output, making columns easier to scan.

The way tibbles work with significant figures is slightly different than you may have learned in primary school math class. Everything in front of the decimal point is always displayed, but each number in front of that decimal point uses up a significant figure (a.k.a. a "sig fig" or "sig dig"). For instance, if you had a number like 666.13. Displaying that to two sig figs would give you 666. Displayed to three sig figs would again be 666. Displayed to four sig figs would be 666.1. Five sig digs would be 666.13. Six sig figs would be 666.130. Seven would be 666.1300, and so on.[6]

To increase the number of sig figs 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 figs 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.

[6]Note that in conventional mathematics, displaying 666.13 to two sig figs would be 670. The first two significant digits are 6 and 6, but the third digit (6) causes rounding up.

3.7 Wide Data vs. Tidy Data

3.7.1 Wide Data

Examining the `skull_cap_partial_wide.csv` file loaded at the beginning of this chapter, we see that the data is structured logically. The first column represents the presumed sex (male or female) of the skulls in each row, as recorded by Thomson and Randall-MacIver (1905). The remaining columns represent skull measurements from a specific period in Egypt's history. Note that the output below displays only the first 8 of 11 columns, and the number of columns visible in your output may vary depending on your screen size.

```
skulls
```

```
# A tibble: 343 × 11
   sex   predynastic c4800BC c4200BC c4000BC c3700BC c3500BC c2780BC
   <chr>       <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
 1 Male         1370    1410    1320    1445      NA    1395    1425
 2 Male         1250    1445    1565    1540      NA    1420    1505
 3 Male         1430    1440    1600    1565      NA    1380    1360
 4 Male         1350    1340    1460    1710      NA    1260    1385
 5 Male         1130    1460    1520    1690      NA    1285    1350
 6 Male         1670    1290    1440    1775      NA    1505    1440
 7 Male         1195    1290    1740    1390      NA    1230    1400
 8 Male         1500    1385    1410    1620      NA    1250    1255
 9 Male         1325    1290    1510    1500      NA    1315    1450
10 Male         1480    1565    1550    1255      NA    1360    1310
# i 333 more rows
# i 3 more variables: c1590BC <dbl>, c378BC <dbl>, c331BC <dbl>
# i 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 breadth of skulls *circa* 4800 BCE, we could just run

```
mean(skulls$c4800BC, na.rm = TRUE)
```

```
[1] 1348.831
```

If we wanted to calculate the mean of each column, we can use the `apply()` function. This function literally *applies* a function of your choosing to either the columns or rows. For example, we might use `apply()` to compute the mean of each column. However, the first column (`$sex`) contains character (`chr`) values which we cannot take the mean of. Trying to do so will generate an error, so we need to exclude such columns before performing our calculations. We can use what we learned about indexing in chapter 1 to ignore that first column (see section 1.4.10).

```r
apply(skulls[, 2:11], MARGIN = 2, FUN = mean, na.rm = TRUE)
```

```
predynastic     c4800BC     c4200BC     c4000BC     c3700BC     c3500BC
   1320.142    1348.831    1434.375    1494.700    1356.429    1336.663

    c2780BC     c1590BC      c378BC      c331BC
   1308.454    1347.488    1285.781    1318.642
```

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 `MARGIN = 2`. `na.rm` is of course an argument belonging to the `mean()` function, which we need to specify so R knows how to handle the `NA` values in the data.

As another example of `apply()`, suppose you wanted to know the maximum value contained in each row, ignoring the first column, you could *apply* the function `max()` to the *rows*. Since our data has 343 rows, we will produce a vector of 343 elements, of which only the first 130 are shown below.

```r
apply(skulls[, 2:11], MARGIN = 1, FUN = max, na.rm = TRUE)
```

```
  [1] 1450 1565 1600 1710 1690 1775 1740 1620 1585 1565 1600 1560 1590
 [14] 1575 1630 1670 1570 1510 1525 1475 1610 1460 1475 1570 1485 1560
 [27] 1570 1570 1510 1540 1495 1520 1640 1600 1515 1590 1450 1740 1515
 [40] 1520 1525 1510 1490 1475 1560 1505 1630 1535 1530 1600 1510 1550
 [53] 1560 1590 1480 1570 1615 1510 1610 1430 1585 1610 1660 1470 1570
 [66] 1665 1440 1520 1560 1460 1485 1385 1530 1485 1470 1510 1430 1420
 [79] 1610 1610 1560 1580 1500 1465 1440 1595 1425 1485 1575 1595 1550
 [92] 1720 1535 1370 1400 1345 1630 1340 1400 1420 1455 1380 1530 1480
[105] 1360 1500 1615 1465 1400 1500 1475 1560 1550 1400 1610 1360 1630
[118] 1760 1325 1400 1510 1640 1635 1620 1435 1305 1490 1545 1465 1540
...
```

The structure of this dataset is what is commonly referred to as the **wide format**. At first glance, R makes it fairly easy to work with data in this format. However, we have not attempted anything particularly complex yet and, apart from the first column, all the columns are conveniently numeric. Even though this layout might seem intuitive, it is not ideal for many types of analysis or visualization. In fact, the original dataset[7] had to be substantially truncated to fit into a usable wide format for our purposes here.

[7]Full data set: https://github.com/statistical-grimoire/thomson-randallmaciver-1905/blob/main/Thomson_Randall-MacIver_1905.csv

3.7.2 Tidy data

Despite its shortcomings, use of the *wide format* is fairly common and gets its name from the fact that it spreads variables across multiple columns. In this case, we can treat of the different historical periods of the skulls (predynastic, c4800BC, c4200BC, c4000BC, etc.) as a single variable in its own right. We might call this variable simply "*period*." That is to say, c4800BC represents a *period* in Egypt's history, c4200BC represents a *period* in Egypt's history, c4000BC represents a *period* in Egypt's history, and so on. Adjunct to this is a second variable, which we could refer to as the "*cranial capacity*" of the skulls. This variable consists of the literal measurements 1370, 1250, 1430, and so on. And of course "*sex*" is a variable in this data as well; however, it is not spread across multiple columns the way *period* and *cranial capacity* are.

When organizing or arranging data, best practices dictate that you **restrict a single variable to a single column**. In this case, the variable *period* is being spread across ten columns. And the variable *cranial capacity* is found within each of those ten columns. To fix this, we can use the *tidyverse* function `pivot_longer()`.

```
skulls_tidy <- pivot_longer(skulls,
  cols = predynastic : c331BC,
  names_to = "period",
  values_to = "capacity"
)
skulls_tidy
```

```
# A tibble: 3,430 × 3
   sex   period      capacity
   <chr> <chr>          <dbl>
 1 Male  predynastic     1370
 2 Male  c4800BC         1410
 3 Male  c4200BC         1320
 4 Male  c4000BC         1445
 5 Male  c3700BC           NA
 6 Male  c3500BC         1395
 7 Male  c2780BC         1425
 8 Male  c1590BC         1440
 9 Male  c378BC          1310
10 Male  c331BC          1450
# i 3,420 more rows
# i Use `print(n = ...)` to see more rows
```

Notice a few differences with this pivoted data. There are now considerably more rows in the data, 3,430 vs. 343, and there are two new columns, `$period` and `$capacity` which have replaced the ten different time period columns. In terms of the `pivot_longer()` function we used, the most important argument we specified is `cols`, as this defines which columns are going

to be collapsed into a single column. 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.

In the above code, we used `:` to specify a range of columns (from predynastic to c331BC), but we could have instead provided a vector of column names, as shown below:

```
cols <- c(
  predynastic, c4800BC, c4200BC, c4000BC, c3700BC,
  c3500BC, c2780BC, c1590BC, c378BC, c331BC
)
```

It is worth noting that the *tidyverse* has numerous methods for selecting multiple columns simultaneously that do not exist as part of base R. For a rundown of each see the R documentation: `?tidyr_tidy_select`. Additionally, if you are operating outside of the *tidyverse*, analogous base R functions will usually require column names to be entered as character strings; though, *tidyverse* functions are typically indifferent to this practice. E.g.,

```
cols <- c(
  "predynastic", "c4800BC", "c4200BC", "c4000BC", "c3700BC",
  "c3500BC", "c2780BC", "c1590BC", "c378BC", "c331BC"
)
```

By collapsing the 10 period columns into one, the data has now 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 skull 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.

Interestingly, because of the restriction that data frames and tibbles have whereby each column needs to contain the same amount of elements, the wide data necessarily had to make use of `NA` values. The pivoting that was done to transform the data from wide to tidy preserved those `NA` values, but now there is no need for them to keep them in the data (because they just represent a non-existent value). As an example, consider row five:

```
skulls_tidy[5, ]
```

```
# A tibble: 1 × 3
  sex   period  capacity
  <chr> <chr>      <dbl>
1 Male  c3700BC       NA
```

Without a value for `$capacity`, this row conveys no meaningful information. As such, it—and any rows like it—can be safely removed from the dataset.

```
skulls_tidy <- drop_na(skulls_tidy, capacity)
skulls_tidy
```

```
# A tibble: 1,449 × 3
   sex   period      capacity
   <chr> <chr>          <dbl>
 1 Male  predynastic     1370
 2 Male  c4800BC         1410
 3 Male  c4200BC         1320
 4 Male  c4000BC         1445
 5 Male  c3500BC         1395
 6 Male  c2780BC         1425
 7 Male  c1590BC         1440
 8 Male  c378BC          1310
 9 Male  c331BC          1450
10 Male  predynastic     1250
# i 1,439 more rows
# i Use `print(n = ...)` to see more rows
```

The function `drop_na()` simply removes (i.e., "drops") any rows that have a `NA` value in the columns you specify. Numerous methods exist for removing `NA` values, this is merely one means by which to do that that remains true to the *tidyverse's* cannon of spell casting.

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

One of the most significant contributions of the *tidyverse* to R has been its seamless implementation of what is known as 'piping' syntax, which allows for an almost otherworldly level of efficiency. However, this is not to say that piping was a concept invented entirely by the *tidyverse*. Rather, the *tidyverse*'s consistent and innovative use of it brought its potential to light, leading to widespread adoption within the R community. The best evidence of this influence is the integration of piping into base R as of version 4.1.0, released in 2021.[8] But, for the uninitiated, what exactly is 'piping'?

[8] Although I have no direct evidence that the *tidyverse* directly motivated the addition of the pipe operator to base R, it seems unlikely to be a coincidence given the ubiquity of `%>%` and the popularity of the *dplyr* package.

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 cranial capacity across all periods of skulls. We could of course insert the `$capacity` column into the mean function like so ...

```
mean(skulls_tidy$capacity)
```

```
[1] 1335.255
```

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

```
skulls_tidy$capacity |> mean()
```

```
[1] 1335.255
```

We could then send that output to something like the `round()` function.

```
skulls_tidy$capacity |>
  mean() |>
  round(1)
```

```
[1] 1335.3
```

As another example, suppose we wanted a tidy data frame that only contained skulls from the predynastic period. The standard methodology would be to specify the data frame within the `filter()` function, like so ...

```
filter(skulls_tidy, period == "predynastic")
```

```
# A tibble: 318 × 3
   sex   period      capacity
   <chr> <chr>          <dbl>
 1 Male  predynastic     1370
 2 Male  predynastic     1250
 3 Male  predynastic     1430
 4 Male  predynastic     1350
 5 Male  predynastic     1130
 6 Male  predynastic     1670
 7 Male  predynastic     1195
 8 Male  predynastic     1500
 9 Male  predynastic     1325
10 Male  predynastic     1480
# i 308 more rows
# i Use `print(n = ...)` to see more rows
```

Alternatively, we could pipe the data into the `filter()` function to achieve the exact same result.

```
skulls_tidy |> filter(period == "predynastic")
```

In addition to this, suppose you did not want the `$sex` column inside the output. To achieve this, this output could be further piped into the *tidyverse's* `select()` function which allows you to grab specific columns.[9]

```
skulls_tidy |>
  filter(period == "predynastic") |>
  select(period, capacity)
```

```
# A tibble: 318 × 2
   period      capacity
   <chr>          <dbl>
 1 predynastic     1370
 2 predynastic     1250
 3 predynastic     1430
 4 predynastic     1350
 5 predynastic     1130
 6 predynastic     1670
 7 predynastic     1195
 8 predynastic     1500
 9 predynastic     1325
10 predynastic     1480
# i 308 more rows
# i Use `print(n = ...)` to see 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 its introduction in R version 4.1.0, 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 `%>%`. This was, and to a certain extent still is, the de facto pipe used by the R community at large. However, the recommended wisdom now (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.

[9] Note that we could obtain the same result by excluding the sex column in the function. i.e., `select(-c(sex))`

3.8.1 Data Manipulation Example

Summarising the Data

The `skull_cap_partial_wide.csv` 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`, pipe it to the `pivot_longer()` function, and then pipe that into the `drop_na()` function.

```
skulls <- read_csv("skull_cap_partial_wide.csv") |>
  pivot_longer(
    cols = predynastic:c331BC,
    names_to = "period",
    values_to = "capacity"
  ) |>
  drop_na(capacity)

skulls
```

```
# A tibble: 1,449 × 3
   sex   period      capacity
   <chr> <chr>          <dbl>
 1 Male  predynastic     1370
 2 Male  c4800BC         1410
 3 Male  c4200BC         1320
 4 Male  c4000BC         1445
 5 Male  c3500BC         1395
 6 Male  c2780BC         1425
 7 Male  c1590BC         1440
 8 Male  c378BC          1310
 9 Male  c331BC          1450
10 Male  predynastic     1250
# i 1,439 more rows
# i Use `print(n = ...)` to see more rows
```

It is 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, `skulls`. 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 we want to know the mean cranial capacity of each period. 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 a very influential *tidyverse* package called *dplyr*.

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., it is 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. 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.

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

```
skulls |>
  summarise(m = mean(capacity))
```

```
# A tibble: 1 × 1
      m
  <dbl>
1 1335.
```

The code we have written is telling the `summarise()` function to apply the `mean()` function to the `$capacity` column. When it did this, it also created a new data frame[10] and stored that calculation as a column called `$m` (though, we could have named the column whatever we wanted).

At present, none of this may 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 `$period` and then summarise the data.

