Chapter 2

Data analysis software and graphs: Excel and R

James Myers

2022/2/10 draft

1. Introduction

Ah, you young people today. When you think about doing statistics, you probably think about turning on a computer program, pushing some buttons, and the right answer magically pops out. But statistics is math, and math isn't the same as computation. In fact, when personal computers were first introduced, mathematicians were among the *last* to adopt them, and even today, some hold it as a badge of honor that they develop and test their mathematical ideas using only their own personal brains. Statistics, as a field, is a couple hundred years old (depending on when you start counting), and so a lot of the statistical concepts and methods that we'll discuss in this book were invented long before computers even existed: probability theory, the bell curve, *t* tests, ANOVA, chi-square tests, and even that latest hot topic of Bayesian statistics.

Just as with those mathematicians, your brain is still far better at thinking logically about math, including statistics, than even the most expensive computers today. So doing statistics is not a matter of sitting back and letting your computer do all the work for you. Instead, the computer is merely a tool for doing the most boring and repetitive work, so that you, the superior human being, get to enjoy the fun parts: thinking about how to give your data the proper respect, and how to interpret the results in theoretically interesting or practically useful terms.

The computer can also provide its own kind of fun, though. As a famous statistician says (Gelman, 2013): "There's so much that goes on with data that is about computing, not statistics.... Statistics can do all sorts of things. I love statistics! But it's not the most important part of data science, or even close."

This chapter introduces the two computer tools that we'll use through the rest of this book: Microsoft Excel (https://products.office.com/excel) and R (https://cran.r-project.org/). I first explain why I chose these particular programs, introduce each by applying them to the same linguistic problem (basic corpus analysis), show how to do some slightly fancier stuff in each program, and then give an overview about how each program makes basic graphs.

By the way, I hate to say this, but don't just sit there! When you're reading this chapter (and the rest of this book, in fact), you really have to try out everything that I demonstrate, by yourself. Learning statistics is more like learning to cook than learning history: it's something that you have to actively do.

2. Excel and R

Why am I making you learn Excel and R? First, these two programs complement each other: Excel is much easier to use for many basic functions, but R is much more powerful. This complementarity is natural: as Spiderman says, with great power comes great responsibility, so the power of R is precisely what makes it harder to use. That's one reason why I still often use Excel, and I suspect that most people who go on and on about how wonderful R is actually use Excel quite a bit too, but are too ashamed to admit it. Excel is also widely used in the "real world", so learning it just might help you get a job. Meanwhile, the popularity of R can be seen everywhere: not only are most of those statistics-for-linguists textbooks (see chapter 1) based on R, but professional statisticians (like Gelman) also tend to prefer it over other statistics software. Among other things, this means that the internet has a lot of information on how to use R.

Second, Excel and R are both free. Well, R is totally free, in both senses of the English word "free": it costs nothing, and it is "open", that is, not locked up in any way. In other words, you can download it without registering, and if you're a computer expert, you can look at its own internal programming and copy or edit any part of it that you like. The freeness of R is reflected in its weird name, in fact. As you may or may not know, the most widely used computer language in the world is **C** (used to program everything from video games to Excel itself), so in 1976, when Bell Laboratories (the inventors of C) wanted to create a new language just for statistics, they called it **S** (for "statistics"). S is not free (not open and not cheap), so in 1997, when poorer researchers wanted to create an open free version, they called it R, because R is the letter before S in the alphabet, and the R program is almost like the S program. R is now maintained and updated by a bunch of people called the R Core Team, as part of the R Foundation for Statistical Computing, based in Vienna, Austria. That's why citations for R look like this: R Core Team (2018) (see reference list for details).

As for Excel, technically it's not free, in either sense: you have to buy it, and you can't look at its internal programming or change it. But if your computer has a Windows operating system, you already have it as part of the Microsoft Office system. And if you don't want to buy Excel at all, there are a number of truly free alternatives, including **OpenOffice** (https://www.openoffice.org/), which contains the Excel-like program **Calc**. In this book I'll assume you have some version of the "real" Excel, but since everybody has slightly different versions, you'll have to adjust my examples to fit your own computer anyway.

Software freeness is not a trivial matter. Linguists tend not to be very rich, and linguistics students are even poorer. So don't be mislead by people talking about IBM's **SPSS** program (www.ibm.com/software/analytics/spss/). While SPSS continues to dominate the statistics software market, this is mainly because it is popular in the business world (the "B" in "IBM" stands for "business"), and in business, people tend to have a lot of money to burn. Since you

don't have to buy SPSS for this book, you can use your savings to buy a round-trip ticket between Taiwan and the US, and still have enough money left over to buy a brand-new high-end laptop. I'm not kidding: SPSS is *extremely* expensive. So if I were to teach you how to use SPSS, basically the only ways you could continue to apply your knowledge after you graduate would be to get a job in a rich institution that already has SPSS, or pirate it (or visit https://www.gnu.org/software/pspp/ to download the amusingly named free imitation PSPP, though even after almost twenty years of development it has not yet reached version 1). By contrast, if you learn Excel and R, you can continue to use them for the rest of your life. (Interestingly, the internet is filled with information for SPSS users wanting to learn R, but nothing in the other direction. This book just saves you that useless first step.)

Third, both Excel and R are logical, again in complementary ways. Excel makes more sense visually (e.g., when taking a bird's-eye view of your data, or making a quick graph to see what's going on), while R is a full-fledged computer programming language that lets you do anything your brilliant little brain can think of. So Excel has a user-friendly (well, sort of user-friendly) **graphical user interface** (**GUI**), with menus that let you choose from the small number of choices that Microsoft gives you, while R uses line-by-line commands to create all sorts of new functions. While you aren't used to a command-line computer program (yet), at least R's language looks and feels quite similar to lots of other widely used computer languages; learning R helps train you in programming, which can't be bad for your job prospects either. By contrast, like Excel, SPSS uses a menu-based GUI that limits what you can do, so if you want to do something even slightly unusual, you have to use a special SPSS language that looks totally different from a "real" computer language. Excel allows for command-line programming too, using something called **macros** [巨集], but we won't use macros in this book, since most Excel users don't use them either.

Moreover, precisely because R is free and open, many outside programmers have creating tools for improving its ease of use. For example, if you want R to look more like Excel or SPSS, you can install the free menu-based **Rcommander** tool (Fox, 2005; https://socialsciences.mcmaster.ca/jfox/Misc/Rcmdr/). If you want R to look more like Mathwork's **Matlab** (http://www.mathworks.com/products/matlab/), another expensive program that's particularly popular with neuroscientists, you can install the free **RStudio** program (https://www.rstudio.com/), which is not a part of R, but rather a platform that runs R, offering you various types of extra menus and windows. You can even use Rcommander and RStudio at the same time. We won't use either of these tools in this book (partly because I want to focus on "pure" R, and partly because I don't use them much myself), but feel free to try them yourself, if you think they'll help you to learn and use R better (while still saving money for your trip and new laptop).

No matter what computer tools you use for doing statistics, remember that you remain smarter than them. Software is intended to make life easier, but long experience has taught me

that the first rule when using software is: Be patient! Computers can be frustrating because they are too complex to fit into the human mental module for "tools", but too stupid to fit into the module for "people" either. So it's normal to want to smash them on occasion (don't really do it, though).

3. Computing word frequency using Excel and R

If linguists have a favorite poem, it's probably "Jabberwocky", which appears in Lewis Carroll's classic 1871 children's book *Through the Looking-Glass* (愛麗絲鏡中奇遇; the sequel to *Alice in Wonderland* 愛麗絲夢遊仙境). "Jabberwocky" starts like this:

'Twas brillig, and the slithy toves
Did gyre and gimble in the wabe;
All mimsy were the borogoves,
And the mome raths outgrabe.

As you can see, the poem is filled with fake words! Carroll loved these kinds of word games, and it's not surprising that linguists love them too: "Jabberwocky" is discussed in textbooks like Fromkin et al. (2018), and the great Chinese linguist Chao Yuan Ren (趙元任) even "translated" it into Chinese (with the help of some fake characters; Chao, 1969).

So here's the question: What is the most common "word" in this poem? Is it one of those fake words, or is it a real word? If it's a real word, is it a content word (like a noun or verb) or a function word (like an article or conjunction)?

The whole text of the poem is freely available on the Internet (e.g., https://en.wikipedia.org/wiki/Jabberwocky), so I created a file containing all of its words, in order, without spaces or punctuation, and called it Jabberwocky_OnlyWords.txt. Like all the files we'll work with through this book, this is supposed to be available from the statistics resources page at http://personal.ccu.edu.tw/~lngmyers/statsresources.html), though I'm not sure how stable my homepage will remain, so I'll also put it into the E-Course folder for this class. Note that the ".txt" part, which tells the world (and your computer) that this is a text file, may be invisible to you, since by default Windows (and maybe Macs?) hides this so-called **filename extension** [副檔名]. Search the internet for how to learn how to make it visible (though this isn't crucial, since R can still see it).

If you open this file in a text editor, you'll see this:

jabberwocky
twas
brillig

and
the
slithy
toves
...

3.1 Computing word frequency in Excel

Let's first try doing this in Excel, after we learn some basic Excel operations.

3.1.1 Excel basics

You can open up Jabberwocky_OnlyWords.txt in your browser and copy/paste it from there into Excel, or you can save the file and open it within Excel (sometimes these two processes work differently, but not in this case).

(Hello...? Did you actually open up Excel and put that file in there? Do it! Stop reading this for a minute and go do it *now*!)

Excel is what is called a **spreadsheet** program, which puts data in a big grid with lots of **cells** arranged in rows and columns. Notice that in Excel's spreadsheet, there is a row of letters at the top and a column of numbers on the left: Excel refers to cells or groups of cells by these letters and numbers. So the cell in the upper left corner is A1, the horizontal row of the first 5 cells is A1:E1, the vertical column of the first six cells is A1:A6, the whole A column is A:A, the two-row by three-column rectangle in the upper left corner is A1:C2, and so on. Normally you don't have to type these identification codes; they magically appear whenever you select a cell or a set of cells (called a **range**).

Let's say you copy/pasted or opened Jabberwocky_OnlyWords.txt so that it appears in column A; specifically, the first word "jabberwocky" is in cell A1 and the last word "outgrabe" is in cell A167. What we have here is a list of **word tokens**, ordered as in the original poem, but to count **token frequency** for every **word type**, the conceptually simplest way is to group tokens of the same word type together. To do this we just sort column A in alphabetical order (I got rid of capitalization so that it doesn't affect the grouping; Excel's sorting function ignores capitalization, which can be annoying for linguistic research). So we select column A by clicking the "A" on top, then look around the menu for a button that looks kind of like A→Z (its precise location and appearance depends on which version of Excel you have). Note that Excel also allows you to sort multiple columns together, putting all rows into the order determined by just one of the columns, or even sorting by one column first, and then sorting the "ties" by another column.

Here we only have one column (column A), so sorting it makes the top look like this:

all
all
and
and

We could count these by hand, but the whole point of Excel is to make it do all of the boring work: don't work hard, work smart! Excel is not just a grid of cells; each of those cells can do calculations. If you type something starting with = into a cell, then Excel assumes you are using a **function** [函數], operating on **arguments** [參數], which can be something you type in, or the values of other cells in your spreadsheet. The syntax is always the same: **=FUNCTION(ARGUMENTS)**.

Excel has many kinds of cell functions. If you forget something about a function, or want to find out what other functions there are, just click on the f_x symbol next to the **function bar**, above the spreadsheet part of the window, and of course, you can also search for help on the internet. But in the meantime, here are some of the most important types.

Math functions include self-explanatory ones like **=COUNT()**, **=SUM()**, and **=MIN()** (e.g., **=MIN(5,3,7)** gives you 3) and there are also functions whose arguments are **character strings** (字串), that is, a bit of text within quotation marks (e.g., **=LEN("five")** gives you 4, since there are four letters in "five"). Math functions ignore character strings, so if you hit return after typing what's in the highlighted cell below, the value of that cell will become 4 (similarly, **=COUNT(B1:D1)** gives 2 and **=MIN(B1:D1)** gives 1). You could also just type the **=SUM(** part, then select cells B1, C1, D1, and Excel will automatically enter **B1:D1** into your function; in this case, Excel will even add the **)** symbol for you when you hit the enter key, though you can't count on it doing this correctly in all cases.

	A	B	C	D
1	=SUM(B1:D1)	1	banana	3

You can also compute sums with + (if you have some specific set of numbers to add, rather than summing up a whole row or column). Similarly, the symbols for subtraction, division, multiplication, and **exponentiation** (冪) are **-**, /, *, ^ respectively (e.g., 3×4^2 is written 3*4^2, read in English as "three times four to the second **power**" or "three times four **squared**"). (R uses exactly the same symbols, as we'll see.) There are also fancier mathematical functions like **=SQRT()** for computing **square roots** (平方根), and of course statistical functions like **=RAND()** for generating random numbers and **=AVERAGE()** for computing averages.

Note that if a number is too large or too small to display in the normal way, both Excel and R use **scientific notation** (科學記數法), where 1E+01 is 10 and 1E-01 is 0.1 ("E" for exponentiation). For example, in Excel the command **=7777777*7777777** will give the result

6.04938E+13, which means $6.04938 \times 10^{13} = 60{,}493{,}800{,}000{,}000$ (obviously rounding a bit from the real answer). Similarly, **=1/7777777** gives 1.28571E-07, which means $1.28571 \times 10^{-7} = 1.28571/10^{7} = 0.000000128571$. So if you see something like 8.98E-20 (and we often will, when we get to ***p* values**), just remember that it's basically equal to zero.

Excel also has character functions like **=CONCATENATE()**, which pastes strings together (i.e., concatenates them: 連接). In both Excel and R, strings have quotations marks around them (either **"** or **'**, as long as you use the same symbol on both ends, which is useful, since if you want to quote a quotation mark, just use the other kind, e.g. **'"'** for ' " '). Just as you can use + for sums, you can concatenate strings with **&** (**=CONCATENATE("book", "store")** and **="book" & "store"** give the same result). Other useful character functions include these: **=LEN("cat")**, **=LEFT("cat",1)**, **=RIGHT("cat",1)**, **=MID("cat",2,3)**, (try them!). By the way, if your version of Excel (or R) is "native" to your operating system, the character functions should work just fine for whatever your native orthography is. For example, **=RIGHT("大象",1)** will yield 象.

Finally, Excel has logic functions like **=IF(X,Y,Z)**, which means "if X is true, then Y, otherwise Z". For example, if you type **=IF(3=3,"yes","no")** into a cell, it will change the cell's appearance to "yes". Similarly, if you type **=IF(3<>3,"yes",SQRT(2))** (where **<>** means "not equal to"), it will display the square root of 2. (The symbols in R for "equal to" and "not equal to" are different from Excel's, as we'll see; this difference can be confusing, so get ready.) Excel also has combination functions like **=COUNTIF()**, which counts how many instances of particular thing there are; for example, if the range A1:D1 has the four strings "cat", "dog", "pig", "cat", then **=COUNTIF(A1:D1, "cat")** will give you 2.

