An R Primer for Lab Biologists, Technicians and Medics

Dr Timothy C. Stone

1 Introduction: Motivation

You might well view R as a glorified calculator or a spreadsheet with written commands. Officially it is a **statistical programming environment**. It is harder to use than a spreadsheet, but more useful.

The world of mouse clicks, toolbars and menus is much less important in R. You will now type in commands when you want something done. Why? A spreadsheet is great for small datasets because all the data is laid out on the screen, but for thousands or millions of numbers, it can get very tedious and slow and Excel might not even be able to cope.

Imagine you are taking some clinical data of tens of thousands of subjects. You want to filter out people with certain medical histories and maybe also remove samples from a particularly unreliable source. Then you want to fit a linear model on the remaining cases and controls, after taking into account the effects of BMI, gender and ethnicity and then save the results, one file for each country. You could do that in Excel, but wouldn't it be a pain? Imagine you are interrogating the entire genome of some cancer patients, would you *honestly* do that on Excel? Please don't.

You could probably manage better on SPSS but it probably won't be so easy either. SPSS is a good reason why many biologists don't learn R. SPSS goes beyond Excel in terms of hard statistical analysis, and it's a decent statistical software tool, but it is still a spreadsheet at heart. Carefully crafted commands on R can do the sort of things that are not that far removed from how you would describe the steps in plain English. Including equivalents to the phrase "do this for every single gene in the genome and order them" or "save a copy for each age group". If you can formulate all the steps of what you want to do in plain English, you are at least half way there to expressing it in R. Programming is like explaining everything you want to do to a person that takes everything literally.

One main point in R's favour, and this is not trivial, is that people who develop new methods of analysis in the biosciences tend to prefer R. R has an army of volunteers who create useful analysis methods and present them in tidy bundles, called **packages**. These can be incredibly useful if not essential. You can often find a helpful R-Package when a new technology is developed. R is also free. We know that academics do not have bottomless expense accounts; they love free stuff. Free stuff, however, is good because everyone uses it.

2 The Really Basic Stuff: Calculator-style R

In this chapter we will type in simple calculations, and by the end of it you will have the functionality of a scientific calculator at your disposal. Of course, there is much more to R than this, but you need to know this first. If you are a false-starter to R, most of this may already be familiar to you, but there still might be some small points that you could learn.

2.1 R-Studio and the R Environment

You obviously need R, but use **R-studio** as well, which is a friendly way of using R. You can use R without R-studio, but R-Studio is really useful so why would you be nasty to yourself?

When you start R-studio, the screen look something like this: (pic of R Studio inserted here)

R proper actually happens on the left. R-Studio is therefore software than runs R. This is software within software (a bit like a *play within a play*). R Studio adds extra functionality to help you and some of that stuff is very useful. A piece of software that gives you a comprehensive set of tools for you to write code is called an **Integrated Development Environment** (IDE). You don't need to remember this.

The lower left-hand area is the **Console**. The left-hand area can split into two. If you start writing and saving code, the console keeps the bottom left-hand area, and the code will go to the upper left. The console is sometimes is called the **command line**. There are more advanced situations where the command line could mean something a little different to the console, so maybe it's better to stick with the word console. You will see a > symbol at the bottom, which is called a **prompt**. Click in this window and you'll see a flashing / character, which is the **cursor**. You probably knew what a cursor was already, but you should know that if you click in any of the other windows, the cursor will stop flashing. You *activate* the console by clicking into it, the cursor flashes indicating that anything you type will get written as a **command**.

2.2 R from the Console

Click into the console and type, 2+2 and press return, it will look like this.

```
2+2
```

```
[1] 4
```

The answer is there, but there's a [1], what is that? It's a counter. It tells you the number of numbers printed so far and for this reason it always starts at [1].

Let's type something incomplete and see what happens. Type *2+* but instead of completing the sum, press return on the incomplete expression.

```
2 +
+
```

We have a new prompt, a plus sign. This is a continuation prompt, it means "Yes, and what else???", it doesn't mean mathematical addition.

If you type another 2 and press return, it will consider the two lines that you have typed to be a complete line. It will join them together and process the two joined lines for you as if they were one.

If, instead of finishing the line with "2", you instead produced another incomplete line (such as "3 +"), you would get another "+" and this will continue until R meets a line that is complete. Let's do this, type 2 + 3 + 4 - 1, but leave each line wanting more by not finishing the addition or subtraction

```
> 2 +
+ 3 +
+ 4 -
+ 1
```

```
[1] 8
```

You might, at some point, expect a calculation to be performed on the command line, but instead you get the "+" prompt when you press return. You got this because you messed up somewhere. If this was a genuine mistake and you want a fresh start, the *ESCAPE* key will stop the continuation and start all over again with a new ">" prompt. The *ESCAPE* key is the go-to panic button in R.

Let's also learn, right now, one of the most important typing shortcuts in R. Press the up arrow. It will recall what you have typed previously and if you press it again, you'll get the thing before that. If you therefore messed up your typing and got an unexpected "+" prompt. You should be able to ESCAPE the continuation line that follows, recall the mistyped line with the up arrow and correct it. It goes without saying that this can be extremely useful and will save you a lot of typing. If you have typed a lot of commands you can actually hold down the up arrow and R will whoosh though all the commands you typed in the past (in true HG Wells time-machine style).

Spaces are generally unimportant in R but they can make your code less dense and easier to read. You can type 2+2 or 2 + 2, it makes no difference to R.

2.3 Using Numbers

2.3.1 The Colon Operator

There is only one answer to 2+2 so you got a pair of square brackets with a one inside it as the counter. This seems a bit unnecessary, but type:

```
1:300 + 2
```

```
  [1]   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19  20
 [19]  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38
 [37]  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56
 [55]  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74
 [73]  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92
 [91]  93  94  95  96  97  98  99 100 101 102 103 104 105 106 107 108 109 110
[109] 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128
[127] 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146
[145] 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164
[163] 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182
[181] 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200
[199] 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218
[217] 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236
[235] 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254
[253] 255 256 257 258 259 260 261 262 263 264 265 266 267 268 269 270 271 272
[271] 273 274 275 276 277 278 279 280 281 282 283 284 285 286 287 288 289 290
[289] 291 292 293 294 295 296 297 298 299 300 301 302
```

(The numbers that count your output will depend on the size of your screen, so they might be different to mine)

You have generated the numbers 1 to 300 and added 2 to each of them. The answers have spilled over into more than one line. The square brackets keep a running total. This won't tell you the final size of the output, though obviously that is information you can easily find out.