[10] Yes, yes—I know, *technically* what we created was a *tibble*. But that's only because `skulls` was already a tibble to begin with. For the sake of sanity (mine and yours), I'll be treating tibbles and data frames as interchangeable for the rest of this book. Purists, feel free to clutch your pearls.

```
skulls |>
  group_by(period) |>
  summarise(m = mean(capacity))
```

```
# A tibble: 10 × 2
   period          m
   <chr>       <dbl>
 1 c1590BC     1347.
 2 c2780BC     1308.
 3 c331BC      1319.
 4 c3500BC     1337.
 5 c3700BC     1356.
 6 c378BC      1286.
 7 c4000BC     1495.
 8 c4200BC     1434.
 9 c4800BC     1349.
10 predynastic 1320.
```

We can now see the mean of each period in the data set and if we wanted, we could create another column, `$n`, showing how many skulls there are in each period total by using the `length()` function to count the skulls.

```
skulls |>
  group_by(period) |>
  summarise(
    m = mean(capacity),
    n = length(capacity)
  )
```

```
# A tibble: 10 × 3
   period          m     n
   <chr>       <dbl> <int>
 1 c1590BC     1347.   203
 2 c2780BC     1308.   152
 3 c331BC      1319.   232
 4 c3500BC     1337.   315
 5 c3700BC     1356.     7
 6 c378BC      1286.    32
 7 c4000BC     1495.    50
 8 c4200BC     1434.    16
 9 c4800BC     1349.   124
10 predynastic 1320.   318
```

If we wanted to add in a column, `$N`, that represented the total amount of skulls across all the periods we could count the number of rows in `$skulls`...

```
nrow(skulls)
```

```
[1] 1449
```

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

```
skulls |>
  group_by(period) |>
  summarise(
    m = mean(capacity),
    n = length(capacity),
    N = nrow(skulls)
  )
```

```
# A tibble: 10 × 4
   period          m     n     N
   <chr>       <dbl> <int> <int>
 1 c1590BC     1347.   203  1449
 2 c2780BC     1308.   152  1449
 3 c331BC      1319.   232  1449
 4 c3500BC     1337.   315  1449
 5 c3700BC     1356.     7  1449
 6 c378BC      1286.    32  1449
 7 c4000BC     1495.    50  1449
 8 c4200BC     1434.    16  1449
 9 c4800BC     1349.   124  1449
10 predynastic 1320.   318  1449
```

Mathematical operations can also be applied to columns created within the `summarise()` function. For instance, historical sources suggest that the ancient Egyptians used a volumetric unit known as the "heqat," primarily for measuring grain, with an estimated value of approximately 4.8 litres (Clagett, 1989). This allows us to convert skull capacities from cubic centimetres to heqats through a simple division. Since 1 litre = 1,000 cm^3, it follows that 1 heqat $\approx$ 4,800 cm^3. Thus if we wanted the mean and median cranial capacity in heqats we simply run ...

```
skulls |>
  group_by(period) |>
  summarise(
    m = mean(capacity),
    n = length(capacity),
    N = nrow(skulls),
    m_heq = m / 4800,
    med_heq = median(capacity) / 4800
  )
```

```
# A tibble: 10 × 6
   period          m     n     N m_heq med_heq
   <chr>       <dbl> <int> <int> <dbl>   <dbl>
 1 c1590BC     1347.   203  1449 0.281   0.280
 2 c2780BC     1308.   152  1449 0.273   0.270
 3 c331BC      1319.   232  1449 0.275   0.275
 4 c3500BC     1337.   315  1449 0.278   0.277
 5 c3700BC     1356.     7  1449 0.283   0.284
 6 c378BC      1286.    32  1449 0.268   0.266
 7 c4000BC     1495.    50  1449 0.311   0.311
 8 c4200BC     1434.    16  1449 0.299   0.306
 9 c4800BC     1349.   124  1449 0.281   0.283
10 predynastic 1320.   318  1449 0.275   0.273
```

Notice that in the expression `m_heq = m / 4800`, we were able to reuse the previously defined variable `m` within the same `summarise()` function. This is one of the more especially convenient aspects of this piping approach.

To finish up, lets include the maximum and minimum capacities found in each period. We will also store this as an tibble called `skull_summary`.

```
skull_summary <- skulls |>
  group_by(period) |>
  summarise(
    m = mean(capacity),
    n = length(capacity),
    N = nrow(skulls),
    m_heq = m / 4800,
    med_heq = median(capacity) / 4800,
    min = min(capacity),
    max = max(capacity)
  )
skull_summary
```

```
# A tibble: 10 × 8
   period          m     n     N m_heq med_heq   min   max
   <chr>       <dbl> <int> <int> <dbl>   <dbl> <dbl> <dbl>
 1 c1590BC     1347.   203  1449 0.281   0.280  1080  1665
 2 c2780BC     1308.   152  1449 0.273   0.270  1030  1660
 3 c331BC      1319.   232  1449 0.275   0.275  1000  1570
 4 c3500BC     1337.   315  1449 0.278   0.277   965  1760
 5 c3700BC     1356.     7  1449 0.283   0.284  1245  1450
 6 c378BC      1286.    32  1449 0.268   0.266  1095  1550
 7 c4000BC     1495.    50  1449 0.311   0.311  1235  1775
 8 c4200BC     1434.    16  1449 0.299   0.306  1110  1740
 9 c4800BC     1349.   124  1449 0.281   0.283  1110  1640
10 predynastic 1320.   318  1449 0.275   0.273  1050  1710
```

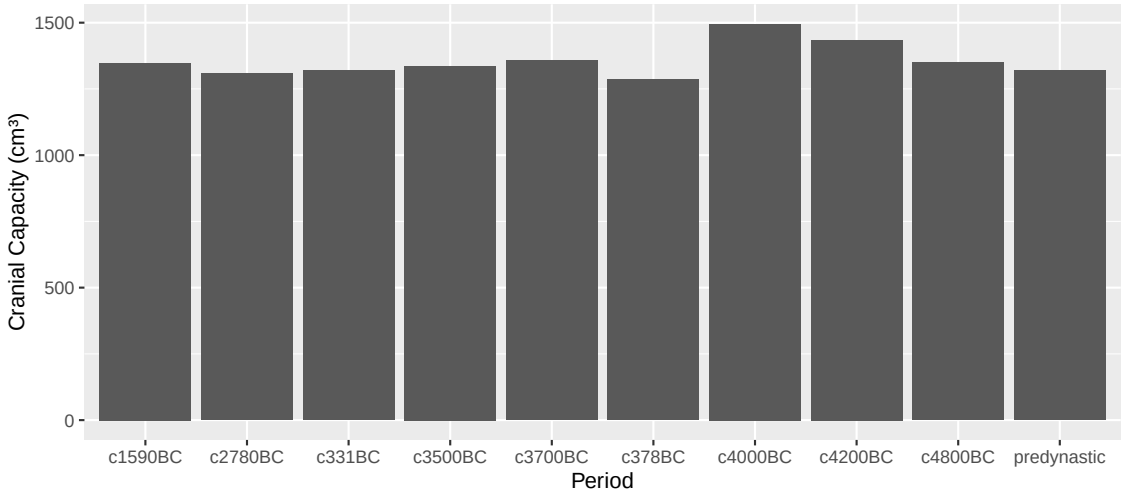
Plotting the Summarised Data

Now that we have neatly organized these summary statistics inside a tibble, we can visualize them. Since the `$period` column contains ten discrete categories, a bar plot is a basic natural choice for representing these data, so that is what we will create.

The basic logic of plotting has been discussed at length in chapter 2, and this discussion will follow from that.[11] 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. Though, it is worth remembering that any of the other numeric (`<dbl>`) columns could be used as well.

```
ggplot(skull_summary, aes(x = period, y = m)) +
  geom_bar(stat = "identity") +
  labs(x = "Period", y = "Cranial Capacity (cm³)")
```

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



The argument `stat = "identity"` is simply telling *ggplot2* to use the values within the `skull_summary` tibble to create the bars. We needed to specify this because *ggplot2* has the ability to take the raw data directly (e.g., `skulls`) 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.

At the moment, going from left to right, the time periods confusingly appear in a non-chronological order. So how do we fix that? This is where the concept of *factors* becomes essential.

3.9 Factors

In statistics, we often refer to a categorical variable as a **factor**. Factors consist of different **levels**, which correspond to the unique categories that variable can take.

For example, in our tidy data, the variable `$period` can be considered a factor. Each distinct time period in that column—such as `predynastic`, `c4800BC`, `c4200BC`, `c4000BC`, and so on—represents a different level of the factor. That is, the factor named `$period` has multiple levels, one for each unique period label.

To summarize: in tidy data, you can think of a "factor" as essentially a categorical variable (or column), and a "level" as one of its possible categories. Just beware that this terminology is specific to tidy data layouts.[12]

[12]While "factors" have a more technical definition in statistics—particularly in the context of experimental design and modelling—this simplified description is sufficient for our current purposes.

- Factor = column
- Level = category within a column

If we examine `skulls`:

```
skulls
```

```
# A tibble: 1,449 × 3
   sex   period      capacity
   <chr> <chr>          <dbl>
 1 Male  predynastic     1370
 2 Male  c4800BC         1410
 3 Male  c4200BC         1320
 4 Male  c4000BC         1445
 5 Male  c3500BC         1395
 6 Male  c2780BC         1425
 7 Male  c1590BC         1440
 8 Male  c378BC          1310
 9 Male  c331BC          1450
10 Male  predynastic     1250
# i 1,439 more rows
# i Use `print(n = ...)` to see more rows
```

You can see that the output is telling us that the `$period` column is a character vector (notice the `<chr>`). In other words, R does not know that `predynastic`, `c4800BC`, `c4200BC`, etc. are categories. It just sees 1,449 individual character values in that particular column. For the purpose of plotting and analyses, it is important that R understands that these 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]

```
skulls$period <- factor(skulls$period)
skulls
```

```
# A tibble: 1,449 × 3
   sex   period      capacity
   <chr> <fct>          <dbl>
 1 Male  predynastic     1370
 2 Male  c4800BC         1410
 3 Male  c4200BC         1320
 4 Male  c4000BC         1445
 5 Male  c3500BC         1395
 6 Male  c2780BC         1425
 7 Male  c1590BC         1440
```

[13]Technically, when we use this function we are replacing an existing column with a new column that happens to be a class of object called 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.

```
 8 Male   c378BC          1310
 9 Male   c331BC          1450
10 Male   predynastic     1250
# i 1,439 more rows
# i Use `print(n = ...)` to see more rows
```

Notice that the `$period` column is now labelled as `<fct>`, which stands for "factor." Additionally, if we isolate this column, the ten levels of the factor are displayed at the bottom of the output.

```
skulls$period
```

```
...
10 Levels: c1590BC c2780BC c331BC c3500BC c3700BC ... predynastic
```

While this implicit listing is convenient, a better and more deliberate way to view the levels of a factor is to use the `levels()` function.

```
levels(skulls$period)
```

```
 [1] "c1590BC"     "c2780BC"     "c331BC"      "c3500BC"     "c3700BC"
 [6] "c378BC"      "c4000BC"     "c4200BC"     "c4800BC"     "predynastic"
```

3.9.1 Ordering Levels

Discerning readers may have noticed that the order of the levels shown match the order of the bars in the graph we created. This is not a coincidence. Whenever you use *ggplot2* to plot or *dplyr* to summarize categorical data, these packages quietly convert the relevant columns into factors behind the scenes when necessary. By default, R arranges factor levels in alphabetical order, which is why the bars appeared in that particular sequence. However, we can override this default by explicitly specifying the order of the levels when we define the factor using the `factor()` function. This allows us to arrange categories in a more meaningful way—such as placing historical time periods in chronological order.

```
skulls$period <- factor(skulls$period,
  levels = c(
    "predynastic", "c4800BC", "c4200BC", "c4000BC", "c3700BC",
    "c3500BC", "c2780BC", "c1590BC", "c378BC", "c331BC"
  )
)
levels(skulls$period)
```

```
 [1] "predynastic" "c4800BC"     "c4200BC"     "c4000BC"     "c3700BC"
 [6] "c3500BC"     "c2780BC"     "c1590BC"     "c378BC"      "c331BC"
```

It is important to emphasize that reordering the levels of a factor does *not* change the actual order of the values in the data frame. The rows remain exactly as they were. What we are doing

instead is instructing R that, for the purposes of plotting or analysis, `predynastic` should be treated as coming before `c4800BC`, which comes before `c4200BC`, and so on. If we now re-run our earlier code to compute summary statistics, you will see that the `$period` column reflects this new ordering and is listed as `<fct>`.

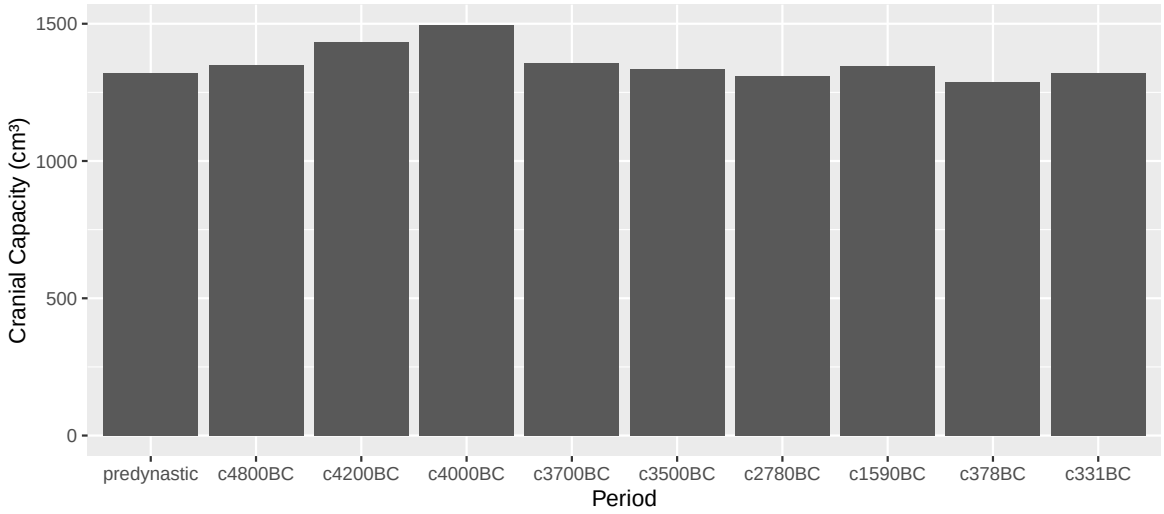
```
skull_summary <- skulls |>
  group_by(period) |>
  summarise(
    m = mean(capacity),
    n = length(capacity),
    N = nrow(skulls),
    m_heq = m / 4800,
    med_heq = median(capacity) / 4800,
    min = min(capacity),
    max = max(capacity)
  )

skull_summary
```

```
# A tibble: 10 × 8
   period        m     n     N m_heq med_heq   min   max
   <fct>     <dbl> <int> <int> <dbl>   <dbl> <dbl> <dbl>
 1 predynas… 1320.   318  1449 0.275   0.273  1050  1710
 2 c4800BC   1349.   124  1449 0.281   0.283  1110  1640
 3 c4200BC   1434.    16  1449 0.299   0.306  1110  1740
 4 c4000BC   1495.    50  1449 0.311   0.311  1235  1775
 5 c3700BC   1356.     7  1449 0.283   0.284  1245  1450
 6 c3500BC   1337.   315  1449 0.278   0.277   965  1760
 7 c2780BC   1308.   152  1449 0.273   0.270  1030  1660
 8 c1590BC   1347.   203  1449 0.281   0.280  1080  1665
 9 c378BC    1286.    32  1449 0.268   0.266  1095  1550
10 c331BC    1319.   232  1449 0.275   0.275  1000  1570
```