For even more complicated operations (which is the usual situation in real data analyses), you can also put functions inside other functions. For example, what value does the following cell function give you, and why? Try it: but be careful that the parentheses match up correctly! Excel shows you how the parentheses are matching up as you type (the ordinary version of R does not give you this kind of help, but RStudio's version does).

=IF(AND((100/2=50),(LEFT("cat",1)="c")), "both true", "at least one is false")

The power of Excel cell functions doesn't end there. If you copy and paste a cell with a function to a new location, Excel will magically redo the calculations based on the function's new location! For example, if cell **B3** contains the function **=A2** (copying the value from the cell that's one row up and one column to the left), then if you copy/paste this to cell **AD472**, its function will now say **=AC471** (still referring to the cell that's one row up and one column to the left, relative to itself).

Sometimes you want this context-dependent change to happen only relative to the rows or only relative to the columns; in that case, Excel gives you a way to "freeze" cell references to make them absolute, instead of contextual. For example, say you have two lists of words

and you want to know how many matches they have. One way to do this would be first to arrange one list in a horizontal row and the other in a vertical column, as shown below:

	A	B	C	D
1		cat	dog	fish
2	cat			
3	dog			
4	monkey			

Then you can type in a logic function in just one of the cells, say B2, as shown below. This function checks if the word in cell B1 ("cat") is identical to the word in cell A2 ("cat"), but notice the "$" symbol: this "freezes" the cell reference (row or column) after it so that even when we copy/paste or drag the cell (by pulling it from its lower right corner) across the spreadsheet, it will still refer to row 1 ($1), where the horizontal list is, and to column A ($A), where the vertical list is. Thus the rest of the cells will automatically get updated like so:

	A	B	C	D
1		cat	dog	fish
2	cat	=if(B$1=$A2,1,0)	=if(C$1=$A2,1,0)	=if(D$1=$A2,1,0)
3	dog	=if(B$1=$A3,1,0)	=if(C$1=$A3,1,0)	=if(D$1=$A3,1,0)
4	monkey	=if(B$1=$A4,1,0)	=if(C$1=$A4,1,0)	=if(D$1=$A4,1,0)

This will generate the following cell values. Now you can use **=SUM(A1:D4)** to count the matches (2).

	A	B	C	D
1		cat	dog	fish
2	cat	1	0	0
3	dog	0	1	0
4	monkey	0	0	0

An Excel cell consists both of the output value (shown in the cell itself) and the function that it's computing (shown in the function bar at the top when you select the cell). What if you only want the output value of a cell, not the hidden function creating this value (e.g., you want to keep a permanent record of the grades that you've been computing all semester)? In that case, copy the cell, and before you paste it again, right click and choose **Paste Special** (S), then click **Value** (V), thereby "freezing" the output value.

3.1.2 “Jabberwocky” in Excel

OK, enough background. How can we take our alphabetized list of “Jabberwocky” word tokens and compute the token frequencies? Here’s one way that works. It’s not the most elegant way, as we’ll see later, but this way lets me teach you more stuff about Excel.

First, slide the alphabetized list down one cell in column A, so it now starts in cell A2. In cell B2 (the second cell of column B), enter a function that compares the current word with the previous one: if the current word is different, it must be the first token of this word, so it gets a token count of 1, and otherwise it adds 1 to the previous token count (B1). Then in column C we mark the last token of a word (i.e., where the following word is different from the current one) with *. The B column cell next to * will thus contain the token frequency for that word. Note that the function in column C will either output the string "*" or the **empty string ""** (a blank cell). So your cell functions will be as shown below.

	A	B	C
1			
2	all	=IF(A2<>A1,1,B1+1)	=IF(A2<>A3,"*","")
3	all	...	...
4	and	...	...
5	and	...	...

When you’re done copying these functions to all the cells in columns B and C, you turn off the functions by copying and pasting “values only”, and then put the rows into order by column C, which will put all the blank cells at the top, so you can easily delete those rows (when you delete them, they disappear from the spreadsheet and all the other rows move up). Now you’ll end up with a list of word types in column A, with each word’s frequency in column B. To find out which word(s) have the highest frequency, just sort again by column B, from largest to smallest.

When I did this, the top six rows of my Excel spreadsheet look like this:

	A	B	C
1	the	19	*
2	and	14	*
3	he	7	*
4	in	6	*
5	jabberwock	3	*
6	my	3	*

I promised you a more elegant way to do this, and here it is. Going back to the original poem starting in cell A1, type **=COUNTIF(A:A,A1)** into cell B1 (or select its arguments by clicking the relevant range and cell), and drag this down column B to the end of the poem. That

will give you a token count for each word. To make the final lexical list above, you'll still have to remove the repeats, but Excel has a built-in tool for that: go to the Data [資料] ribbon, look for the Data Tools [資料工具], and click the Remove Duplicates [移除重複] button. The lesson is: to do things more efficiently in Excel, you need to know more Excel tools and cell functions. And the lesson to that lesson is: learn how to search for more information about Excel on the internet.

In any case, this little exercise shows that, unsurprisingly, the most frequent words in "Jabberwocky" are real function words. Still, one fake word makes the top-five list too: the Jabberwock, the monster at the center of the poem.

3.2 Computing word frequency in R

Now let's do this all over again, this time using R. Of course, we need to learn some R basics first.

3.2.1 R basics

To get your own free copy of R (by itself, not via RStudio, which I won't discuss in this book), search the web for "R cran" (CRAN = Comprehensive R Archive Network), which should bring you to https://cran.r-project.org/, and find your computer type (Windows, Macintosh, Linux). For Windows, you want to start with the **base** link (basic R stuff), which brings you to a page with a link to the most updated version (while I'm typing this, it's R-4.1.2-win.exe). For Macs, you need to choose the version that suits your specific operating system (while I'm typing this, they currently offer R-4.1.2.pkg for macOS 10.13 [High Sierra] and R-4.1.2-arm64.pkg for macOS 11 [Big Sur]) - whatever that means, I don't have a Mac. In fact, if you Mac users have questions about R as you go through this book you'll have to do some internet searches for help. A good place to start is R's official Mac FAQ [frequently asked questions] page: https://cran.r-project.org/bin/macosx/RMacOSX-FAQ.html). Of course there's also one for Windows: https://cran.r-project.org/bin/windows/base/rw-FAQ.html. There's also a general R FAQ: https://cran.r-project.org/doc/FAQ/R-FAQ.html. But searching the internet will bring up lots and lots of other helpful websites hosted by other organizations and even private citizens.

In any case, when you install R, by default it will be "nativized" to the language of your computer's operating system, though of course this only affects the menus and some basic warning messages, not the R language itself, which is based on English. If you want everything to be in English (like me!), then unclick the "Message translations" box during set-up, or at least that's how it works in Windows; as I said, I don't have a Mac. However, I do happen to know that the Mac menus look a bit different from those in the Windows version; for example,

in order to change the **working directory**, i.e., the folder where R looks for data files and saves your results files, in the Windows version of R you by using the "File" menu, but in the Mac version you use the "Misc" menu.

(Um.... You actually did this, right? You actually downloaded R and set it up, right? Just reading this chapter passively isn't going to do you any good! OK, enough nagging.)

If all went well, when you run R you'll get one big window with a few menu items on top, and in the body of the window you'll see some blue words and then the **prompt** symbol **>** in red (though you can actually change these colors, and the background color, to anything you want; in the Windows version, look in the **Edit** menu for **GUI preferences...**, and for Macs, there's an **Application** menu with **Preferences** under it).

What you're looking at is just an interface to R, not R itself; in fact, as just noted, the Windows and Mac interfaces aren't even the same, and RStudio uses yet another type of interface. R itself is a computer language, and whichever GUI you uses is just where you enter commands and get results. So using R doesn't feel the same as using Excel or most of the other programs that you may be used to. For example, while the R GUI has menus, they're just for changing the file folder (directory), changing the appearance of the window, installing extra **packages** (tools with special functions), and other basic stuff like that, not for actually running analyses. To do an analysis, you have to write a little computer program (a **script** if it's very little, or more generally, **code**). While this gives you great power and flexibility, it means you have to memorize (or look up) the proper **command** words and syntax. Fortunately, the R computer language is pretty simple compared to some other computer languages: similar to Excel's cell functions, the syntax is almost always **function(argument)** (without = at the front). If you forget what an R **function** does, you can look R's built-in help files by typing **?** before the function name. The help files will show you what the function's syntax is supposed to be, and what values the function generates. However, because R was written for free by statistics nerds, these help files are not always very helpful. I often prefer just to search for help on the internet, where usually you can find somebody who has asked exactly the same question, along with a pretty clear answer (mostly in English, but R is used by lots of Chinese speakers too).

In addition to giving you complete freedom to do any kind of analysis you want, the R language also encourages you to play actively with your data, instead of just sticking it into a black box and passively waiting for the results, as you do with most statistics programs (including Excel and SPSS). Every time you enter a new command into R, you can do a new thing to your data, including reorganizing it, changing it (e.g., dropping bad data, as we'll explain in later chapters), making a graph, making a different graph, trying one analysis, trying a different analysis, and so on. This is how real statisticians work.

If you're typing R code directly into R's window, one useful tip to know is that you can reenter commands by pressing the up-arrow and down-arrow keys, which scrolls through all the commands you've entered since you started running R. But I usually don't type my

commands directly into R, but instead type them in a text editor first, saving my work as I go along (as a **.txt** file). Don't use Word to write your code, since unless you change your settings it will automatically "fix" your spelling and even change your quotation marks (R functions recognize **"** but not "). As a Windows user, I sometimes use the built-in Notepad [記事本] program (like Mac's built-in TextEdit [文字編輯]), but for Windows even better is the free non-Microsoft program Notepad++ (https://notepad-plus-plus.org/), which has some extra features to the programming life a bit easier. For example, it fills in brackets for you, so if you type **(**, **[**, or **{**, it will automatically add **)**, **]**, or **}**, respectively, and as in Excel, if you click on one of them later, the other one will be highlighted in the same color; otherwise, if you have a lot of brackets, it can get quite confusing about which bracket goes with which. Notepad++ also lets you change the default colors and font size to something easier on the eyes, and it can supposedly even help you remember R commands if you turn on auto-completion (https://npp-user-manual.org/docs/auto-completion/), though I've never tried that feature. For the Mac, fancy text editors similar to Notepad++ include Brackets (http://brackets.io/), Textmate (https://macromates.com/), and Sublime Text (https://www.sublimetext.com/; this also works in Windows).

When I'm ready to run a script, or part of a script, I usually just copy/paste it into the R window. You can also load an entire R script file with the function **source(filename)**, where **filename** is a string naming the script file, but I tend not to do it that way, since it doesn't show you the script actually working, making it harder to find programming errors. If you use RStudio, it will help with all of these things too (complete brackets, run code, etc), plus instantly update your graphs when you change the code that creates them.

When you save your R script on your computer, you can continue working on your data the next day (or next year), or share it with colleagues (more and more academic journals encourage authors to post their R code along with their paper, so readers can check the analyses for themselves). You can even apply your analysis to a totally different data set, or edit it for a new but similar situation (very useful if it took you a long time to figure out the original analysis). All of this is basically impossible to do with Excel (as you can already see in this chapter, it's a lot harder for me to explain how to do stuff in Excel than in R, where I can just give you code to copy/paste). Many times I've labored for hours working through some Excel-based procedure, rearranging and deleting things as I go along, and then when I want to redo the whole process later on, I have no clear memory how I did it, or don't even know if I made any mistakes along the way. By contrast, to redo an R analysis, I just have to search my computer for my old R script and edit it a bit to fit my new situation (later I'll explain an even fancier to save and share R code).

Now that I've sold you on how wonderful R is, let's actually start using it.

Even though R may look weird to you at first, to understand its "soul", you basically only have to remember two principles. First, R is an **object-oriented language**. That means that

whenever you run a command, you are actually creating a "thing", not merely doing an action; you can then name this object and do other stuff with it, like look inside it to see what properties it has (we'll see lots of examples of this shortly). This is different from Excel, where the cell functions just change the value of the cell, but don't create an object.

Second, because R is designed for statistics, which is all about analyzing data sets, the most common type of R object is a **vector** (向量), an ordered sequence of elements that are all of the same type (e.g., all numbers, or all character strings). It's called a vector because when the sequence consists of numbers, you can think of it as an arrow with a certain direction and length (e.g., the vector (1,2) could represent an arrow starting at the center point in a 2D plane and ending at the point (1,2), i.e., where x = 1 and y = 2).

Sorry for the weird name; basically, a vector is just a sequence of numbers or strings. Because a vector is an object in R, you can look at its properties, including each of its elements and its total length. For example, if you have a vector named **banana** (why not? you can name your R objects whatever you like) that has 253 elements (why not? R permits super-long vectors, limited only by your computer's memory), then the 179th element can be referred to with **banana[179]**. The thing inside the **[]** is called the **index** (plural: **indices**), since it helps R "look up" the referred-to element, like an index at the back of a book.

You have to confront both of these two principles as soon as you write even the simplest R command. For example, say you want to add 2 and 3, using the following command. (In all of the R examples in this book, the bold part represents the red part you type after R's red > prompt, and the non-bold part represents the response that R gives you in blue. So feel free to just copy/paste the bold part from this e-file to your R window.)

```
2+3
[1] 5
```

The answer is 5 (of course), but what does R mean by that [1]? R treats 5 as a vector with just one element in it, and the location of the first element in this one-element vector is represented by the index [1]. In essence, our command asked R to compute something and create an unnamed one-element vector object, which it then displays for us, because all we did was "describe" the object with our command.