This command made a lot of numbers, but R has also performed the same operation on 300 numbers all at once. R treats all the numbers as a big block so you don't have to loop through them. If you have worked in other programming languages, then it is a joy to see R doing such a thing. R is orientated to the manipulation of large sets of numbers as a block.

You can therefore generate lots of numbers as a simple counting sequence using a colon and it looks almost exactly like the way you would specify these numbers in everyday life. This is a quick, easy and very useful tool that you will see and use a lot. The colon is referred to as an **operator**. Any symbol that acts directly on numbers is an operator. You can use this operator to generate the numbers 1 to 1000000 as easily as the numbers 1 to 10.

Let's generate a million numbers, for fun. Type:

```
1:1000000
```

You will see a lot of numbers and then something like this

[945] 945 946 947 948 949 950 951 952 953 954 955 956 957 958 959 960

[961] 961 962 963 964 965 966 967 968 969 970 971 972 973 974 975 976

[977] 977 978 979 980 981 982 983 984 985 986 987 988 989 990 991 992

[993] 993 994 995 996 997 998 999 1000

[reached getOption("max.print") – omitted 999000 entries]

Numbers will spill out to the screen, but after it gets to 1000, R stops. R anticipates that throwing out a lot of stuff to the screen past a certain point probably isn't going to be of any use to you and it makes a decision to not to continue any more.

The colon operator can create numbers in the downwards direction as well.

```
10:1
```

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

The colon operator can also deal with non-integers, but always in steps of +1 or -1.

```
0.5:10.5
```

```
 [1]  0.5  1.5  2.5  3.5  4.5  5.5  6.5  7.5  8.5  9.5 10.5
```

```
10.99:0.99
```

```
 [1] 10.99  9.99  8.99  7.99  6.99  5.99  4.99  3.99  2.99  1.99  0.99
```

```
0.5:11
```

```
 [1]  0.5  1.5  2.5  3.5  4.5  5.5  6.5  7.5  8.5  9.5 10.5
```

You can see in the final example (0.5:11) that 11 is not a destination but a limit. The command effectively says *0.5 upwards in whole number steps, but no larger than 11*, 11 itself is never printed.

Exercises

1. Use the colon operator to generate the numbers 150 to 200
2. How many numbers are produced with

a. 1:10
b. 0.5 : 10
c. 0:10

2.3.2 Negatives

A negative sign before a number makes it negative, as you'd expect. This negative sign is an operator, known as **unary minus**. The operator can work on numbers but also other single items, including items containing many numbers. Unary minus takes whatever on the right of it and makes it negative. The example below is a negative countdown. The start and finish points are both negative, so we need two negative signs (spaces are optional, but added here for readability).

```
-1 : -10
```

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

This is a downwards count, even though the actual numbers are getting bigger in magnitude, because becoming more negative is considered a downwards move. The up-down analogy spills into other concepts in R. Upper limits for rounding numbers are called **ceilings**, lower limit are **floors**. We will return to this idea.

If you recall from your high-school mathematics, the negative of a negative is a positive. You can actually confirm this in R. Type this (spaces are optional but added here for clarity):

```
- - 1
```

```
[1] 1
```

And R will actually go to the trouble of turning two negatives into a positive. In reality you would not do such a (pointless) thing, but let's continue being silly for a little bit longer. Type a lot of negatives followed by a 1.

```
----------------------1
```

```
[1] 1
```

R will take any number of negatives followed by a number and flip between successive negatives and positives until it has worked through them all. You should have no need for this feature, but this does mean that if you type two negatives by mistake, R does not view this as an error but as something to process.

Unary positive (i.e. +2) is also a "thing" in R, but I never use it and I don't really see why you should (I suspect that a lot of R programmers don't even know it exists). I can imagine it might be useful that R can read a file of fastidious data that has such positive signs included, but I have yet to encounter such a scenario especially as most spreadsheets obliterate them. In summary, a single negative is used to negate whatever is next to it. You can combine multiple unary negatives or positives or both, but don't.

Exercises

1. How many numbers are produced by:

a. 5: -5
b. 4.5 : -5
c. +0 : -0

2. What is the value of:

a. ++-+-+ 15

b. ——-0
c. +++++100

2.3.3 Multiplication, Division, Integer Division and Modulo

Multiplication and division are the same symbols as used in Excel and other programming languages, * for multiplication, / for division. Something that might be less familiar is the modulo operator, which is two percent signs %%. This gives you the leftovers after whole-number division. When you first learned division at the elementary levels of education, you might recall the concept of *remainder*, where 17 divided by 5 was not 3.4, but "3 remainder 2". A common use for the modulo operator is with the number 2 to determine if a number is even or odd. Modulo 2 gives the code: 1=odd, 0=even. If you wish to "chop off" the last digit of a long number, modulo 10 will do this and do it with long numbers of different sizes.

We can verify now that 100 is even and 47 is odd using modulo 2.

```
100 %% 2
```

```
[1] 0
```

100 is an even number because 100 modulo 2 is equal to 0.

```
47 %% 2
```

```
[1] 1
```

and 47 is an odd number because 47 modulo 2 is equal to 1.