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

```
ggplot(skull_summary, aes(x = period, y = m)) +
  geom_bar(stat = "identity") +
  labs(x = "Period", y = "Cranial Capacity (cm³)")
```



3.9.2 Naming Levels

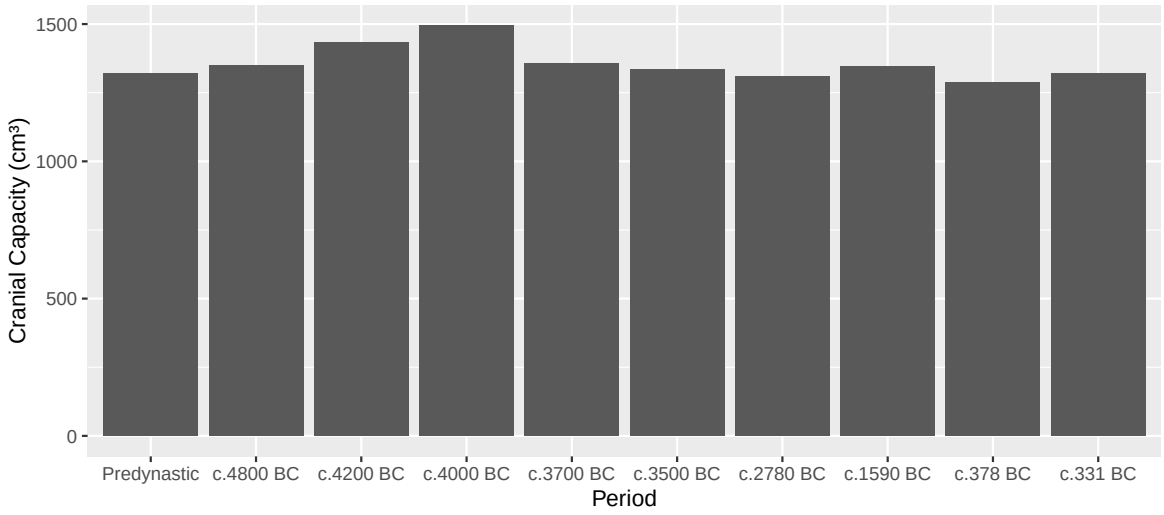
On occasion, it will be useful to rename the levels of a factor. For instance, previously we had used the `labels` argument inside *ggplot2*'s `scale_x_discrete()` function to adjust the x-axis labelling. However, an alternative strategy would have been to relabel the factor levels. We can do this using the `levels()` function from earlier. And we have the option of renaming the levels of the `skulls` or `skull_summary` data frames. We will do the latter so that we do not need to re-run the code that produced `skull_summary`.

```
levels(skull_summary$period) <- c(
  "Predynastic", "c.4800 BC", "c.4200 BC", "c.4000 BC", "c.3700 BC",
  "c.3500 BC", "c.2780 BC", "c.1590 BC", "c.378 BC", "c.331 BC"
)
skull_summary
```

```
# A tibble: 10 × 8
   period          m     n     N m_heq med_heq   min   max
   <fct>       <dbl> <int> <int> <dbl>   <dbl> <dbl> <dbl>
 1 Predynastic 1320.   318  1449 0.275   0.273  1050  1710
 2 c.4800 BC   1349.   124  1449 0.281   0.283  1110  1640
 3 c.4200 BC   1434.    16  1449 0.299   0.306  1110  1740
 4 c.4000 BC   1495.    50  1449 0.311   0.311  1235  1775
 5 c.3700 BC   1356.     7  1449 0.283   0.284  1245  1450
 6 c.3500 BC   1337.   315  1449 0.278   0.277   965  1760
 7 c.2780 BC   1308.   152  1449 0.273   0.270  1030  1660
 8 c.1590 BC   1347.   203  1449 0.281   0.280  1080  1665
 9 c.378 BC    1286.    32  1449 0.268   0.266  1095  1550
10 c.331 BC    1319.   232  1449 0.275   0.275  1000  1570
```

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

```r
ggplot(skull_summary, aes(x = period, y = m)) +
  geom_bar(stat = "identity") +
  labs(x = "Period", y = "Cranial Capacity (cm³)")
```



It would be remiss not to mention that *ggplot2* provides a dedicated method for updating x-axis labels—one that doesn't require modifying the factor levels directly. Specifically, you can use the `labels` argument within the `scale_x_discrete()` function. This approach simply involves supplying a character vector that specifies the new labels in their current plotting order, like so...

```r
ggplot(skull_summary, aes(x = period, y = m)) +
  geom_bar(stat = "identity") +
  labs(x = "Period", y = "Cranial Capacity (cm³)") +
  scale_x_discrete(
    labels = c(
      "Predynastic", "c.4800 BC", "c.4200 BC", "c.4000 BC", "c.3700 BC",
      "c.3500 BC", "c.2780 BC", "c.1590 BC", "c.378 BC", "c.331 BC"
      )
  )
```

Concerning the manipulation of factors, a brief word of warning is in order: **do not** confuse the `levels` *argument* used inside the `factor()` function with the `levels()` *function* itself. While they sound similar, they serve very different purposes.[14]

- `levels = ...` (argument inside `factor()`) is used to *specify the order* of factor levels.
- `levels()` (function) is used to *rename* existing factor levels.

[14]To further complicate things (because of course it does), the `factor()` function also includes a `labels` argument that allows you to rename levels at the time of creation. See the R documentation for details: `?factor`

For beginners, factors can feel particularly troublesome. But they are foundational to R's thaumaturgy and cannot be avoided. So rather than resisting, it is best to embrace their evil, arcane nature—otherwise you will never be at peace with yourself.

3.10 Adding Error Bars

Recall that in addition to the mean estimated cranial capacity for each period, `skull_summary` also includes the smallest and largest measured capacities, stored in the `$min` and `$max` columns, respectively.

```
skull_summary
```

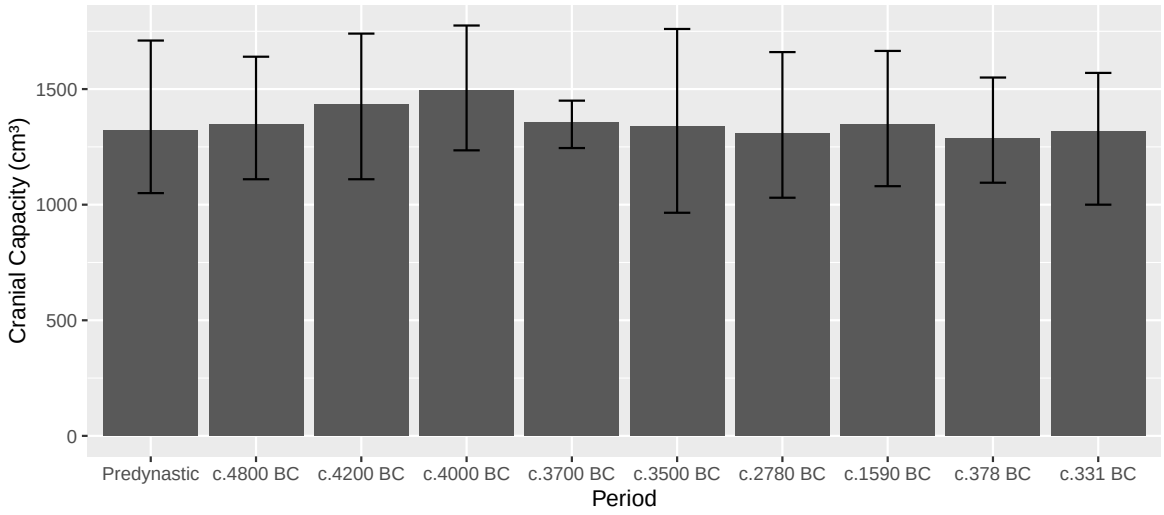
```
# A tibble: 10 × 8
   period          m     n     N m_heq med_heq   min   max
   <fct>       <dbl> <int> <int> <dbl>   <dbl> <dbl> <dbl>
 1 Predynastic 1320.   318  1449 0.275   0.273  1050  1710
 2 c.4800 BC   1349.   124  1449 0.281   0.283  1110  1640
 3 c.4200 BC   1434.    16  1449 0.299   0.306  1110  1740
 4 c.4000 BC   1495.    50  1449 0.311   0.311  1235  1775
 5 c.3700 BC   1356.     7  1449 0.283   0.284  1245  1450
 6 c.3500 BC   1337.   315  1449 0.278   0.277   965  1760
 7 c.2780 BC   1308.   152  1449 0.273   0.270  1030  1660
 8 c.1590 BC   1347.   203  1449 0.281   0.280  1080  1665
 9 c.378 BC    1286.    32  1449 0.268   0.266  1095  1550
10 c.331 BC    1319.   232  1449 0.275   0.275  1000  1570
```

We can incorporate this information into our graph using **error bars**. Error bars provide a visual representation of the data's *spread*, and the difference between the maximum and minimum values corresponds to a classic measure of spread known as the *range*.[15] While the range is generally not recommended as a primary measure of spread, it has the advantage of being intuitive and serves our current illustrative purposes well enough.

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

```
ggplot(skull_summary, aes(x = period, y = m)) +
  geom_bar(stat = "identity") +
  geom_errorbar(aes(ymin = min, ymax = max), width = 0.25) +
  labs(x = "Period", y = "Cranial Capacity (cm³)")
```

[15] If this concept isn't entirely clear yet, don't worry—spread, as a formal statistical idea, will be explored in more detail in later chapters.

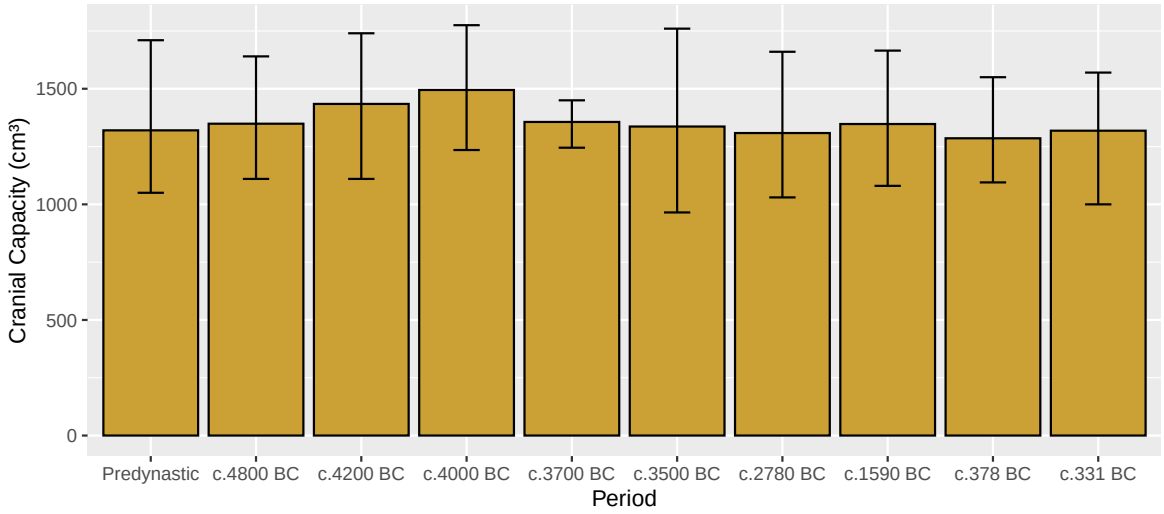


3.11 Bar Fill Colour

To further enhance the bar plot's visual appeal, we could adjust the fill colour of the bars to reflect the corresponding time periods.

But we should pause for a moment because, strictly speaking, this is something we should NOT do—unless we have a compelling reason. And in this case, we do not have a compelling reason. The purpose of a plot like this is to compare the heights of the bars—that is, the y-axis values. From a scientific standpoint, it's best to ensure that each bar has equal visual weight, and using a single consistent colour accomplishes exactly that. Since the categories are already labelled on the x-axis, additional colours only serve as unnecessary distractions that might bias or obscure the comparisons you are trying to see. For instance, something like this would be scientifically acceptable:

```
ggplot(skull_summary, aes(x = period, y = m)) +
  geom_bar(stat = "identity", colour = "black", fill = "#CBA135") +
  geom_errorbar(aes(ymin = min, ymax = max), width = 0.25) +
  labs(x = "Period", y = "Cranial Capacity (cm³)")
```



That said, if you're collaborating on a project, your teammates may insist on rainbow-coloured bars or other such nonsense regardless of this sound logic. And if they outnumber you, they'll win the vote—and possibly the fistfight. Science offers no defence against the tyranny of "aesthetics."

Setting aside everything we just said, if we do decide to adjust the colour of the bars, we need to be mindful of how the x-axis is structured. Technically, the x-axis represents a *discrete* scale—not a *continuous* one (see Section 2.9.1 for more on this distinction). However, while *ggplot2* treats it as discrete (because it's a bar plot), the underlying variable—ordered time periods—does follow a natural continuum theoretically. So, in this case, using a continuous colour palette is not entirely unjustified. To that end, we can make use of one of R's many HCL palettes discussed in Section 2.9.5. The "Green-Brown" palette, in particular, is a solid choice—it provides a smooth gradation that maps well onto temporal data without being garish or overly distracting.[16]

Given that we have 10 separate categories of dates, we can define our palette in the following way.[17]

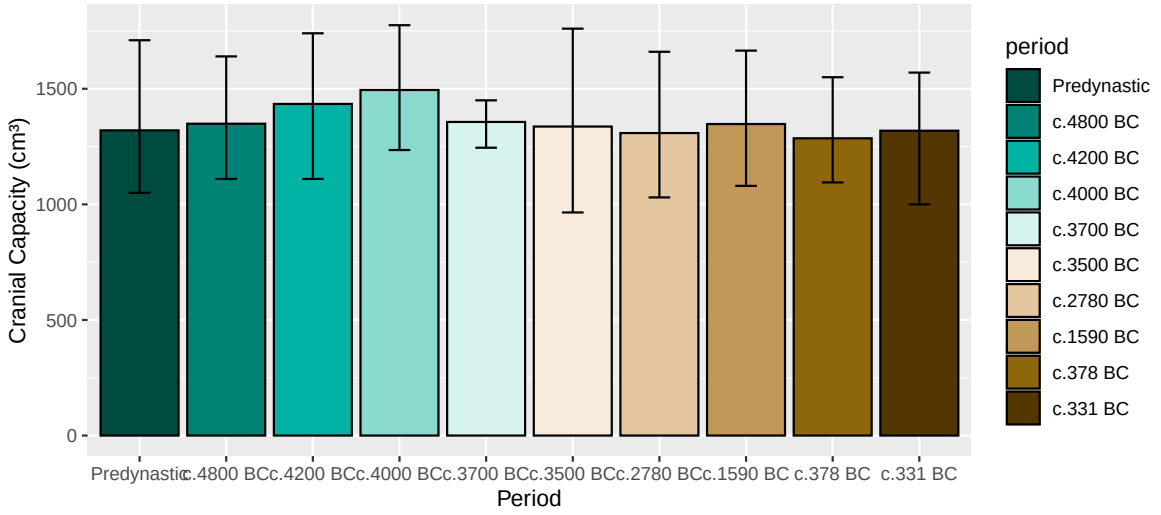
```
1 egypt_pal <- hcl.colors(n = 10, palette = "Green-Brown")
```

[16] A visual guide to all HCL palettes is available in Appendix B.