Now let's put the output of our command into a new object (I'll often use the word **variable** to describe one-value objects that can change their contents, like the answer to a math problem). Again, we can call it anything we want, so let's call it **x**:

```
x = 2+3
```

This time R doesn't give a response, but when we type **x**, R responds just as if we had typed **2+3** directly:

```
x
[1] 5
```

By the way, in this book I prefer to use = to create and name objects, including variables, but many R authors prefer to use the arrow-like symbol **<-** instead: **x <- y** means "put the value of the object **y** into the object **x**". One advantage of **<-** is that it makes it clear that we are not talking about "equal to" here. For example, for beginning programmers, the command below looks like a logical impossibility, but actually it's a command that takes the value of **x**, adds 1 to it, and then updates **x** with this new value:

```
x = x+1
x
[1] 6
```

The arrow notation also lets you assign variables "backwards" if you want (I don't see the point of this at all, but whatever):

```
3 -> y
y
[1] 3
```

One difference from Excel is that R does care about the difference between uppercase and lowercase letters; **SUM()** is not a function in R, but **sum()** is. You can use it just as you would in Excel:

```
sum(3,1,4,1,5)
[1] 14
```

But **sum()** is actually an unusual R function in this way, because R normally wants to know exactly how many arguments it's dealing with. Since we can sum any number of numbers, R would prefer for us to put all the numbers into one object, namely a vector. The most general way to create a vector to use the **c()** function ("c" for "combine"), which lets you combine anything of the same type (e.g., all numbers or all strings):

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

That [1] refers just to the first number in the vector (3), not the whole thing. You can see that's what's going on if you make R's window super-narrow, so the text has to wrap around:

c(3,1,4,1,5)
[1] 3 1 4 1
[5] 5

I could have explained this to you directly in the command itself, as in the command line below. Anything written after the # symbol is treated as a **comment** and ignored by R (for human eyes only). Commenting is very useful if you need to explain your code to somebody else, or to help you understand your own code when you look at it later.

c(3,1,4,1,5) # The [1] in the answer refers to the first element, not the whole vector

R also gives you lots of other ways to create special types of vectors (as explained in the comments):

a = c(9,6,4,8,3,2,0) # Creates the vector a with these values in this order
b = rep(1,5) # Uses repetition to create the vector b = (1,1,1,1,1)
c = 3:7 # Creates c = (3,4,5,6,7)
d = rep(6:9,2) # Creates d = (6,7,8,9,6,7,8,9)
e = seq(1, 5, by=2) # Uses a sequence to create the vector e = (1, 3, 5)
f = c(a,b,c,d,e) # Creates f = (9,6,4,8,3,2,0,1,1,1,1,1,3,4,5,6,7,6,7,8,9,6,7,8,9,1,3,5)

So if you want to use the **sum()** function in the usual R way, it would be better to put our numbers in a vector, for example using **c()**, and then put this vector inside **sum()**:

sum(c(3,1,4,1,5))
[1] 14

To see why it's important to use R functions in the way they expect, consider the statistical function **mean()**, which computes the same thing that Excel's **=AVERAGE()** computes (see next chapter for details). Here's the right way to compute the average of our set of numbers (compare the result with **=AVERAGE(3,1,4,1,5)** in Excel):

mean(c(3,1,4,1,5))
[1] 2.8

Even though a few R commands can automatically convert a list of elements into a vector, this is very rare: most cannot, including **mean()**. So if you forget to make the vector (using the **c()** command) for **mean()**, R will just respond with the first number in your data set, because it treats the first argument as a vector and just averages it all by itself, ignoring all the rest of the numbers:

```
mean(3,1,4,1,5)
[1] 3
```

This kind of mistake can be hard to detect (in fact, I just did it myself the other day!), since R doesn't give you a syntax error (we will see a few more examples over this book of this annoying aspect of the programming life). And also be careful not to confuse **[]** (for vectors) and **()** (for functions)! I did that myself the other day too (my excuse is that I had just edited the code and didn't notice that I needed to change this bit too), and while this time R gave me an error message, it was totally baffling and useless:

```
mean[c(3,1,4,1,5)]
Error in mean[c(3, 1, 4, 1, 5)] :
  object of type 'closure' is not subsettable
```

Wha...?? Apparently functions in R have the **type** called "closure", and the **[]** tell R that you're trying to find a subset of it (i.e., just the elements indicated by the indices), which of course makes no sense. But I wish R would just say "Don't use [] with functions, dummy!" If R ever gives you a baffling error, just copy/paste it into an internet search and you'll find some other dummy getting an explanation.

In case you're curious, you can see the type of an R object using the **typeof()** function ("double" is old-timey computer terminology for "ordinary number", so called because it uses two memory portions to store the stuff before and after the decimal point [小數點]).

```
typeof(mean)
[1] "closure"

typeof(3)
[1] "double"

typeof("banana")
[1] "character"
```

Fortunately you don't have to worry a lot about object types, since unlike most computer languages, R is very flexible, and will usually change the type for you (later I'll warn you about situations where it doesn't). For example, you can easily turn a logical variable into a numerical variable (FALSE = 0, TRUE = 1), just by applying arithmetic to it:

```
1*(3 == 1:5)
[1] 0 0 1 0 0
```

Since we've introduced the character type, you should know that, like Excel, R also has lots of functions for them. For example, to concatenate (remember, this means to paste strings

together), use **paste()**. The **default** (built-in) separator for **paste()** is a space (**sep = " "**), but if I'm making a compound word and I don't want a space, I can change it to **sep = ""** (the empty string).

```
paste("book","store",sep="")
[1] "bookstore"
```

How did I learn about the **sep** argument in **paste()**? I started by opening up R's help file (it opens a page in your Web browser, but it's actually a file on your computer):

```
?paste
```

In the Usage section of the help page, it says the general syntax is **paste(..., sep = " ", collapse = NULL)**, and in the Arguments section, it explains that **...** represents "one or more R objects, to be converted to character vectors". That tells me that **paste()** is like **sum()**: one of the rare R functions that lets me enter any number of arguments in the first position. The Arguments section also explains that **sep** represents "a character string to separate the terms", and since the Usage section shows **sep = " "**, that means that the default character string is a space. So from all this I deduced that to concatenate without a space, I need to change **sep** to **""**. The final argument is **collapse**, with the default of **NULL**, but I don't need to change the default on that here. Basically, **collapse** acts like **sep** when we enter our strings as a vector, which can be useful in complex scripts:

```
paste(c("book","store"),collapse="")
[1] "bookstore"
```

R also has functions similar to, but sometimes crucially different from, Excel's **=MID()** and **=LEN()**. Try and see below. In particular, remember that to R, even a simple value is a one-element vector, so for functions that do one thing to one string (or number), **function(vector)** will do that same thing individually to each of the elements:

```
substring("cat",2,3) # Notice the logic is a bit different from Excel!
substring(c("cat","dog"),1,1) # Like most R functions, this can process vectors too
nchar("cat") # Like Excel's =LEN() function
nchar(c("one","two","three")) # Again, it can work across vectors
```

Note that R's **length()** function doesn't act like Excel's **=LEN()** function, but rather more like Excel's **=COUNT()** function:

```
length(c(9,6,5))
[1] 3
```

To illustrate how R handles if-then logic, I first have to clarify a set of confusing symbols. Remember that Excel cell commands all start with =, which basically makes the cell equal to the output of the command; Excel uses this same symbol to represent "equal to" in logic functions, while <> is used for "not equal to". By contrast, as we saw above, R uses =, **<-**, or **->** to name or update the object created or modified by a command. R also uses the = symbol to assign the values of function arguments (e.g. **sep = ""** in **paste()** above). For if-then logic, however, "equal to" must be written as ==, while "not equal to" is **!=**.

The basic syntax of R's **if()** function is **if(X) {Y} else {Z}**, which means: if X is true, give the value Y, otherwise give the value Z (without the optional **else** part, the function outputs nothing if X is false):

```
x # Still 6, based on the commands we ran earlier
[1] 6

if (x != 6) {paste("orange")} else {paste("banana")}
[1] "banana"

if (x != 6) {paste("orange")} # Nothing happens
```

A final note before we get into "Jabberwocky": If it takes you more than one R session to try out this chapter's R code, you might want to save all the new objects you create along the way, so you don't have to start over again from the start of the chapter when you turn on R again. For example, if you started with a fresh R session with this chapter, right now your R workspace should have the following boring variables that we've just created, as you can see with the **ls()** function:

```
ls()
[1] "a" "b" "c" "d" "e" "f" "x" "y"
```

Every time you turn off R, it will ask "Save workspace image?" or the equivalent in whatever language you used to set up R, which means to save a copy of the workspace with al your objects in it. If you choose "Yes", while exiting R it will also create two files in your active directory: **.RData** (which is the workspace image file, with all your objects in it) and **.RHistory** (which is the record of all the commands you ran in R during your session). Now, when you start a new session, after changing the directory to your working folder (remember, in Windows that's "Change dir..." in the File menu), you can reload your old workspace by choosing "Load Workspace..." in the File menu, and selecting **.RData**. Try it, and check what **ls()** says!

3.2.2 "Jabberwocky" in R

OK, now that we know some R basics, how do we put the Jabberwocky_OnlyWords.txt file into R? Unlike Excel, pasting data into the R window won't work (you just get a long list of syntax errors, except where Carroll's words coincidentally match existing R commands, none of which we'll use in this book). Since this file is on the web, it's actually possible for R to get it itself, since it knows how to access the web, but in real life, your data files will usually be on your own computer, so I'll normally assume you downloaded the file first, and saved it in some **directory** (folder).

The easiest way to tell R how to find the file on your computer is to use R's menus to change the working directory: click **File**, then **Change dir...**, and then find the folder where you put Jabberwocky_OnlyWords.txt (note that R's file menu system may start way at the "top" of your computer's file system, so you need to go through the "Users" and "Admin" levels before you get down to the more normal folders). You can confirm that you're in the right directory by using the **dir()** function:

```
dir() # This doesn't require any argument
```

In my case, I see "Jabberwocky_OnlyWords.txt" in there, so I know I'm in the right folder. Note that there are quotation marks around the file name, since R treats it as a string, which is useful if you want to write R functions that can find or change file names. If your folder has lots of other stuff, each filename will appear next to [1], [2], and so on, since the object created by **dir()** is actually a character vector, listing the names of all the files in the folder.

Our file also contains a character vector (the poem). R has a variety of ways of opening files and putting their contents into R's **workspace**, depending on the file and data types and what you want to do with them. Right now, R's workspace only has **x** and **y** in it (the variables we created while playing around above):

```
ls() # ls for"list"
[1] "x" "y"
```

As with many "raw" corpora, our file is just a series of lines of text (in this case, just one word each), so the simplest function to use to load it in is: **readLines()** (note the capital L).

```
readLines("Jabberwocky_OnlyWords.txt")
```

But that doesn't quite do what we want: **readLines()** created an object and just displayed it, instead of storing it as a variable: **ls()** shows that the workspace doesn't actually have it. To

add the poem to our workspace, we need to give the object a name. I'll choose something easy to remember and type:

jabberwocky = readLines("Jabberwocky_OnlyWords.txt")

Now if we type **jabberwocky**, we'll see whole the poem again. And since it's an object (specifically, a character vector), we can also look at just parts of it:

jabberwocky[1] # Just first word
head(jabberwocky) # First six words
tail(jabberwocky) # Last six words

You might think that after all this, R is so much harder to use than Excel that it's just not worth the trouble. But here comes the magic. Since R is a statistics program, it has lots of built-in functions designed to answer just the kinds of question we started with. In particular, if we want to know what the token frequency is for each of the words in "Jabberwocky", all we need is a single function: **table()**. This creates a **frequency table** counting how often unique elements appear in a vector. So all the steps we did in Excel, partly by hand, are compressed into just one automatic process:

table(jabberwocky)

To see just the top six most frequent words, parallel to what we did in Excel, we first create the frequency table (using **table()**), then sort it from highest to lowest (using **sort(..., decreasing=TRUE)**), then show the first six (using **head()**), all embedded in one command:

head(sort(table(jabberwocky),decreasing=TRUE))

```
jabberwocky
        the         and          he          in  jabberwock          my
         19          14           7           6           3           3
```

The results are not only exactly the same as what we got with Excel, but they were fully automated. With the Excel method, we have to rely on our memory for what we did if we want to explain it to somebody else or reuse it ourselves later, and there's also a risk that we'll accidentally lose information halfway (e.g., when we delete the repeats in the sorted word list). By contrast, with R, if we want to exactly the same thing to a new character vector, we can simply save the following code in a text file, edit the filename as our new situation requires, and rerun it:

wordlist = readLines("filename.txt") # This won't work unless you really have this file!
freqlist = sort(table(wordlist),decreasing=TRUE)

If, for some reason, we wanted R to do this job in a more Excel-like step-by-step fashion, R still makes it easier than Excel. For example, to extract just the unique elements from a vector containing repeated elements (as we did laboriously in our first Excel analysis of "Jabberwocky"), we can just use **unique()**:

```
unique(c(3,1,4,1,5))
[1] 3 1 4 5
```

However, it would be dishonest to pretend that Excel is not easier to use than R in general, which is why I also include Excel in this book as much as possible. This easiness comes not only from its GUI system, but also from its being better integrated into the specific type of computer you are running. This latter point is especially important for linguists, who work with many strange languages with strange writing systems (like Chinese!). In particular, base R (at least for Windows) is not very good at handling Unicode, the "universal" system for computer coding of written symbols. This means that R can mess up any symbols that aren't in the very small **ASCII** set (American Standard Code for Information Interchange, that is, basic digits, punctuation, and letters without any diacritics), including the IPA and even letters that are standard in non-English spelling systems (e.g., German "ü"). It can even mess up Chinese characters, doing particularly badly with very low-frequency ones. Shortly I'll explain various ways of dealing with this stupid problem.

4. More tips and tricks

In this section I want to mention a few other tips and tricks, unique to Excel or R. Of course the rest of this book will be filled with many more, but we have to start somewhere.

4.1 More Excel tips and tricks

The "Jabberwocky" poem is actually a kind of linguistic **corpus**, and we just did a kind of corpus analysis: computing frequencies. We can also do another thing that corpus linguists often do with corpora: we can **tag** the word tokens for special properties. In this case, let's tag this corpus to mark which words are real English words and which ones are fake. In real life, linguists do this kind of thing all the time; for example, in studying child language, they transcribe a child's speech and then tag all the words for syntactic category (nouns, verbs, and so on), so they can study the child's syntactic structure. The "Jabberwocky" corpus only has 167 word tokens, so tagging wouldn't be very hard to do by hand, but real corpora tend to be thousands of times bigger. Is there any way Excel can make the job a little bit easier?

I'm glad you asked. Instead of manually tagging all 167 word tokens, all we have to do manually tag the 91 distinct word types (the ones we counted the token frequencies for). So

let's say we did that, maybe with the help of Word's spell-check tool to indicate the fake words with wiggly red underlines. We record this information in the same Excel file where we pasted in the full corpus (poem), but in a different **sheet**. Let's call the sheet with the full corpus **Corpus** (you can type the name into the little tag at the bottom of the sheet), and the sheet with the lexicon **Lexicon**, with a column for the words, another column for the frequencies, and a new column showing "real" or "fake" for each word. Since Lexicon has the word types in column A, the frequencies in column B, and the (now useless) stars in column C, let's type the tags by hand into column D.

How can we automatically add these real/fake tags to the corpus? The crucial function is **=VLOOKUP()**, which lets you look up something in a table by searching vertically ("V") row by row. In our case, for each word token in Corpus, we want to look it up in the list of word types in Lexicon, then find the associated real/fake tag, then put that information next to the word token in Corpus. The basic syntax looks like this (the **FALSE** is to force an exact match; if an approximate match is OK, use **TRUE**):

= VLOOKUP(cell-to-look-up, table, column-number-in-table, FALSE)

Since the word tokens are listed in column A in the Corpus sheet, let's put the tags in column B. We start by typing **=VLOOKUP(** into the first B cell in Corpus, then for the cell to look up, we click the first A cell in Corpus (which has "jabberwocky" in it), and for the table, we change the sheet to Lexicon (which will automatically be marked in the function by the appearance of **Lexicon!** before the cell or range, with an exclamation point), then select the entire lexicon table, crucially including both column A (with the words to be looked up) and column D (with the tags). Since column D is the fourth column from A, the column number is 4. Then we add that **FALSE** to look for exact matches, and end the function with the close parentheses mark. Since we're going to copy/paste or drag this function down all of column B in the Corpus sheet, we need to fix the absolute location of the table in Lexicon with those $ marks, to fix both columns and rows. The result is that the function in cell B2 in sheet Corpus will look like this:

= VLOOKUP(A1,Lexicon!$A$1:$D$91,4,FALSE))

Assuming "twas" is a real word, we end up with a tagged corpus in the Corpus sheet that starts like this:

	A	B
1	jabberwocky	fake
2	twas	real
3	brillig	fake
4	and	real
5	the	real

By the way, we probably want to name our columns something more memorable than just "A" and "B". We could do this by shifting all the content down one row, then entering something like "Text" and "Tags" into cells A1 and B1, respectively. We can even fix this row so it doesn't move as we scroll down, by selecting the whole row (click the "1" row label), changing the menu to **View**, and then clicking that little icon with the shaded first row.

