Part 1 Section 1: R Basics

Aaron Robotham

Coding in R Basics

- **R** is free implementation of **S** (I kid you not), a language developed at Bell labs in 1976 that was focused on statistics. The design of **R** is also heavily influenced by **Scheme** and **Lisp**. It is called **R** (not **T**, which might be more obvious since it came later) because the earlier designers were "Ross Ihaka" and "Robert Gentleman", and they saw it as a slight pun on their first names.
- It is usually classed as a functional programming language rather than an imperative / procedural language (think **C** or **Java**), or more commonly now an object orientated programming (OOP) language. It has support for all of these paradigms (as does **Python**), but the default structure lends itself to functional code (whereas **Python** users are prodded towards imperative and OOP).
- It is the go-to language for statistical researchers, and is heavily focused on data analysis.
- It is the 6^{th} most popular general language on the 2025 PYPL Rankings, and top 14^{th} on the 2025 TIOBE index. It is usually the highest ranked domain specific language for data science, and a mixture of **Python**, **R** and **MATLAB** dominate for data science in general. In many fields (ecology, biophysics, geophysics) **R** is the dominant language.
- Has the "Central R Archive Network", or CRAN, a mirrored database of contributed and well maintained **R** extensions. About 18k are currently available, doing just about anything with data you can imagine. All *must* be fully documented, so **R** provides comprehensive help for all packages.
- Language design:
  - Deep down written in **C**, **C++**, **Fortran** and (recursively) **R**. As such, it can natively ran code written in these other languages (I do this often with **Fortran**).
  - An interpreted language (i.e. it returns answers as you go, like **IDL**, **MATLAB** et al). Also has compile and JIT options.
  - Dynamic type assignment, so no need to specify objects as being e.g. scalar / vector or double / integer etc up-front.
  - 1 indexed for vectors and matrices, like Fortran (note this, and be careful is coming from **C** or **Python** etc which are 0 indexed).
  - Case sensitive (no strict rules on style, but I like smallCamelCase (as opposed to BigCamelCase), under_score and dot.separate for naming, Just_not.allAt_once).
  - Has lexical scoping (this affects how variables can be accessed internally within function etc).
  - Free-form syntax guided largely by parentheses and expressions, and not code layout or indentations.
  - As of **R** 3.0.0 it is head-to-toe 64 bit, so **R** can access 2^{64} bits of RAM.
  - Uses S3, S4, Reference Class (RC or R5), R6 and S7 based object-orientation. They are increasingly complex and are intended for increasingly ambitious software (e.g. GUIs). Most packages do not use any OO, and are very simple to read. I use S3 when I am developing more complicated code. In principle S7 is meant to be integrated into base **R** to be the future of its OO system, but as of 2025 this has not happened (still a package on CRAN).
- To understand computations in R, two slogans are helpful (via John Chambers, a major contributor to **R**):
  - Everything that exists is an object.
  - Everything that happens is a function call.
- Nowadays most folks use **R-Studio** for interacting with **R**, writing and running scripts, and developing complex code/packages.

- You can script and run your **R** code from simple text files. On UNIX like systems you can also run **R** scripts like an executable by putting '#! /usr/bin/Rscript' on the first line.

Get R and R-Studio

R will work well on pretty much any modern operating system (e.g. OSX, Linux, Unix and Windows) with almost entirely inter-changeable code. This course will use additional packages, but it will attempt to keep to a minimal set in order to remove potential installation and version issues.

The first thing you should do is get version 4.4+ of **R** at cran.rstudio.com/

You will definitely want to use the **R-Studio** version of **R**, it is much more user friendly:

www.rstudio.com/products/rstudio/download/

Mac Users

You will likely also need to install **Xquartz** in order for openGL packages to work (we use these in some later parts of the course). You can get that here:

https://www.xquartz.org

You should not need to install separate compilers with any **R** after v4.0.0, but in case you are stuck on a museum version you can follow the extra instructions here:

https://cran.r-project.org/bin/macosx/tools/

Windows Users

Windows users might need to go through a couple of additional steps depending on how their system is set up, but most likely you will need to at least install *Rtools* for later parts of this course, which are available at https://cran.r-project.org/bin/windows/Rtools/ and follow the instructions about how to link these into your system path. You will know it is working because the following will not be empty:

```
Sys.which("make")
```

Linux Users

There are too many OS variants to detail how to get **R** for Linux working, so you will need to do your own sleuthing. My understanding is it often provided with the OS (but old versions), and easy to update with the supported package managers. But if you are using Linux at all you are a brave breed of student, so I am sure you are used to this. Good luck!

Useful Resources

These initial lecture will try hard to be self sufficient, but inevitably there will be times you need a bit more information.