[17] If you want to be a bit fancy—and make your code more robust—you can write something like `egypt_pal <- hcl.colors(length(levels(skulls$period)), palette = "Green-Brown")`. This way, if you ever remove levels, the number of colours will automatically adjust, and you won't need to revise your code manually.

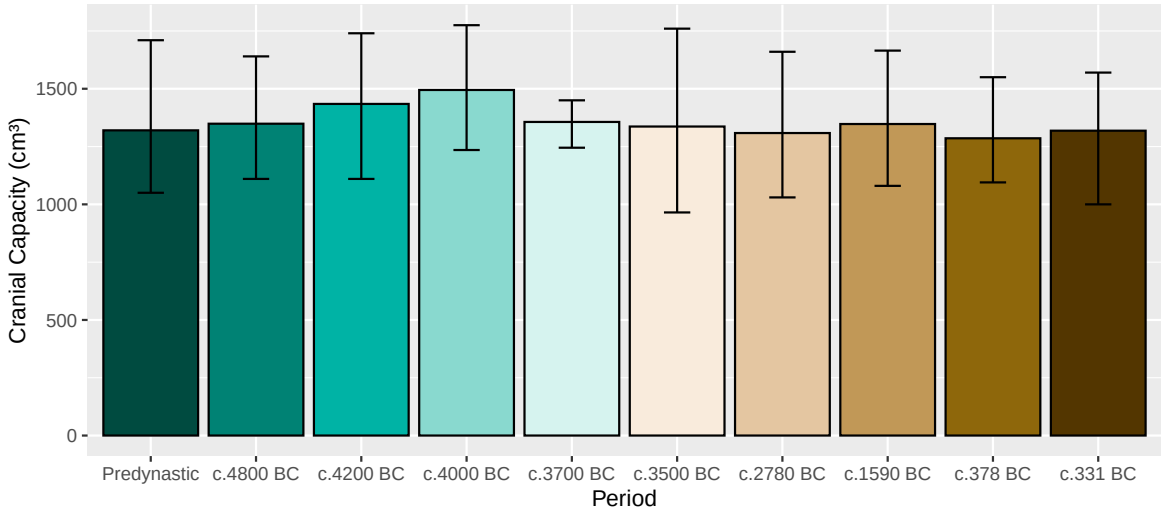
We can then use `egypt_pal` to adjust the fill colour of the bars in the plot.

```
ggplot(skull_summary, aes(x = period, y = m)) +
  geom_bar(stat = "identity", colour = "black", aes(fill = period)) +
  geom_errorbar(aes(ymin = min, ymax = max), width = 0.25) +
  labs(x = "Period", y = "Cranial Capacity (cm³)") +
  scale_fill_manual(values = egypt_pal)
```



ggplot2 quite sagely adds a legend when you map fill colours to a variable; however, in this particular case the legend is redundant with the information our x-axis provides and is thus taking up space unnecessarily. To remove the legend, there are different methods that could be employed. Since we only have the fill aesthetic mapped, it is easy enough to just add `guide = "none"` to the `scale_fill_manual()` function.

```
ggplot(skull_summary, aes(x = period, y = m)) +
  geom_bar(stat = "identity", colour = "black", aes(fill = period)) +
  geom_errorbar(aes(ymin = min, ymax = max), width = 0.25) +
  labs(x = "Period", y = "Cranial Capacity (cm³)") +
  scale_fill_manual(values = egypt_pal, guide = "none")
```



3.12 Putting It All Together

To consolidate everything covered in this chapter, it's helpful to revisit the analysis one final time—but in a more realistic, streamlined, end-to-end format. Doing so not only reinforces how the various components work together in a complete R script, but also provides an opportunity to enhance the graph by incorporating some faceting[18] based on the previously ignored variable `$sex`.

```r
# Load the tidyverse (and praise the our dear leader Hadley)
library(tidyverse)
```

```r
# Load the data
skulls <- read_csv("skull_cap_partial_wide.csv") |>
  # Pivot to the tidy format
  pivot_longer(
    cols = predynastic:c331BC,
    names_to = "period",
    values_to = "capacity"
  ) |>
  # Remove NAs
  drop_na(capacity)
```

```r
# Factor and order "period"
skulls$period <- factor(skulls$period,
  levels = c(
    "predynastic", "c4800BC", "c4200BC", "c4000BC", "c3700BC",
    "c3500BC",     "c2780BC", "c1590BC", "c378BC",  "c331BC"
  )
)
```

[18] Facets were covered in Chapter 2, section 2.6.

```r
# Rename the factor levels
levels(skulls$period) <- c(
  "Predynastic", "c.4800 BC", "c.4200 BC", "c.4000 BC", "c.3700 BC",
  "c.3500 BC",   "c.2780 BC", "c.1590 BC", "c.378 BC",  "c.331 BC"
)
```

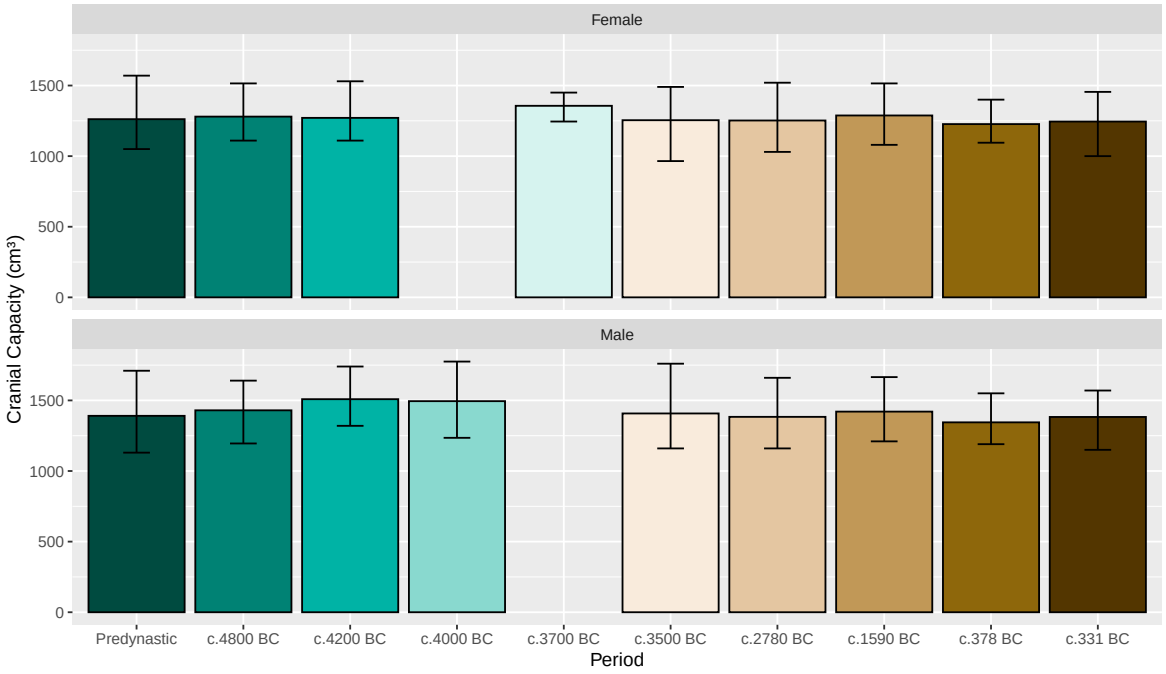
```r
# Calculate stats for plot, grouping by 'period' and 'sex'
skull_summary <- skulls |>
  group_by(period, sex) |> # Note the addition of a second factor to group_by()
  summarise(
    m = mean(capacity),
    min = min(capacity),
    max = max(capacity)
  )

skull_summary # Output shows group-wise means and ranges, separated by sex
```

```
# A tibble: 18 × 5
# Groups:   period [10]
   period      sex        m   min   max
   <fct>       <chr>  <dbl> <dbl> <dbl>
 1 Predynastic Female 1262.  1050  1570
 2 Predynastic Male   1391.  1130  1710
 3 c.4800 BC   Female 1280.  1110  1515
 4 c.4800 BC   Male   1430.  1195  1640
 5 c.4200 BC   Female 1271   1110  1530
 6 c.4200 BC   Male   1509.  1320  1740
 7 c.4000 BC   Male   1495.  1235  1775
 8 c.3700 BC   Female 1356.  1245  1450
 9 c.3500 BC   Female 1255.   965  1490
10 c.3500 BC   Male   1408.  1160  1760
11 c.2780 BC   Female 1252.  1030  1520
12 c.2780 BC   Male   1384.  1160  1660
13 c.1590 BC   Female 1288.  1080  1515
14 c.1590 BC   Male   1421.  1210  1665
15 c.378 BC    Female 1227.  1095  1400
16 c.378 BC    Male   1345.  1190  1550
17 c.331 BC    Female 1245   1000  1455
18 c.331 BC    Male   1383.  1150  1570
```

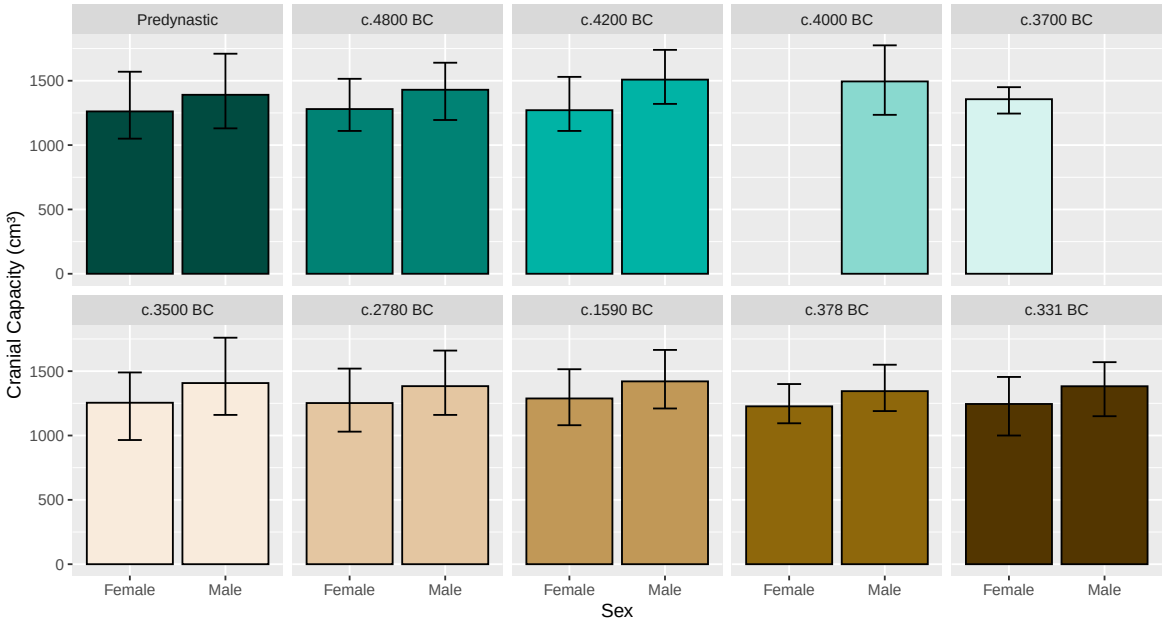
```r
# Store desired colours
egypt_pal <- hcl.colors(n = 10, palette = "Green-Brown")
```

```r
# Plot data
ggplot(skull_summary, aes(x = period, y = m)) +
  geom_bar(
    stat = "identity",
    colour = "black",
    aes(fill = period)
  ) +
  geom_errorbar(aes(ymin = min, ymax = max), width = 0.25) +
  scale_fill_manual(values = egypt_pal, guide = "none") +
  labs(
    x = "Period",
    y = "Cranial Capacity (cm³)"
  ) +
  facet_wrap(~ sex, ncol = 1) # Note the use of facet_wrap
```



Finally, it is worth demonstrating just how easily—and dramatically—we can adjust *ggplot2* to suit different analytical goals. As it currently stands, the plot is structured to compare cranial capacity across time periods within each sex. However, with just two minor changes (see line 2 and 14), we can reorient the layout to instead compare the sexes within each time period. Specifically, we place `$sex` on the x-axis and facet according to `$period`.

```
1  # Plot data
2  ggplot(skull_summary, aes(x = sex, y = m)) +
3    geom_bar(
4      stat = "identity",
5      colour = "black",
6      aes(fill = period)
7    ) +
8    geom_errorbar(aes(ymin = min, ymax = max), width = 0.25) +
9    scale_fill_manual(values = egypt_pal, guide = "none") +
10   labs(
11     x = "Sex",
12     y = "Cranial Capacity (cm³)"
13   ) +
14   facet_wrap(~ period, ncol = 5)
```



PART II

Descriptive Statistics - Seeing Without Asking

This part opens the gate to those ancient tools which allow us to extract form from the formless. We do not yet ask questions of the gods; we simply take stock of the offering they have laid before us.

Chapter 4

Taxonomies of the Profane - Variables, Scales, and Their Unholy Properties

THERE is a kind of grim devilry in the act of classification. The moment you categorize a thing—whether it be a small volume of blood, the reaction time of a startle, or the flickering presence of a belief—you strip it from the chaos of the unknown and chain it down, trembling, to a scale. Statisticians call them variables, but do not be fooled: these are not gentle creatures. They are twisted reflections of reality that must be bound in measurement and tortured into obedience. Nominal. Ordinal. Interval. Ratio. These are the sigils we etch into our grimoires of data, each one whispering what kind of rituals—summations, correlations, regressions—we may dare perform. But beware: misuse the wrong scale, or confuse the nature of your variable, and the results may turn on you, distorted and cursed. This chapter delves into the infernal art of measurement, uncovering the hidden laws that govern how data can be named, ranked, counted, or quantified. Prepare yourself—for to wield statistics is to practice a kind of taxonomy, yes ... but one written in the ink of madness, precision, and utter cruelty.

