Understanding Uncertainty

An Introduction to Probability and Statistics

Dylan Spicker

2024-09-03

Table of contents

Preface

What are these notes?

These notes were originally written to supplement the Winter 2024 offering of STAT 1793 at the University of New Brunswick on the Saint John campus. They have since grown into a more comprehensive resource, attempting to direct a path through introductory probability and statistics material, both for students in a Statistics program, as well those who require probability or statistics for their major. Owing to the wide range of possible audiences, not all sections of the notes will be relevant to all students. I have tried to clearly indicate requirements throughout the text.

Further, these notes remain a work-in-progress. There is additional material being added, old material being revised or clarified, and some restructuring going on. If at any point you have comments, questions, suggestions, or corrections to these notes, please do reach out!

In the writing of these notes, I am indebted to the numerous introductory texts that I have previously read, and the multitude of Statistics instructors that I have had the pleasure of studying with. As a result, I am certain that I have drawn inspiration from many sources for examples, exercises, or techniques of description. Where parallels with other work are explicit and intentional, those works are mentioned alongside the relevant passages. Where parallels are the result of internalized knowledge coming forth as inspiration, I am not able to draw direct connections. However, I will advise that the following books are all excellent resources, and each has played a part in my current approach to teaching statistics, either directly or indirectly. For introductory topics consider Engelhardt and Bain (1991); Navidi (2010); Mendenhall et al. (2013) among countless others. For problems, consider Bassett et al. (2000); Grimmett and Stirzaker (2001); Schiller, Alu Srinivasan, and Spiegel (2012).

One final note is that our coverage of concepts throughout reflects my personal biases and proclivities towards what is important in a sequence on probability and statistics. For instance, compared to others I may have a larger emphasis on probability. This is reflected both in the ordering of material (we start at Probability not at Statistics), and in the weight that each topic is given. It is my belief that you need to be able to fluently speak the language of probability in order to study statistics, and the notes begin with a strong focus on this. If these notes are being used in a two course sequence, it is my belief that the first term should consider almost exclusively probability, leaving statistical results for a second course.

Using These Notes

These notes are best viewed online. You can access them from anywhere at https://dylanspicker.github.io/Understanding-Uncertainty/index.html. Viewing online will allow you to take advantage of interactivity (such as through code blocks and solutions hiding), the search feature (visible in the left hand margin), as well as better overall formatting. There is a version of these notes in PDF format available. To download them, press the PDF icon button () next to the note title in the left hand bar. The PDF notes contain all the same content, and will be updated alongside the web notes. The formatting of the PDF document may be less visually appealing compared to the web notes. Though an effort will be made to ensure that everything remains legible, and well-formatted, there may be formatting differences. Please reach out if any components of the PDF are illegible.

Studying Statistics (and other quantitative subjects)

To successfully use these notes, I would recommend a several step procedure:

1. When reading the material, or watching lecture videos, focus on developing your understanding. Statistics is not a subject that is predominantly memory-driven. It is not about remembering the facts, formula, or question types, it is about understanding how these all come together.
   - If you get lost along the way, ask questions.
2. When you get to examples in these notes, do not look at the solutions directly. Instead, you should try to solve the examples yourself. If you get stuck, look at the solution, but only enough to see remove the current confusion, and then continue.
   - The process of learning statistics, and most quantitative subjects for that matter, relies on *doing* the work in these subjects.
   - Looking at solutions will often undermine your own understanding. It is a much different process to look at a solution and understand why it is correct, than it is to determine a solution for yourself.
3. Complete the practice questions.
   - Note: struggling with problems is a key component of learning in any mathematical discipline. It is only by struggling to understand how the pieces fit together that you will develop a proper understanding for yourself.
4. Throughout your study, make sure you are continually asking yourself how this fits together, how this confirms or contradicts what you already know, and how this may be used in your life beyond these pages.

- Drawing personal connections to the material is a very effective way of ensuring that it sticks better.

Throughout the notes there will be references to computer code in a programming language known as R. R is the most common language used for statistics by practicing statisticians, and it is a great utility for many of the types of problems we will be considering throughout these notes. It is possible to pass through the notes without writing R code, however, familiarity with it will make the process of statistics far nicer to engage with. If you are using the web notes this code can all be run directly in the browser. If you are using the PDF notes, the code and output is still provided. In either case, I would suggest getting R running on your own device, if possible. It is free to download and install. I would also suggest installing R Studio, which is a program specifically designed for writing and running R code with a lot of nice features.

Some Mathematical Background

These notes require high school mathematics to complete. Some sections of the notes require calculus (limits, derivatives, and integrals). It is possible to pass through the core of these notes skipping any sections that specifically require calculus: these are noted in text. For all sections, the assumption is that you will be comfortable manipulating mathematical expressions, solving basic equations, using a calculator, and so forth. The following are a handful of topics which will come up throughout the notes that are assumed to be known, presented here in brief as a quick reference.

Logarithms and Exponent Rules

Recall that if $a^x = b$, then we can solve for x through the use of logarithms. Specifically, $x = \log_a(b)$, where we read this as "the logarithm with base a of b." The expression means that "x is the exponent we put on a to get b." In these notes[1] we use the notation $\log(x)$ to represent the **natural logarithm**. The natural logarithm is $\log_e(x)$, where $e \approx 2.71828$ is Euler's number. The **exponential function** is given by $e^x = \exp(x)$. Which notation is used depends on the specific setting, but both are equivalent.