Now let's chop off the final digit from some long numbers using modulo 10

```
54321 %% 10
```

```
[1] 1
```

```
99999999 %% 10
```

```
[1] 9
```

Another nifty modulo trick, which is useful for computational-mathematical puzzles, but might also find use processing id numbers (you never know), is to use the modulo operator to break a number into two pieces and extract the end. You can extract the final digit on a number using modulo-10

```
538 %% 10
```

```
[1] 8
```

```
6354 %% 10
```

```
[1] 4
```

R can perform integer division which is the type of elementry-school division mentioned above (i.e. decimal free). This is the achieved via the awkward symbol-cluster of percent-division-percent %/%. In reality, you almost never see integer division as R has functions to truncate numbers that do exactly the same thing and the explicitness of that method makes it more understandable than this obscure symbol. Even so, let's do a party trick and use integer division and modulo operators to answer the question: if the literary classic *One Thousand and One Nights* was converted to years and days, what would the new title be? First of all, the years are obtained by integer division.

```r
1001 %/% 365
```

```
[1] 2
```

And now for the days, BUT before you retype this command. Let's get some shortcut practise. Press the up arrow to recall the integer division you have just typed, move the cursor to delete the "/" and press return. Get into the habit of doing such things when you are doing similar things one after the other.

```r
1001 %% 365
```

```
[1] 271
```

1001 nights is equal to 2 years and 271 nights. Let's give a big ironic round of applause to R for taking the romance out of the title of this literary classic.

Exercises

1. There are 5/8 of a mile in a kilometre, use R to convert

a. 100 kilometres to miles
b. 100 miles to kilometres

2. Using integer division, find out how many 12-egg eggboxes would you need to store 1000 eggs
3. How many hours and minutes are there in a million minutes? (using %/% and %%)
4. What do you think the answers are to these, make a guess first and check:

a. 10.5 %% 0.5
b. 10.75 %% 0.5

5. What is:

a. 3.14159 %/% 0.1
b. 2.718282 %/% 0.01

2.3.4 Strange and exotic numbers

2.3.4.1 Infinity and Not a Number R has provision for infinity. You may find yourself generating this number unintentionally through division by zero. Other languages often give you an error message and refuse calculation if they encounter this process, but R generates an infinity marker. This means that processing can still continue while other computer languages would grind to a halt when using the same data. Of course, these infinities may create problems for you further down the analysis pipeline, but they can be removed once created. Ultimately there's not a huge difference between a pre-processing removal of zeros and a post-processing removal of infinities, but at least other people's packages and functions can work immediately with your zero-populated data.

Let's make infinity. Type:

```
1 / 0
```

```
[1] Inf
```

You can input infinity, why you would want to do this is another matter. You have to be exact with the capitalisation. You might think that infinity + infinity gives you *even more infinity* but infinity is already the limit and there is no beyond. R agrees:

```
Inf+Inf
```

```
[1] Inf
```

On the other hand, mathematicians have generally agreed that infinity minus infinity is not zero but undefined. R makes provision for undefined numbers by having **NaN**, which means not a number.

```
Inf-Inf
```

```
[1] NaN
```

2.3.4.2 Not Available

R has a placeholder for missing data, which is **NA**. NA means *not available*. If you are entering your raw data in a spreadsheet (like Excel) to export to R, it might be a good idea to start using "NA" for your missing data. R will automatically assume that it is dealing with missing data if it encounters NA without you having to specify what missing data marker you are using.

NaN and NA are completely different things. NaN means that there does not exist a mathematically-agreed number for the calculation you are attempting. NA means a number is missing in your data. We will leave further discussion on NA until we address the issues of processing large batches of data.

2.3.4.3 Imaginary and Complex Numbers

R can also work with the world of imaginary and complex numbers (these are numbers that include multiples of the square root of -1). R works with imaginary numbers but always expresses them as a complex number. Complex numbers are numbers which have a real and imaginary part. We will briefly mention them now, and forget about them, because, as a practical biologist, you are unlikely to ever encounter them. You can skip the rest of this section if you don't want to be bothered any further.

You input imaginary numbers into R as a real number followed directly by *i*. As mentioned before R always displays an imaginary number as a complex number, even if the real part is zero. Typing *1i* gives you the square root of minus one as a complex numbrer.

```
1i
```

```
[1] 0+1i
```

Complex numbers are used extensively in certainly branches of physics and engineering, but only turn up in the more densely mathematical end of theoretical biology. I think it's fair to say that most biologists and medics will never have any use for this type of number. Complex numbers are always explicitly defined, you cannot "accidentally" create them as a result of calculations in R. What do I mean by this? There are many types of number that can be created via other means. Creation of negative numbers is trivial, you subtract a big number from a small number. Creation of infinities is also fairly trivial as they can be created from division by zero, but R doesn't care for the casual creation of complex numbers. If you were calculating the square root of minus one using everyday data, there's a big chance that you've made a mistake, and in this respect, it is almost certainly a good thing that R throws an error rather than dives into a world of imaginary weirdness. If you are anticipating complex numbers in your output, you have to include a letter *i* somewhere in your input, even if the amount of *i* in your input is zero. It's almost as if you are signalling to R that you know what you are getting into and will accept the consequences. We see here that our *casual* attempt to create a complex number fails.

```
(-1)^0.5
```

```
[1] NaN
```

You get NaN, even though there is actually a number, i, that R could have used.

The correct way to calculate the square root of -1 in R is like this:

```
(-1 + 0i)^0.5
```

```
[1] 0+1i
```

This is possibly the last time you will ever type a complex number into R. If you didn't understand this, don't worry. It really isn't a problem for understanding anything else We have paid lip-service to complex numbers. Let's leave them be.