4.1 A Practical Problem

Consider the complete craniometric dataset provided by Thomson and Randall-MacIver (1905), available in the file `Thomson_Randall-MacIver_1905.csv`,[1] a "small" excerpt of which is displayed in Table 4.1. The file contains a wide range of craniometric measurements along with other useful, and sometimes missing, contextual information, such as the estimated date range for each skull, the ruling dynasty at the time, and the archaeological site of origin.

[1]The data file can be obtained at this book's GitHub repository: https://github.com/statistical-grimoire/thomson-randallmaciver-1905

table	start_date	end_date	start_era	end_era	dynasty	location	sex	gol	ool	bbh	mb	biaurb	bizygb	bnl	bal	nah	nh	nw	fai	ga	po	cc
1			BC	BC	Early Predynastic	Abydos	Male	178.0	177	138.0	131	113	120.0	98	89	68.0	49	23.0	91.0	76.0	A	1370
1			BC	BC	Early Predynastic	Abydos	Male	179.0	179	131.0	125	111	121.0	97	92	67.0	48	23.0	95.0	73.0	B	1250
1			BC	BC	Early Predynastic	Abydos	Male	185.0	185	134.0	136	112	116.0			68.0	47	24.0				1430
1			BC	BC	Early Predynastic	Abydos	Male	183.0	180	132.0	131	112	122.0	103	99	64.0	50	26.0	96.0	74.5	C	1350
1			BC	BC	Early Predynastic	Abydos	Male	169.0	169	132.0	119	106	119.0	100	96	64.0	44	25.0	96.0	74.0	B C	1130
1			BC	BC	Early Predynastic	Abydos	Male	202.0	202	143.0	136	119	130.0	107	100	75.0	54	24.0	93.0	73.5	A	1670
1			BC	BC	Early Predynastic	Abydos	Male	185.0	185		114					68.0	47	23.0				
1			BC	BC	Early Predynastic	Abydos	Male	175.0	175		128											
1			BC	BC	Early Predynastic	Abydos	Male	190.0	190		146											
1			BC	BC	Early Predynastic	Abydos	Male	188.0	188		127											
1			BC	BC	Early Predynastic	Abydos	Male	177.0	177	122.0	130											1195
1			BC	BC	Early Predynastic	Abydos	Male	192.0	189		136											
1			BC	BC	Early Predynastic	Abydos	Male	187.0	187	137.0	138	114	123.0	96	89	76.0	56	25.0	93.0	70.5	A	1500
1			BC	BC	Early Predynastic	Abydos	Male	176.0	174	134.0	132	100		98	86	65.0			88.0	79.5	- A	1325
1			BC	BC	Early Predynastic	Abydos	Male	192.0	192	130.0	139	115	125.0	103	108	72.0	48	28.0	105.0	66.0	E F	1480
1			BC	BC	Early Predynastic	Abydos	Male	187.0	187		132											
1			BC	BC	Early Predynastic	Abydos	Male	187.0	185	136.0	125	105	119.0	101	93	66.0	48	25.0	92.0	76.0	A	1350
1			BC	BC	Early Predynastic	Abydos	Male	181.0	177	134.0	131	112	125.0	102	102	76.0	51	25.0	100.0	68.0	C	1350
1			BC	BC	Early Predynastic	Abydos	Male	194.0	191	134.0	134	127		109	99	72.0	51	26.0	91.0	77.5	A	1480
1			BC	BC	Early Predynastic	Abydos	Male	191.0	189		130											
1			BC	BC	Early Predynastic	EL Amrah	Male	161.0	161		129											
1			BC	BC	Early Predynastic	EL Amrah	Male	181.0	180	138.0	129		123.0	106	95	65.0	50	24.0	90.0	80.0	A	1370
1			BC	BC	Early Predynastic	EL Amrah	Male	188.0	188	138.0	135		113.0			67.0	47	26.0				1490
1			BC	BC	Early Predynastic	EL Amrah	Male	169.0			140											
1			BC	BC	Early Predynastic	EL Amrah	Male	189.0	188		138											
1			BC	BC	Early Predynastic	EL Amrah	Male	189.0	189		125											
1			BC	BC	Early Predynastic	EL Amrah	Male	182.0	184		128											
1			BC	BC	Early Predynastic	EL Amrah	Male	195.0	193													
1			BC	BC	Early Predynastic	EL Amrah	Male	193.0	191	140.5	131		128.0									1495
1			BC	BC	Early Predynastic	EL Amrah	Male	190.0	188		125											
1			BC	BC	Early Predynastic	Hou	Male	177.0	176	121.0	134	116	125.0	96	95	72.0	53	25.0	99.0	68.0	B C	1220
1			BC	BC	Early Predynastic	Hou	Male	189.0	188	129.0	126	111	126.0	105	109	68.0	51	26.0	104.0	68.0	F	1305
1			BC	BC	Early Predynastic	Hou	Male	190.0	188	142.0	133	114		106	107	64.0			101.0	71.0	E	1525
1			BC	BC	Early Predynastic	Hou	Male	180.0	179	136.0	132	121	129.0	102	100	66.0	50	27.0	98.0	72.0	C D	1375
1			BC	BC	Early Predynastic	Hou	Male	181.0	179	140.0	141	114	125.0	102	100	72.0	51	27.0	98.0	70.0	C	1520
1			BC	BC	Early Predynastic	Hou	Male	174.0	171	134.0	131	116	130.0	99	97	69.0	54	22.0	98.0	71.0	C	1300
1			BC	BC	Early Predynastic	Hou	Male	178.0	178	137.0	135	112	122.0	109	103	74.0	50	24.0	94.5	74.0	A B	1400
1			BC	BC	Early Predynastic	Hou	Male	176.0	173	133.0	132	112	126.0	98	93	72.0	53	26.0	95.0	71.5	A B	1315
1			BC	BC	Early Predynastic	Hou	Male	181.0	179	136.0	139	116		98	96	70.0	50	27.0	98.0	70.0		1460
1			BC	BC	Early Predynastic	Hou	Male	185.0	183	131.0	132	111	124.0	104	101	68.0	49	26.0	97.0	73.0	C	1360
1			BC	BC	Early Predynastic	Hou	Male	184.0	184	133.0	126	114	125.0	103	102	66.0	51	28.0	99.0	72.0	D	1310
2			BC	BC	Early Predynastic	Hou	Male	186.0	185	135.0	135	116	125.0	105	103	66.0	47	26.0	98.0	73.0	D	1440
2			BC	BC	Early Predynastic	Hou	Male	176.0	175	124.0	134	110	123.0	96	93	74.0	53	23.0	97.0	69.0	A B	1245
2			BC	BC	Early Predynastic	Hou	Male	187.0	186	134.0	128			110	103	67.0	50	27.0	94.0	77.0	B C	1360
2			BC	BC	Early Predynastic	Hou	Male	181.0	181	130.0	130	117	129.0	103	104	69.0	49	25.0	101.0	69.5	D E	1300
2			BC	BC	Early Predynastic	Hou	Male	185.0	182	135.0	138	121	136.0	104	100	78.0	55	27.0	96.0	70.0	A B	1470
2			BC	BC	Early Predynastic	Hou	Male	184.0	184	132.0	128	106	117.0	97	93	72.0	53	25.0	96.0	70.5	A B	1320
2			BC	BC	Early Predynastic	Hou	Male	184.0	184	129.0	127	116	131.0	102	106	63.0	48	28.0	104.0	68.5	F	1290
2			BC	BC	Early Predynastic	Hou	Male	185.0	183	136.0	131	117	129.0	111	114	73.0	54	27.0	103.0	68.5	E F	1400
2			BC	BC	Early Predynastic	Hou	Male	180.0	179	138.0	124	114	120.0	101	101	62.0	46	25.0	100.0	71.5	D E	1310
2			BC	BC	Early Predynastic	Hou	Male	181.0	180	139.0	131	113	121.0	98	92	71.0	53	24.0	94.0	73.0	A	1400
2			BC	BC	Early Predynastic	Hou	Male	183.0	182	138.0	131	111	126.0	102	98	73.0	49	24.0	96.0	71.0	B	1410
2			BC	BC	Early Predynastic	Hou	Male	183.0	180	140.0	137	127	133.0	103	103	74.0	52	22.0	100.0	69.0	C	1495
2			BC	BC	Early Predynastic	Hou	Male	184.0	180	125.0	128	112	121.0	98	95	64.0	44	24.0	97.0	72.5	C	1250
2			BC	BC	Late Predynastic	El Amrah	Male	185.0	185	138.0	124	115	122.0	102	101	67.0	48	26.0	99.0	71.0	D	1350
2			BC	BC	Late Predynastic	El Amrah	Male	178.0	176		138											
2			BC	BC	Late Predynastic	El Amrah	Male	189.0	187	134.0	133		129.0	99	97	65.0	48	30.0	98.0	72.0	C D	1430
2			BC	BC	Late Predynastic	El Amrah	Male	183.0	182		142											
2			BC	BC	Late Predynastic	El Amrah	Male	183.0	183		141											
2			BC	BC	Late Predynastic	El Amrah	Male	194.0	194		134											
2			BC	BC	Late Predynastic	El Amrah	Male	189.0	189	134.0	138		120.0	100	98	67.0	45	26.0	98.0	71.0	C	1490
2			BC	BC	Late Predynastic	El Amrah	Male	190.0	190		132											
2			BC	BC	Late Predynastic	El Amrah	Male	190.0	186	129.0	148	130	134.0	100	104	69.0	51	24.0				
2			BC	BC	Late Predynastic	El Amrah	Male	172.0	171	124.0	126	110	117.0	98	95	61.0	45	27.0	97.0	74.0	C D	1140
2			BC	BC	Late Predynastic	El Amrah	Male	191.0	187	136.0	135	115	128.0	104	98	71.0	52	25.0	94.0	74.0	A B	1490
2			BC	BC	Late Predynastic	El Amrah	Male	185.0	184	145.0	132	119	130.0	102	100	71.0	54	23.0	98.0	70.5	C	1505
2			BC	BC	Late Predynastic	El Amrah	Male	187.0	185		134											
2			BC	BC	Late Predynastic	El Amrah	Male	187.0	189	130.0	133		132.0	103	102	71.0	48	25.0	99.0	70.0	C D	1375
2			BC	BC	Late Predynastic	El Amrah	Male	176.0	176	134.0	131	112	119.0	103	96	71.0	50	25.0	93.0	74.5	A B	1315
2			BC	BC	Late Predynastic	El Amrah	Male	189.0	186	137.0	133			102								1465
2			BC	BC	Late Predynastic	El Amrah	Male	191.0	189		134											
2			BC	BC	Late Predynastic	El Amrah	Male	171.0	172		120											
2			BC	BC	Late Predynastic	El Amrah	Male	173.0	173	125.0	133	111	119.0	95	94	70.0	46	24.0	99.0	69.0	C	1220
2			BC	BC	Late Predynastic	El Amrah	Male	191.0	190	120.0	144			99								1400
2			BC	BC	Late Predynastic	El Amrah	Male	196.0	195		132											
2			BC	BC	Late Predynastic	El Amrah	Male	183.0	183	136.0	136											1440
2			BC	BC	Late Predynastic	El Amrah	Male	185.0	182	131.0	131		127.0	104	98	67.0			94.0	75.0	B	1350
2			BC	BC	Late Predynastic	El Amrah	Male	187.0	187	130.0	122		118.5	103	91	66.5	49	25.0	88.0	79.5	- A	1260
2			BC	BC	Late Predynastic	El Amrah	Male	194.5	192	133.0	132		124.0	104	104	74.0	48	23.5	100.0	69.0	C	1450
2			BC	BC	Late Predynastic	El Amrah	Male	193.0	193	128.0	131											1380
2			BC	BC	Late Predynastic	El Amrah	Male	196.0	194	132.0	136		126.0									1500
2			BC	BC	Late Predynastic	El Amrah	Male	188.0	187	134.0	129											1380
2			BC	BC	Late Predynastic	Hou	Male	179.0	178		132	111	119.0	101	96	66.0	50	23.0	95.0	74.0	B	
2			BC	BC	Late Predynastic	Hou	Male	186.0	184	136.0	133	116	127.0	107	103	72.0	53	27.0	96.0	72.5	B C	1430
2			BC	BC	Late Predynastic	Hou	Male	185.0	183	139.0	131	118	127.0	105	98	70.0	51	26.0	93.0	75.0	A B	1430
2			BC	BC	Late Predynastic	Hou	Male	187.0	185	136.0	131	114	124.0	102	99	77.0	56	25.0	97.0	69.0	A B	1420
2			BC	BC	Late Predynastic	Hou	Male	194.0	194	134.0	138	109	116.0	100	98	68.0	49	24.0	98.0	71.0	C	1525
2			BC	BC	Late Predynastic	Hou	Male	190.0	189	136.0	130	116	127.0	107	104	75.0	53	25.0	97.0	71.0	B C	1430
2			BC	BC	Late Predynastic	Hou	Male	182.0	180	128.0	131	111	126.0	99	98	63.0	45	25.0	99.0	71.0	D	1300

Table 4.1: An excerpt showing the first two, of 32, data tables presented in the appendix of Thomson and Randall-MacIver (1905).

All totalled, there are 23 separate columns of information each with close to 1000 or more values to conduct an analyses with. This raises a couple of important questions:

1. What exactly do we mean by "analyses" in this context?

2. Given the sheer number of distinct values, how can we discuss this data in a practical and meaningful way?

Listing the complete set of values each time we want to reference the data would be wildly impractical. And even if we were absurdly committed to doing so, it is safe to say that our meagre primate brains simply are not equipped to juggle that much information at once. What we are after then are clear, compact, and accurate descriptions of the data that still capture its essential features. Put another way, we need to distil the data's chaos into something intelligible. That is the essence of "conducting" an analysis and, while this may seem a hopeless task given the sheer volume of data, there is often a surprisingly large amount of order buried within this seeming chaos.

4.2 Descriptive and Inferential Analyses

The analysis of data is typically driven by two main objectives. The first is descriptive: to summarize the data in ways that are intuitive and meaningful. Perhaps unsurprisingly, this is commonly referred to as a **descriptive analysis**. The second objective is inferential: to use those summaries to make conclusions that reach beyond the data at hand. These conclusions are generally assumed to have some form of practical relevance[2]—for example, perhaps they help to answer a key research question. This process is known as **inferential analysis**, and it always builds upon descriptive analysis as a necessary foundation.

In many respects a *descriptive analysis* should be a purely empirical endeavour. To the best of the analyst's ability, it seeks to answer a simple question: what can be said with certainty about this data? This often involves computing summary statistics that characterize the dataset as a whole. For example, calculating familiar measures such as the mode, median, and mean, or visualizing the distribution using graphs like histograms, are all methods that tell us something about which values are most commonly observed in the data set and how the values as a whole are distributed.