A general summary is at:

https://education.rstudio.com/learn/beginner/

One of the main **R** authors and package contributors has curated a large amount of useful information:

https://r4ds.had.co.nz/introduction.html

A nice free interactive book can be found at:

https://intro2r.com

There are lots of nice online tutorials, but this one is neat because they are interactive (only some parts are free to access):

https://www.w3schools.com/r/

In general, Google is your friend when trying to find resources and solve problems when using **R**. If you feel overwhelmed by the non-**R** related hits (turns out **R** is not the most Google friendly name they could have picked), then try https://rseek.org, which uses Google but cleans out all of the non-**R** hits for you.

Getting Going

For most people (using **R-Studio**) you will just need to launch the application. If you are running from the command line then you will need to type 'R' in the terminal. This will print out a bunch of boiler plate about the version you are running etc, and give you a live prompt in which you can directly enter commands interactively. If you have a R.app (Mac) or R.exe (Windows) you can also launch **R** by double-clicking the relevant icon.

If you come across a package that we are using that you do not already have then this can be installed from CRAN via the **install.packages** function, e.g.:

```
install.packages('new_package') #you need speechmarks here
```

You then load it with the **library** function, e.g.:

```
library(new_package) #you do not need the speechmarks for loading
```

We will mostly be using base **R**, but we will explicitly load packages the first time they are needed in each chapter of the course.

Libraries needed for this chapter: magicaxis

```
library(magicaxis, quietly=TRUE)
library(bit64, quietly=TRUE)
```

Basic Usage and Getting Help

There is no type assignment necessary in **R**, so you can do this:

```
a = 10
a = a + 2
a = a/10 + sqrt(a) + a^1.3 - 1/a #notice we don't need to divide by 10.0
a
```

```
## [1] 29.87
```

Words followed by brackets, e.g. **sqrt** and **summary** above, are inbuilt (base **R**) functions. To find out more we can use '?'. This is very helpful for figuring out what we are seeing.

If in doubt in **R**, check out the ever-present help files. E.g. ?sqrt will tell us all about the square root function built into **R**.

One of the big strengths of **R** is its comprehensive documentation. All CRAN contributed packages *have* to meet a minimum level of documentation.

If you want to find something but only have an idea of the name you can use '??', e.g. ??hist for various functions that might be relevant to histograms via partial matching.

For online help www.rseek.org is very useful since 'R' is hard to Google. Also, the 'Astrostatistics' Facebook group is a useful forum for **R**, astronomy and statistics questions.

We Welcome our Robot Overlords

Since late 2022 **ChatGPT** (from OpenAI) has somewhat changed the game when it comes to coding assistance. I would encourage students to explore the suggestions it can provide and ask it questions, but be careful- I found it give confidently misleading answers for many of the assignment questions and would produce a failing effort. Also, getting something wrong because **ChatGPT** provided the wrong

answer is not a valid excuse (for this course, or professional work in general). Warnings aside, let's see how it does (this was Chat GPT v3.5 BTW):

"Hey ChatGPT, create a square matrix filled with numbers 1 to 16, multiply it by itself, and extract the square root of diagonal elements of the result."

```
# Step 1: Create a square matrix filled with numbers 1 to 16
n <- 4 # Define the size of the square matrix (n x n)
matrix_1_to_16 <- matrix(1:16, ncol = n, byrow = TRUE)

# Step 2: Multiply the matrix by itself
result_matrix <- matrix_1_to_16 %*% matrix_1_to_16

# Step 3: Extract the square root of diagonal elements from the result
square_root_diagonal <- sqrt(diag(result_matrix))

# Print the original matrix, the result matrix, and the square root of diagonal elements
print("Original Matrix:")
```

```
## [1] "Original Matrix:"
```

```
print(matrix_1_to_16)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8
## [3,]    9   10   11   12
## [4,]   13   14   15   16
```

```
print("Result Matrix (Matrix Multiplied by Itself):")
```

```
## [1] "Result Matrix (Matrix Multiplied by Itself):"
```

```
print(result_matrix)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]   90  100  110  120
## [2,]  202  228  254  280
## [3,]  314  356  398  440
## [4,]  426  484  542  600
```

```
print("Square Root of Diagonal Elements of the Result Matrix:")
```

```
## [1] "Square Root of Diagonal Elements of the Result Matrix:"
```

```
print(square_root_diagonal)
```

```
## [1]  9.486833 15.099669 19.949937 24.494897
```

And the accompanying explanation:

- In Step 1, we create a 4x4 matrix filled with the numbers 1 to 16 using the matrix() function and specifying byrow = TRUE to fill the matrix row-wise.
- In Step 2, we multiply the matrix_1_to_16 by itself using the %*% operator to perform matrix multiplication.
- In Step 3, we extract the diagonal elements of the result matrix using the diag() function and then calculate the square root of these elements using the sqrt() function to obtain the square root of diagonal elements.
- Running this code will display the original matrix, the result matrix (matrix multiplied by itself), and the square root of diagonal elements of the result matrix.

Re the use of ChatGPT (and similar) in coursework: you must fully understand your own solutions! If we are suspicious about the independence of assignment solutions (which includes thoughtless collusion etc) then you can be interviewed to assess your comprehension of the submitted work.