4.2 More R tips and tricks

How did I convert the Wikipedia version of "Jabberwocky" into my character vector? If I had done it the Microsoft way, I could have used Word: open the original poem in Word, do a global replace to delete all punctuation and remove the extra line breaks between the poem's sections (stanzas), then convert everything to lowercase.

But instead I did it in R, using the following script. I first show the code, then explain how it works, using the commented line numbers for reference. The script assumes that you have already downloaded Jabberwocky_Original.txt and R's directory has been changed to the folder containing it.

```
jabberwocky.orig = readLines("Jabberwocky_Original.txt") # (1)
jabberwocky.orig = subset(jabberwocky.orig, jabberwocky.orig != "") # (2)
punctuation = c("\"","'","\\.",",","—","\\?","!",";",":","-") # (3)
for (punc in punctuation) { # (4)
  jabberwocky.orig = gsub(punc,"",jabberwocky.orig) # (5)
} # (6)
jabberwocky.orig = tolower(unlist(strsplit(jabberwocky.orig," "))) # (7)
write(jabberwocky.orig,"Jabberwocky_OnlyWordsNew.txt") # (8)
```

In line (1), I read in the poem line by line, using **readLines()** again, because again this file is not a data table. I decided to give the original version a new name, **jabberwocky.orig**, so I wouldn't write over the **jabberwocky** object that I put in R's workspace earlier in this chapter. R doesn't treat the "." symbol as anything special inside object names; since R distinguishes lower and uppercase, I could also have used **JabberwockyOrig** or something.

In line (2), I removed the line breaks between poem stanzas (i.e., all empty strings in the character vector **jabberwocky.orig**). We'll see the **subset()** function a lot in this book, since it's very useful for playing around with real datasets.

In line (3) I just list all the punctuation marks I noticed (with my own eyes) in "Jabberwocky", putting them into the character vector **punctuation**. Some of them have special usages in R's character functions, so to make them behave like normal characters, I had to **escape** them with the \ or \\ symbols (don't worry if this is confusing; we'll never need to do this again for the rest of the book, though it's crucial for linguists who do a lot of computer text analysis).

Lines (4) through (6) represent a **loop**: the **for()** function looks at each element in the character vector **punctuation**, names the element **punc**, does the operation within the **{ }** symbols on **punc**, and then outputs the results (unlike other computer languages, R loops normally don't output anything until the whole loop is finished). Notice that I indented line (5) a bit to show that it's inside the **{ }** symbols.

The operation in line (5) is basically like Word's global replace: the **gsub(X, Y, Z)** function ("global substitution") changes every instance of X into Y within Z. Since X here is **punc** and Y = **""** (the empty string) and Z is **jabberwocky.orig**, the effect is to remove all of the punctuation. (Note that if we didn't escape the "." symbol, **gsub()** would replace everything with nothing, since in string search functions, "." is a **wildcard** representing any character.)

Line (7) does three things in sequence, starting with the most deeply embedded function. The **strsplit()** function splits strings, in this case splitting all strings in the character vector **jabberwocky.orig** at the spaces, in other words, separating each line into separate words. (Excel can do something similar: select a column you want to split, then look around the menus for an icon showing one column of boxes being split into multiple columns). Because each line of the poem may have a different number of words, **strsplit()** does not create a vector (where all elements have to be the same type), but rather a different kind of object called a **list**, which is an ordered sequence of any type of object (in this case, each element is a different-length vector of words). The function **unlist()** then flattens the list into an ordinary vector, where each element is a word (character string). Finally, **tolower()** puts everything into lowercase.

Line (8) then saves our work. Just as R has several different functions for importing files, it also has several ways to export them. In this case, **write()** saves our character vector as a text file. If you really ran this code, look in your directory: do you see the new file that was created?

This little exercise highlights yet another difference between R and Excel: R can do loops. Loops are the conceptually simplest way to make a computer do the same stupid thing over and over again, so we don't have to. For example, the following **for()** loop sums the numbers from 1 to 5.

```
total = 0 # Initialize the variable that will contain the total sum
for (i in 1:5) {# Programmers like to use i as the index variable
  total = total + i      # Again, note the indenting, to keep track of what's inside what
}
total
[1] 15
```

Of course that's a dumb example, since we could have done the same thing a lot more easily with **sum(1:5)**. But it's not always obvious that a non-looping approach is possible. For example, the following code creates a set of ten random values between 0 and 1, finds the minimum, and replaces this value with 0. Run it to see what happens.

```
randvar = runif(10)      # "Uniform" random distribution, like Excel's =RAND()
newvar = randvar
for (i in 1:10) {
  if (randvar[i] == min(randvar)) { # min() is just like Excel's =MIN() function
    newvar[i] = 0       # Replace just this element, if it's truly the minimum
  }
}
cbind(randvar,newvar) # Bind the vectors into columns for comparison
```

But it turns out that it's actually possible to do this without looping, in a kind of tricky way; the results are the same as above:

```
newvar2 = randvar # So we don't lose the original newvar
newvar2[randvar==min(randvar)] = 0 # Tricky!
cbind(randvar,newvar,newvar2) # newvar = newvar2
```

How does the tricky step work? Remember that the **[ ]** brackets mark the indices for vector elements. You can refer to vector elements directly by number, as we did earlier when we displayed the first word in **jabberwocky** just by typing **jabberwocky[1]**. We could also refer to the first three words with **jabberwocky[1:3]**, or the first three odd-numbered words with **jabberwocky[seq(1,5,2)]**. But we can also refer to vector elements by using formal logic (i.e., the branch of math relating to truth, falsity, and operators like NOT, AND, and OR). That is, **vector[logical]** refers to all of the elements in **vector** for which the **logical** part is true. Thus the object **newvar2[randvar==min(randvar)]** refers to the element of the vector **newvar** such that it is true that **randvar==min(randvar)**; in plain English, it refers to **randvar**'s minimum element. When we then assign this object to have the value 0 (using **newvar2[randvar==min(randvar)] = 0**), the outcome is the same as the loop: the minimum value is replaced by zero.

The way R performs this magic is to treat the logical indices inside the **[ ]** as just another kind of vector, namely a **logical vector**:

```
1==1
[1] TRUE

typeof(1==1)
[1] "logical"
```

You can actually see the logical vector if you just type the logical part by itself: it's a list of TRUEs and FALSEs, showing when **randvar** is or is not equal to **min(randvar)** (the location of your TRUE might be different from mine, depending on the results of **runif()**).

```
randvar==min(randvar)
[1] FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

By the way, note that the all-caps **TRUE**s and **FALSEs** have no quotation marks: these are logical values, not strings. This is also why in the **sort()** function earlier, one of the arguments was **decreasing=TRUE**; I could have used **T** for short, but not **"TRUE"** or **true**.

You are right to think that this non-looping method is more confusing than the looping method, but try not to get too intimidated. R is still following its own logic: as I noted earlier in the chapter, the two key principles are just that R creates and manipulates objects, and that these objects are usually vectors. Just like objects in the real world, R objects have various components and properties; in the case of vectors, the components are the elements, which can be referred to in a variety of ways, including by means of logic.

Unfortunately, as you progress in your R skills, you'll learn that simple-minded looping can be much slower than the more confusing non-looping tricks. For example, R functions that take vectors as arguments can seem to produce output almost instantaneously:

```
a = 1:10000000 # Creates the vector a = (1,2,3, ... , 10000000)
b = sqrt(a) # Computes the square root for each value in a
head(b) # See first few results instantly
```

By contrast, if we try to do this same job with a loop, it goes noticeably slower:

```
b= numeric(10000000) # Create an all-zero vector of this length
for (a in 1:10000000) {
  b[a] = sqrt(a)
}
head(b) # See first few results after a delay
```

We can actually quantify how much slower the loop runs by surrounding each of the scripts above with the following two lines: the first stores the computer's internal clock time with **proc.time()** and the last subtracts it from the time after finishing the script. The third variable shows the total time difference, in seconds (the other two show how long R itself was running, ignoring all of your computer's other processes). The result below is what I got after running the looping script above, showing a total time of 0.7 seconds. By contrast, when I tested the vector-based script, it only required 0.14 seconds.

```
now = proc.time() # "now" is just my own clever variable name
# ... Insert the script you want to time here ...
proc.time() - now
   user  system elapsed
   0.12    0.09    0.70
```

So it really helps to mentally internalize the vector principle of R. This is kind of like how when you learn a second language, you have to restructure your brain a bit to think more like a native speaker.

One final difference between R and Excel worth mentioning here is that R easily lets you create your own new functions (in Excel this would require those macros that we're not discussing in this book). For example, suppose you want to determine if an **integer** (整數) is odd or even. One way to do this is to divide the integer by two, and see if the result is still an integer. This latter step can be done with the help of the function **floor()**, which removes the **decimals** (e.g., **floor(3.7)** = 3): if **x/2** is equal to **floor(x/2)**, **x** must be even.

Thus we can write a little script like so:

```
if (x/2 == floor(x/2)) {paste("x is even")} else {paste("x is odd")} # Remember x = 6
[1] "x is even"
```

But suppose you need to do this test a lot, as part of some larger project. Wouldn't it be nice if there were a function like **is.even(x)** that would output TRUE if **x** is even and FALSE if it's odd...? But unlike Excel (**=ISEVEN()**), R has no such built-in function.

No problem, we'll just create it ourselves:

```
is.even = function(x) {          # argument name x only within this function
  return(x/2 == floor(x/2))      # returns (outputs) a logical value: TRUE or FALSE
}
```

As the script comment says, the **x** is just used internally to define the argument; the function works the same way if you run it using a different variable name:

```
is.even(y) # Remember y = 3
[1] FALSE
```

Anyway, now we can use our newly invented function inside **if()**, for example like so:

```
if(is.even(y)) {paste("y is even")} else {paste("y is odd")}
[1] "y is odd"
```

New functions can be as complex as we need them to be, taking whatever arguments we want them to take. But after we create them, we just use them like regular functions. For

example, even though our invented **is.even()** function takes just one numeric argument, R's vector principle means that it will automatically be applied to entire vectors, testing the evenness of each element.

```
(1:10)[is.even(1:10)]
[1]  2  4  6  8 10
```

By the way, note the parentheses around the vector **1:10**; if we leave them off, R will try apply the **[]** index just to 10, as shown below. This will confuse R, since 10 is a one-element vector, so it doesn't have all the elements implied by the five-element Boolean vector **is.even(1:10)** part. The result will be a syntax error complaining about values that are **NA** (not available) or even **NaN** ("not a number"):

```
1:10[is.even(1:10)]
Error in 1:10[is.even(1:10)] : NA/NaN argument
In addition: Warning message:
In 1:10[is.even(1:10)] :
  numerical expression has 5 elements: only the first used
```

You can see where this error arises by trying this bit of nonsense:

```
10[is.even(1:10)]
[1] NA NA NA NA NA
```

In other words, as we saw before, R is too stupid to tell you exactly what went wrong, more like a crying baby than a friendly teacher. To find your mistake, you may have to do an internet search. Lucky you're a lot smarter than R or any other mere computer program. Just don't smash your computer while you're trying to figure it out.

4.3 Excel and R can play together

So far we've practiced using Excel and R separately, but I doubt I'm unusual in using both together. The visual, hands-on nature of the Excel interface makes it much easier to use it for certain things (like running quick analyses or making quick graphs), while the statistical programming language nature of R makes it essential for doing other things (like running fancy analyses or making complex graphs).

In particular, even if I mainly want to do my analyses in R, it's very useful to first take a look at my data in Excel, or even perform some basic adjustments to it. For that reason, it's very common for me to start my R analysis by copy/pasting data from an Excel file into a text file, and then loading the text file into R. (R actually has tools for loading in Excel files directly,

but since Excel cells can contain lots of non-R functions and formatting and such, this doesn't seem to me the best way to do it.)

To illustrate how this works, and to teach you some more really basic Excel and R concepts, let's look at the Excel file Jabberwocky_ExcelFreqs.xlsx. This contains the results of the manipulations I described earlier when we were calculating word frequencies. It looks like the following, since I've also created **headers** for the two crucial columns at the top. By the way, it's often useful to "freeze" the header row in a different way from our "freezing" above, namely by clicking the top row number (1), going to the View [檢視] menu, opening the Freeze Panes [凍結窗格] options and choosing Freeze Top Row [凍結首欄]. Then you can scroll down a long spreadsheet while keeping the headers visible.

	A	B
1	Word	Freq
2	the	19
3	and	14
4	he	7
5	in	6
6	jabberwock	3
...	...	...

Now let's load it into R. As noted above, the most flexible way to start this process is to first copy/paste from the Excel file into a text file; let's call it Jabberwocky_Freqs.txt. Notice that this process automatically separates the columns with **tabs** (that is, the big spaces you get when you hit the tab key [製表鍵]), but since the words aren't all the same length, in the text file the columns are all ragged. Don't worry; R will just look for the tabs so it doesn't have to look nice.

Since this file contains two columns (the words and their frequencies), we shouldn't use **readLines()**, since that creates a vector. Instead we'll load it into R as a **data frame**. Data frames are the usual way that R likes data; they are formatted like an Excel spreadsheet, except that each column must be a vector (of numbers or character strings or whatever) that you can name (as we did here). Since this is a **tab-delimited file**, we will use the function **read.delim()** (notice the dot - lots of R functions have dots in them), and as usual, we should give the loaded-in thing a name:

jabfreq = read.delim("Jabberwocky_Freqs.txt")

Maybe someday instead of a tab-delimited file you'll want to load in a **CSV file** (with **comma-separated values**), which separates the columns with commas rather than tabs; such files are produced by many programs that linguists use (since tabs don't always work the same way on different computer systems, but commas always do), such as PsychoPy, the free

software for designing and running psychology experiments (Peirce & MacAskill, 2018). In that case, you use the function **read.csv()**.

To see what we loaded in, we could type the name of the data frame, but it's got lots of lines, so if we just want to get a general idea of its contents, we could use the **head()** function to show the top six lines:

head(jabfreq)

	Word	Freq
1	the	19
2	and	14
3	he	7
4	in	6
5	jabberwock	3
6	my	3

So now R knows there's a data frame called **jabfreq** that contains columns called Word and Freq. It doesn't know Word and Freq by themselves, though:

Word
Error: object 'Word' not found

Freq
Error: object 'Freq' not found

If we want to refer to the columns (as we do in order to do anything with them), we could copy the **jabfreq** variables into R's main workspace by entering **attach(jabfreq)** and later, when we want to hide them again, entering **detach(jabfreq)**, but that can get confusing (especially when you're working with different data frames that happen to have variables with the same names). So most of the time, a better (though uglier) way to refer to data frame columns is to use the **$** symbol, as shown below (the syntax is always **X$Y**, where **Y** is an object "inside" the main object **X**). So typing **jabfreq$Word** will show us just the words and **jabfreq$Freq** will show us just their frequencies:

jabfreq$Word
```
 [1] "the"       "and"      "he"       "in"         "jabberwock" "my"
 [7] "through"   "all"      "as"       "beware"     "borogoves"  "brillig"
 ...
```

jabfreq$Freq
```
[1] 19 14 7 6 3 3 3 2 2 2 2 2 ...
```

Another useful thing to know about loading data frames relates to missing data. In real life it's common for something to go wrong (e.g., somebody skips an item in an experiment), leaving a "hole" in your neatly organized data set. In that case, when creating the data file in another program (Excel or otherwise), you can put **NA** (for "not available") in the cells with missing data (and this will happen automatically if R loads in a tab-delimited file with gaps in it, assuming the number of tabs is the same in all of the rows). After you load the data into R, you can do various things with the rows with missing data, including removing (omitting) these rows entirely with the **na.omit()** function. Since **RTdat** isn't missing any data, I'll demo this with a tiny data frame created using the **data.frame()** function. Note that **NA** is a type of object, not a string; note that the whole row is deleted, not just the NA itself; and note that the row ID number is deleted too.