Inferential analysis goes a step further by introducing reasonable assumptions that allow us (the analyst) to make predictions or generalizations that extend beyond the data at hand. In most cases, we are not interested in the dataset for its own sake—we care about what it represents

[2]While practical relevance should be a requirement for any inferential analysis, many such analyses are performed more out of tradition than genuine purpose. This isn't cynicism—just the voice of experience, tinged with a bit of jadedness.

more broadly. That is, we want to know what can it tell us about a broader population, what trends might it reveal about an underlying system, what future outcomes might it help us predict? Statistical methods that allow us to extrapolate in this respect are inferential in nature.

4.3 Data

The word data has appeared frequently throughout this book, often without much reflection as to what it actually means. Given that data is the central subject of both *descriptive* and *inferential* analysis, it is perhaps worth taking a moment to clarify. At its core, **data** refers to a collection of observations about *something*. The singular form, **datum**, refers to just one of those observations. The "something" in question is usually the phenomenon the researcher is investigating—this could be the number of cells in a slice of brain tissue, the rate of deaths per capita, how quickly participants learn a behavioural response; or any number of other things that can be measured or classified in some way.

Before going further, it is worth drawing a distinction between what we will refer to as *statistical data*—the kind typically used in research and analysis (e.g., see Table 2.1 and Table 4.1)—and the kind of data used to train machine learning models (e.g., pictures of cats, playlists of Eurodance hits, or whatever else the algorithm gods demand). In the latter case, what is being fed to the model is perhaps more akin to collections of stimuli than it is data in the traditional statistical sense of the term. That said, in machine learning contexts, the terms *data* and *stimuli* are often used interchangeably. When this book refers to "data", you can assume it means *statistical data*.[3]

4.4 Variables

Examining Table 4.1, we can see that each row corresponds to a single skull examined by Thomson and Randall-MacIver (1905). Each column captures a different type of information recorded for that skull. Some columns contain categorical details—such as the ruling dynasty at the time of burial, the archaeological site where the skull was found, or the presumed sex of the individual—while others include numeric measurements, like the glabello-occipital length (gol), ophryo-occipital length (ool), and basi-bregmatic height (bbh), just to name a few.[4]

Each column in a dataset represents what we call a **variable**—a single characteristic or property that can differ (i.e., *vary*) across the observations. It is worth noting that this use of the term "variable" is slightly different from how it is often used in something like algebra, which is probably the most familiar context in which the term appears. In algebra, a variable is a symbol

[3]Yes, I'm aware I haven't defined the term "statistic" yet. But some knowledge comes at a price—and you haven't bled nearly enough.

[4]View the data file's README document for the complete listing.

that stands for an unknown value or set of values (hence the classic phrase "solve for x", with x being the variable). Still, the two meanings have an underlying connection because, in both cases, we are referring to something—be it a symbol, label, or phrase—that can represent different values. For example, in Table 4.1 *sex* is a variable that (from Thomson and Randall-MacIver's perspective) has two possible values: "male" and "female." Cranial capacity (*cc*) is also a variable, but it is a variable which can take on a theoretically infinite amount of possibilities that extend anywhere from 0 to infinity in the positive direction.[5]

The way you conduct a descriptive or inferential analysis hinges on the nature of your variables. For instance, it makes little sense to try to compute the mean average of categorical traits such as eye colour, biological sex, or whether a plant is alive or dead. While this may seem self-evident, these types of variables are often represented numerically in data files. For example, "Alive" might be coded as 1 and "Dead" might be coded as 0, giving the illusion that arithmetic operations are appropriate. But calculating a mean assumes the data have specific numerical properties: namely, that the values lie on a scale where addition and division are meaningful. When these conditions are not met, the result is not just meaningless—it is misleading. For example, if you have 20 plants that are alive and 30 that are dead, calculating the mean would yield an "aliveness" score of 0.4. But what does that actually tell you? That the average plant is 40% alive? As we will see, the mean is intended to reflect the value most typical of the data. But what exactly is "typical" about 0.4 when, under our coding scheme, the only valid values are 0 (dead) and 1 (alive)? No plant in the dataset is 40% alive. The absurdity of this result lays bare the danger of treating categorical variables as if they were genuinely numerical. Just because something is *represented* by a number does not mean it *is* a number.

This issue strikes at the very core of what it means to measure something. Once a research question is identified, what needs to be measured often feels self-evident—duration, height, topography, severity, intensity, hardness, rate, and so on. But how to measure it is an entirely different matter. Suppose we want to assess the height of 100 people. One approach is straightforward: use a tape measure and record each individual's height in metres. Alternatively, we could assign rankings—giving the tallest person a score of 100 and the shortest a score 1. A third option would be to sort them into broad categories: short, medium, and tall. Or we might simply ask them, via questionnaire, "How tall are you in metres?" While one of these methods clearly rises above the others in terms of scientific precision, each method produces its own kind of height "measurement".[6] And therein lies the deeper problem: not every measurement is created equal.

[5]This is not to suggest that Godzilla-sized skulls are in any way feasible. An asymptote lurks somewhere along the continuum — and long before we reach kaiju level proportions, the laws of physics (and the poor creature's neck) would surely intervene.

[6]To show how flexible measurement can be, in Canada, distance is commonly measured in units of time. For example: "Eh bud, d'ya know how far Timmies is?" 'Oh, bout five minutes. Just make a larry, it's kitty-corner from the rink.' "Thanks eh."

4.5 Measurement and The Problem of Measurability

Prior to the late 1940s, scientific measurement was primarily understood as the assignment of numerals to real-world magnitudes—quantities that were assumed to exist independently of the observer. The central idea being that mathematical relationships could meaningfully represent relationships among physical objects or phenomena. However, a philosophical dilemma began to take shape as the field of psychology sought to align itself with the standards of the natural sciences. There was heated debate about whether mathematical relations could validly capture the complexities of the human mind. At the heart of this controversy was a deceptively simple question: Is it possible to measure human sensation? (Stevens, 1946, p. 677).

To get a sense of the problem being grappled with, suppose you ask participants in a study to rate their happiness on a 11-point scale, where 0 represents the absence of any happiness and 10 represents the happiest they could conceivably be. For simplicity, imagine you only have two participants—one selects a 3, the other a 5. A common research practice is to compute the mean of the scores, which in this case would be 4 (see equation 4.1).

$$\frac{3+5}{2} = \frac{8}{2} = 4 \tag{4.1}$$

This seems straightforward enough. The average happiness level across these two people is 4. However, there is a potential problem lurking here: the "psychological distance" between the numbers on the scale may not be consistent across individuals. What one person considers a 5, another might interpret as a 4, or as a 6, or as a 7, or as a 8.4, or a 2.66, or some other value. That becomes an issue when we calculate a mean because the process of adding the values in the numerator assumes these values have a standardized meaning independent of the person. For instance, if instead the participants had reported 2 and 6 we would similarly arrive at 8 in the numerator (see equation 4.2).

$$\frac{2+6}{2} = \frac{8}{2} = 4 \tag{4.2}$$

However, we have no compelling reason to assume that a mathematical equality such as $(3+5) = (2+6)$ should hold in this context (see 4.3), because these numbers reflect *subjective* judgments—not objective quantities grounded in a standardized unit of measurement. Is it mathematically true that both $(3+5) = 8$ and $(2+6) = 8$? Yes, arithmetically. But in the realm of subjective ratings, such an equality only holds if participants are interpreting the scale in a comparable way. While that is possible, it is far from guaranteed given the vast differences in individual physiology and lived experience people have.

$$(3 + 5) \stackrel{?}{=} (2 + 6) \stackrel{?}{=} 8 \tag{4.3}$$

And the problems do not end there. More fundamentally, there is no objective means of verifying the accuracy—or even the honesty—of any subjective report. We are left to trust that participants are both willing and able to faithfully describe their internal states. This issue, often referred to as the *problem of introspection*, has haunted Psychology since its inception, tracing back to Wilhelm Wundt's 19th-century laboratory, where the first systematic efforts to understand "minds" began.[7]

None of this is to suggest that subjective assessments should be dismissed outright as pseudoscience and you would be hard pressed to find anyone in the modern day dismissing these types of measures out of hand. As Labovitz (1967) contends, there may be some practical value—however impure—in treating such ratings as more numerical than they truly are. Despite their crude, unstable, and potentially erroneous nature, these measures may still contain just enough precision to tease some signal from the noise. By analogy, the literal sound an engine makes is not what powers a vehicle, but a skilled mechanic can sometimes diagnose a problem from the sound alone.[8] Subjective assessments may be similar in this respect. Introspection is, without doubt, a murky cauldron—but an obsession with methodological purity does risk sacrificing potentially useful data on the altar of perfectionism. We ought not discard what might yield insight simply because it falls short of an ideal.

4.6 Scales of Measurement

The most influential attempt to resolve the problem of measurability came from Stevens (1946), who defined measurement as "the assignment of numerals to objects or events according to rules" (p. 677)—a view that aligned with the operationalist philosophy dominant at the time. When we use a rule to assign numbers to aspects of objects or events, we create a *scale*, and Stevens proposed that measurements could be classified into four distinct types: nominal, ordinal, interval, and ratio. These scales differ in how closely the numbers we use reflect the real-world properties we are trying to measure. Stevens devised these scales based on the degree of correspondence (i.e., isomorphism) between the properties of numbers in a series and the the real-world properties we are trying to measure. This relationship, in turn, determines which mathematical operations are meaningful for a given measurement. The four scales are often viewed as forming a hierarchy of increasing numerical richness.[9]

---

[7] One could argue that this problem traces back even earlier, to the foundational work of Ernst Heinrich Weber and Gustav Fechner in psychophysics—research that Wundt deeply admired and which heavily influenced his own.

[8] Though, one does rather expect a mechanic to occasionally look under the hood as well.

[9] Hang in there—it's less arcane than it sounds (even with words like *isomorphism* thrown around). I promise it'll make more sense by the third reread. For a deeper dive into these measurement scales and their theoretical foundations, see Stevens 1951.

4.6.1 Nominal Scales

As was previously mentioned, numbers can be used to represent the values of a variable. For instance, if you have catalogued the "handedness" of various individuals, you might assign 0 for ambidextrous, 1 for left-handed, and 2 for right-handed. One of the most basic properties of numbers is that each number is distinct from every other number: for example, $1 \neq 2$, $1 \neq 3$, $1 \neq \sqrt{3}$, and so on ad infinitum. Likewise, identical numbers are treated as equivalent: $1 = 1$, $2 = 2$, $\sqrt{3} = \sqrt{3}$, etc. **Nominal scales** of measurement preserve this property of distinctness and equivalence that numbers have. In other words, they allow us to determine whether two values refer to the same category or not, but they do *not* carry any information about order, magnitude, or arithmetic relationships. In our handedness example, we are simply labelling three distinct categories. The numbers here are symbolic: 2 does not imply that right-handedness is "greater" than left-handedness, only that $1 \neq 2 \neq 0$.

Mathematics aside, nominal measurement scales are fundamentally about categorizing observations into mutually exclusive[10] unordered groups. The numbers are just convenient labels. Common examples include binary responses like "yes" or "no," taste qualities such as sweet, sour, salty, bitter, and savoury, or biological traits like feeding strategy (e.g., herbivore, carnivore, omnivore, detritivore), disease presence (e.g., infected vs. not infected), or treatment group (e.g., placebo, drug A, drug B). What matters is not the number itself (any set of numbers can be used), but the category it stands in for.

At the risk of undermining what was just said, in many cases, researchers do not actually bother coding nominal categories as numbers at all. For example, Table 4.1 stores the nominal variables (dynasty, location, sex) as plain text. Modern computers handle character strings efficiently, and it is often easier to interpret variable levels when they are stored in a human-readable format. Of course, exceptions exist—especially in contexts like regression analysis, where categorical variables must be converted into numerical representations to be included in the model.

4.6.2 Ordinal Scales

Ordinal scales share the properties of distinctness and equivalence found in nominal scales, but they also introduce the crucial property of order. That is, the numbers can be arranged meaningfully along a ranked continuum. For example, 1 is *greater* than 0, 2 is *greater* than 1, 3 is *less* than 4, and so on. In other words, ordinal scales impose a logical sequence—such as $0 < 1 < 2 < 3 < 4$—on the categories they represent. When coding variables that have an inherent order, the numerical values used should reflect that ordering.

A classic example is letter grades. A grade of *A* reflects a higher level of academic perfor-

[10] *Mutually exclusive* means that each observation can belong to only one category. You cannot, for instance, be both right-handed and ambidextrous.

mance than a B, which in turn reflects higher performance than a C, and so forth. These categories are often translated into grade point values—e.g., $A = 4.0$, $B = 3.0$, $C = 2.0$, $D = 1.0$—which makes the ranking explicit: $4.0 > 3.0 > 2.0 > 1.0$. However, this is where the usefulness of the numbers stops. We cannot meaningfully say that an A is "twice as good" as a C, or that the difference between a B and a C is the same as the difference between an A and a B. This is because the criteria for assigning these grades differ across courses, disciplines, and institutions. The numeric labels indicate order, but not magnitude or difference (see Box 4.1).

Other examples of ordinal scales include developmental stages in biology (e.g., larva → pupa → adult), where the stages follow a clear order but the differences between them are not necessarily equal in duration or complexity. In chemistry and safety contexts, hazard levels (e.g., flammable → highly flammable → extremely flammable) represent increasing danger, but again without uniform steps. In astronomy, seeing conditions are rated subjectively as "poor," "fair," "good," and so on, these are ordered categories that describe atmospheric clarity without precise measurement. Similarly, the Fujita Scale for tornadoes ranks storm intensity from F0 to F5 based on observed damage, providing an ordered but not evenly spaced classification.

4.6.3 Interval Scales

The **interval scale** is the first scale in which measurements are meaningfully numerical, supporting most standard mathematical operations with minimal risk of misapplication. Like ordinal scales, interval scales preserve both distinctness (different numbers represent different things) and order (larger numbers represent greater amounts). But interval scales go a step further by allowing meaningful statements about the differences between values—because the units along the scale are empirically equal.