As of 2025, I find Microsoft Co-Pilot (which can be integrated into many IDEs and GitHub) is probably the best LLM software for assisting with **R** related problems and questions. In general it is not free though.

Not All Assignments Are Equal

Notice in the first example **R** code ($a = 10$) use the '=' operator to assign. Style wise, most books will tell you to use the **R** specific '<-' operator. E.g.:

```
a <- 10
a <- a + 2
a <- a/10 + sqrt(a) + a^1.3 - 1/a
a
```

```
## [1] 29.87
```

Now here I shall commit a big **R** sin, and tell you to use '=' rather than '<-'. For starters it is two characters rather than one. In fact the reason it exists at all is old APL keyboards used to have an explicit arrow key!

The other issue is that to a non-**R** native it just looks like you are asking 'is a less than minus 10?', so it is making you code unnecessarily obscure. In practice they are *very nearly* identical except for a few things:

```
b <- 4
a <- b
a
```

```
## [1] 4
```

```
b
```

```
## [1] 4
```

That is probably what you expected, but you can also do this weird assignment:

```
a <- 10
a -> b
a
```

```
## [1] 10
```

```
b
```

```
## [1] 10
```

I.e. '=' always assigns right to left, but the arrow assignment follows the direction of the arrow. Whilst perhaps handy, this is vanishingly rarely used, and if you are using it then you should probably be rewriting your code since it is hard to read. It does serve a purpose when piping though (which we see later). There is also this lethal typo to fear:

```
a< -10
```

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

This will actually answer the aforementioned question 'is a less than minus 10?'. If you are do this in a function things will break!

There is one special case (that we will discuss in more detail later) that the arrow notation does something powerful and different:

```
d <<- 10
d
```

```
## [1] 10
```

In practice you should also never use this type of assignment because it produces side effects in functions, and that is something you should avoid when practising functional programming (which we will be here). So basically, stick with '=', ignore the haters, and thank me later. So:

```
a = 10
a = a + 2
a = a/10 + sqrt(a) + a^1.3 - 1/a
a
```

```
## [1] 29.87
```

R Control Functions

R has the usual core control functions **if**, **for**, **else** and **while**. These take the following forms:

```
if(cond) expr
if(cond) cons.expr  else  alt.expr

for(var in seq) expr
while(cond) expr
```

We will not dwell on the details here since we will pick the syntax usage up from context in later examples in this chapter. Note that **while** is usually not used in safe **R** code since it can create infinite loops quite easily if you are not careful.

R Coding Style

I will not impose a very strict style in this course, but you will notice it is quite verbose when using brackets (like LISP). This is for safety, since both of the following work:

```
for(i in 1:2)
  for(j in 1:2)
  print(i + j)
```

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

```
for(i in 1:2){
  for(j in 1:2){
    print(i + j)
  }
}
```

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

But if you wanted to do anything else (like print $10i$ in the higher level loop), then things look a bit odd here:

```
for(i in 1:2)
  print(10*i)
```

```
## [1] 10
## [1] 20
```

```
  for(j in 1:2)
  print(i + j)
```

```
## [1] 3
## [1] 4
```

But work more logically in this case:

```
for(i in 1:2){
  print(10*i)
  for(j in 1:2){
    print(i + j)
  }
}
```

```
## [1] 10
## [1] 2
## [1] 3
## [1] 20
## [1] 3
## [1] 4
```

The confusion comes from the fact that code layout and indentation means almost nothing in **R** because it is expression based, and parentheses (i.e. () and {}) denote expressions. This is counter to languages like **Python**, where indenting it actually a requirement and means something syntactically. Style wise it is usually a good idea, but for the above we could have written:

```
for(i in 1:2) print(10*i)
```

```
## [1] 10
## [1] 20
```

```
for(j in 1:2) print(i+j)
```

```
## [1] 3
## [1] 4
```

Which makes it clearer that these two for loops are not embedded in this context, hence the outputs.

Written in this manner, the previously clearer example using {} now becomes harder to read!

```
for(i in 1:2){print(10*i)
for(j in 1:2){print(i+j)}}
```

```
## [1] 10
## [1] 2
## [1] 3
## [1] 20
## [1] 3
## [1] 4
```

What it is really doing is running everything inside the first for loop {}, and then everything inside the second for loop {}, fairly regardless of the newlines, and completely regardless of the indenting.

For the most part we will stick with verbose use of () and {}, with plenty of new-lines and indenting for clarity.