[1]And in most math books and courses.

Recall further the following exponent and log rules:

$$\begin{aligned}
a^b \times a^c &= a^{b+c} \\
a^b \times c^b &= (ac)^b \\
a^{-b} &= \frac{1}{a^b} \\
\left(a^b\right)^c &= a^{bc} \\
a^0 &= 1 \\
\log(ab) &= \log(a) + \log(b) \\
\log(a^b) &= b\log(a) \\
\log(1) &= 0 \\
\log(e) &= 1.
\end{aligned}$$

Summation and Product Notation

If we wish to add up a series of numbers, $x_1, x_2, \dots, x_n$ then we can use summation notation to record this. Specifically,

$$x_1 + x_2 + \cdots + x_n = \sum_{i=1}^{n} x_i.$$

Here i is an indexing variable just used to keep track of which number we are currently working on. Note that the following quantities will hold

$$\begin{aligned}
\sum_{i=1}^{n}(x_i + y_i) &= \sum_{i=1}^{n} x_i + \sum_{i=1}^{n} y_i \\
\sum_{i=1}^{n}(x_i - y_i) &= \sum_{i=1}^{n} x_i - \sum_{i=1}^{n} y_i \\
\sum_{i=1}^{n} cx_i &= c\sum_{i=1}^{n} x_i.
\end{aligned}$$

We can also use a similar shorthand to discuss repeated products. That is, if we want to multiply $x_1, x_2, \dots, x_n$, then we can write

$$x_1 \times x_2 \times \cdots \times x_n = \prod_{i=1}^{n} x_i.$$

Again, i is used as an indexing variable to keep track of what value is currently being multiplied.

The following results hold

$$\prod_{i=1}^{n} x = x^n$$
$$\prod_{i=1}^{n} b^{x_i} = b^{\sum_{i=1}^{n} x_i}$$
$$\prod_{i=1}^{n} cx_i = c^n \prod_{i=1}^{n} x_i.$$

Other Important Points

The following are some additional points that are useful to know, but which do not necessarily fit together nicely.

- For any two values x and y, we get $(x+y)^2 = x^2 + 2xy + y^2$.
- The integers are the set of all whole numbers, $\{0, \pm 1, \pm 2, \dots\}$. We will often denote the set of integers using $\mathbb{Z}$.
- The natural numbers are the set of all counting numbers, either $\{0, 1, 2, 3, \dots\}$ or else $\{1, 2, 3, 4, \cdots\}$ depending on the source. We often denote the set using $\mathbb{N}$.
- The real numbers are the set of all numbers that you think of when thinking of numbers. They include the integers, and any fractions, and anything that cannot be written as a fraction (like π and e) as well. We denote the real numbers as $\mathbb{R}$.
- Intervals of real numbers are denoted as (a, b) or $[a, b]$ (or a mixture, giving $[a, b)$ or $(a, b]$). A round bracket means that the endpoint is **not** included in the set, where a square bracket means that it is. That is, a value of x is in (a, b) if $a < x < b$, where it is in $[a, b]$ if $a \leq x \leq b$.
- We write "x is in the interval (a, b)" mathematically using the $\in$ symbol. Specifically, $x \in (a, b)$.
- Infinity (∞) is not a number. Rather, it is a statement regarding the lack of a bound. When we say that something is infinity, or infinite, we simply mean that it is larger than any of the natural numbers. If you were to be given any number, then infinity will be larger than that.
- **Limits:** note that, while limits are from calculus, they are useful for formally understanding a few concepts in class. The limit of a function, $f(x)$, is simply the value that the function approaches as x approaches a specific point. So, for instance, $\lim_{x \to a} f(x)$ is the value that $f(x)$ approaches as x approaches a. If $f(a)$ exists then it is simply $f(a)$. However, $f(a)$ may not exist, at which point, we look at where the function is tending to.
- **Infinite Limits:** Of specific importance to us are infinite limits. Specifically,

$$\lim_{x \to \infty} f(x)$$

captures the beahviour of the function $f(x)$ as x gets larger and larger. In many cases, $f(x)$ will approach some specific value, which we assign to be the limiting value of the function.

Additional Resources

The following are helpful resources which may be used to supplement the material in the class. If you have any suggested resources, please feel free to send them to me, and I will include them here.

1. Open Intro Statistics.
2. Probability Course.
3. Introduction to R.
4. Learning Statistics with R.
5. Mathigon.

References

Part I

Part 1: Probability

1 Introduction to Probability

1.1 What is Probability?

At its core, statistics is the study of uncertainty. Uncertainty permeates the world around us, and in order to make sense of the world, we need to make sense of uncertainty. The language of uncertainty is **probability**. Probability is a concept which we all likely have some intuitive sense of. If there was a 90% probability of rain today, you likely considered grabbing an umbrella. You are not likely to wager your life savings on a game that has only a 1% probability of paying out. We have a sense that probability provides a measure of *likelihood*. Defining probability mathematically is a non-trivial task, and there have been many attempts to formalize it throughout history. While we will spend a good deal of time formalizing notions of probability, we first pause to emphasize the familiarity with probability that you are likely starting with.