Exercises

1. What do you think are the answers to these, guess them before you type them:

a. Inf * Inf
b. Inf / Inf
c. 53 * Inf
d. 53 / Inf
e. 53 %% Inf
f. 53 %/% Inf
g. 53 + NA
h. 53 / NA
i. NaN + NA
j. Inf * 0
k. Inf * 1 / 0
l. NaN + 2

2.3.5 Very Big and Very Small Numbers

R will probably work very easily and accurately with all the numbers you can ever create with your data. Like Excel and a scientific calculator, R uses scientific notation when the numbers get very big or small. Type a hundred thousand as a number and you will observe it:

```
100000
```

```
[1] 1e+05
```

For the small numbers, it take a mere three zeros after the decimal point for the negative exponent to come into use. The initial zero, the one before the decimal point, is option, though often included for clarity:

```r
0.0001
```

```
[1] 1e-04
```

```r
.0001
```

```
[1] 1e-04
```

You can create these numbers using a lowercase or uppercase *e*, R accepts both, however, use the lowercase version because an unwritten rule of computer programming is that you use capital letters if and only if they add something useful, and in this case, capital *E* adds nothing.

```r
1e-10
```

```
[1] 1e-10
```

```r
300e18
```

```
[1] 3e+20
```

The numerical range of numbers used by R is huge, and as a biologist, you are unlikely to create data that exceeds them. The upper limit is around 1 x 10^{305}, and the lower limit is 1 x 10^{-323} (at least on my current version of R with my computer). If you exceed that, R overflows to Inf, and for numbers that are too small R underflows to zero.

```r
1e310
```

```
[1] Inf
```

```r
1e-324
```

```
[1] 0
```

2.3.6 Exercises

1. 9.99 %% 10 is 0.99, but 9.99999999999999 %% 10 works out as 10 on R due to underflow errors. Use trial-and-error to work out the maximum number of nines after the decimal point that you can have before R gets it *wrong*?
2. 1e-325 * 1e150 * 1e175 **should** be equal to 1, instead it is equal to 0 on R. Why does this happen?
3. Does the double unary minus (–1 = 1) work with powers of 10? Try it.
4. The colon operator cannot always cope with big numbers? Try these and see if you can understand what is going on, type:

a. 1e2:1e3
b. 1e10:1e11
c. 1e150:1e151

2.3.7 Precedence

A concept that all computer languages address is that of precedence. Precedence determines which operators "get their hands on" the numbers first. All operators have a place in this order.

In the world of computers:

4 + 3 * 2

is equal to 10, and not 14, because multiplication is done before addition. This is the order of precedence that R uses. You don't need to remember all the details. It might be useful to remember that the basic mathematical operations (* / + -) are at the bottom of the pile, with the simplest (+ -) at the very bottom.

1. Raising to a power (ˆ)
2. Unary minus (-), Unuary plus (+)
3. Colon operator (:)
4. Modulo (%%), Integer division (%/%)
5. Multiplication (*), Division (/)
6. Addition (+), Subtraction(-)

Given that the range of numbers in the colon operator determines the number of items in the output, the precedence rules make a big difference to the length of the output in these two examples, even though they both involve 1:5

```
1:5 ^ 2
```

```
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
```

```
1:5 * 2
```

```
[1]  2  4  6  8 10
```

In the first example, 5ˆ2 was performed first, making the colon operator 1:25, in the second example, a run of 1 2 3 4 5 had each element multiplied by 2. Obviously there will be situations when you want a lower priority operation to happen before a higher one, such *queue-jumping* is achieved by using brackets.

2.3.8 Exercises

1. Work out the answer in advance that R will give to these, and check them:

a. 10 - 8 * 8
b. 1:10 + 2
c. 1:10 * 2
d. 1:10 ˆ 2
e. 1:10 %% 2
f. 1:10 %/% 5
g. 1:-10 ˆ 2

2.3.9 Brackets

Conventional round brackets () are known specifically as **parentheses**, especially in the context when you wish to distinguish them from curly { } and square [ ] brackets which are all used in different contexts in R. Different types of bracket are distinct and cannot be freely interchanged. Parentheses indicate that everything inside them is to be treated as one whole unit for mathematical processes. If you are familiar with Excel calculations, you will understand how R uses brackets because they are used in exactly the same way.

When unary minus acts on a bracket, it makes the entire contents negative. In light of this, you should be able to see the (important) difference between these two expressions.

```
-1:10
```

```
 [1] -1  0  1  2  3  4  5  6  7  8  9 10
```

```
-(1:10)
```

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

In the first example the unary minus goes first, due to its high precedence. It works only on the one, and then the colon operator takes over. This produces a sequence from -1 to 10. In the second example, the colon operator is moved to the highest order of precedence due to the brackets. This produces the numbers one to ten and then unary minus makes the whole lot negative.

Remember from the precedence rules that the calculation:

```
2 + 3 * 2^2
```

```
[1] 14
```

is equal to 14 (and not 100 which is the left-to-right answer).

We can replace the steps manually in turn to see the priority. First the power 2^2=4:

```
2 + 3 * 4
```

```
[1] 14
```

Then the multiplication 3*4=12, leaving the addition:

```
2 + 12
```

```
[1] 14
```

If you wanted to perform the calculation in left-to-right fashion, you would need to produce three brackets that are tucked inside each other or **nested**, just like Russian dolls. Here's how you force the calculation to be done in left-to-right style:

```
(((2 + 3) * 2) ^2)
```

```
[1] 100
```

Did you notice anything unusual as you typed in the above command? R automatically produced not one bracket, but a pair of matched brackets, "(" and ")", when you typed "(". This is a device that enables efficient typing. As you type the bracket contents, the right-hand bracket shunts along. If you attempt to add a closed bracket, R simply moves the cursor past the automatically-created bracket.

This pair-creation and cursor shunting feature can occasionally be annoying as it can sometimes give you an unwanted closed bracket or fail to add an additional closed bracket when you actually want one. I think it is worth the occasional irritation to have this useful finger-saving feature. You do have the option of switching this off this feature via the options menu if you find it too annoying.

2.3.10 Exercises

1. Predict the output of these, then try them for yourself:

a. -(-1:-10)
b. -(1:-2)ˆ-2
c. ((2 + 3) * 3) - 4 + (7 %/% 3) * 4

2.4 Functions

2.4.1 Fundamental Mathematical Functions

Functions transform data into new data. Operators do this too, but operators are symbols wheras functions are words. Functions are generally more complex and the format for using them is different than symbols. If you are familiar with Excel, you'll be familiar with functions. Parentheses are used to parcel-up the numbers and commas are used as a separator if the function requires more than one number. When functions are being described in instruction manuals, they usually have a pair of empty parentheses beside them. This is a standard convention for identifying a function (like a *calling card*). Let's start with the sort of mathematical functions you might find on a calculator.