```
dumb = data.frame(X=c("Noun","Verb","Adj"),Y=c(4,NA,6))
dumb
     X  Y
1 Noun  4
2 Verb NA
3  Adj  6

na.omit(dumb)
     X  Y
1 Noun  4
3  Adj  6
```

So that's loading data into R. How do we get the data out again? If you only want R to use it again later, you can save a lot of space on your computer if you create an R-specific file (unreadable by a text editor, Excel, or humans), by using the **save()** function. Of course, even our original text file was pretty tiny, so doing this doesn't make a big difference here:

```
save(jabfreq, file="jabfreq.R") # You don't need the ".R" filename extension; my habit
```

If we want to process this in R again later, you use the load() function, which puts an object into the workspace with the name you created earlier (which you may have forgotten, so look for it using ls()):

```
load("jabfreq.R")
```

But since we're talking about communication between R and Excel, how do we save an R result in a format the Excel can read? One way is to use the write.table() function, but since by default this separates by spaces rather than by tabs, you also need to specify tabs as the separator symbol, using the R-friendly code **"\t"** (that's a backslash, not a slash):

write.table(jabfreq, "jabfreq.txt", sep = "\t") # "\t" = tab (with backslash)

Now you can open this with a text editor and copy/paste the contents into Excel, or maybe even open it directly using Excel.

4.4 An important new R dialect: the tidyverse

Hopefully by now you agree with me that R can do some interesting things, but probably you also agree with me that it doesn't always do things in the most obvious or elegant way. This is inevitable with any computer system, so pretty soon somebody comes along with a new format that they hope will become standard, but inevitably not everybody likes that format either, so other formats get proposed, and so and so forth, until the situation becomes messy again, just at a higher level (for a classic cartoon about this problem, see: https://xkcd.com/927/). This applies to statistics textbooks too, of course, which is why I keep saying you shouldn't rely just on this one.

Anyway, the point of the story is that an influential data scientist and R programmer named Hadley Wickham (for how he got his influence in the first place, I'll tell you soon) has recently created what's quickly becoming a new "standard" version of R that he calls **tidyverse** (cute-ese for "tidy universe" of course); for the full details, see https://www.tidyverse.org/, along with Wickham & Grolemund (2016) and Wickham et al. (2019). As Wickham and his collaborates openly admit, it is an "opinionated" collection of packages that allows you to interface with confusing R functions in a (hopefully) less confusing way. Some linguists find that the tidyverse (i.e., the world created by the **tidyverse** package)
is so useful that they rely on it exclusively to teach statistics to their students (e.g., Winter, 2019), and indeed when you search for R help on the web nowadays, the helper often will assume that you use it too.

On the one hand, these recent developments don't make all the non-tidyverse textbooks and websites just disappear, so while searching for help you have to be ready to understand both the old R dialect and the new tidyverse dialect, which is annoying (see above cartoon again). On the other hand, the tidyverse basically just adds some new functions to help do stuff that you can also do without them, so it's not the end of the world if you don't use it. Indeed, my impression is that like other of the fancier corners of R, the tidyverse is really mainly aimed at high-powered data scientists, not ordinary linguists who just want to do some basic analyses (let alone their thesis committee members or conference reviewers or journal reviewers, who may not know how to use R at all). So... where appropriate in this book, I'll explain tidyverse stuff, but I won't insist that you use it.

So what makes the tidyverse tidy? It doesn't do statistics in a new way, but it does try to help humans better understand the summarizing and modeling of data (two of the key goals noted in chapter 1). After explaining how to get it, I'll review three subsets of its tools. For

linguists the most useful are those relating to character strings, next most useful are tools relating to the way you format and interact with data, and the least useful (at least for ordinary linguists rather than high-powered data scientists) are tools for automating the reporting and sharing of analyses.

4.4.1 How to install tidyverse (or any other R package)

Even if you never use **tidyverse**, it is very useful to learn how to install new packages into R, and we'll be doing this off and on throughout the whole book, so we may as well practice now.

Before installing **tidyverse** (or any other R package), you may as well first check to see if you already have it, by seeing if R can load its "library" of functions:

library(tidyverse)

If you get the following error, than you don't got it yet:

Error in library(tidyverse) : there is no package called 'tidyverse'

The easiest way to install a package in the R Windows version is to go to the **Package**s menu and choose **Install package(s)...**, which will give you a little pop-up window with a list of mirrors (as usual, my Mac readers will have to figure out the equivalent for their system, including Mac-specific problems like packages that download but that R somehow can find - Windows sometimes has this problem too, and you just have to try various pieces of internet advice until it works, starting with https://cran.r-project.org/bin/windows/base/rw-FAQ.html#Packages).

When installing new packages, the first mirror listed has the mysterious name "0-Cloud": that's actually run by the RStudio company (but if you're actually running RStudio you would be installing packages using the RStudio menus anyway: it's in **Tools > Install Packages**). But you can also choose to be patriotic and use a local mirror, like "Taiwan (Taipei)" (Taichung used to have one but I guess they gave up). After you've chosen the source you want to use, you'll see an extremely long list of R packages. The list is in alphabetical order, so just keep scrolling until you find the package you want, in this case **tidyverse**. This package is just a package of other packages, so it will install those too, as you'll see from a bunch of text scrolling down the screen.

Again, this shows how clicking a menu option is really just short-hand for running command lines. In fact, you can get the same result by typing the following (the disadvantage

is that you'll have to memorize this specific function - note the plural "s" even though we're only installing one package - instead of just remembering which menu items to click).

install.packages("tidyverse")

Fortunately, you only need to install a package once (theoretically even after updating the whole R program, though doing this may also require checking that FAQ). Package creators often update their creations, though, so you can also check for updates with the **Packages/Update packages...** menu option. Every time you use a non-base package, just start it up with the function **library()**, as I'll illustrate throughout the rest of the book.

Now that you know how to install packages, I'd like to remind you that you might also want to try installing the **Rcmdr** package, which runs the Rcommander tool that I mentioned earlier. Rcommander provides R with a more user-friendly GUI, with menus more similar to those in Excel or SPSS. It even nativizes to your computer's default language (you can change this to a different language by choosing Edit from R's main menus, selecting **GUI preferences...**, then typing your preferred language into the box at the upper right). After installing **Rcmdr**, load it with **library(Rcmdr)**, and if you happen to close it while keeping R open, you can restart it with the function **Commander()**. As I said, I won't use Rcommander in this book, but if you like it, go ahead and use it. Note also that the philosophy of **Rcmdr** is the total opposite of that of **tidyverse**, namely visual but limited old-timey-stats style vs. abstract but powerful data science style.

4.4.2 Dealing with Unicode problems

Two of the packages in the tidyverse are called **readr** and **stringr** (the latter based on an older and more complex package called **stringi**; Gagolewski, 2021); as the names suggest, these are packages for reading in data and working with character strings, linguists' favorite things. Crucially, this means the tidyverse helps R work well with Unicode, including Chinese characters.

In order to illustrate the problems that base R has with Unicode, we need a file with non-ASCII symbols, so we have to take a break from "Jabberwocky" for a bit. We'll work with the file TsaiFreq.txt, which contains 13,058 Chinese distinct characters (in the "Char" column) and their frequencies (in the "TsaiFreq") column, derived from a 171,882,493-character corpus compiled by Chih-Hao Tsai (http://technology.chtsai.org/charfreq/). By the way, if you view this file on the web the characters may already look garbled, but that's because browsers are dumb too; it's actually in perfectly good Unicode, as we'll see eventually below.

Let's first see what happens in English Windows, which is what I use. We go to load in the data frame like so, naming it **tf** (for Tsai frequency):

tf = read.delim("TsaiFreq.txt")

No error message, so all seems well. But then we look inside it:

head(tf)

	Char	TsaiFreq
1	çš„	6538132
2	æ˜¯	3200626
3	ä¸\u008d	2831612
4	æ^‘	2584497
5	ä¸€	2542556
6	æœ‰	2289333

What the heck...? R can't read Chinese, so it turns everything into nonsense. It does particularly badly with the third character, where actually spells out a Unicode code number (though even that is garbled, as we'll see in a moment). And all of these are super-common characters too, so they're listed from most to least frequent. Pretty pathetic, R!

Before I learned about **stringr**, I created some functions for reading and writing Unicode files and put them up on my homepage. If you want to try them, just type in **source("http://personal.ccu.edu.tw/~lngmyers/UnicodeTools.R")** - yes, the **source()** function that I mentioned earlier can actually run code from the internet. This will add three new functions to your workspace: **readLines.uni()**, which loads Unicode text as a single string as we did with "Jabberwocky", **read.delim.uni()**, which loads a tab-delimited Unicode data frame file, as we'll be using in most of the rest of this book, and **write.uni()**, which saves Unicode text or a data frame as text file.

These functions work OK in English Windows. For example:

source("http://personal.ccu.edu.tw/~lngmyers/UnicodeTools.R") # Just do this once
tf = read.delim.uni("TsaiFreq.txt")
head(tf)

	Char	TsaiFreq
2	<U+7684>	6538132
3	<U+662F>	3200626
4	<U+4E0D>	2831612
5	<U+6211>	2584497
6	<U+4E00>	2542556
7	<U+6709>	2289333

Don't worry about the Unicode codings shown here; that's not the fault of my functions, but of how R displays data frames in English Windows. The contents themselves are OK, as we can see if we just look at the Char column (I'll explain the "levels" thing in a later chapter):

```
head(tf$Char)
[1] 的 是 不 我 一 有
13058 Levels: ...
```

There are still some serious problems, however. With very very low frequency characters (even rarer than those at the bottom of the Tsai frequency list), R keeps them in Unicode coding and never shows them as characters at all, and in fact can't even count them correctly if they're in a string. Even worse, if you use Chinese Windows, my functions don't work at all. (And as usual, I don't know anything about Mac R, but I'm not optimistic.) Whether you use the usual **read.delim()** or my special **read.delim.uni()**, you get a fatal error, with the message saying R can't recognize certain characters from the very first line (and indeed, as shown by the English message below, this happens even if you use English R in Chinese Windows):

```
tf = read.delim("TsaiFreq.txt")
Error in type.convert.default(data[[i]], as.is = as.is[i], dec = dec,    :
  invalid multibyte string at '<e7><9a><84><09>6538132'
```

This is the kind of nonsense that special-purpose file-reading and string-processing packages were invented to solve. Specifically, using the **tidyverse** package's **readr** package, we can get the results we want, in both the English and Chinese versions of Windows. Instead of using base R's **read.delim()** function, we the tidyverse's **read_delim()** function (note that **tidyverse** function names never contain dots, but many contain underlines, making them visually distinct from base R functions). Technically, **read_delim()** creates a data frame that's inside a special wrapper that I'll explain in the next section, so to avoid explaining that stuff now, we'll also use the function **as.data.frame()** to convert it back to an ordinary data frame:

```
library(tidyverse) # You only have to do this once per R session
tf = as.data.frame(read_delim("TsaiFreq.txt")) # The key package here is readr
```

Both of the above commands will generate a bunch of tidyverse gobbledygook, but you can ignore all that. The main thing is that everything is OK with our Unicode now. In fact, if we look inside this data frame in Chinese Windows, we can even see the characters directly:

```
head(tf)
```

	Char	TsaiFreq
1	的	6538132
2	是	3200626
3	不	2831612
4	我	2584497
5	一	2542556
6	有	2289333

If you want to other things with strings, in Unicode or otherwise, the many functions in the **stringr** package may make things easier than with base R; see the cheat sheets at https://stringr.tidyverse.org/.

4.4.3 Tables vs. tibbles

As noted in the previous section, the tidyverse actually wraps data frames in what it thinks is a simpler-looking object, even though deep down it's still just a data frame. Consistent with the cutesy style of the tidyverse, this kind of object is called a **tibble** (sounds like "table", get it?), and that's also the name of the package inside the **tidyverse** package that handles them. I won't use tibbles in this book, but since you may encounter them on the web and other textbooks (particularly Winter, 2019), I thought I should explain them anyway (see the creator's introduction at https://r4ds.had.co.nz/tibbles.html). Moreover, the tidyverse also comes with the package **dplyr** (sounds like "d(ata frame) pliers", get it?), which has some useful functions for manipulating both tibbles *and* ordinary data frames (see the introduction at https://r4ds.had.co.nz/transform.html).

The main difference between tibbles and ordinary data frames is how they are displayed. To see this, let's go back to the Tsai frequencies and load it as a tibble this time (i.e., without using **as.data.frame()**):

```
library(tidyverse) # Remember, you only have to do this once per R session
tft = read_delim("TsaiFreq.txt") # Final "t" for "tibble"
```

When we use the **head()** function on the tibble, we get pretty much the same thing as before, except now it tells us the dimensions of the thing we have created (6 rows and 2 columns; R always mentions rows before columns) and the types for each of the variables (Char is a character vector and TsaiFreq is a "double" [= ordinary number] vector).