Types

Whilst **R** will dynamically create the number type for you (numeric double or integer are the most common), you can explicitly switch between them using the generic **as.???** interface. This method allow you to convert most **R** data types and structures back and forth. Here is an example of this:

```
vec.num = seq(0.5, 9.5)
vec.num
```

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

```
as.integer(vec.num)
```

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

One gotcha is that sometimes numbers that look like integers are in fact double (or "numeric" in **R** speak). This very rarely matters in **R** since it will convert types on the fly when doing mathematical calculations. If you *really* do need integers then you have two options: force the type (as above) or use the **L** notation:

```
is.integer(1)
```

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

```
is.integer(as.integer(1))
```

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

```
is.integer(1L)
```

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

However, for nearly any integer-like operation (say indexing etc) you can safely use the **R** double type and it will all work fine. In fact these integer-like doubles are sometimes called integer-ish for that reason, since the following will always work in **R**:

```
1L == 1
```

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

This does not work in some languages that are highly pedantic about data types, but it makes sense really - 1 is still 1 whether counting on a discrete or continuous number line.

Number Ranges

Like any language, **R** only supports a certain range of numbers:

For numerics this will be +/- 1.797693e+308, which is stored internally:

```
.Machine$double.xmax
```

```
## [1] 1.797693e+308
```

For integers this will be +/- 2^{32} - 1 = 2,147,483,647.

R does cheat a bit though. If you try to store an integer outside that range it will silently convert it to a double. This allows the accurate storing of integers to +/- 2^{52}, but not beyond:

```
print(2^53, digits=16) #correct
```

```
## [1] 9007199254740992
```

```
print(2^53 + 1, digits=16) #wrong, same as above!
```

```
## [1] 9007199254740992
```

If you need to work with integers beyond this then the best option is the **bit64** package which support 64 bit integers:

```
as.integer64(2^53) #correct
```

```
## integer64
## [1] 9007199254740992
```

```r
as.integer64(2^53) + 1 #correct
```

```
## integer64
## [1] 9007199254740993
```

Vectors

R can assign vectors in a number of ways. Imagine we want to make a vector containing the elements 1,2,3,4,5,6,7,8,9,10. The most long-winded way of doing this would look much like classic **Fortran** code.

```r
b0 = vector('integer', length=10) #pre-define the length of the vector to fill
for(i in 1:10){
  b0[i] = i
}
b0
```

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

Notice in the above that **R** is one-indexed!! This is very important to note. Many languages (**C**, **C++**, **Python**) are zero-indexed, meaning the first element is accessed via something like $b0[0]$, whilst in **R** it is accessed via $b0[1]$. There is a whole internet war over which is *more correct*, but all you need to know is that for various reasons **R** is one-indexed.

Anyway, forget the indexing for now. Luckily we can do the above assignment process *much* more simply in **R**:

```r
b1 = c(1,2,3,4,5,6,7,8,9,10)
```

Where **c** is the concatenate function. Even easier than this we can do:

```r
b2 = 1:10
```

The colon here is a special sequence operator, where $a : b = a, a + 1, a + 2, ..., b$. We can do this more verbosely using the **seq** function:

```r
b3 = seq(1,10, by=1) # By specifiying the gap between sequence elements
b3 = seq(1,10, len=10) # By specifiying the length of the sequence
```

We can check if all 10 elements are the same for $b1$ and $b2$ using vector boolean operation.

```r
b1 == b2
```

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

We can check for the exact equivalence of all elements with the **all** function

```r
all(b1 == b2)
```

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

```r
all(b1 == b3)
```

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

```r
all(b2 == b3)
```

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

Sometimes to save space in a script we can invoke a newline without writing one by using a semi-colon.

```r
all(b1 == b2); all(b1 == b3); all(b2 == b3)
```

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

Imagine now we want to multiply each element of our b vectors. We can do this the long-winded way:

```
for(i in 1:10){
  b0[i] = b0[i]*10
}
b0
```

```
##  [1]  10  20  30  40  50  60  70  80  90 100
```

Or the simple vectorised way:

```
b1 = b1*10
b2 = b2*10
b3 = b3*10
b1
```

```
##  [1]  10  20  30  40  50  60  70  80  90 100
```

```
b3*1  # This will not do anything
```

```
##  [1]  10  20  30  40  50  60  70  80  90 100
```

```
b3*1:2  # R will repeat the shorter vector as many times as required
```

```
##  [1]  10  40  30  80  50 120  70 160  90 200
```

```
b3*1:5  # which here is twice
```

```
##  [1]  10  40  90 160 250  60 140 240 360 500
```

```
b3*1:10  # and here is once
```

```
##  [1]   10   40   90  160  250  360  490  640  810 1000
```

Nearly all **R** functions are also vectorised, e.g.

```
sqrt(1:10)*(1:10)^2
```

```
##  [1]   1.000000   5.656854  15.588457  32.000000  55.901699  88.181631
##  [7] 129.641814 181.019336 243.000000 316.227766
```

An important aspect of structures in **R** is that you can nearly always apply logical operations to them, and get a similar shaped result, e.g.:

```
b3 %% 30==0 #where %% does modular arithmetic
```

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

If you want to know the locations of these TRUE outcomes you can wrap the logical statement in **which**:

```
which(b3 %% 30==0)
```

```
## [1] 3 6 9
```