Square root, sqrt(), all positive numbers are accepted, negative numbers produce an error (complex numbers would be processed, but we aren't ever going to use them):

```r
sqrt(100)
```

```
[1] 10
```

```r
sqrt(-1)
```

```
Warning in sqrt(-1): NaNs produced
```

```
[1] NaN
```

Absolute value, abs() is a simple but useful function that makes all negatives into positives (and leaves the positives untouched):

```r
abs(-10)
```

```
[1] 10
```

```r
abs(5)
```

```
[1] 5
```

Exponentials ("*e to the power of x*"), exp() and the reverse function, log():

```r
exp(1)
```

```
[1] 2.718282
```

```r
log(7)
```

```
[1] 1.94591
```

Factorials were written with an exclamation mark in your mathematics class, but R has a dedicated function. These are progressive multiplication functions that rapidly produce huge numbers. 4!=1 * 2 * 3 * 4, the R equivalent to 4! is:

```r
factorial(4)
```

```
[1] 24
```

It's actually possible to calculate factorials of decimal numbers, and there's also a mathematical function, gamma(), that is essentially a laggard factorial. The two functions, though very similar, arrived at different times in mathematics history, and R gives you both. Gamma is the same as a factorial but one whole number behind, gamma(x) = (x-1)!. We can check this in R:

```r
factorial(3.5)
```

```
[1] 11.63173
```

```r
gamma(4.5)
```

```
[1] 11.63173
```

We also have all the tools of trigonometry at our disposal: sin(), cos() and tan(), and their variants. Put an ***a*** in front of them for an inverse and an ***h*** at the end for the hyperbolic versions. If you don't know what that means, it's likely you are never going to need them, but they are there for you, just in case.

Let us also introduce pi, because these functions use radians, not degrees and the radian measure of angles is based on the number pi. PI is so ubiquitous in mathematics that R stores a copy of this number for your convenience, just like all scientific calculators and spreadsheets do. It must be typed as a lowercase English word **pi** with no function brackets, because it's not a function (this is **not** the same as Excel, which treats it as an empty function just because it cannot deal with bare words, but R uses the bare word and is more correct).

Let's do some trig! By all means try all the functions if you want, but I will include only some of them.

```
pi
```

```
[1] 3.141593
```

```
cos(2*pi)
```

```
[1] 1
```

```
tan(46)
```

```
[1] -2.086614
```

```
asin(1)
```

```
[1] 1.570796
```

```
cosh(0)
```

```
[1] 1
```

```
atanh(0.5)
```

```
[1] 0.5493061
```

We already calculated the cosine of two-pi, now lets calculate the sine, which, if you remember from your mathematics class, is zero.

```
sin(2*pi)
```

```
[1] -2.449294e-16
```

What???? What is going on? This is an example of a **precision error**. The true number pi is an infinite decimal. R approximates it (as all computers do) and when you do some calculations with it, the differences between *practical pi* (i.e. computer-stored pi) and *theoretical pi* show up.

R is very good with precision, but you need to be aware that very occasionally you might encounter precision issues and it might be a problem. The most obvious example that I can think of is if you wanted to remove all the zero values from your processed data, or count all the zeros in your data, but because of precision errors, some data that actuallly **should** be zero is ever-so-slightly not zero and misses the count. It should be said that such errors don't normally happen. They are very rare and normally involve some quirky situation to get them (if, for example, you are using pi or some other approximated infinite constant). R has a suite of functions to round and truncate numbers that will solve this issue, but it is a very rare problem.

2.4.2 Function Parameters

We've seen that we can do logarithms.

```r
log(2.71)
```

```
[1] 0.9969486
```

The answer to this example is nearly one, because the input was nearly *e* the base of natural logarithms. But wait a minute?! *Any* number can act as a base for logarithms. In fact, Base-10 logs and base-2 logs are quite common in biology. Base-2 logs are nicely intuitive, every time a base-2 log increases by one the *true* value has doubled, likewise, every time a base-10 log increases by 1 the *true* number is multiplied by 10 (which is like adding a zero). Base-2 and base-10 logs are good human-intuitive ways to shrink numbers that have a huge range. How can we evoke them?

Actually these two bases of logarithm are indeed so useful that R has supplied specific functions for these bases, which are log2() and log10(), and even though I'm about to tell you an alternative way to get get such logs, you should always use these easily-readable versions in preference.

```r
log2(64)
```

```
[1] 6
```

```r
log10(10000)
```

```
[1] 4
```

Most R functions are adjustable. You can change the underlying settings used for calculations. These are called **parameters** of the function. Base-10 logs are achieved in the log() function by adjusting the *base* parameter. The base parameter setting in the log() function is normally e, which is 2.171. . . , but if we specify the value, R will use whatever we give it, including 2 or 10.

```r
log(64, base = 2)
```

```
[1] 6
```

```r
log(10000, base = 10)
```

```
[1] 4
```

R assumes you want a natural log, unless you specify a base parameter. This is an example of a **default parameter**. Most R functions have them. If you are using a complicated function for the first time from a specialised package, it's probably likely that you will want to stick with the default values. The authors of functions aim to give you good *sensible* default values. A good principle to work on is to only change the default values when you know what you are doing. The log() function has two parameters, one is called x which is the main input number, the other is base with a default value of 2.171. . . (e), and indicates the type of log you wish to calculate.

Some function parameters are obligatory and have no default. x in the log() function is obligatory, which makes sense. These two examples fail because x was not provided.

```
log()
Error: argument "x" is missing, with no default

log(base=10)
Error: argument "x" is missing, with no default
```

When we typed log(100, base = 10), we specified the parameter name base but not x. What's going on there? If you don't give R the parameter names, R has an expected list of the order in which things should arrive for every function. It's actually part of the function set-up. In the case of logs, it's *x* first, *base* second. The calculation below would therefore be equivalent to the one above, because we supplied the numbers to the function in the correct order.

```
log(100,10)
```

```
[1] 2
```

When you specify the parameter by name, the order is still retained, but the parameter in question drops out of the order list. You can therefore use a mixture of naming and order. This is ok if the order of importance of the first few numbers are obvious, but it can create difficult-to-read code.