Suppose that two friends, Charles and Sadie, meet for coffee once a week. During their meetings they have wide-ranging, deep, philosophical conversations spanning across many important topics.[1] Beyond making progress on some of the most pressing issues of our time, Charles and Sadie each adore probability. As a result, at the end of each of their meetings, they play a game to decide who will pay. The game proceeds by having them flip a coin three times. If two or more heads come up Charles pays, and otherwise Sadie pays.[2]

We can think about the strategy that they are using here and *feel* that this is going to be "fair". With two or more heads, Charles pays. With two or more tails, Sadie pays. There always has to be either two or more heads *or* two or more tails, and each is equally likely to come up[3]. The outcome of their game is uncertain before it begins, but we know that in the long run neither of the friends is going to be disadvantaged relative to the other. We can say that the probability that either of them pays is equal. It's 50-50. Everything is balanced.

[1]For instance they ask "do we all really see green as the same colour?" or "why is it that 'q' comes as early in the alphabet as it does? it deserves to be with 'UVWXYZ'?"

[2]This game, and the ensuing development relating to this game stems from a historical story at the dawn of our modern understandings of probability and chance. Specifically, questions related to this style of game were addressed by Blaise Pascal and Pierre de Fermat in letters they wrote. Their story, a fascinating read, is detailed by Devlin (2010).

[3]There has been some recent literature, see Bartoš et al. (2023), which may suggest that a coin is ever so slightly more likely to land on the same side it started on, perhaps undermining this assertion.

Now imagine that one day, in the middle of their game, Charles gets a very important phone call[4] and leaves abruptly after the first coin has been tossed. The first coin toss showed heads. Sadie, recognizing the gravity of the phone call, pays for the both of them, while realizing that Charles was well on the way to having to pay.

Example 1.1 (Basic Probability Enumeration)**.** Suppose that Sadie pays for coffee if there are two or more tails in three tosses of a coin, and Charles pays otherwise. If Charles leaves after the first coin is tossed, showing heads, what is the probability that Sadie would have had to pay?

Solution

Sadie figures that any coin toss is equally likely to show heads or tails. Because the first coin showed heads, then there are four possible sequences that could have shown up:

1. H, H, H;
2. H, H, T;
3. H, T, H;
4. H, T, T.

In three of these situations [$H, H, H,\ H, H, T$, and H, T, H] there are two heads and so Charles would have to pay. In one of them there are two tails, and so Sadie would have to pay. As a result, Charles would have to pay in 3 of the 4 (with probability 0.75) and Sadie in 1 of the 4 (with probability 0.25).

Looking at Example **??**, we can see that Sadie should likely have not paid. Only one out of every four times would Sadie have had to pay, given the first coin being heads. However, we can not be certain that, had all three tosses been observed, Sadie would *not* have paid. It is possible that we would have observed two tails, making Sadie responsible for the bill. This possibility happens one time out of four, which is more likely than the probability of rolling a four on a six-sided die[5]. Fours are rolled on six-sided dice quite frequently[6], and so it is not all together unreasonable for Sadie to have paid.

This seemingly simple concept is the core of probability. Probability serves as a method for quantifying uncertainty. It allows us to make numeric statements regarding the set of outcomes that we can observe, by quantifying the frequency with which we expect to observe those outcomes. Probability does not *remove* the uncertainty. We still need to flip the coin or roll the die to know what will happen. All probability gives us is a set of tools to quantify

[4]Someone has just pointed out the irony in the fact that there is no synonym for synonym. Technically, there are the words *metonym* and *poecilonym* and *polyonym*, but these are rarely used and Charles would wager that there is a very high probability you have never seen them.

[5]This event happens one time out of every six.

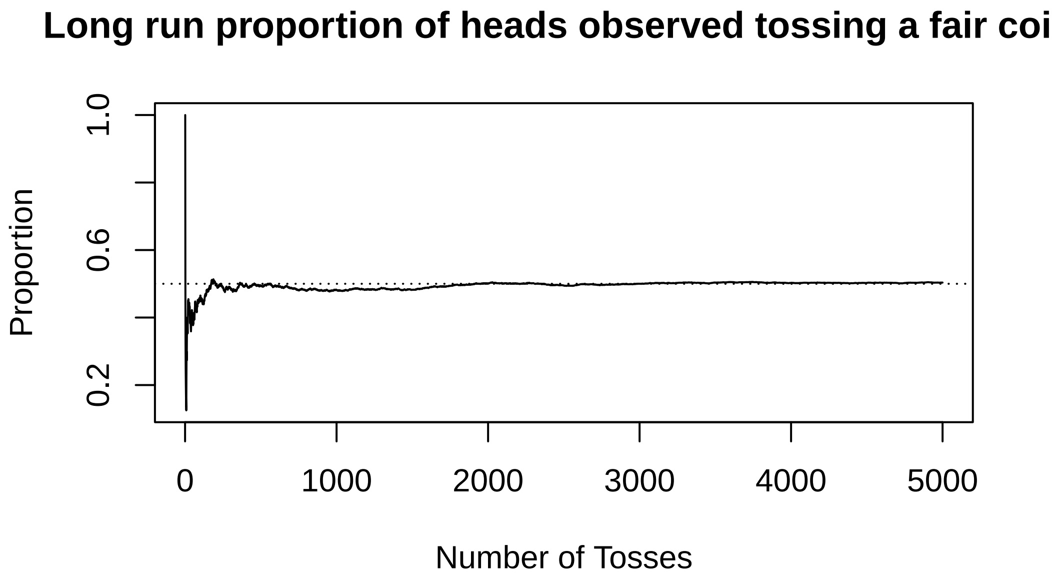
[6]Again, about one out of every six rolls

this uncertainty. These tools are critical for decision-making in the face of the ever-present uncertainty around us.