Matrices

R natively supports matrix programming. To make a 3x3 matrix with elements 1:9:

```
mat1 = matrix(1:9,nrow=3)
mat1
```

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

```
mat2 = cbind(c(1,2,3),c(4,5,6),c(7,8,9)) # Using 'cbind' for binding columns
mat2 #same as mat1
```

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

```
mat3 = rbind(c(1,4,7),c(2,5,8),c(3,6,9)) # Using 'rbind' for binding rows
mat3 #again, same as mat1
```

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

```
summary(mat1) # Summary will give details of each column by default
```

```
##        V1            V2            V3
##  Min.   :1.0   Min.   :4.0   Min.   :7.0
##  1st Qu.:1.5   1st Qu.:4.5   1st Qu.:7.5
##  Median :2.0   Median :5.0   Median :8.0
##  Mean   :2.0   Mean   :5.0   Mean   :8.0
##  3rd Qu.:2.5   3rd Qu.:5.5   3rd Qu.:8.5
##  Max.   :3.0   Max.   :6.0   Max.   :9.0
```

```
sqrt(mat1) # Vector treatment works on matrices too, no need to loop!
```

```
##          [,1]     [,2]     [,3]
## [1,] 1.000000 2.000000 2.645751
## [2,] 1.414214 2.236068 2.828427
## [3,] 1.732051 2.449490 3.000000
```

```
mat1[1,3] # We can access matrices with [i,j], where i=row and j=column
```

```
## [1] 7
```

It is important to note that matrices collapse to vectors along columns first:

```
as.vector(mat1)
```

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

Like with vectors, we can execute logical statements on matrices:

```
mat1 > 5
```

```
##       [,1]  [,2] [,3]
## [1,] FALSE FALSE TRUE
## [2,] FALSE FALSE TRUE
## [3,] FALSE  TRUE TRUE
```

```
which(mat1 > 5) #TRUE positions when matrix is collapsed to a vector
```

```
## [1] 6 7 8 9
```

```
which(mat1 > 5, arr.ind=TRUE) #TRUE positions in matrix [i,j] notation.
```

```
##      row col
## [1,]   3   2
## [2,]   1   3
## [3,]   2   3
## [4,]   3   3
```

You can do matrix multiplication two ways:

```
mat1 * 1:3 # For multiplying each column vector by a 1:3 vector
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
```

```
## [2,]    4   10   16
## [3,]    9   18   27
```

```
mat1 %*% 1:3 # For traditional matrix multiplication
```

```
##      [,1]
## [1,]   30
## [2,]   36
## [3,]   42
```

The latter is equivalent to:

$$\begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix} \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} = \begin{pmatrix} 1 \times 1 + 4 \times 2 + 7 \times 3 \\ 2 \times 1 + 5 \times 2 + 8 \times 3 \\ 3 \times 1 + 6 \times 2 + 9 \times 3 \end{pmatrix} = \begin{pmatrix} 30 \\ 36 \\ 42 \end{pmatrix}$$

```
mat1 * mat2^2 # This will multiply each [i,j] element of mat1 by mat2^2
```

```
##      [,1] [,2] [,3]
## [1,]    1   64  343
## [2,]    8  125  512
## [3,]   27  216  729
```

```
mat1 %*% mat2^2 # Again, this is traditional matrix multiplication
```

```
##      [,1] [,2] [,3]
## [1,]   80  368  872
## [2,]   94  445 1066
## [3,]  108  522 1260
```

Other basic functions (transpose, determinant, eigen values etc):

- Transpose of a matrix

```
t(mat1)
```

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

- Determinant of a matrix

```
det(mat1)
```

```
## [1] 0
```

- Inverse of a matrix

```
mat4 = matrix(c(1,-2,3,-1),nrow=2) # Create an invertible matrix
inv4 = solve(mat4) #I agree, calling the matrix inverse 'solve' is a terrible name!
inv4
```

```
##      [,1] [,2]
## [1,] -0.2 -0.6
## [2,]  0.4  0.2
```

By definition $I = A^{-1}A$, where I is the identity matrix:

```
inv4 %*% mat4
```

```
##      [,1]          [,2]
## [1,]    1 -1.110223e-16
## [2,]    0  1.000000e+00
```

This does not work perfectly due to numerical noise, because it should be:

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

- Extract the diagonal components of NxN matrix, or create a diagonal matrix

```
diag(mat1)
```

```
## [1] 1 5 9
```

```
diag(1:3)
```

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

- Eigen values and eigen vectors of a matrix (these are covered in proper detail later in the course)

```
eig1 = eigen(mat1)
```

By definition of eigen vectors and matrices $A = V\lambda V^{-1}$:

```
eig1$vectors %*% diag(eig1$values) %*% solve(eig1$vectors)
```

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

- Singular value decomposition of a matrix

By definition of SVD $A = U\Sigma V^T$, where $UU^T = I$, $VV^T = I$, and Σ only has non-zero terms on the diagonal, where this non-zero diagonal can be noted with a vector D.