R has a very useful feature with functions on the command line. Type

log(

and press the tab key. R will list the parameters of the function just above the cursor. It will also list the order in which it expects the parameters. You should see information on the first parameter. Press return, R will insert the parameter name with an equals sign. You should see this (don't forget that R automatically added the closing bracket).

log(x =)

Let's give x the value of 100 and add a comma and a space for readability.

log(x = 100,)

Now press the tab again. You will see the information on the second parameter, including information about the default value. Press return to get the parameter name and we can complete the calculation.

```
  log(x = 100, base = 10)
```

```
[1] 2
```

The tab is a very useful key. It summons R's labour-saving autocomplete mechanism.

We will now create different examples of the same calculation (the base 10 log of 100), but using every possible way of providing input to the function. They are ultimately processed by R in exactly the same way and always produce the same answer (which is 2). Some of these are examples of readable presentation, others are not, but they all still work.

The way most people would write the calculation in R is this:

```
log(100,base = 10)
```

```
[1] 2
```

If you wished to be thorough or if tab-autocomplete is filling in the parameter names, you will do this:

```
log(x = 100,base = 10)
```

```
[1] 2
```

This next example is not so good. The second parameter is not named and that's actually the parameter that is changed from its default.

```r
log(x = 100, 10)
```

```
[1] 2
```

This is also not so good. The parameters are named, but the orders are swapped. R does the right thing because the parameter naming overrides the order, but it would possibly confuse someone reading your code.

```r
log(base = 10, x = 100)
```

```
[1] 2
```

This final example works, but is very bad, and actually slightly crazy. You would do well to understand why it still works but also why it is misleading. The x parameter is named but appears in the second position on the function call. The only other item on the function parameter-list is *base* but it is not named and not in second position. R uses it only because it finds a number and has a position available for its use. If you read this line quickly, you might think it was to calculate log 10 in base 100. This is legitimate code, but it is badly written code.

```r
log(10, x = 100)
```

```
[1] 2
```

2.4.3 Functions to Round Numbers

We have mentioned the rounding of numbers already, so let's finaly see how these are done. As mentioned above, the language of up and down pervades R's view of numbers and is somewhat independent of the magnitude of a number. An upper limit is a **ceiling**, a lower limit is a **floor**, 100 is higher than 1, -100 is lower than -1. Let's use pi with some ceilings and floors and we'll throw in unary negative as well.

```r
ceiling(pi)
```

```
[1] 4
```

```r
floor(pi)
```

```
[1] 3
```

```r
ceiling(-pi)
```

```
[1] -3
```

```r
floor(-pi)
```

```
[1] -4
```

If we just wanted to cut-off or truncate the decimal part, the trunc() function will do this.

```r
trunc(pi)
```

```
[1] 3
```

```r
trunc(-pi)
```

```
[1] -3
```

Often, instead of moving to the ceiling() or the floor(), we want to go to the nearest whole number. You COULD do this by adding 0.5 and moving to the floor(), or subtract 0.5 and move to the ceiling. For example, this method will get 1.25 rounded down to 1 and 1.75 rounded up to 2.

```r
floor(1.25 + 0.5)
```

```
[1] 1
```

```r
floor(1.75 + 0.5)
```

```
[1] 2
```

```r
ceiling(1.25 - 0.5)
```

```
[1] 1
```

```r
ceiling(1.75 - 0.5)
```

```
[1] 2
```

But this is actually not a good way to do it and we shall see why. Instead, the function round() should be your go-to method of rounding data.

The two parameters of round() are *x* and *digits*, which is the number of decimal places. For rounding to the nearest integer, the number of decimal places is zero. We will be specifying the parameters into the function by the order they appear in the function call, and not explicitly by parameter name. For a simple, well-known, two-parameter function like round(), this is generally considered acceptable, even if someone else is likely to read you code.

```r
round(1.25, 0)
```

```
[1] 1
```

```r
round(1.75, 0)
```

```
[1] 2
```

That does also mean that you could round these numbers to one decimal place as well. Let's do that and pay attention to the results.

```r
round(1.25, 1)
```

```
[1] 1.2
```

```r
round(1.75, 1)
```

```
[1] 1.8
```

Did you see what happened to the two numbers? Both numbers ended in a 5, but in the first example the number went DOWN, in the second the number went UP!

This is not a bug.

Consider, for a moment if you used ceiling-rounding or floor-rounding on a large set of numbers. If you decided to take 0.05 to the ceiling every time – to the nearest ceiling of 0.1, you would be pushing all those numbers upward and that's an artificial source of bias. Likewise, if you used floor-rounding, you would be sinking your numbers ending in a 5.

round() is clever, it takes all the numbers ending in a 5 and shunts them to the nearest decimal ending in an even number. This is why 1.25 goes down to 1.2 and 1.75 goes up to 1.8. If your input numbers ending in a five were distributed evenly on the previous decimal place, this will minimise any artificial drift that you will get from rounding your numbers.

In the same family as the round() function is the signif() function which does significant digits. round() is absolute, and always refers to the decimal point, signif() is relative and doesn't care where the decimal point is. It also does the clever bias-reducing even number trick that we just observed.

```r
signif(1.25, 2)
```

```
[1] 1.2
```

```r
signif(1.75, 2)
```

```
[1] 1.8
```

2.4.4 Multiple Input and Nested Functions

R functions can work with many numbers all at once, just like it did for addition. Let's generate nine numbers with the colon operator but do that inside a function call.

```r
sqrt(1:9)
```

```
[1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751 2.828427
[9] 3.000000
```

The "Russian doll" approach to brackets also works with functions. We can take the output of a function and use it as the input to a new function, so that they are nested.

```r
sqrt(exp(abs(-2)))
```

```
[1] 2.718282
```

Here, -2 encounters the abs() function first, then the exp() function, then finally the sqrt().