1.2 How to Interpret Probabilities (like a Frequentist)

We indicated that, intuitively, probability is a measure of the frequency with which a particular outcome occurs. This intuition can be codified exactly with the **Frequentist interpretation** of probability. According to the Frequentist interpretation (or frequentism, as it is often called), probabilities correspond to the proportion of time any event of interest[7] actually occurs in the long run. For a Frequentist, you imagine yourself performing the experiment you are running over, and over, and over, and over, and over again. Each time you answer "did the event happen?" and you count up those occurrences. As you do this more and more and more, wherever that proportion lands corresponds to the probability.

Figure 1.1: This plot simulates the repeated tossing of a coin. The x-axis represents the number of coins being tossed, and the y-axis plots the proportion of times that heads has shown up cumulatively over the tosses thus far. We can see in the long run that this proportion tends towards 0.5.



To formalize this mathematically, we first define several important terms.

Definition 1.1 (Experiment)**.** Any action to be performed whose outcome is not (or cannot) be known with certainty, before it is performed.

[7]Be that flipping heads, rolling a four, or observing rain on a given day.

Definition 1.2 (Event [Informally]). A specified result that may or may not occur when an experiment is performed.

Suppose that an experiment can be performed as many times as one likes, limited only by your boredom. If you take k_N to represent the number of times that the event of interest occurs when you perform the experiment N times, then a Frequentist would define the probability of the event as

$$\text{probability} = \lim_{N\to\infty} \frac{k_N}{N}.$$

If you have never taken a calculus course, and thus are unfamiliar with the concept of limits, that is not a barrier to understanding this statement. When you see a statement of the form

$$\lim_{x\to\infty} f(x),$$

simply think "what is happening to the function $f(x)$ as x grows and grows (off to ∞)?"

In practice this means that, in order to interpret probabilities, we think about repeating an experiment many, many times over. As we do that, we observe the proportion of time that any particular outcome occurs, and take that to be the defining relation for probabilities. The reason that we say the probability of flipping heads is 0.5 is because if we were to sit around and flip a coin[8] over, and over, and over again, then in the long-run we would observe a head[9] in 0.5 of cases.

Example 1.2 (Probability Interpretation). How do we interpret the statement "the probability that Sadie would have had to pay, given a head on the first toss, is 0.25"? Recall that in the game, they toss three coins, and Sadie pays if two of them show tails.

Solution

This statement means that, if Sadie were to repeatedly be in the situation where one head has shown and there are two coins left to toss, then in 0.25 of these situations (in the limit, as this is repeated an infinite number of times) will end up showing two tails.

Many situations in the real world cannot be run over and over again. Think about, for instance, the probability that a particular candidate wins in a particular election. There is uncertainty there, of course, but the election can only be run once. What then? There are several ways through these types of events.

First, we can rely on the **power of imagination**. There is nothing stopping us from envisioning the hypothetical possibility of running the election over, and over, and over, and over again. If we step outside of reality for a moment, we can ask "if we could play the day of the

[8]Our experiment.
[9]Our event.

election many, many, many times, what proportion of those days would end with the candidate being elected?" If we say that the candidate has a 75% chance of being elected, then we mean that in 0.75 of those imagined worlds, the candidate wins. It is crucial to stress that in our imagination here, we need to be thinking about the **exact same day** over and over again. We cannot imagine a different path leading to the election, different speeches being given in advance, or different opposition candidates. If we start from the same place, and play it out many times over, what happens in each of those worlds?

This repeated imagining is not for everyone. As a result, alternative proposals to the interpretation of probability have been made, with the **Bayesian interpretation** (or subjectivist interpretation) being particularly prominent. To Bayesians, probability is a measure of subjective belief. To say that there is a 50% chance of a coin coming up is a statement about one's knowledge of the world. The Bayesian view, built around subjective confidence in the state of the world, can be formalized mathematically as well. A Bayesian considers the *prior evidence* that they have about the world[10] and combine this with current observations in order to update their subject beliefs, balancing these two sources of information.

Remark (Bayesian Probabilities and Belief Updating). Suppose that a Bayesian is flipping a coin. Before any flips have been made the Bayesian understandably believes that the coin will come up heads 50% of the time. However, when the coin starts to be flipped, the observations are a string of tails in a row.

After having flipped the coin five times, the individual has observed five tails. Of course, it is totally possible to flip a fair coin five times and see five tails, but there is a level of skepticism growing.

After 10 flips, the Bayesian has still not seen a head. At this point, the subjective belief is that there is likely something unfair about this coin. Even though the experiment started with a baseline assumption that the coin was fair the Bayesian no longer believes that the next flip will be a head.

As this goes on, you can imagine the Bayesian continuing to update their view of the world. To them, the probability is an evolving concept, capturing what was believed and what has been observed.

For the election example, the Bayesian interpretation is somewhat easier to think through. To say that a candidate has a 75% chance of winning the election means that "based on everything that has been observed, and any prior beliefs about the viability of the candidates, the subjective likelihood that the candidate wins the election is 0.75". If we disagree about prior beliefs, or have experienced different pieces of information, then we may disagree on the probability. That is okay.