Given a matrix A, **R** lets us compute U, V and the diagonal element vector D:

```
svd1 = svd(mat1)
svd1
```

```
## $d
## [1] 1.684810e+01 1.068370e+00 5.039188e-17
##
## $u
##            [,1]        [,2]       [,3]
## [1,] -0.4796712  0.77669099  0.4082483
## [2,] -0.5723678  0.07568647 -0.8164966
## [3,] -0.6650644 -0.62531805  0.4082483
##
## $v
##            [,1]       [,2]       [,3]
## [1,] -0.2148372 -0.8872307 -0.4082483
## [2,] -0.5205874 -0.2496440  0.8164966
## [3,] -0.8263375  0.3879428 -0.4082483
```

```
svd1$u %*% diag(svd1$d) %*% t(svd1$v)
```

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

```
svd1$u %*% t(svd1$u)
```

```
##              [,1]         [,2]         [,3]
## [1,] 1.000000e+00 2.220446e-16 1.387779e-16
```

```
## [2,] 2.220446e-16 1.000000e+00 1.665335e-16
## [3,] 1.387779e-16 1.665335e-16 1.000000e+00
```

```
svd1$v %*% t(svd1$v)
```

```
##              [,1]         [,2] [,3]
## [1,] 1.000000e+00 1.665335e-16    0
## [2,] 1.665335e-16 1.000000e+00    0
## [3,] 0.000000e+00 0.000000e+00    1
```

The last two matrices should be identity matrices I, but due to numerical noise you will rarely see this in practice.

Arrays

R also supports multi-dimensional arrays (i.e. 3D data cubes and higher dimensions), see ?array for details. Every comma inside the square brackets indicates an extra dimension:

```
array3D = array(1:(2*2*2), dim=c(2,2,2))
array3D
```

```
## , , 1
##
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
##
## , , 2
##
##      [,1] [,2]
## [1,]    5    7
## [2,]    6    8
```

```
array3D[1,2,2]
```

```
## [1] 7
```

And more complex:

```
array5D = array(1:(2*2*2*2*2), dim=c(2,2,2,2,2))
array5D
```

```
## , , 1, 1, 1
##
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
##
## , , 2, 1, 1
##
##      [,1] [,2]
## [1,]    5    7
## [2,]    6    8
##
## , , 1, 2, 1
##
##      [,1] [,2]
## [1,]    9   11
## [2,]   10   12
##
## , , 2, 2, 1
```

```
##
##      [,1] [,2]
## [1,]   13   15
## [2,]   14   16
##
## , , 1, 1, 2
##
##      [,1] [,2]
## [1,]   17   19
## [2,]   18   20
##
## , , 2, 1, 2
##
##      [,1] [,2]
## [1,]   21   23
## [2,]   22   24
##
## , , 1, 2, 2
##
##      [,1] [,2]
## [1,]   25   27
## [2,]   26   28
##
## , , 2, 2, 2
##
##      [,1] [,2]
## [1,]   29   31
## [2,]   30   32
```

```
array5D[2,1,2,1,2]
```

```
## [1] 22
```

Data Frames

Data frames are similar to matrices, but different in a few key respects.

They are 2D structures that can hold $N \times M$ rows and columns of data. The key difference compared to a matrix is that a matrix *must* be of the same data type (e.g. all integer, or all double).

```
badmat = cbind(strings=rep('a',10), numbers=1:10)
badmat # Everything is forced to be a string
```

```
##       strings numbers
##  [1,] "a"     "1"
##  [2,] "a"     "2"
##  [3,] "a"     "3"
##  [4,] "a"     "4"
##  [5,] "a"     "5"
##  [6,] "a"     "6"
##  [7,] "a"     "7"
##  [8,] "a"     "8"
##  [9,] "a"     "9"
## [10,] "a"     "10"
```

```
# Numeric operations will throw an error:

try(badmat[,2]*2) # The try function will allow code to keep running even if a task fails
```

```
## Error in badmat[, 2] * 2 : non-numeric argument to binary operator
```

Data frames are more flexible though: each column can be a different **R** data type. This means we can create tables that are a mixture of characters and numerics using the *data.frame* function, and still carry out numerical operations on the numerics:

```
gooddf = data.frame(strings=rep('a',10), numbers=1:10)
gooddf # That's better!
```

```
##    strings numbers
## 1        a       1
## 2        a       2
## 3        a       3
## 4        a       4
## 5        a       5
## 6        a       6
## 7        a       7
## 8        a       8
## 9        a       9
## 10       a      10
```

```
# Numeric operations will work fine now:

gooddf[,2]*2
```

```
##  [1]  2  4  6  8 10 12 14 16 18 20
```

If you read in astronomy survey data from a file it will usually have a mix of data types: IDs might be integer and RA and Dec doubles, or even strings. **R** will automatically recognise the mix, and will store the object as a data frame.