```
head(tft)
# A tibble: 6 x 2
  Char  TsaiFreq
  <chr>    <dbl>
1 的     6538132
2 是     3200626
3 不     2831612
4 我     2584497
5 一     2542556
6 有     2289333
```

However, we don't even need to use **head()**, since unlike the case with data frames, when you type the name of the tibble it just shows you the top few rows:

tft

```
# A tibble: 13,058 x 2
   Char  TsaiFreq
   <chr>    <dbl>
 1 的     6538132
 2 是     3200626
 3 不     2831612
 4 我     2584497
 5 一     2542556
 6 有     2289333
 7 大     1891383
 8 在     1715554
 9 人     1598855
10 了     1507218
# ... with 13,048 more rows
```

If you do want to scroll through the entire tibble, you have to use a totally different command:

print(tft, n = Inf) # Inf = infinity, a number that R actually recognizes

Tibbles also don't want to show you all of your columns if you have too many, so in case you do want to show all columns, you would do this:

print(superwidetibble, width = Inf) # For a super-wide tibble (if you had one)

While tibbles themselves don't seem all that useful to me, most of the other tidyverse functions that process tables produce tibbles as output, so you have to be ready in case one pops up. Moreover, the functions in the **dplyr** sub-package do seem useful, and they work for ordinary data frames as well. Let me illustrate the five main ones here on the data frame **tf**, comparing how they work with the way base R would do the same things. I won't use them in the rest of the book, but that doesn't mean you can't use them!

The first function lets you add new columns by changing ("mutating") other column variables, which we'll be doing a lot later on. In particular, we will often want to take the logarithm (對數) of frequencies, for reasons I'll discuss in a later chapter. The R function for this is **log()**. Here's the ordinary way to do it:

tf.orig = tf # Let's save the original version so we can demo each function separately
tf$LogFreq = log(tf$TsaiFreq)

head(tf)

	Char	TsaiFreq	LogFreq
1	的	6538132	15.69316
2	是	3200626	14.97886
3	不	2831612	14.85636
4	我	2584497	14.76504
5	一	2542556	14.74868
6	有	2289333	14.64377

Now here's the tidyverse way:

tf = tf.orig # Go back to the original for this demo
tf = mutate(tf, LogFreq = log(TsaiFreq))
head(tf)
[... same as above...]

The second function lets you select a subset of columns. For the ordinary method, remember that in R, rows are always specified before columns, so if you put nothing before the comma inside **[]** that means you want to select all of the rows. So here's how to look at just the frequency columns:

tf.freqs = tf[,c("LogFreq","TsaiFreq")] # Or tf[,2:3], i.e. columns 2 through 3 for all rows
head(tf.freqs)

	LogFreq	TsaiFreq
1	15.69316	6538132
2	14.97886	3200626
3	14.85636	2831612
4	14.76504	2584497
5	14.74868	2542556
6	14.64377	2289333

Here's the tidyverse way:

tf.freqs = select(tf, LogFreq:TsaiFreq)
[... same as above...]

The third function lets you filter out rows by a logical variable. Here's the ordinary way to do this, filtering out just the rows with the lowest frequency:

tf.lowestfreq = tf[tf$TsaiFreq==min(tf$TsaiFreq),] # Gap after comma = all columns

tf.lowestfreq # Not many of these, so we can list them all

	Char	TsaiFreq	LogFreq
13052	踧	4	1.386294
13053	紂	4	1.386294
13054	縪	4	1.386294
13055	薆	4	1.386294
13056	顁	4	1.386294
13057	鶹	4	1.386294
13058	鶔	4	1.386294

Here's the tidyverse version, with the only difference being that it doesn't use the original row numbers:

tf.lowestfreq = filter(tf, TsaiFreq == min(TsaiFreq))
tf.lowestfreq

	Char	TsaiFreq	LogFreq
1	踧	4	1.386294
2	紂	4	1.386294
3	縪	4	1.386294
4	薆	4	1.386294
5	顁	4	1.386294
6	鶹	4	1.386294
7	鶔	4	1.386294

The fourth function lets you order ("arrange") the rows. Here's how to do it in the ordinary way, arranging the rows from least to most frequent:

tf.sorted = tf[order(tf$TsaiFreq, decreasing = F),] # Decreasing is default; here increasing
head(tf.sorted) # Reorders row numbers too

	Char	TsaiFreq	LogFreq
13052	踧	4	1.386294
13053	紂	4	1.386294
13054	縪	4	1.386294
13055	薆	4	1.386294
13056	顁	4	1.386294
13057	鶹	4	1.386294

And here's how tidyverse does it:

tf.sorted = arrange(tf, TsaiFreq) # Ascending is default; cf. desc(TsaiFreq) for descending

head(tf.sorted) # Doesn't reorder row numbers

	Char	TsaiFreq	LogFreq
1	踧	4	1.386294
2	紨	4	1.386294
3	繹	4	1.386294
4	薆	4	1.386294
5	顁	4	1.386294
6	鷦	4	1.386294

The fifth and final major function lets you create a summary table from your data. This makes most sense when your data has groups, so let's split the Tsai frequency data into groups of characters above and below the mean log frequency, and then calculate the mean for each group. Here's the ordinary way:

tf$Group = "Upper" # We'll fill in the "Lower" part shortly
tf$Group[tf$LogFreq <= mean(tf$LogFreq)] = "Lower" # Less than or equal to mean
head(tf) # You check yourself!
tail(tf) # You check yourself!

Just for your information, here's one way to do this in one line in the tidyverse way (can you figure out how it works?):

tf2 = mutate(tf, Group=c("Upper","Lower")[1+(LogFreq <= mean(LogFreq))])
head(tf2) # You check yourself!
tail(tf2) # You check yourself!

Now let's compute by-group means in the ordinary way:

tapply(tf$LogFreq, tf$Group, mean)

Lower	Upper
2.665762	8.112729

And here's the tidyverse way; the main differences are that the output is a tibble and that the displayed values are rounded a lot more (this is just for the display - the actual values are the same as above).

summarize(group_by(tf, Group), Mean = mean(LogFreq)) # "Mean" is my own variable
 # Note: if you're from New Zealand like the tidyverse creator, you can use "summarise()"
A tibble: 2 x 2

	Group	Mean
	<chr>	<dbl>
1	Lower	2.67
2	Upper	8.11

In the tidyverse, the tibble philosophy also applies to other types of tables encountered in R, in particular those reporting statistical results. But we'll save that for a later chapter!

4.4.4 Workflow

In order to make a report for our fellow human beings, we have to get the statistical or modeling results, and before that we have to get the data, and doing that requires using a computer program like R. Computer experts call this kind of process the **workflow**. Before explaining how tidyverse advocates like Wickham & Grolemund (2016) and Winter (2019) want us to do this, let me first explain how I personally do it.

As I emphasized above, two of the biggest advantages of R over Excel are that your script lets you automatically do a series of complex things (since it's a computer language) and that you have a permanent record of what you did (since the script is written down). The automating aspect even applies to finding and saving the data on your computer, even if they are in more than one folder (e.g., if your raw experimental results fill up one folder so you want to put the overall analysis in a separate place to avoid mixing up different types of files).

So what I usually do is write my script piece by piece, testing it each piece until I'm sure it works, and then I put all the pieces together, run it, and save the results (sometimes just copy/pasting it into my R script file, with # marks since it's not code), and later when I type up the report for my fellow human beings, I copy/paste these results, including graphs, into a Word file. That's my way, but it's not Wickham's or Winter's, and it doesn't have to be yours.

Regarding finding files, say I have a folder called MyStudy and inside this is a folder called MyData and another folder called MyAnalyses. I put my R script into MyAnalyses but I want it to look inside the MyData for the data file experiment.txt and then save the results in MyAnalyses. So I first use R's menu to set the directory to MyStudy, and then I write a script like this:

```
exp.data = read.delim("MyData/experiment.txt") # Look inside MyData for data
# ... do analysis, creating stats.results
write.table(stats.results,"MyAnalyses/results.txt") # Write results in MyAnalyses folder
```

I can even move up a folder by using **".."**. For example, if my working directory has been set to MyAnalyses using the R menu, I can still reach MyData like this:

```
exp.data = read.delim("../MyData/experiment.txt") # Go up and then down into MyData
```

This works perfectly (at least in Windows; things may work differently in Mac or Linux systems, but I don't think so). Even if I send a colleague the folder MyStudy, containing

MyData and MyAnalyses, that person should have no trouble running my code, since the locations of all the files are all relative to each other, not absolute for just my computer.

So how do Wickham, Winter et al. want us to do things? The first thing they recommend, since they are computer nerds, is to make R more like Unix (or Linux), the computer operating system, by adding the concept of the **pipeline** (管道). This takes an object, puts it into the first function in the pipeline, then puts that result into the next function in the pipeline, and so on. What's better for good workflow than a good pipeline?

In Unix this operation is represented by the character "|" because it looks like a pipe, but since this symbol already has other purposes in R (as we'll see later), and since the next-best symbol would be ">" (like an arrow) but that's already used too (as we've already seen), and since R has other operators that surround already-used symbols with "%" on each end, the result is that that tidyverse pipe symbol is the awful-looking **%>%**. This is handled within tidyverse by the **magrittr** package, which again its a joke name, this time coming from the Belgian surrealist painter René Magritte [雷內·馬格利特] due to his famous painting of a pipe: https://en.wikipedia.org/wiki/The_Treachery_of_Images.

To make our discussion less surreal, if you have the functions **fun1()** and **fun2()** and the objects **X**, **Y**, **Z** (even if **Y** and **Z** are empty, i.e. you only have object **X**), then this...

```
X %>% fun1(Y) %>% fun2(Z)
```

... does the same thing as this (note that the object before **%>%** gets treated as the first argument of the function after **%>%**)...

```
X1 = fun1(X,Y)
fun2(X1,Z)
```

... or in one line...

```
fun1(fun2(X,Z),Y)
```

The second version uses a temporary variable that you have to memorize while reading the script even though you don't really care about it, while the third version is totally confusing to read. That's why many programmers like piping.

Here's a real example you can run:

```
library(tidyverse) # Again, you only need to do this once per session
x = c(3,1,4,1,5) %>%      # Put %>% at the end of each line to make reading even easier
  -5    %>%    # Subtract 5 from all elements in the vector (indenting helps readability)
  rep(2)    %>%    # Repeat the vector 2 times
  mean       # You don't even need () for one-argument functions
```

```
x # First line assigned a value to x, so result is a new value for x
[1] -2.2
x = c(3,1,4,1,5)       # What if we define the object before we start piping?
x    %>%               # Now we start the pipeline here
  -5    %>%
  rep(2)    %>%
  mean                 # It outputs the same result, but...
[1] -2.2

x                      # ... the original object isn't changed
[1] 3 1 4 1 5
```

This is doing the same thing as this ordinary R code:

```
x = c(3,1,4,1,5)
x1 = x # So we don't change the original object
x1 = x1-5
x1 = rep(x1,2)
mean(x1)
[1] -2.2
```

I won't use pipelines in this book, but if you think they're for you, go for it. You might see it on the web or in other books anyway.

The other big thing that tidyverse experts want you to do with your analyses and reports is to "knit" them into Web- and Word-friendly HTML files using the **knitr** package (Xie, 2015). This is easiest to do if you're working in RStudio (there are special-purpose menus for it) but it can also be done in ordinary R (https://yihui.org/knitr/). I've never tried it, though, so if you're curious, search the internet for help. As I said, I'm currently happy with copy/pasting all my stuff by hand.

5. Graphs

I end this survey of the fundamentals of Excel and R with a pretty big topic in itself: how to make graphs. Then I apply some of these graphing methods to our "Jabberwocky" data again (in case you haven't forgotten about it after all this time).

5.1 Graph basics

Human beings, like our primate relatives, are visual animals: a picture is worth a thousand words. That's why **graphs** or **plots** are so useful for expressing scientific information. This is not only to communicate information to the audience of your report, but also to communicate information intuitively to yourself, while you're still doing your research. There have been

times when I've broken my head trying to figure out why my statistical analysis was coming out so weird, and only when I make a graph I see that my assumptions about the data were all wrong. So I've learned my lesson: make graphs all along the way, not just at the last minute.

Graphs are more intuitive than **tables**, but they may also be less precise. Use tables instead of graphs if you don't have a huge amount of data, and/or the precise values are important to what you want to communicate. Don't give the exact same information in both a table and a graph: it wastes space and misleads your reader into thinking that there must be some crucial difference between them, when there really isn't.

The choice of graph depends partly on the **measurement scale** of your data. The scale also affects the choice of statistical analysis (as we'll see in later chapters).

A **nominal scale** puts things into different categories (*nominal* = "relating to names") without claiming that there is an order or values between the categories. For example, in a child language study you could divide children into girls and boys, and divide the words they produce into nouns, verbs, function words, and "other". All other types of scales used in statistics are **quantitative**, where there is a natural numerical way to describe each item.

An **ordinal scale** is based on ranking, not on the specific number associated with each item. For example, you could rank languages by the number of speakers, or the degree of acceptability of a sentence on a **Likert scale** (named after American psychologist Rensis Likert [1903-1981]), from 1 = impossible to 7 = perfect. The numbers in an ordinal scale are always positive integers, so values like 3.5 don't make any sense.

C**ontinuous** scales (e.g., where 3.5 would make sense) are probably the most common in statistics, especially the **ratio scale**, which has a zero point so that ratios can be defined (e.g., 4 is twice as big as 2, with 0 as the baseline). For example, **reaction time** (RT) is the number of milliseconds (ms) between a stimulus and a response in a psycholinguistic experiment, with 0 ms being the lowest logically possible RT.

Line graphs (折線圖) are almost always restricted to numerical scales like ordinal or continuous ones. For example, the Southern Min graph in chapter 1, repeated below in Figure 1, is an appropriate line graph because the points form a natural order (shorter to longer utterances).

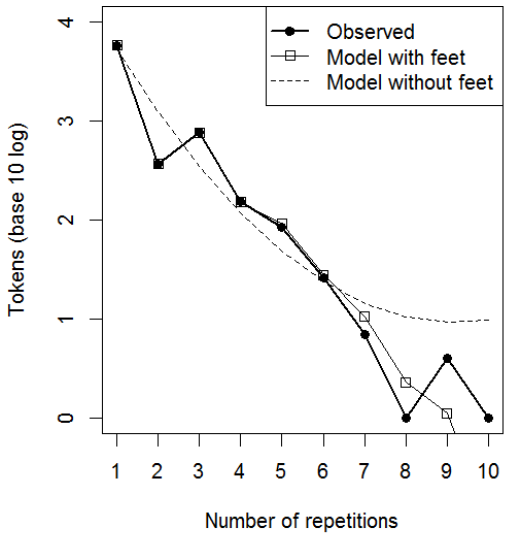

Figure 1. A good line graph (from Myers & Tsay, 2015) (made in R)

By contrast, the graph in Figure 2 is bad, since each point represents a different experimental participant, so not only is there no natural ordering across the people, but it makes absolutely no sense to talk about measurements that somehow lie "between" them.

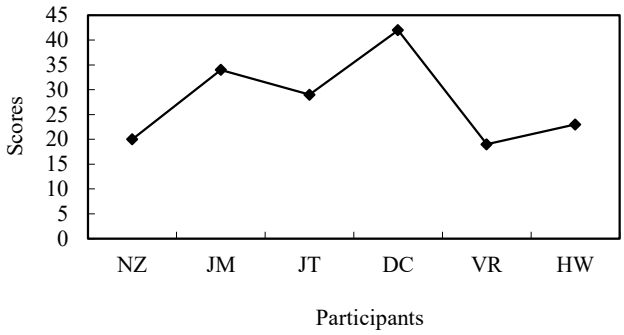

Figure 2. A bad line graph (made in Excel)

The data in Figure 2 should instead be plotted with a **bar graph** or **bar plot** (長條圖), where each bar represents a distinct category on a nominal scale. Further examples are shown in Figure 3, where the nominal categories are word types or experiments. A bar plot of a frequency table is called a **histogram** (直方圖、矩形圖), which will become very important starting in the next chapter.

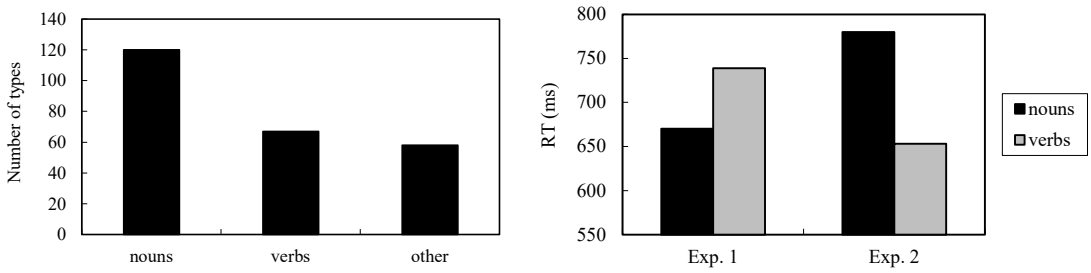

Figure 3. Good bar graphs (made in Excel)

Since later we'll redraw the bar graph on the right of Figure 3 in R, let me explain how I made this version in Excel. First, I typed in my fake data, along with labels at the top and left, like so (if this were real data, I could have computed it using cell functions, perhaps with **=AVERAGE()** across a sample of real reaction times). Then I just selected this 3 × 3 range of cells, poked around Excel's menus (e.g., under **Insert**), clicked the icon that looks like a bar plot, and then adjusted the colors and borders and added the y-axis label.

	A	B	C
1		Exp. 1	Exp. 2
2	nouns	670	780
3	verbs	739	653

Note that I made sure the colors in my bar graph print clearly on black-and-white printers. Believe it or not, some people like to print things out when they want to read them carefully, and most people don't have access to a color printer. It's fine to use colors for electronic displays, including in presentation slides, but try to make sure they don't all end up looking the same when you print them in black and white.

Another important type of graph is called a **scatter plot** (散佈圖), where each dot represents a single entity that has two continuous numerical values. The graph in Figure 4, for example, shows fake data for a bunch of fake adults, each of whom has an age and a vocabulary size. Scatter plots are especially important when looking for **correlations** (and as I mentioned in the first chapter, correlations are closely related to **modeling**, one of the key jobs of statistical analysis). Thus the scatter plot in Figure 4 seems to show that there is no correlation between age and vocabulary: no matter what age you choose on the **x-axis**, the average location of the dots remains about the same on the **y-axis** (vocabulary size).

A scatter plot also reflects another key concept in statistics: **variability**. Even if there were a pattern here, the dots would still form a cloud, just a cloud with some sort of shape or direction. (Note that the apparent dip in vocabulary size around 40 years is due to pure **randomness**: as I said, I faked these data to make sure there was no real pattern in them.)

Figure 4. A scatter plot (made in Excel)

5.2 "Jabberwocky" in histograms and scatter plots

Let's end the chapter by seeing what histograms and scatter plots can tell us about "Jabberwocky".

As I just noted, a histogram is a bar plot for a frequency table. We already created the frequency table for "Jabberwocky", twice in fact, once using Excel and then again using R. How can we plot it as a histogram?

In later chapters we'll see that both Excel and R have special functions for plotting histograms, but only for continuous scales (divided up into so-called **bins**, to make the discrete categories needed for a bar plot). In our "Jabberwocky" analysis, the entries in our frequency table are separate words, so our data are already on a nominal scale; neither Excel and R have special functions for plotting histograms in this type of case.

No problem; we can just plug the frequencies into an ordinary bar plot. This is much easier to do in Excel than in R (in fact, all of the figures above were made in Excel, except for the Southern Min one, which was made in R). This is unsurprising because unlike R, where even to draw pictures you have to write out the instructions as a script, Excel has a visual-style graphical user interface (GUI), where the user mainly pokes around in menus, so that making graphs in Excel feels more like drawing than writing.

So to turn our "Jabberwocky" frequency table (sorted from most to least frequent word type) into a histogram, all we have to do is select the columns containing the words and the frequencies, find the menu item for inserting graphs, choose the icon that looks like a bar plot, and we're pretty much done. We're not totally done, though, because the easiness of Excel also means that it treats you like a big baby, making lots of decisions for you, and you may have to fight back to make it do what you want. For example, it may want to insert a **legend** (explanation box for a figure: 圖例) identifying your variables, but in this case we only have one variable (the words), making a legend totally useless here. It's also up to you to tell your readers what the x-axis and y-axis represent, and maybe to give a title to the whole thing. And maybe change the colors too, including the color of the plot border.

When I did all that, I ended up with the graph in Figure 5. Note that even despite all my slapping, Excel still decided not to show all of the words on the x-axis, but I can't really blame it; it just ran out of room. (By the way, you can also create graphs in Word, but since its graphing functions are more limited than Excels, the non-R graphs in this book were all made in Excel and pasted into Word.)

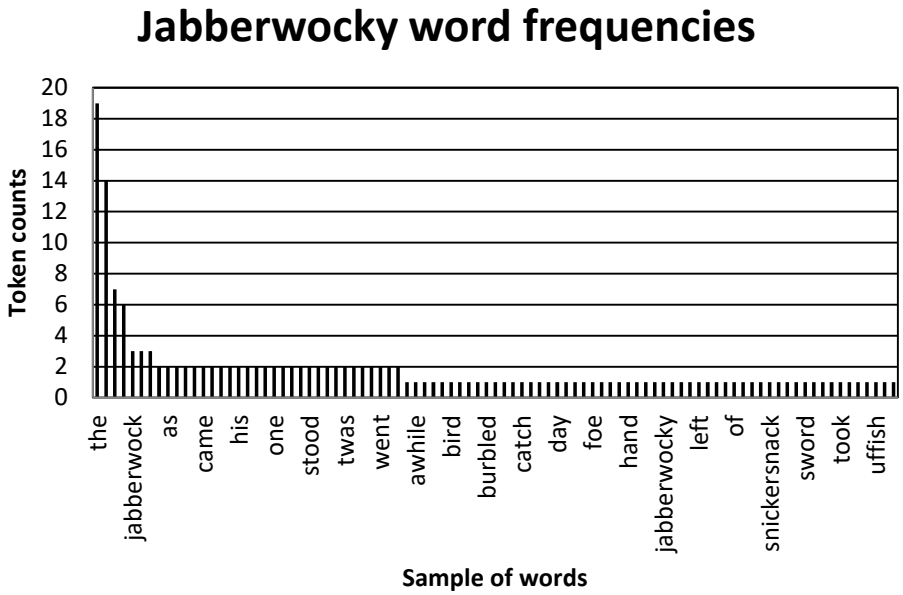

Figure 5. "Jabberwocky" word frequencies

If you are paying attention so far, you should be asking yourself one question right now: So what? Well, a histogram shows how our data are **distributed**, that is, where most of the data are and how they vary. This particular histogram makes a simple truth quite clear: most of the lexical items in "Jabberwocky" are rare (all those stretching way out on the right side of the graph), and only a few are common (like "the" and the few other words squeezed onto the left side). It turns out that this is what word frequency distributions always look like, even in huge corpora. This generalization (called **Zipf's law**, after a guy named Zipf; see, e.g., Baayen, 2001) will be discussed again in later chapters, but there are a couple of implications I can mention now.

First, this distribution shape is what makes it so annoying to learn a second language: the vast majority of the words that you have to learn are so rare that you'll hardly ever need to use them. In fact, no matter how many words you learn, there will always be even rarer words out there yet to learn. Wouldn't it be nice if every word was about equally common, and then all the effort you put into learning each one would be repaid by an equal amount of usefulness? Nope, sorry the world doesn't work like that.

Second, this distribution doesn't look anything like the famous **bell curve**, and the bell curve is what many statistical analyses assume that your data distribution is shaped like. So if

your data are actually distributed like Figure 5, you had better either use some other type of statistical analysis, or figure out how to **transform** your data to make it a bit more bell-like. I'll explain both approaches in later chapters.

Making a bar plot in R is a bit more annoying, but because R doesn't treat you like a big baby, you can get it to do exactly what you want (as long as you don't smash your computer during the process). For example, to replicate the Excel bar plot on the right side of Figure 3, we first put the (fake) RTs into a **matrix** (矩陣; plural **matrices**), which is like a rectangle of values (like a 2D, 3D, or any-D vector; a regular vector is like a 1D matrix).

```
results.mat = matrix(c(670,739,780,653),nrow=2)
rownames(results.mat) = c("nouns","verbs")
colnames(results.mat) = c("Exp. 1","Exp. 2")
```

As usual, you can see what the matrix object looks like by typing its name:

```
results.mat
       Exp. 1 Exp. 2
nouns     670    780
verbs     739    653
```

As the above code shows, the properties of a matrix object include the row and column **names**, which you can change separately from the numerical values. This trick is also used by the **table()** function, which creates a vector of frequencies where each element gets its name from the original character vector elements; you can extract these names using the **names()** function (as we'll do later below).

After putting the data into a matrix, we can make the bar plot from it, using the **barplot()** function. The two versions below are exactly the same, but in the first one I split it up so I can show you comments explaining each step. The result, which pops up in its own window (if you're following along in R, as you should be), is shown below in Figure 6:

```
barplot(results.mat,                    # table of values
  beside=T,                             # draw bars next to each other, not on top
  names.arg=c("Exp. 1","Exp. 2"),       # the names at the bottom
  legend.text=c("noun","verb"),         # the names in the legend box
  ylim = c(0,1100),                  # min & max y-axis (so legend doesn't cover bars)
  ylab = "RT (ms)"                          # y-axis label
)

# Same thing, but not split up (you can type it all as one line; R will wrap it for you)
barplot(results.mat, beside=T, names.arg=c("Exp. 1","Exp. 2"),
  legend.text=c("noun","verb"), ylim = c(0,1100), ylab = "RT (ms)")
```

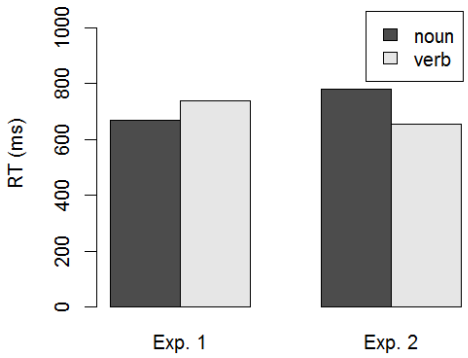

Figure 6. Basically the same as Figure 3, but using R instead of Excel

Notice that unlike Excel, the bars in R bar plots always start from zero, and in fact it is quite difficult to change this default (for one way to do get around the default, see the cheat sheet StatsFunctions.pdf). This limitation is built into R on purpose: the height of each bar should reflect the actual measurement for the bar's category, so putting the bottom of the graph higher than zero can exaggerate what is actually a small difference. So the pattern looks much more dramatic in Figure 3 than in Figure 6, even though both graphs plot exactly the same data. Lying with statistics indeed!

Now, to make the histogram for the "Jabberwocky" frequencies in R, we simply take the same R script that we just wrote and change only what we need to change, creating Figure 7:

```
jabberwocky.freq = sort(table(jabberwocky), decreasing=TRUE)
barplot(jabberwocky.freq, beside=T, names.arg=names(jabberwocky.freq),
  main="Jabberwocky word frequencies", # Main title for plot
  xlab = "Sample of words", # x-axis label
  ylab = "Token counts") # y-axis label
```

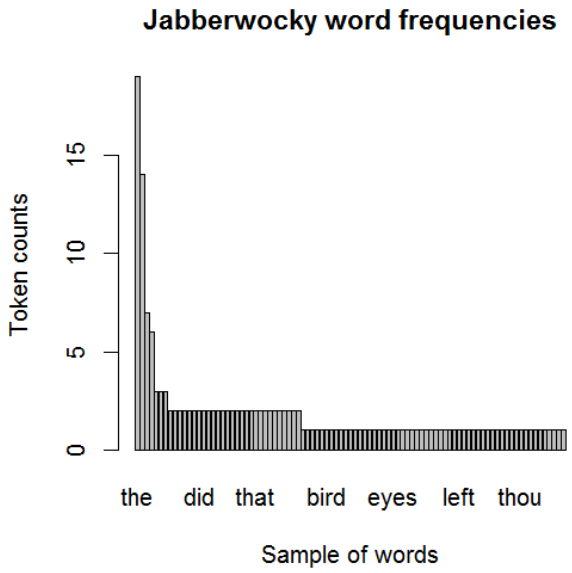

Figure 7. R's version of Figure 5 ("Jabberwocky" word frequencies)

Since histograms, and the frequency tables they come from, are so important to statistics, R has an even simpler way to create Figure 7: just use the generic **plot()** function. This function is context-sensitive in that it behaves differently depending on what argument you put in it, sort of like "taking a bath" and "taking a walk" are very different kinds of taking. So if the argument is a table object like **jabberwocky.freq**, **plot()** will assume that you want to plot a frequency table as a histogram. Try the alternative commands below to see what happens!

```
plot(jabberwocky.freq) # Histogram without labels
plot(jabberwocky.freq, main="Jabberwocky word frequencies",
  xlab = "Sample of words", ylab = "Token counts") # Include same labels as Fig. 7
```

Let's turn now to scatter plots, which is another of R's favorite kind of graph. Why? Probably because of the central role that modeling plays in statistics, and by showing you how the variable on the x-axis relates to the variable on the y-axis, a scatter plot is showing you a model of how the two variables are related (e.g., how well you can predict y from x). In fact, scatter plots are the default plot type in R: the **plot()** function will draw a scatter plot unless you tell it otherwise. For example, run the following code, and you get a scatter plot with the random vector on the y-axis plotted against their indices (**1:100**, for the 100 random numbers) on the x-axis (the exact results depend on what random numbers you get when you run **runif(100)**):

```
plot(runif(100))
```

Just for fun, you can put this inside a loop and make some animated snow!