Take, for example, the span of time between certain historical events. The difference between the years 1347 and 1014 is the same as the difference between 1666 and 1333: in both cases, exactly 333 years separate the two points. These differences are equivalent because a year is a standardized, empirically defined unit—it corresponds to the time it takes the Earth to complete one full orbit around the Sun, or more precisely, the time between two vernal equinoxes (when the Sun crosses the equator heading north and a blood sacrifice is demanded). Crucially, this unit does not change depending on when it is measured. A year is a year, whether you are counting from the fall of Rome or the end of the world.[11]

Often, when we measure something to be a value of zero, we are not just assigning a number—we are declaring the absence of the thing itself. For instance, a height of 0 metres implies no person. A scale reading 0 kilograms suggests nothing is being weighed. However, interval scales

[11] A small caveat: we are treating the empirical unit of a year as if it were perfectly constant, when in fact it fluctuates by fractions of a second due to gravitational interactions, axial wobble, and orbital quirks. These variations are imperceptible to humans and rarely matter in practice—but over time, the errors accumulate. This is, of course, the reason leap years exist: to reconcile our tidy calendar with the untidy behaviour of the cosmos.

Box 4.1: The GPA Illusion

Let's take a moment to talk about something sacred: the Grade Point Average (GPA). This little number wields tremendous power. It determines scholarships, grad school admissions, and sometimes whether your parents buy you dinner. But there's a secret no one likes to talk about: GPA is a mathematical illusion. It's the *mean* average of grade point values ($A = 4.0$, $B = 3.0$, $C = 2.0$, etc.), which themselves are just numbers slapped onto ordinal categories. We know that an A is better than a B, and a B is better than a C—but is the "distance" between them truly equal? Does moving from a B to an A represent the same leap in achievement as going from a D to a C?

If you're thinking "probably not," you'd be right. Taking the mean of ordinal data is technically a no-no because the maths assumes equal spacing between the numbers—something ordinal scales don't guarantee. It's like claiming the difference between silver and gold medals is the same as between bronze and a participation ribbon. If schools were being honest and fair, they'd use the *median* average instead: it respects rank without pretending the scale is evenly spaced. It's also a notably *robust* statistic—so a few dodgy classes won't derail your grad school dreams. But honesty doesn't feed the machine. The *median* flattens distinctions. It produces ties. It refuses to give the illusion of precision that bureaucracies crave. The *mean*, in contrast, offers lovely decimals—3.47 vs. 3.52—that suggest meaningful differences where none may actually exist. It is ideal for sorting, and thus ideal for systems that would prefer not to think too hard.

Your instructors probably know better. Most of them, anyway. But the people making the rules—the ones who decide how grades are processed—tend to prioritize convenience over correctness.[a] Why else would an entire institution choose a method so statistically indefensible? Their unofficial motto might as well be: *Numerus est veritas, etiam si mendacium*—"The number is truth, even if it's a lie."

[a] I am going to get in so much trouble for writing this.

eschew this. On interval scales a value of zero does not represent the absence of the thing being measured. For example, if we measure time in years, the year 0 does not indicate the absence of time—it is simply a reference point.[12] Consider temperature as another case. One unit on the Celsius scale represents 1/100th of the interval between the freezing and boiling points of water at standard atmospheric pressure. Zero degrees Celsius is defined as the freezing point of water, but this does not represent the absence of temperature. The molecules in the water are still moving—still vibrating and colliding—meaning there is still kinetic energy present. In short, zero on an interval scale is arbitrary, not absolute.[13]

This fact about zeros and interval scales has big implications for the kind of mathematics we can conduct. Interval scales allow us to talk meaningfully about the *difference* between measurements; however, they do not allow us to make meaningful statements about the *ratio* of two measurements. For example, we cannot say that 10°C is twice as warm as 5°C (see Equation 4.4).

$$\frac{10°\text{C}}{5°\text{C}} \neq 2 \tag{4.4}$$

This is one of the core issues with having an arbitrary, rather than absolute, zero. A ratio such as Equation 4.5—

$$\frac{10}{5} = 2 \tag{4.5}$$

—implies that 10 contains two full units of whatever quantity is represented by 5. But that logic only works when zero represents the complete absence of the thing being measured.

Interestingly, while interval scales do not support ratios of values, they do support ratios of *differences*. For example, if the temperature on Monday was 10°C and the temperature on Tuesday was also 10°C, the difference is 0°C—an absolute value indicating no change in temperature (i.e., 10°C − 10°C = 0°C). In this context, zero *does* mean "none of the thing"—it reflects a complete absence of temperature *change*. In other words, talking in terms of differences gives us an absolute zero that we can utilize in the context of ratios.

For instance, it is perfectly valid to say that the temperature change between 10°C and 0°C is twice as large as the change between 5°C and 0°C:

[12] Casually referring to year 0 is a bit of a historical no-no. In the calendar used by most historians (the Gregorian calendar), there is no year 0—time jumps straight from 1 BCE to 1 CE with no pause for breath. The concept of zero didn't exist in Roman numerals, so early timekeepers just skipped it. Astronomers, being a more mathematically inclined bunch, use a system that does include a year 0—because trying to do calculations across 1 BCE/1 CE without one is annoying.

[13] This is why the concept of absolute zero exists, it is the theoretical point at which all molecular motion stops: $-273.15°C$

$$\frac{10°\text{C} - 0°\text{C}}{5°\text{C} - 0°\text{C}} = 2 \tag{4.6}$$

Now admittedly, if you look at how Equation 4.6 arrives at a value of 2, you might be confused—because it appears to be doing exactly what Equation 4.4 said was not permissible. Step by step, the calculation proceeds as follows:

$$\frac{10°\text{C} - 0°\text{C}}{5°\text{C} - 0°\text{C}} = \frac{\mathbf{10°C}}{\mathbf{5°C}} = \frac{10\cancel{°C}}{5\cancel{°C}} = 2 \tag{4.7}$$

The second step (in bold text) looks identical to Equation 4.4. But it is not the same. In Equation 4.4, the numerator and denominator represent absolute temperatures—values on an arbitrary scale where zero does not represent an absence of temperature. In contrast, in Equation 4.7, both the numerator and denominator represent temperature differences, and differences on an interval scale are meaningful because they reflect quantities with consistent units.

Unfortunately, this is a distinction that the maths alone can obscure. The symbolic operations may look identical, but the interpretation depends entirely on what the numbers represent.

4.6.4 Ratio Scales

Of the four scale types identified by Stevens (1946), **ratio scales** are the most robustly numeric. They retain all the properties of nominal, ordinal, and interval scales, but with one critical addition: a true zero point that signifies the complete absence of the measured attribute. This allows for meaningful ratio comparisons—for example, one value can be said to be twice another—and eliminates concerns about the appropriateness of mathematical operations like multiplication or division. Ratio scales are especially prevalent in the natural sciences, where quantities such as mass (e.g., of tissues), cell count, blood volume, elapsed time, concentration (e.g., parts per million), frequency, reinforcer amount, and behavioural response duration are measured from zero and up, enabling a full range of mathematical analysis.

4.6.5 Implications of Scale Type for Statistical Analysis

The concept of measurement scales carries with it a deceptively simple but important warning: it is entirely possible to use numbers incorrectly. That is, one can feed numerical values into a statistical formula, obtain a result, and walk away with a clean-looking number—but that number may bear little resemblance to anything meaningful in the real world (Roberts, 1985). Worse, it might produce an interpretation that is subtly misleading and difficult to detect.

Think of a statistical procedure like a meat grinder: it does not care what you put in it. Toss in high-quality ingredients, and you will get something palatable. Toss in a boot, and you

will still get an output—just not one you probably want to serve. Likewise, statistical methods are indifferent to the scale of measurement used. They will happily crunch any numbers you supply, whether those numbers actually warrant mathematical manipulation or not. But just as pureed leather makes a poor sausage, treating ordinal or nominal data as if it were interval or ratio can yield results that are technically valid but conceptually indigestible.

The entire project of Stevens (1946) was, in many ways, aimed at avoiding this exact mistake. His advocacy centred on the principle of invariance—that we should use statistical and mathematical procedures that preserve the properties of the underlying scale (Stevens, 1968). By identifying the scale on which a variable is measured, researchers can more appropriately select analyses that respect the structure of the data and preserve its intended meaning.

To see why this matters, consider a simple experiment designed to test whether a drug reduces headache severity. One group of participants receives a placebo, another group receives an active treatment. An hour after taking the placebo or treatment, participants rate their current headache intensity on a 5-point Likert scale, using the labels shown in Table 4.2.

Likert Value	Description	Interpretation	True Discomfort Value
1	No headache	No pain, no symptoms	1
2	Mild headache	Noticeable but not bothersome; no impairment	3
3	Moderate headache	Interferes somewhat with activities; may require rest or over-the-counter medication	10
4	Severe headache	Significant discomfort; activities impaired; likely requires medication	32
5	Extremely severe/ incapacitating headache	Cannot function; may require lying down or medical intervention	100

Table 4.2: Example of a 5-point Likert scale for headache severity, including an illustrative mapping to an underlying true discomfort value.

As the researcher, you do not have direct access to participants' internal experience of pain. There's no standardized unit of discomfort that you can measure like you would millimetres or kilograms. The Likert scale values are, in effect, numerical placeholders for subjective states.

Now suppose—just for argument's sake—that you *do* know the true underlying discomfort each Likert value represents. Rather than simple integers from 1 to 5, perhaps the "true" subjective intensity of the headache follows an exponential (1, 3, 10, 32, and 100) progression when compared to the Likert scale divisions. Notice that this alternative scale preserves the order of the original ratings, but the intervals between points are no longer equal. The leap in discomfort from a "1" to a "2" on the Likert scale is not actually equivalent to the leap from "4" to "5"; the subjective change

is far greater in the latter. Put another way, the Likert scale preserves the rank order of discomfort but fails to represent *differences* between discomfort levels in a meaningful or proportionate way to the person's "true" experience.

This has real consequences for how we interpret certain statistical results. For instance, researchers commonly use a statistic called Cohen's d to quantify the size of a treatment effect. In this context, Cohen's d would give us a measure of how effective the drug was at reducing headache severity compared to the placebo. However, Cohen's d relies on means, which assume the data are measured on at least an interval scale—that is, where equal differences between values reflect equal differences in the underlying construct. The Likert scale used here, by contrast, preserves only the ordering of values, not the spacing between them. It is ordinal, not interval. The "true" discomfort values we have listed, on the other hand, do meet the criteria of an interval scale.

To explore the consequences of this mismatch, we can simulate this experiment in R thousands of times, randomly generating data and calculating Cohen's d for each iteration—once using the raw Likert scores, and once using the true discomfort values. If the scale of measurement genuinely does not matter to our Cohen's d effect size statistic, then the values we obtain should be roughly equivalent across both cases. In other words, we should observe similar values of Cohen's d regardless of which scale we use.

Figure 4.1 displays the results across 10,000 simulated experiments. The divergence between the two versions of Cohen's d is striking. When based on the Likert values, the effect size appears consistently larger, overstating the drug's effectiveness. In contrast, the effect size calculated from the true values tells a more modest story. This mismatch illustrates just how misleading it can be to apply interval-based statistics to ordinal data. The sausage may look the same on the plate—but if you grind in the wrong ingredients, the result may not sit well. Respecting the scale of measurement reduces the risk that the outcome of a statistical test is an artifact of numerical representation rather than a valid insight.

In this vein, it is worth noting that alternative effect size measures designed for ordinal data—such as Cliff's delta (δ)—are not susceptible to this distortion. Because Cliff's δ relies only on order information, it produces equivalent results whether calculated from the Likert scores or the true discomfort values. In other words, when the tool matches the scale, the message stays consistent.

Because of these issues, Stevens and others attempted to prescribe which statistical methods were appropriate for each type of measurement scale to help researchers avoid making precisely these kind of mistakes (Robinson, 1965; Senders, 1958; Stevens, 1946). However, that endeavour—and Stevens' broader framework—has not gone unchallenged.

One common criticism is that his classification system is not as clear-cut as it initially appears. Many real-world variables seem to occupy a gray area. Take intelligence (I.Q.) scores,

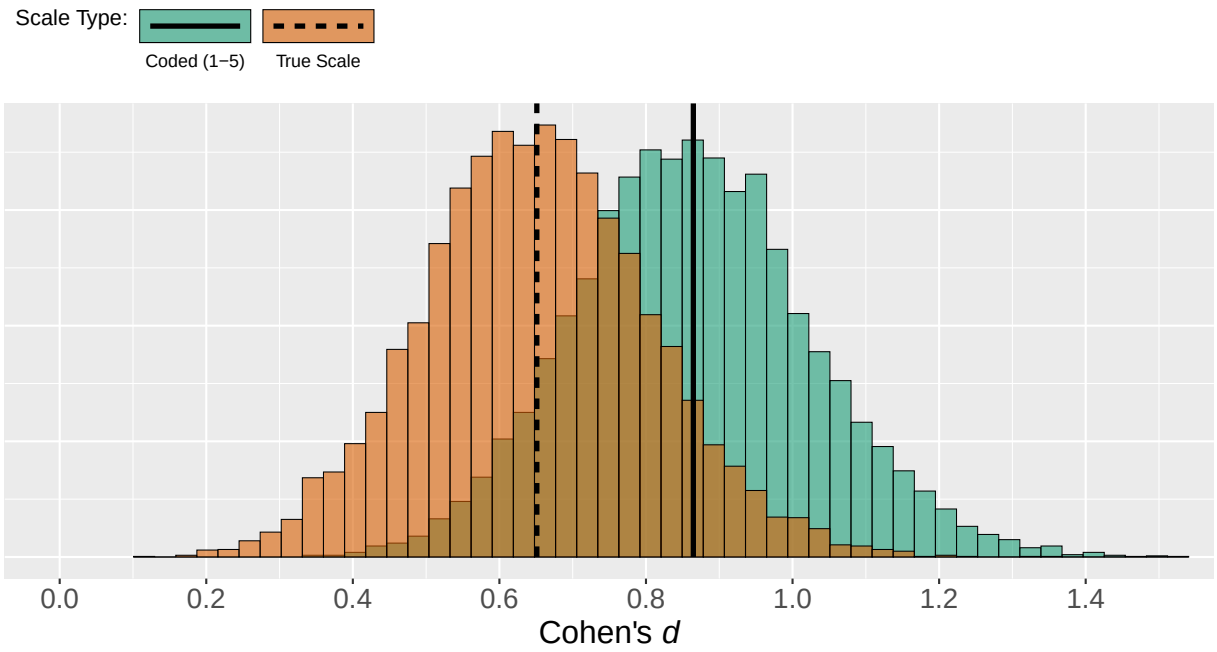


Figure 4.1: Results from 10,000 simulated experiments comparing effect size estimates using Cohen's d based on Likert-scale values versus "true" exponentially scaled discomfort values. Although both scales preserve ordinal ranking, the unequal intervals in the true values reveal that the Likert-based d consistently overestimates the effect size. Black horizontal lines indicate the mean Cohen's d for each scale type. This highlights how treating ordinal data as interval can distort results—like mistaking shoe leather for sausage. Simulation code available at: https://github.com/statistical-grimoire/ordinal-data-simulation

for example: while technically ordinal, they are often treated as though they possess interval properties. Researchers frequently assume that differences in I.Q. scores reflect meaningful psychological differences, even though the scale may not meet the strict criteria required for interval measurement.[14]

Additionally, many statistical procedures that theoretically require interval or ratio-level data often yield results similar to those produced by ordinal methods when applied to ordinal-scale data (Baker et al., 1966). This robustness has led some to argue that, in practice, violating scale assumptions may not always lead to catastrophic errors—at least not when the primary concern is *statistical significance* (e.g., p-values). However, this reassurance comes with important caveats:

- The consequences of misusing statistical methods can be more serious in high-stakes contexts or when findings are generalized beyond their original scope.
- The apparent robustness often vanishes when we move beyond p-values to other statistics—such as Cohen's d—which can be heavily distorted by inappropriate assumptions about

[14]This is not to suggest that differences between IQ scores *do* or *do not* reflect meaningful psychological differences—only that some researchers proceed as though they do.

scale.