You can access data frames in a few different ways:

```
gooddf[3,2] # Obvious enough, just like a matrix
```

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

```
gooddf[3,'numbers'] # Mixture of column name and row number
```

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

```
gooddf$numbers[3] # Hmm...
```

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

```
gooddf[[2]][3] # What the...?!
```

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

The last two techniques probably look confusing, but it because internally **R** stores data frames as a special type of 'list', which we will explore in more detail next. In both cases the syntax work by extracting the column of interest as a vector, and then the '[3]' on the rightmost tells **R** to extract the 3rd element of the extracted vector. This chaining of data extraction can be used almost everywhere, creating very compact but sometimes obscure code.

An important part of accessing data frames is using row logic to select subsets. E.g. you can do things like:

```
gooddf[gooddf$numbers > 5,]
```

```
##    strings numbers
## 6        a       6
## 7        a       7
## 8        a       8
## 9        a       9
## 10       a      10
```

Data Tables

It would be remiss not to mention data tables, which are available via the excellent **data.table** package:

```
library(data.table, quietly=TRUE)
```

```
##
## Attaching package: 'data.table'
```

```
## The following object is masked from 'package:bit':
##
##     setattr
```

They are very similar to data.frames, and can also be accessed via the $ symbol Why use them? Well, imagine we have the following two tables:

```
df1 = data.frame(a=1:10,b=21:30,c=(1:10)*4+5,d=c(rep(1,3),rep(2,4),rep(3,3)))
df1
```

```
##     a  b  c d
## 1   1 21  9 1
## 2   2 22 13 1
## 3   3 23 17 1
## 4   4 24 21 2
## 5   5 25 25 2
## 6   6 26 29 2
## 7   7 27 33 2
## 8   8 28 37 3
## 9   9 29 41 3
## 10 10 30 45 3
```

```
dt1 = data.table(a=1:10,b=21:30,c=(1:10)*4+5,d=c(rep(1,3),rep(2,4),rep(3,3)))
dt1
```

```
##          a     b     c     d
##     <int> <int> <num> <num>
##  1:     1    21     9     1
##  2:     2    22    13     1
##  3:     3    23    17     1
##  4:     4    24    21     2
##  5:     5    25    25     2
##  6:     6    26    29     2
##  7:     7    27    33     2
##  8:     8    28    37     3
##  9:     9    29    41     3
## 10:    10    30    45     3
```

They look pretty similar, but imagine you want to add a fifth column "e" that is $a^b + c - d$. The **data.frame** way would be extremely cumbersome:

```
df1$e = df1$a^df1$b+df1$c-df1$d
df1$e
```

```
##  [1] 9.000000e+00 4.194316e+06 9.414318e+10 2.814750e+14 2.980232e+17
##  [6] 1.705817e+20 6.571236e+22 1.934281e+25 4.710129e+27 1.000000e+30
```

But it can be done very elegantly with **data.table**:

```
dt1[,e:=a^b+c-d]
dt1$e
```

```
##  [1] 9.000000e+00 4.194316e+06 9.414318e+10 2.814750e+14 2.980232e+17
##  [6] 1.705817e+20 6.571236e+22 1.934281e+25 4.710129e+27 1.000000e+30
```

It is also easier to select row subsets:

```
df1[df1$a > 5,]
```

```
##     a  b  c d            e
## 6   6 26 29 2 1.705817e+20
## 7   7 27 33 2 6.571236e+22
## 8   8 28 37 3 1.934281e+25
## 9   9 29 41 3 4.710129e+27
## 10 10 30 45 3 1.000000e+30
```

```
dt1[a > 5,]
```

```
##        a     b     c     d            e
##    <int> <int> <num> <num>        <num>
## 1:     6    26    29     2 1.705817e+20
## 2:     7    27    33     2 6.571236e+22
## 3:     8    28    37     3 1.934281e+25
## 4:     9    29    41     3 4.710129e+27
## 5:    10    30    45     3 1.000000e+30
```

Now imagine we want to know the sum of a for subsets defined by d (which are valued 1, 2 or 3). Again the data.frame way is a bit of a pain (I admit I had to do some Googling, because I would *never* do this personally):

```
aggregate(df1$a, by=list(df1$d), FUN='sum')
```

```
##   Group.1  x
## 1       1  6
## 2       2 22
## 3       3 27
```

or less compactly:

```
df1sub = {}
for(i in unique(df1$d)){
  df1sub = c(df1sub,sum(df1[df1$d==i,'a']))
}
df1sub
```

```
## [1]  6 22 27
```

And **data.table**:

```
dt1[, sum(a), by=d]
```

```
##        d    V1
##    <num> <int>
## 1:     1     6
## 2:     2    22
## 3:     3    27
```

And if we instead wanted the sum to be of $a * b - c$? Easy!

```
dt1[, sum(a*b-c), by=d]
```

```
##        d    V1
##    <num> <num>
## 1:     1    95
## 2:     2   458
## 3:     3   662
```

This is an extremely powerful and fast (super fast, much faster than base **data.frame** operations) way to process data by group. Try doing the above with a **data.frame**. Or indeed in pretty much any other language.