In these notes, we focus on the Frequentist interpretation. This choice is not a strong stance on the relative merits of the two viewpoints. There is some research to suggest that Frequentist

[10]Any relevant evidence that has previously been collected.

interpretations are fairly well understood by the general public[11]. However, it is important to know and recognize that there is a world beyond Frequentist Probability and Statistics, one which is very powerful once it is unlocked.[12]

If probability measures the long term proportion of time that a particular event occurs, how can we go about computing probabilities? Do we require performing an experiment over and over again? Fortunately, the answer is no. The tools of probability we will cover allow us to make concrete statements about the probabilities of events without the need for repeated experimental runs. However, before completely dismissing the idea of repeatedly running an experiment, it is worth considering a tool we have at our disposal that renders this more possible than it has ever been: computers.

1.3 R Programming for Probability and Statistics

Throughout these notes we will make use of a programming language for statistical computing, called R. Classically, introductory statistics courses involved heavy computation of particular quantities by hand. The use of a programming language (like R) frees us from the tedium of these calculations, allowing for a deeper focus on understanding, explanation, decision-making, and complex problem-solving. This is **not** to say we will *never* do problems by hand[13], however, these notes emphasize the use of statistical computing. The sections on R programming throughout the notes are self-contained, and separate from the main material. Studying these sections will give you familiarity and comfort with reading, modifying, and writing R scripts for statistical programming.

It may be useful to have R open alongside the notes to ensure that you can get the same results that are printed throughout. In this section we will cover the very basics of using R, and reading R code. If you are interested, there are plenty of resources to becoming a more proficient R programmer.

1.3.1 Basic Introduction to R Programming

When programming, the basic idea is that we are going to write instructions in a **script** which we will tell our computer to execute. These instructions are[14] performed one-by-one, from the top to the bottom of the script. We can have instructions which operate on their own, or

[11]See, for instance Gigerenzer et al. (2005), where they study how people interpret the statement "there is a 30% chance of rain tomorrow." Interestingly, most people can convert this into a frequency statement (3 out of 10, say), even if the specific meaning is sometimes lost. They conclude that there are other issues in attempting to understanding this statement, issues which we will address later on.

[12]More on this in later books, if you so desire!

[13]To the contrary, some amount of by-hand problem-solving helps to solidify these concepts.

[14]Typically. There are some exceptions to this, but if this is your first time programming, you need not worry about that!

which interact with previous (or future) instructions to add to the complexity. The trick with programming then is to determine which actions you need the computer to perform, in which order, to accomplish the task that you are setting out to do.

To begin, we may consider an R program that uses the programming language as a basic calculator.

```
## [1] 33554420
```

Here, we ask the computer to perform arithmetic operations. We use `+` for addition, `-` for subtraction, `*` for multiplication, `/` for division, and `^` for exponentiation. With the use of parentheses, any expressions relating to these basic operations can be performed. Note that here the result is output after it is computed. Try modifying the exact expressions being computed, allowing you to feel comfortable with these types of mathematical operations.

```
## [1] 33554420
## [1] 40
```

Here, we have two lines of math running with arithmetic operations. Each is output when it is computed. These two results have no ability to interact with one another, and if we were to add more and more lines beneath, the same would continue to happen.

If we want different commands to be able to interact with one another, we need a method for storing the results. To do so, we can define **variables**. In R, to define a variable, we use the syntax `variable_name <- variable_value`. We can choose *almost* anything that we want for the variable name[15] and the variable value can also be of many types.[16] The arrow between the two is the **assignment operator**. It tells R to assign the `variable_value` to be accessible from the `variable_name`.

```
## [1] 5
## [1] 8
```

In this code we assign the variable `my_5` to contain the value 5, and the variable `my_8` to contain the value 8. We can output these as expressions themselves by typing the variable name. Directly outputting these variables is not of particular interest, however, we can use the variables in later statements by including the variable name in them.

```
## [1] 40
```

[15]The variable name must start with a letter and can be a combination of letters, numbers, periods, and underscores.
[16]more on this later

Here, instead of outputting the variables, we multiply them together. We could have used `5 * 8` in this case for the same result, however, we are afforded a lot more flexibility with this approach. Much of this flexibility comes from our capacity to *change* the values of variables over time. Consider the following script, and try to understand why the output is the way that it is.

```
## [1] 40
## [1] 80
```

At the top point in the script, before the first `my_5*my_8` call, the variable `my_5` has the value `5`. However, after this is called, the value is updated to be `10`. Then, when we call `my_5*my_8` again, this is now simplified to `10 * 8`, giving the result. Perhaps more importantly, we can take the value of an expression and assign that to a variable itself.

```
## [1] 40
```

Here, we define another variable. This time, `result` now contains the result of multiplying our previous two variables. Thus, when we output it, we get the same value. Take a moment to read through the following script, and try to understand what will happen at the end.

The result is 1600. Why? We can read through this step-by-step in order to understand this. First we set our variables to have the values of `5` and `8`. Then, `result` is made to be the product of our two variables, which in this case is `40`. After that, we set the value of `my_5` to be the same as the value of `result`, which gives `my_5` equal to `40`. At this point, `result` equals `40`, `my_5` equals `40`, and `my_8` is equal to `8`. The next line updates `my_8` to be the same as the value of `my_5`, which we just clarified was `40`. As a result, all the variables we have defined now take on the value of `40`. The final line before output, `result <- my_5 * my_8` updates the value of `result` to be the product of the two variables again, this time giving `40 * 40` which gives 1600.