Some critics go even further, questioning whether there is any necessary connection at all between level of measurement and the validity of a statistical technique (Gaito, 1980). This is arguably true—again—if the only statistic you care about is the p-value. Others warn that strict adherence to Stevens' taxonomy may encourage a kind of statistical "mindlessness," where researchers rely on rigid checklists instead of thoughtful judgment.[15]

The debate over Stevens' framework is long-standing, and many critiques are directed less at his core principles than at overly simplistic—or at times uncharitable—interpretations of them (see Zand Scholten & Borsboom, 2009, for a summary). Yet despite the criticism, Stevens' central insight is difficult to dispute: researchers must remain mindful of how the scale of measurement can influence both the validity and the interpretation of statistical conclusions.

But the story does not end there. The idea of measurement scales is just one piece of a larger field known as Measurement Theory, which in turn sits within an even broader philosophical landscape that includes competing frameworks such as operationalism, realism, and information theory, to name a few. As fascinating (and important) as these topics are, this book is not primarily concerned with the philosophy of measurement. The discussion so far has been driven by pragmatism: in the context of doing research, Stevens' taxonomy remains genuinely useful for thinking about data and informing analytic decisions.

Stevens gave us a practical and widely adopted classification system—but in the eyes of the most rigorously developed modern theory of measurement, *Representational Measurement Theory* (RMT; Luce & Suppes, 2001), his framework does not quite qualify as real measurement. From the RMT perspective, Stevens' definition is too loose, and some of his scales—nominal and ordinal—do not meet the criteria for genuine measurement at all. For RMT, assigning numbers is not enough; measurement must involve assigning numbers that preserve the structure of the empirical world.

So, Stevens' scales are not so much wrong as they are just incomplete. They provide a helpful, practical way to classify data types and choose appropriate methods, but they do not offer a rigorous foundation for what measurement truly means. You can (and should) still use them—just do not treat them as the final word.

Think of it this way: Stevens' framework is like Newtonian physics—practical, useful, and perfectly adequate for most everyday purposes (including landing on the Moon). But just as Newton's laws break down at relativistic speeds or near black holes—where Einstein's theory is

[15]This is an odd complaint, as it seems to assume that, had Stevens kept quiet, researchers would have spontaneously developed better judgment on their own. Still, the concern isn't entirely without merit. Any framework, however well-intentioned, can be misapplied as dogma. And when tools like measurement scales are treated as gospel—especially by novices—they risk crowding out the deeper, more nuanced reasoning that good statistical practice demands.

needed—Stevens' model breaks down when we start asking deeper questions about the nature of measurement. In that analogy, RMT is Einsteinian: more precise, more demanding, and more accurate at the theoretical edges (but also more complex).

One especially intriguing development within this more rigorous framework of RMT is additive conjoint measurement, which demonstrates that, under certain conditions, meaningful interval-level representations can be constructed from purely ordinal data—so long as specific axioms are satisfied. This has powerful implications for the social sciences, where directly measurable quantities are rare, but ordinal judgments (like rankings or preferences) are common. In short: with the right structure, even simple orderings can yield valid and mathematically rich measurements.

4.7 Other Distinctions Between Variables

Variables are not only distinguished by their scale of measurement; they can also be more broadly categorized as either *qual*itative or *quant*itative. A **qualitative variable** represents non-numeric characteristics or categories. These categories may be ordered (e.g., letter grades) or unordered (e.g., handedness), but they should always be mutually exclusive—that is, each observation should fall into one and only one category. For instance, a person cannot be both left-handed and right-handed simultaneously; if such a case exists, it necessitates a distinct category (e.g., ambidextrous). In practice, qualitative variables serve to classify observations into discrete groups, which is why the terms qualitative variable and categorical variable are often used interchangeably. Accordingly, it should be clear that any variable measured on a nominal or ordinal scale would also be considered qualitative in nature.

A **quantitative variable** tells you something about how much there is of something—it deals in actual numbers that represent real amounts or magnitudes. Unlike qualitative variables, which categorize or rank observations, quantitative variables support meaningful arithmetic operations like addition and subtraction. They are always measured on an interval or ratio scale and can be either discrete (e.g., number of siblings) or continuous (e.g., temperature, height).

A **discrete variable** is one that takes on a finite number of values (e.g., the number of cells in a slice of brain tissue) or a countably infinite set of values (e.g., the set of odd numbers). In other words, discrete variables are countable using the natural numbers: 1, 2, 3, and so on. Calling a variable "discrete" means there are no possible values in between—the values are separate, distinct steps. For example, it does not make sense to count 12.5 brain cells. Even if, under a microscope, a cell appears partially missing, you must decide whether to count it as a whole cell or not at all.

By contrast, if it makes sense to talk about values between any two measurements, the variable is no longer discrete—it is continuous. A **continuous variable** can, at least in theory, take on any numeric value within a given range. These variables have uncountably infinite possibilities.

Variable	Scale	Description	Examples
Qualitative (Categorical)	Nominal	Categories with no inherent order.	Handedness, Eye colour, Species
	Ordinal	Values with a meaningful order, but differences between values are indeterminate.	Letter grades, Likert-scale ratings, Percentile ranks
Quantitative (Numerical)	Interval	Numeric values with equal spacing, but no true zero	Temperature in Celsius, Calendar Years
	Ratio	Numeric values with equal spacing and a meaningful zero	Height, Weight, Reaction time

Table 4.3: Summary of variables and corresponding measurement scales

For example, temperature is a continuous variable. Between 20°C and 21°C, there are infinitely many possible values—like 20.1°C, 20.01°C, or 20.0001°C. No matter how small the interval, there is always another possible value in between. Of course, in practice, temperature often appears discrete—especially when measured with a thermometer that rounds to the nearest degree. But this discreteness is an illusion caused by the limitations of our measuring tools. In reality, temperature varies continuously; it is our instruments (and our finite senses) that force us to round or approximate.

Bringing this back to scales of measurement: any variable measured on an interval or ratio scale is, by definition, also a quantitative variable.

To Be Continued...

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`

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.

continuous variable A **quantitative variable** that can take on any numeric value within a given range.

CRAN Comprehensive R Archive Network

data A collection of observations about something.

data frame An object class in R with rows and columns resembling a spreadsheet structure. R Documentation: `?data.frame`

datum The singular form of the word **data**.

delimiter A character within a data file used to delimit (i.e., define the limits of) individual values.

dependent variable The variable that you are trying to explain, predict, or measure the effect on. i.e., It is "dependent" on changes to other variables and is also known as the **response variable** or **outcome variable**. In R formulas, the dependent variable typically appears on the left-hand side of the tilde (e.g., `dependent variable ~ predictor variable`).

descriptive analysis A type of data analysis focused on summarizing and organizing data in a way that reveals its features without making claims beyond the data at hand.

directory An address that *directs* you to a file

discrete variable A **quantitative variable** that consists of countable values.

error bar A graphical representation of the variation surrounding a measure of central tendency. Displayed as lines (also called "whiskers") extending above and below a plotted value. They typically represent statistics such as the standard error or confidence intervals; however, they can, in principle, illustrate any statistic that conveys variation in the data.

factor A class of object in R that has a defined set of possible values called **levels**. Factors are used to represent categorical data, control the order of categories, and influence how data is processed or displayed in visualizations and models. 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 and followed by one to four characters typically. 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

inferential analysis A type of data analysis that uses reasonable assumptions to make generalizations, predictions, or decisions that extend beyond what a descriptive analysis of the data

alone can show.

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.

interval scale A measurement scale that provides both an order of values and equal spacing between them, but lacks a true zero point.

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.

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."

nominal scale A measurement scale used for labelling or categorizing without implying any order or quantity.

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.

ordinal scale A scale that categorizes and arranges items in a meaningful order. The intervals between items are not necessarily equal.

outcome variable The variable that you are trying to explain, predict, or measure the effect on.

i.e., It is the "outcome" that results to changes in other variables and is also known as the **response variable** or **dependent variable**. In R formulas, the outcome variable typically appears on the left-hand side of the tilde (e.g., `outcome ~ predictor`).

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.

qualitative variable A **variable** that represents non-numeric characteristics or categories. These categories can be either ordered or unordered, but they must be mutually exclusive.

quantitative variable A **variable** that represents a numerical magnitude and supports meaningful arithmetic operations. It is measured on an interval or ratio scale and may be either a **discrete variable** or a **continuous variable**.

ratio scale A measurement scale that includes all the properties of an interval scale—ordered values with equal intervals—*plus* an absolute zero point, which represents a true absence of the quantity being measured.

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`

response variable The variable that you are trying to explain, predict, or measure the effect on. i.e., It "responds" to changes in other variables and is also known as the **dependent variable** or **outcome variable**. In R formulas, the response variable typically appears on the left-hand side of the tilde (e.g., `response ~ predictor`).

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., 666130000000000 can be written as 666.13×10^{12} or `666.13e+12`

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.

significant figures The digits in a numerical value that carry meaning about its precision. This includes all nonzero digits, any zeros between nonzero digits, and trailing zeros in a decimal number. Leading zeros are not considered significant. Significant figures are also commonly referred to as significant digits.

subdirectory A directory nested within another directory.

syntactic name An object name that begins with a letter or a dot (not followed by a number) and may include letters, numbers, dots, or underscores. 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 which consists of a single **response variable** and sometimes one or more predictor variables.

variable A single characteristic or property of that can differ from one observation to another. Often represented by the columns of a data set.

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 format A way of structuring data such that variables are spread 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:

```
sum(x = 1:5)
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.

```
sum(x <- 1:5)
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...

```
rational_pi = function() {
  rat_pi <- round(pi)
  return(rat_pi)
}
```

When we run the function, it straightforwardly spits out a 3

```
rational_pi()
```

```
[1] 3
```

But when we run object `rat_pi` we get an error message saying the object cannot be found:

```
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:

```
rational_pi = function() {
  rat_pi <<- round(pi)
  return(rat_pi)
}

rational_pi()
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:

```
rational_pi = function() {
  rat_pi <- 10
  rat_pi <<- round(pi)
  return(rat_pi)
}

rational_pi() #Notice the function produces 10
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

Figure B.1: Sequential Palettes

Figure B.2: Diverging Palettes

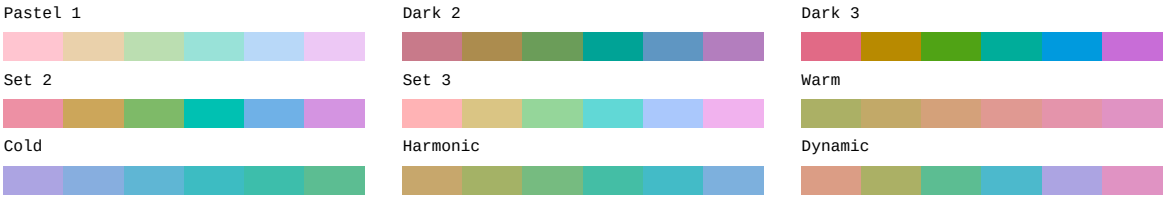


Figure B.3: Qualitative Palettes

References

Baker, B. O., Hardyck, C. D., & Petrinovich, L. F. (1966). Weak measurements vs. strong statistics: An empirical critique of s. s. stevens' proscriptions on statistics. *Educational and Psychological Measurement*, *26*(2), 291–309. https://doi.org/10.1177/001316446602600204

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

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.

Clagett, M. (1989). *Ancient Egyptian science : A source book.* American Philosophical Society.

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. (2024). *What is free software?* Retrieved August 26, 2024, from https://www.gnu.org/philosophy/free-sw.html

Gaito, J. (1980). Measurement scales and statistics: Resurgence of an old misconception. *Psychological Bulletin*, *87*(3), 564–567. https://doi.org/10.1037/0033-2909.87.3.564

Hunt, P. (2024). *Source code pro* [version 2.042R-u_1.062R-i]. https://github.com/adobe-fonts/source-code-pro

Labovitz, S. (1967). Some observations on measurement and statistics. *Social Forces*, *46*(2), 151–160. https://doi.org/10.2307/2574595

Luce, R. D., & Suppes, P. (2001). Representational measurement theory [Volume 4: Methodology in Experimental Psychology]. In J. T. Wixted & H. Pashler (Eds.), *Stevens' handbook of experimental psychology* (3rd ed., pp. 1–41, Vol. 4). John Wiley & Sons.

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

Roberts, F. S. (1985). Applications of the theory of meaningfulness to psychology. *Journal of Mathematical Psychology*, *29*(3), 311–332. https://doi.org/10.1016/0022-2496(85)90011-2

Robinson, R. E. (1965). Measurement and statistics: Towards a clarification of the theory of "permissible statistics". *Philosophy of Science*, *32*(3/4), 229–243. http://www.jstor.org/stable/186516

Senders, V. L. (1958). *Measurement and statistics: A basic text emphasizing behavioral science applications*. Oxford University Press.

Stevens, S. S. (1946). On the theory of scales of measurement. *Science*, *103*(2684), 677–680. https://doi.org/10.1126/science.103.2684.677

Stevens, S. S. (1951). Mathematics, measurement, and psychophysics. In S. S. Stevens (Ed.), *Handbook of experimental psychology* (pp. 1–49). John Wiley & Sons.

Stevens, S. S. (1968). Measurement, statistics, and the schemapiric view. *Science*, *161*(3844), 849–856. https://doi.org/10.1126/science.161.3844.849

Thomson, A., & Randall-MacIver, D. (1905). *Ancient races of the Thebaid: Being an anthropometrical study of the inhabitants of upper egypt from the earliest prehistoric times to the mohammedan conquest based upon the examination of over 1,500 crania*. Clarendon Press.

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

Wilkinson, L. (2005). *The grammar of graphics*. Springer Science.

Zand Scholten, A., & Borsboom, D. (2009). A reanalysis of Lord's statistical treatment of football numbers. *Journal of Mathematical Psychology*, *53*(2), 69–75. https://doi.org/10.1016/j.jmp.2009.01.002

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/