If we nest a function with its inverse, we get the original number back, and in fact, we see absolutely no precision problems here:

```
log(exp(7))
```

```
[1] 7
```

3 Going Beyond the Calculator, Your Tools for R-Adulthood

A conventional R course will teach you all the different data structures, and how to analyse data. We will obviously get there too, however, as a professional R Programmer, there are some good practises that set you up for a considerably less-stressed programming life and the sooner you learn them, the better. We are therefore going to do some things that are considered more *advanced* but not because they are conceptually difficult. It's just that they are often pushed to a later stage in order that you learn more basic stuff. A lot of them are actually optional; you can muddle on without them, but that's the point. You want to avoid as much muddle as possible and you ought to get these muddle-free habits ingrained into you as soon as possible. We will therefore get familiar with **packages** and **package management**, with **pipes** which will make the structure of your code less computery and closer to the structure of English prose. We will also aim to make you think on a more structured level about the code that you write and where it will live. We will look at **projects** and **version control** which are standard ways that programmers keep track of the development of their code. We haven't written any programs yet, but that's actually a great time to learn good coding practises.

3.1 Packages

R is all about the packages. Think of it like computational blood donation. Someone (you probably don't know who they are) is doing you a favour and making your life bearable by their altruism and it's a life-saver. People all over the world write loads of functions and sometimes include useful data as well. They then bundle them up all together into the computational equivalent of a boxed set that you can freely download and use. These packages do incredibly useful stuff and come at no expense to yourself. This makes R a remarkable enterprise.

We are going to install a package right now, and then we shall use it. We will start with a smallish package that offers a very useful way to write code, called **magrittr** The *install.packages()* command well automatically go online, download the package you need and place it in a folder that R can access. It will give you a running report of its progress. The function to do this is made of two words that are joined by a dot. Incidentally, R commands are always a single word, and *install.packages* is in fact a single word in R's eyes, even though it looks like two words to you. The dot and underscore are sometimes used as a pretend word separator. Let's install the package *magrittr*

```
install.packages("magrittr")
```

You will see stuff in red and black that looks like this:

```
trying URL 'https://cran.rstudio.com/bin/macosx/contrib/4.0/magrittr_1.5.tgz'
Content type 'application/x-gzip' length 152671 bytes (149 KB)
==================================================
downloaded 149 KB
```

```
The downloaded binary packages are in
    /var/folders/7h/zhgkyzk13fq5phcs5qps5x2r0000gn/T//RtmpnMtTje/downloaded_packages
```

If you get a message with the word *warning* in it, that's not necessarily a problem. R warnings indicate that something has gone ahead, but R is alerting you to some possibilities. Nonetheless, the installation worked. There is an R function, warnings(), that will recall the previous warnings you experienced.

If you get an message with the word *error* in it, that's a problem and you must fix it because you can't continue any further. If you are using a work/office computer, it could be that you do not have persmissions granted on your computer to freely download functional stuff from the internet. Offering solutions to any errors like this is beyond the scope of this primer. You can try and fix the problem yourself. You will have to search the error message online (that's what I'd do), and see if anyone else has experienced and solved the problem. Take a phrase from the error that looks like it couldn't be confused for anything else, and Google it.

Most R programmers will encounter a problem that they cannot solve themselves and will be looking for solutions online. Possibly the most well-known forum for community-offered solutions is a site called *stack overflow.* It is highly unlikely that a problem that you are experiencing hasn't already been experienced elsewhere. It is also unlikely that a question hasn't appeared on Stack Overflow with a solution.

We will now assume you have installed this package. The package now sits in your computer memory ready for activation, like an unopened boxed set.

Click on the **Packages** tab on the lower right sub-window of RStudio. You will see all the R packages that have been installed. If someone else has been using R on your computer, you might see a lot of packages already. I have about 200 of them. I haven't necessarily installed all 200 of them, in fact I certainly haven't. It's just that some R packages piggy-back off other R packages. This is a system known as **dependencies** . The *magrittr* package is itself a dependeny of a very popular package called **dplyr** which we will encounter soon enough. In fact, I could have asked you to install *dplyr* and we would have had magrittr installed too. I just wanted to keep this simple right now.

Every R package contains where you can view an instruction manual. magrittr's is quite fancy by R standards.

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

You can see there is a little *Install* button below the packages tab. You could have installed the magrittr package by clicking on that button. Clicking opens a window where you can type the package name. I prefer command-line installation myself, but that is an alternative way in which you can install packages. It's your choice.

Update is a button you should press from time to time, when you don't need R to do anything else (so coffee break and lunchtimes are good). People change packages, sometimes there are small bugs that need fixing.

Let's mention package longevity. Occasionally I will receive a warning from a command that the way I am writing the command is not the way is *old-fashioned* they want me to write it in another way in the future. They may say that at some point they will stop allowing me to write the command in that manner. The people writing the package have their reasons for such an approach. It is not a common occurrence, but it happens often enough that you might wish to consider using *Packrat*, which is the third button on the Packages tab.

Packrat is one of those facilities that I ought to use but don't yet. The idea is to make sure your code of the day doesn't get messed up by any future updates, either in the package you are using or its dependencies. Occasionally some packages undergo massive rewrites and these updates can alter the package so much that they mess up other packages that depend on them.

If I was using something that needed some longevity, such as my very own R package or a piece of long-term code, I might want to make sure that my code had long-term stability. Currently I can get away without using Packrat.

With the Packages Tab, now explained. Let's click back in the console.

magrittr is in my computer memory but it is effectively boxed up. In order to *unwrap it* we use the library command.

The library command *summons* all the commands and data associated with a particular package.

```
library(magrittr)
```

This is good, we will use magrittr for the next new concept.

3.2 Piping

a **pipe** feeds the output of one colaculation into the input of another without you having to store the middle-value yourself.

You don't have to use pipes to do this. Another way to do this would be to nest functions. We have seen this already. The problem with nesting functions is that the whole thing can become an unreadable mess. Piping solves this problem by splitting the processes out into distinct steps that follow the order of calculation. Piping is therefore a valuable tool for simplification and understanding. It is slightly more cumbersome to type with pipes, but you are far less likely to fill your code with errors.