1.3.2 Function Calls in R

Up until this point we have only used numerical operations and variable assignment. While this allows R to serve as a very powerful calculator, we often want computers to do much more than arithmetic. As a result, we need to explore **functions** in R. A function is a piece of code which takes in various arguments and outputs some value (or values). Most of the way that we will use R in these notes is through the use of function calls.[17] This is exactly analogous to a mathematical function: it is some rule which maps input to output. In fact, some of the most basic functions in R are functions which relate to mathematical functions.

[17]Note: when you begin to write programs for yourself, a lot of your time will be spent writing custom functions. If this is of interest to you, I suggest looking into programming more! For now, we will not need to define our own functions.

```
## [1] 22026.47
## [1] 3.162278
## [1] 2.302585
```

1. Computes the exponential function applied to `x`, that is $e^x = e^{10}$.
2. Computes the square root of `x`, that is $\sqrt{x} = \sqrt{10}$.
3. Computes the natural logarithm of `x`, that is $\log(x) = \log(10)$.

The basic format of a function call will always be `function_name(param1, param2, ...)`. Each of these functions required only a single parameter, however, there are some functions which take more than one parameter. If we have decimal numbers, for instance, we may wish to round them. To do so, we can use the `round` function in R, which takes two parameters: the number we wish to round, and how many digits we wish to keep.

```
## [1] 3.142
```

There are a few things to note about this sequence of function calls. First, notice that we assign the output of a function to a new variable. This behaves exactly as we saw above with numeric calculations. Next, consider that the output of the function[18] has a value of 3.142. That is: we rounded the value to 3 decimal points, exactly as would be expected. Finally, notice that the second parameter passed to the function call is **named**. That is, instead of calling `round(unrounded_value, 3)`, we have `round(unrounded_value, digits = 3)`. If you had made the first call this would have worked perfectly.[19] However, R also provides the ability to pass in parameter names alongside the parameter values with the syntax `function_name(param_name = param_value, ...)`.

The benefit to doing this is two-fold. First, it is easier to read what is happening, especially for function calls that you have never seen before. Second, it removes the need to have the parameters ordered correctly. It is best practice to **always** include parameter names where you can.

Now, you may be wondering "how do you know what the parameter names should be?" The names are built-in to the different functions that you are working with, and with experience you will become quite familiar with them. However, at any point you can also run the command `?function_name`, replacing `function_name` with the name of a function you are interested in, to open documentation about that function. There you will see not only the names of the different parameters, but useful information regarding what the function does, and examples of how to call it.

When we run this code, we are given *a lot* of information. We can see the function name, the details, how it's called, and so forth. In the **Arguments** section we get a list of all the arguments we can pass to the function, along with a description of them. In this case we see

[18]Which we have stored in the variable `rounded_value`
[19]Try it out to convince yourself!

that `round` can take a parameter called `x` for the number to be rounded, and `digits` (which we have previously seen).

```
## [1] 3.142
```

This code produces the exact same output, where now both parameters are named. Specifically, we could read this function call as "round the value of `x` to have `digits` decimal points." If you had instead written `round(3, unrounded_value)`, you would get the value `3`, since now we are rounding `3` to `3.141592` decimal points.

1.3.3 Moving Beyond Numeric Data

So far, everything that we have looked at has been numeric data. We have seen integers, and decimals. You can have negative results, say by taking `my_var <- -5`. And while numbers are frequently useful, we will require further types of data to write useful computer programs. In these notes, we will focus on three additional data types: textual data which (referred to as **strings**), true and false binary data (referred to as **logicals** in R), and lists of the same data type (referred to as **vectors**). They will behave in much the same way as numeric data, with different functions and techniques which can be applied to them.

To define a string of text, we encapsulate the text that we are interested in within quotation marks (either single `'` or double `"` quotation marks will work).

```
## [1] "This is a string."
```

Two commonly used functions which rely on strings are `paste` and `print`. Each will take in strings as input, and they do not need to be named. The function `paste` can take in as many strings as you would like. It will "paste" together all the strings provided, creating a longer string out of these. The function `print` will display the string that is passed as output. Until now, we have been running these programs in a way where all calls are displayed as output: this will not always be the case, and so print can come in handy there.

```
## [1] "Hello! Welcome to R programming, Dylan !"
```

Note that `paste` has several additional options which can be investigated in the documentation. This is only the simplest use case for it. In general, strings are particularly helpful when we wish to have output from the computer that will be human-readable. Where strings are largely for humans, logicals are largely for computers.

Much of what computer programming entails is checking whether certain conditions hold, and then taking different actions depending on what is found. In order to do this, the computer needs a way to represent true and false statements. In R, these are codified with the values `TRUE` and `FALSE`. Note, the capital letters here make a difference. You cannot use `true` or `True` or any other combination thereof. More important than being able to specify the values `TRUE` and `FALSE` directly is the ability to detect whether certain statements are `TRUE` or `FALSE`. For this we require comparison operators.

If we think of mathematical comparisons we can state whether two things are equal, or not equal, and whether one thing is less than (or equal to) or greater than (or equal to) another. We can run all of these same checks in R.

- To check whether two quantities are equal you use `quantity_1 == quantity_2`. This statement will be `TRUE` if `quantity_1` and `quantity_2` are exactly the same, and will be `FALSE` otherwise.
- To check whether two quantities are not equal, you can use `quantity_1 != quantity_2`. This statement will be `TRUE` if the quantities differ from one another.
- To check whether one quantity is larger than another, you can use `quantity_1 > quantity_2`. If you want to know whether it is greater than *or* equal to, you can use `quantity_1 >= quantity_2`.
- To check whether one quantity is smaller than another, you can use `quantity_1 < quantity_2`. If you want to know whether it is less than *or* equal to, you can use `quantity_1 <= quantity_2`.