```
for (i in 1:100) {
  plot(runif(100))
}
```

If you put two vectors of the same length into the generic function **plot()**, it will assume that you now want to make a scatter plot, so it will put the first vector on the *x*-axis and the second vector on the *y*-axis. As shown in Figure 8, the two vectors might be related by a function (so the dots are not really scattered as real data would be):

```
plot(1:20,(1:20)^2) # The second vector gives the square for each element
```

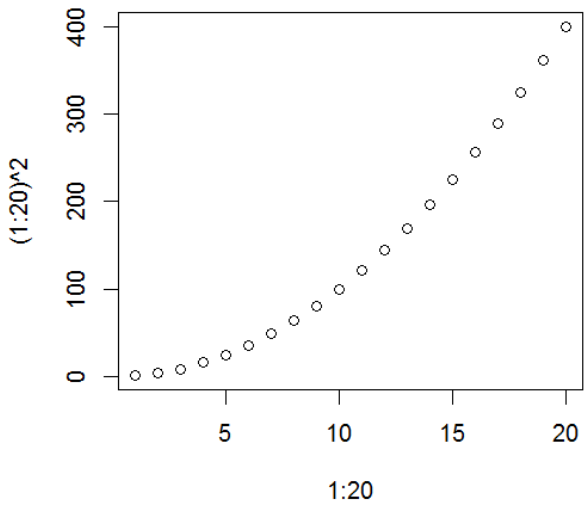

Figure 8. A not very scattered scatter plot

You can change the default dots (little open circles) by changing the **pch** argument (**pch=1** is the default "plot character"). It's impossible to remember what all the dot symbol codes are, but you can find all the choices by searching the internet, or writing little bits of code, like the one that creates Figure 9:

```
plot(1:20,(1:20)^2, pch=1:20) # Plot each dot with a different symbol
```

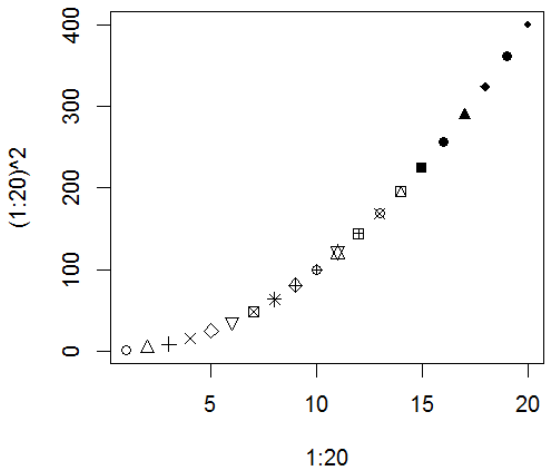

Figure 9. Some of R's dot types

You can also change the colors using the **col** argument (e.g., **col=2** makes the dot red, and so does **col="red"**). Try the following code, and modify it however you like:

plot(1:20,(1:20)^2, pch=15, col=1:20) # Plot each square with a different color

plot(1:20,(1:20)^2, pch=15, col=rainbow(20)) # So pretty!

The following command (which creates Figure 10) shows that R treats line graphs just as a special case of dot plots, with the argument **type="l"** (that's a lowercase "L", for "line"), that is, with line segments connecting the invisible dots. You can change the line type using the argument **lty** (e.g., **lty=1** is the default, and **lty=2** makes a dashed line), and change the width with **lwd** (try it!)

plot(1:20,(1:20)^2, type="l")

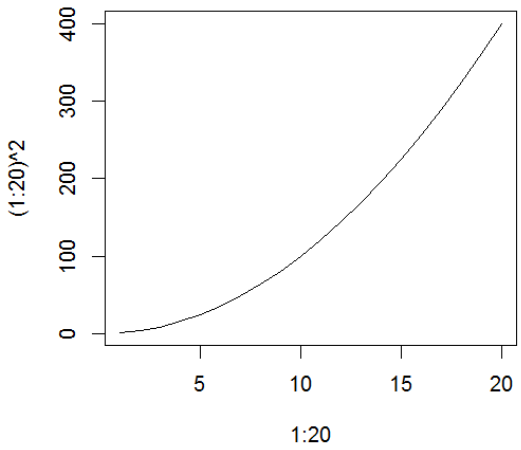

Figure 10. A line plot

Sometimes, it's useful to draw a plot in steps: first draw the basics using **plot()**, then add other stuff to it with other graph commands like **lines()** to lines to an existing plot object, **points()** to add dots, **text()** to add character strings, or **legend()** to add a legend box (try it!):

```
plot(1:20,(1:20)^2, type="l")
lines(1:20,(1:20)*2, type="l", lty=2, lwd = 2, col="blue")
points(5,200,pch=15,col="green")
text(5,250,"A green square", col="darkgreen")
legend("topleft",legend=c("x^2","x*2"),lty=c(1,2),lwd=c(1,2),col=c("black","blue"))
```

How might a scatter plot be useful to studying our "Jabberwocky" corpus? Well, not only did Zipf observe that with a large enough corpus, words fall into that non-bell-shaped frequency distribution, but he also noticed that they tend to show another interesting pattern: the more common the word, the shorter it is (see, e.g., Piantadosi et al., 2011). This pattern implies that speakers tend to try save energy when repeatedly using the same words.

To find out if this generalization also holds in the tiny fake-word "Jabberwocky" corpus, we first have to measure the lengths of all the words in the poem, then plot these lengths against their frequencies. We could do this in Excel using the **=LEN()** function and then use Excel's graphing tools; here's how to do it using R's **nchar()**, **names()**, and **plot()** functions (see Figure 11).

```
plot(jabberwocky.freq,nchar(names(jabberwocky.freq)))
```

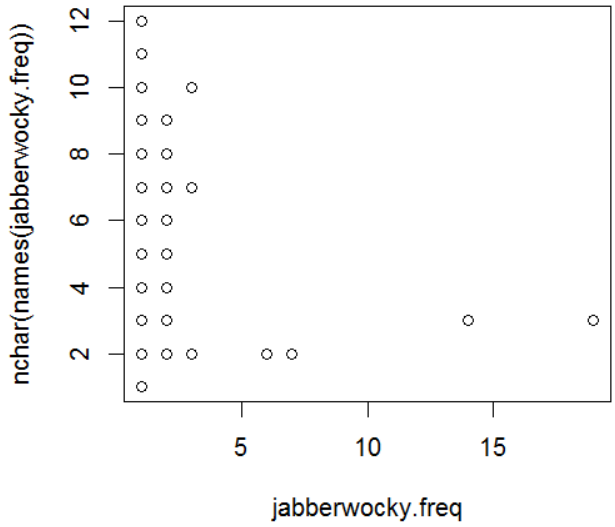

Figure 11. The correlation between word frequency and word length in "Jabberwocky"

Even without prettying up the plot (e.g., by using the **xlab** and **ylab** arguments for more meaningful x-axis and y-axis labels), it's clear that Zipf's word-length generalization basically holds here, with more common words (on the right side of the x-axis) being shorter (lower on the y-axis). Of course, given the tiny corpus, the pattern doesn't look very smooth: basically a lot of very long words on the left (almost all content words), and a few very short words on the right (all function words).

As noted above, you can make exactly the same scatter plot in Excel; please try it yourself. In fact, because of Excel's GUI-based design, the job is a lot easier to do, though keep in mind that deep inside Excel's own programming, it's actually converting each of your menu choices into command lines (albeit in C, not R).

5.3 Another way to make graphs in R: ggplot2

What I've explained so far are some of R's built-in **base** functions, but since R is open, people are constantly trying to improve it by creating new packages with new functions. For the past decade or so the most popular non-base graphics package is **ggplot2**. It gets this weird name because it is apparently the updated version of a "grammar of graphics" (non-linguists love abusing our terminology), most prominently advocated in Wickham (2009); other useful introductions to it include Chang (2013) and Healy (2018). If you have a good memory, you may recognize the name "Wickham"; it was arguably the great popularity of **ggplot2** that boosted the same guy's **tidyverse** into the mainstream as well. But as with all of the tidyverse component packages, you can install and load **ggplot2** all by itself.

However, despite its popularity, just as with the tidyverse more generally, I won't use **ggplot2** much in this book, for two main reasons. First, its defaults are designed for electronic color displays, not for black and white printing (see my complaints about that earlier). Second, the syntax is very different from R's normal syntax, making it confusing to learn while also learning ordinary R (and Excel!) at the same time. Even Winter (2019), apparently a computer genius, repeatedly confesses that he found it very hard to learn. Just as most R users probably also use Excel but don't want to admit it, I bet most **ggplot2** users probably just copy/paste sample code from the internet and never quite understand how it works. In this very book I will also occasionally offer some such code for you to edit for your own purposes.

But if you still want to get some sense of how this so-called "grammar of graphics" is supposed to work, the key idea is this: any graph is composed of a number of partially independent, partially interacting parameters, including the raw data itself, the graph's shapes (**geom**s, for geometric elements) used to represent it (e.g., points for a scatter plot or lines for a line plot), and **aesthetic** (審美的) features like shape (indicated with the **aes()** function). Unlike R's base plotting functions, **ggplot2** uses its own special syntax intended to combine these various parameters: basically you type one command, type "+", then type another command, until all your graph elements and features have been added to your satisfaction.

To demonstrate **ggplot2** in action, let's draw the bar graph in Figure 3 and 6 for a third time. While R's base function **barplot()** required us to put our data into a matrix object, **ggplot2** expects bar plot data to be in a data frame (or a tibble, if you want go all-tidyverse). So let's put our fake data into one, using the **data.frame()** function:

```
results.data = data.frame(Experiment = c(rep("Exp. 1",2), rep("Exp. 2",2)),
  WordType = rep(c("Noun","Verb"),2), RT = c(670,739,780,653))
```

You can see what this object looks like if you type its name:

```
results.data
```

	Experiment	WordType	RT
1	Exp. 1	Noun	670
2	Exp. 1	Verb	739
3	Exp. 2	Noun	780
4	Exp. 2	Verb	653

After **ggplot2** has been installed (by itself or as part of the **tidyverse** package), you turn it on (for your current R session) using the **library()** command:

```
library(ggplot2)
```

And finally, here's how **ggplot2** replicates that bar plot:

```
ggplot(results.data,                    # Data frame with data
  aes(x=Experiment,                     # Aesthetics, with bottom label Experiment
    y=RT,                               # y-axis variable is RT
    fill=WordType)) +                   # fills in bar colors based on WordType;
                                        #    + links parameters (cannot start a line)
ylab("RT (ms)") +                       # y-axis label
geom_bar(position="dodge",              # "dodge" acts like "beside" in barplot()
  color="black",                        # use black edges on the bars
  stat="identity") +                    # "identity": use actual values, not counts
scale_fill_manual(values=c("black","gray")) +  # colors of bars
theme_bw()                          # makes background white instead of default gray
```

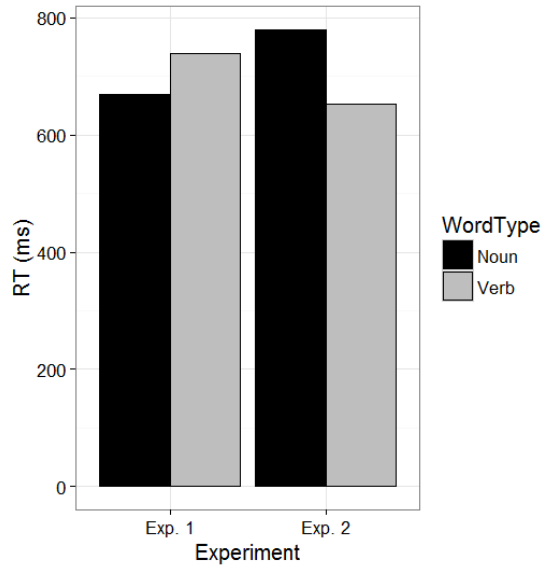

Figure 12. **ggplot2** tries to imitate Figures 3 and 6

If we don't care about imitating Figures 3 and 6 exactly, we can make the command slightly simpler by sticking with **ggplot2**'s defaults (try it!):

```
ggplot(results.data, aes(x=Experiment, y=RT, fill=WordType)) +
  ylab("RT (ms)") + geom_bar(position="dodge", stat="identity")
```

Personally, I don't find the results any better than R's base **barplot()** function (maybe even a little bit worse because I hate the color choices, which remind me of a 1960s American highway motel), and as I said, the syntax is quite different from the rest of R (those "+" symbols would be ungrammatical everywhere else in R). However, it is nice that **ggplot2**'s functions have a similar look and feel for all types of plots (bar plots, scatter plots, line plots, histograms, and so on), rather than being all different as in base R, and that the function names are *almost* English (with the typical tidyverse underlines). And if you search for help on R graphs, everybody will assume you're using **ggplot2** anyway. So go ahead and use it if you want.

6. Summary

This chapter introduced the basics of Excel and R, including how to put data into them, how to compute various things with them, and how to make graphs with them. You can review (some of) the different operations and commands that we discussed in the online "cheat sheet" StatsFunctions.pdf.

Along the way we also talked a bit about some important statistical properties of language, like Zipf's two laws, and introduced several crucial statistical concepts that will come up repeatedly throughout the rest of this book: data sets (including vectors and matrices), frequency tables, histograms, averages, correlations, and randomness.

References

Baayen, R. H. (2001). *Word frequency distributions*. Dordrecht: Kluwer.

Chang, W. (2013). *R graphics cookbook*. O'Reilly.

Chao, Y. R. (1969). Dimensions of fidelity in translation with special reference to Chinese. *Harvard Journal of Asiatic Studies, 29*, 109-130.

Fox, J. (2005). Getting started with the R commander: a basic-statistics graphical user interface to R. *Journal of Statistical Software, 14*(9), 1-42.

Fromkin, V., Rodman, R., & Hyams, N. (2018). *An introduction to language*. Cengage Learning.

Gagolewski, M (2021). stringi: Fast and portable character string processing in R. *Journal of Statistical Software*.

Gelman, A. (2013). Statistics is the least important part of data science. Blog post on *Statistical Modeling, Causal Inference, and Social Science*, 14 November 2013 <http://andrewgelman.com/2013/11/14/statistics-least-important-part-data-science/>.

Healy, K. (2018). Data visualization: A practical introduction. Princeton University Press.

Myers, J., & Tsay, J. (2015). Trochaic feet in spontaneous spoken Southern Min. In Hongyin Tao, Yu-Hui Lee, Danjie Su, Keiko Tsurumi, Wei Wang, & Ying Yang (Eds.), *Proceedings of the 27th North American Conference on Chinese Linguistics, Vol. 2*, 368-387. Los, Angeles: UCLA.

Piantadosi, S. T., Tily, H., & Gibson, E. (2011). Word lengths are optimized for efficient communication. *Proceedings of the National Academy of Sciences, 108*(9), 3526-3529.

Peirce, J., & MacAskill, M.R. (2018). *Building experiments in PsychoPy*. Sage.

R Core Team (2018). R: A language and environment for statisticalcomputing. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

Xie, Y. (2015) *Dynamic documents with R and knitr*. 2nd edition. Chapman and Hall/CRC.

Wickham, H. (2009). *ggplot2: Elegant graphics for data analysis*. Springer.

Wickham, H., Averick, M., Bryan, J., Chang, W., McGowan, L., François, R., Grolemund, G., Hayes, A., Henry, L., Hester, J., Kuhn, M., Pedersen, T., Miller, E., Bache, S., Müller, K., Ooms, J., Robinson, D., Seidel, D., Spinu, V., … Yutani, H. (2019). Welcome to the tidyverse. *Journal of Open Source Software, 4*(43), 1686.

Wickham, H., & Grolemund, G. (2016). *R for data science: Import, tidy, transform, visualize, and model data*. O'Reilly Media, Inc.

Winter, B. (2019). *Statistics for linguists: An introduction using R*. Routledge.