Pipes make your R code a bit longer, but **far** more readable.

Let's recall this command from before the previous chapter.

```
sqrt(exp(abs(-2)))
```

```
[1] 2.718282
```

We read from left to right, however, the brackets make this command work from right to left. If you are describing the chain of events in this command, we say something like *take minus 2, make it a positive number, use that as the input in an exponential and take the square root of the result.* Pipes basically mirror the structure of this prose. We use this symbol: %>%

That symbol means *use the left result as an input to whatever is on the right*

Let's rewrite the previous command with pipes:

```
-2 %>% abs() %>% exp () %>% sqrt()
```

```
[1] 2.718282
```

So this says: *take minus 2, stick it in the abs() function, stick that result into the exp() function, stick that result into the sqrt() function and serve.*

I mentioned that brackets are the *calling card* of functions, but this is not strictly true. There are times where brackets are optional. The following bracket-free piece of code will actually work

```
-2 %>% abs %>% exp %>% sqrt
```

```
[1] 2.718282
```

In general with computer languages, there is an unwritten principle that unnecessary typing will be minimised if possible. This principle works here. All the brackets can be left off because the functions are all simple and can all work with just one input number and default parameters. If you are going to change any parameters, you will need to wrap that number in brackets. You should specify the parameter name for clarity.

```
-2 %>% abs %>% log(base = 3) %>% sqrt
```

```
[1] 0.7943109
```

If you typed *log(base = 3)* on its own, the command would produce an error because the x parameter is required, but the command will work in this format because the previous commands generate a value for x that is piped into the command.

The pipe feeds the log() function with the parameter x. But what if you wanted something more complicated? What if, instead of using the output of abs(-2) as the x-imput for log(), you wanted instead to feed the piped input to the base parameter of the log function and supply your own value of x? You can do this. The magrittr package has a special variable, which is "dot" (.) and this stands for the input number being passed around. This may make your commands look a little strange but the dot is a valid variable name. We will therefore supply 64 as the value to x and feed in the result of abs(-2) to the *base* parameter of log().

```
-2 %>% abs %>% log(x = 64, base = .) %>% sqrt
```

```
[1] 2.44949
```

The dot notation does not with operators (symbols such as _+-/*_). This line takes the original pipeline and tries to adds two to the result. It doesn't work and produces an error.

```
-2 %>% abs %>% exp %>% sqrt %>% . + 2
Error in .(.) : could not find function "."
```

The pipe symbol at the end of a line makes the line incomplete, and this means that R will create a continuation line if it sees a line ending in a pipe. We can use this to break up the calculation and put each step on a seperate line making a very clear script. It doesn't change the calculation in the slightest, but it makes the code more readable.

```
-2 %>%
  abs %>%
  log(base = 3) %>%
  sqrt
```

```
[1] 0.7943109
```

Pipes are particularly useful when functions with parameters are piled up. The parameter usage is obvious with a pipe, and it is confusing with traditional nested bracket coding. This is what I mean by a parameter pile-up. Look at this mess:

```
signif(sin(round(log(sqrt(log(pi , base = 3)), base = 2), digits = 2)), digits = 4)
```

```
[1] 0.03
```

This example is a little bit artificial, but messy pile-ups are quite common. Two functions, signif() and round() both have parameters with the same name (*digits*), which isn't surprising as they do similar things. There are two log() functions here, both with different *base* parameters.

To analyse what this function is doing, let's see it described in plain English:

Take the log to the base 3 of tne number pi, then take the square root of the result, then take the log to the base 2 of that, then round that number to 2 decimal places, then take the sine of that number and keep the first 4 significant figures.

Now imagine you would taking that text and coding it using nested brackets.

so you start by writing the log to the base 3 of pi:

log(pi, base = 3)

Then you add the square root function to all that. You would stick sqrt(and the beginning, then you'd move to the end of the line and close the bracket.

sqrt(log(pi, base = 3))

Then you would take the log of base 2 of all that. You would put log(at the front, then move to the end of the line and add a comma to seperate out the parameters, add the parameter name and value, and close the bracket.

log(sqrt(log(pi, base = 3)), base = 2)

And so on, back and forth until you complete the line. You would be constantly shuttling between the front and the back, or you would do all the stuff at the front and move to the back and add oll the parameters and close all the brackets. This is just crying out to go wrong.

If you get the order of the brackets mixed-up, your code will have errors that will be very difficult to unravel.

This is almost correct, but one of the parameters is in the wrong place. Can you see where it is wrong?

```
signif(sin(round(log(sqrt(log(pi , base = 3)), base = 2), digits = 2),digits = 4))
Error in sin(round(log(sqrt(log(pi, base = 3)), base = 2), digits = 2), :
2 arguments passed to 'sin' which requires 1
```

Now let's see how piping works. Let's remind ourselves of the English language version:

Take the log to the base 3 of tne number pi, then take the square root of the result, then take the log to the base 2 of that, then round that number to 2 decimal places, then take the sine of that number and keep the first 4 significant figures.

- Take the log to the base 3 of the number pi

```
log(pi, base = 3)
```

```
[1] 1.041978
```

- Then take the square root of the result

```
log(pi, base = 3) %>% sqrt
```

```
[1] 1.020773
```

- Then take the log to the base 2 of that result.

```r
log(pi, base = 3) %>% sqrt %>% log(base = 2)
```

```
[1] 0.02966244
```

- Then round that number to 2 decimal places

```r
log(pi, base = 3) %>% sqrt %>% log(base = 2) %>% round(digits = 2)
```

```
[1] 0.03
```

- Then take the sine of that

```r
log(pi, base = 3) %>% sqrt %>% log(base = 2) %>% round(digits = 2) %>% sin()
```

```
[1] 0.0299955
```

- Then keep the first 4 significant figures

```r
log(pi, base = 3) %>% sqrt %>% log(base = 2) %>% round(digits = 2) %>% sin() %>% signif(digits = 4)
```

```
[1] 0.03
```