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

There are two key points to note beyond this. First, we will of course not normally compare two constants to one another. We already know that `5==5`, so we would not need to check it. We can, however, plug-in variables, perhaps with unknown values, and have the same types of statements being made. Second, the checks for equality and inequality also work with other data types (like strings).

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

The final checks may be slightly odd. Here we are comparing across different types of data. When we do this R will automatically try to convert from one type to the other. With strings and numbers this is not too challenging. If they can be converted nicely between types, then they are and the values are compared. Otherwise, R will conclude they are not equal by default. For logicals, it is important to note that `TRUE == 1` and `FALSE == 0`. We will often use these values interchangeably.

The final data type that we will consider are vectors. Vectors store multiple values, of the same type, in a single object. Thus, we may have a vector of numeric data, or a vector of strings, or a vector of logicals. The vectors will always contain the same type throughout, but they are stored in a single object (and as such, for instance, can be stored in a single variable). To define a vector we call `c(...)`, where the `...` contains the set of objects we want to store in the vector. The `c` stands for **c**oncatenate, as we are *concatenating* together the set of items into a single container.

```
## [1] "vector"  "of"      "strings"
## [1] 3 1 4 1 5
## [1]  TRUE  TRUE FALSE  TRUE
## [1] FALSE  TRUE FALSE
```

We see that each of these vectors holds one type of object. Vectors can be of arbitrary and different lengths. It is also possible to combine multiple vectors *of the same type* into one, by using the `c` function again.

```
##  [1] 3 1 4 1 5 9 2 6 5 3
```

Here we combine different numeric vectors together. We also show, when forming `combined_v3`, how numeric vectors can have single items added onto them. That is, if you have a single number, it can be treated as a vector with one element in it. This becomes very useful when building up vectors within code.

In addition to combining multiple vectors together, we can also select elements out of a vector. To do this, we use a set of square brackets after the vector's name, with a number within those square brackets specifying the **index** of the vector we are interested in. The index is the element position starting at 1[20] and running to the length of the vector. We can include a vector of indices to select multiple elements at once.

```
## [1] "D"
## [1] "Y"
## [1] "L"
## [1] "A"
## [1] "N"
## [1] "D" "Y" "L" "A" "N"
```

Note that, each element is selectable individually, giving a single item of that type (in this case, strings). If you select multiple of the elements together, it will create a vector of that type (in this case, a string vector). In addition to selecting elements in this way by their indices, you can also update the elements in the same way.

```
## [1] 2 0 2 3
## [1] 2 0 2 4
```

In this example we are changing the last element of the vector. Sometimes we may not know how long the vector actually is, if for instance, it is being built-up as our code runs. If we ever want to check the length of a vector, we can call the function `length` which takes as input only one vector, and outputs the numeric value of its length.

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

1.3.4 Program Control Flow

We have seen different types of data, different ways of manipulating data, functions, and variables so far. In order to bring all of these concepts together into useful programs we need some way to control the flow of our programs. We have seen that, by default, programs execute from the top until the bottom. However, it will often be the case that we want to have certain code running only if certain conditions hold, or that we want to repeat some piece of code many times over. To accomplish these tasks we require **control flow statements**. We will

[20]If you have programmed in the past there is a good chance the language you have learned is "0 indexed" rather than "1 indexed". In R, all vectors start at position 1 and count up, which is not the case in many languages. Be careful of this.

consider only two types of control flow statements now, which will serve well enough to read most of what needs to be read for these notes.

The first type of statement is the **conditional statement**. Conditional statements execute only when certain conditions hold. The simplest conditional statement is an **if** statement. The format to define an if statement is `if (condition){ ... }`, where `condition` is some logical condition to be evaluated. If the condition is `TRUE` then the code contained in { … } is evaluated. If the condition is `FALSE` it is not.

```
## [1] "My number is a positive."
```

In this case, these conditional statement check to see whether the number we entered is larger than zero and whether the number we entered is smaller than zero, respectively. When run, notice that at most one of the statements is executed.[21] In this case, we know that only one (or neither) of these statements can be true. When that is the case it may make sense to make use of `else` clauses in our conditional logic.

```
## [1] "My number is not a positive."
```

Here, if the number is greater than `0`, then we run the first block of code, otherwise we run the second block of code. Thus, whenever a positive number is entered, we see "My number is a positive", and whenever a non-positive number is entered, we see "My number is not a positive." We can extend `else` blocks to be `else if` blocks, where further conditions can be specified.

```
## [1] "My number is a large positive value."
```

In these case we can pass through each conditional statement in order. First, is the number larger than `50`? If so, print the statement, otherwise we check the next condition, is the number greater than `0`? Note that if we are checking this condition we *know* that the number is less than or equal to `50` since it failed the first check. We continue through the rest of this procedure down until the last else block. This block is run only when all of the other conditions fail: that is, our value is not larger than `50`, or larger than `0`, or smaller than `-50`, or smaller than `0`. The only value that satisfies this is `0` itself.

Sometimes, we wish to check compound conditions. That is, we want to know whether multiple conditions hold, or perhaps, whether at least one of many conditions hold. These statements can be converted into "and" and "or" statements, respectively. To denote "and" statements we use `&&` and to denote "or" statements we use `||`. Thus, the check `my_val > 0 && my_val < 100` returns true only if `my_val` is both above `0` **and** below 100. The check `my_val < -50 || my_val > 50` returns true whenever **either** `my_val` is less than `-50` **or** `my_val` is greater than `50`.

[21]In fact, if `my_number` is set to `0` then none of the statements are executed.

```
## [1] "Either 5 or a negative value."
```

Conditional statements can grow to be very complex, however, with these rules you can read through them top-to-bottom, substituting for “and” and “or” where necessary. It is also possible, where required, to place one conditional statement inside the code block for another, and to combine them with any of the other techniques that we have learned thus far.

The final piece of control flow that we will consider for now is the `for` loop. The idea with a `for` loop is that we want to repeat the same action either a certain number of times, or for every item in a set of items. To do so, we use the syntax `for(x in vector){...}`, where the code in `...` will be performed once for every single item in the vector. Within the code block specified by `...`, the value `x` will take on the current value in the loop.

```
## [1] 1
## [1] 2
## [1] 3
```

Notice that three values are printed, in order, `1` then `2` then `3`. The loop code is run three different times, one for each element in the list. Each time the loop is running the next value from the list gets assigned to the variable `x`. The first time it runs it gets the first element, and so forth. As a result, we can use these values in our calculations in whatever way we need to.

```
## [1] "The square of 1 is 1"
## [1] "The square of 2 is 4"
## [1] "The square of 3 is 9"
```

Whenever we are trying to form a numeric vector with consecutive elements, as we are in `c(1,2,3)`, we can make this easier on ourselves by specifying the upper and lower bounds of the range, separated by a colon. That is `c(1,2,3) == 1:3`. This is often useful when specify a loop as we very often want to repeat something a set number of times. Note that we do not ever *need* to use the value of the looping variable. Sometimes, we just want things to repeat, and so the loop is a convenient way to do that.

```
## [1] "This will get printed 5 times."
## [1] "This will get printed 5 times."
## [1] "This will get printed 5 times."
## [1] "This will get printed 5 times."
## [1] "This will get printed 5 times."
```

1.3.5 Reading Through a More Complex R Program

Take a moment to read through the following R program and try to understand what is happening exactly. There are comments throughout which will assist in the parsing of the script. We have seen comments up until now, without drawing explicit attention to them. In R, anything placed after a `#` on the line is considered a comment. The programming language ignores these and so they are only there to help other individuals who may be reading through. It is good practice to comment your code to help others, and also to help yourself whenever you return to it in the future. In these notes code will typically be commented. If you are reading the PDF version of the notes often these comments will be annotations beside the code (numbering certain lines) with the comments provided below, for the sake of legibility.

Note, this combines everything that we have learned, and it is entirely understandable if it takes some time to process. Fortunately, you can always try running the script yourself, and playing with different components of it. Remember, if you do not know what a function call does, you can use the documentation[22] and try playing with it some yourself. To help with the interpretation here, note that this is an implementation of the game that Charles and Sadie have been playing.

```
## [1] "Now starting round 1"
## [1] "The flip was a H which benefits Charles"
## [1] "Now starting round 2"
## [1] "The flip was a T which benefits Sadie"
## [1] "Now starting round 3"
## [1] "The flip was a T which benefits Sadie"
## [1] "Sadie has scored enough points to win."
## [1] "After flipping the coin 3 times, Charles scored a total of 1 points"
## [1] "while Sadie scored a total of 2 points."
## [1] "As a result Sadie won the game and will not have to pay!"
```

1.3.6 R Programming for Probability Interpretations

Recall that the motivation for the discussion of R was the Frequentist interpretation of probability. Computers are very effective at repeatedly performing some action. As a result, we can use computers to mimic the idea of repeatedly performing an experiment. Consider the case of flipping a coin over and over again.

We can use `sample(x, size)` as a function to select `size` realizations from the set contained in `x`. Thus, if we take `sample(x = c("H","T"), size=1)` we can view this as flipping a coin one time. If we use the loop structure we talked before, then we can simulate the experience of repeatedly flipping a coin. Consider the following R code. Note, any time that we are doing

[22]The internet is also a wonderful resource, one which even very experienced developers make frequent use of.

something which is randomized in R (such as drawing random samples) we also will make use of the `set.seed()` function. This function takes in an integer value as an argument, and by providing the *same* integer value we can make sure to always get the same random numbers generated.[23] This helps to ensure the repeatability of any R analysis, and it is good practice to do. To see what happens without seeding, try modifying the following code without a seed, and running it several times. Then, set the seed (to any number you like) and do the same process.

```
## [1] 0.522
```

1. A seed ensures that the random numbers generated by the program are always the same. This helps to be able to reproduce our work.
2. This is how many times we want to repeat the experiment.
3. This is where we are going to store the results of our tosses. It creates an empty list for us.
4. Here we are going to loop over the experiments, one for each run.
5. This is our coin toss. We are going to sample 1 from either 'H' or 'T'
6. If the coin toss is heads, then we add a 1 to the list. Otherwise, we add a zero to the list.
7. Return the mean of all of the tosses.

It is worth adjusting some of the parameters within the simulation, and seeing what happens. What if you ran the experiment only 5 times? Ten thousand times? What if instead of counting the number of heads, we wanted to count the number of tails? What if we wanted to count the number of times that a six-sided die rolled a 4? All of these settings can be investigated with modifications to the provided script.

References

[23] Technically, we cannot use a computer to generate random numbers. We can only generate *pseudo random* numbers, which are close enough for most purposes.