Lists

R lists are highly flexible and generically complicated structures:

```
list1 = list(a=a, mat=mat1, vec=b1, note='Example list')
list1 # It is all there!
```

```
## $a
## [1] 29.87
##
## $mat
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
##
## $vec
##  [1]  10  20  30  40  50  60  70  80  90 100
##
## $note
## [1] "Example list"
```

There are a few ways to access this information:

- Select an element of a list, but keeps it in a list data structure (notice the $mat)

```
list1[2]
```

```
## $mat
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

- Select an element of a list, but uses the native data structure of the element in the list selected (so in this case the output is a matrix and no $mat)

```
list1[[2]]
```

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

- Select an element by the assigned name within the list, using the element's data structure

```
list1$mat
```

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

- Select an element by the assigned name within the list, using the list data structure

```
list1['mat']
```

```
## $mat
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

- Select an element by the assigned name within the list, using the element's data structure

```
list1[['mat']]
```

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

All of these options might be useful in different circumstances. You might want to loop through the list by the number of elements, or if the structure is complex you might want to extract a single element by name.

Lists can be much more complicated, and endlessly recursive:

```
list2 = list(sublist=list1, note='Even more complicated!')
list2 # There is no limit to how complex lists can be!
```

```
## $sublist
## $sublist$a
## [1] 29.87
##
## $sublist$mat
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
##
## $sublist$vec
##  [1]  10  20  30  40  50  60  70  80  90 100
##
## $sublist$note
## [1] "Example list"
##
##
## $note
## [1] "Even more complicated!"
```

This can make accessing multi-depth lists complicated. Here we will extract the matrix part, and then take out the first 2 rows of the 3rd column (7 and 8):

```
list2[[1]]$mat[1:2,3]
```

```
## [1] 7 8
```

The flexibility with ways you can access lists can be overwhelming:

- Select the first element of list2

```
list2[[1]]
```

```
## $a
## [1] 29.87
##
## $mat
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
##
## $vec
##  [1]  10  20  30  40  50  60  70  80  90 100
##
## $note
## [1] "Example list"
```

- Select the second element of the first element of list2

```
list2[[c(1,2)]]
```

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

- Select the 6^{th} of the 2^{nd} of the 1^{st} element of list2

```
list2[[c(1,2,6)]]
```

```
## [1] 6
```

Select elements 1 and 2 of list2 (which is the whole of list2!)

```
list2[c(1,2)]
```

```
## $sublist
## $sublist$a
## [1] 29.87
##
## $sublist$mat
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
##
## $sublist$vec
##  [1]  10  20  30  40  50  60  70  80  90 100
##
## $sublist$note
## [1] "Example list"
##
##
## $note
## [1] "Even more complicated!"
```

You do need to be careful when concatenating lists though. Consider the following:

```
c(vector=1:3, list(text='hello'))
```

```
## $vector1
## [1] 1
##
## $vector2
## [1] 2
##
## $vector3
## [1] 3
##
## $text
## [1] "hello"
```

```
c(vector=list(1:3), list(text='hello'))
```

```
## $vector
## [1] 1 2 3
##
## $text
## [1] "hello"
```

```
c(list(vector=1:3), list(text='hello')) #the same as the above
```

```
## $vector
## [1] 1 2 3
##
## $text
## [1] "hello"
```

In fact even empty lists are considered to be non-NULL objects, whereas an empty vector is NULL:

```
is.null(list())
```

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

```
is.null(c())
```

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

Which means even the following will change the type from a vector to a list:

```
c(1:3, list())
```

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

The advantage of using lists is that you can keep your workspace nice and clean. All the outputs of a function that produces many matrices (say, different versions of the same image produced during an image reduction process) can be stored in a single list structure. This make book keeping significantly easier.

Functions

R is usually used via functions (simple or complex). In fact, it is usually called a functional programming language. **R** does support object orientated programming, but for data analysis this is often discouraged for its use of opaque side effects. Almost everything that happens in **R** is a function call: this includes operators like '+' and '-'; sub-setting brackets '[]'; list dollars '$'; even normal brackets like '()' and '{}'.

We have seen some functions already (e.g. **sqrt**), but now we will write our own:

```
newfunc = function(x){
  return(x*10)
}
newfunc(4)
```

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

We can create default parameter value and multiple parameters easily.

```
newfunc2 = function(a=4, b=10){
  return(a*b)
}
newfunc2()
```

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

```
newfunc2(2)
```

```
## [1] 20
```

```
newfunc2(2, 40)
```

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

Notice in the above the inputs are matched by order, but you can also match by name.

```
newfunc2(b=40, a=2)
```

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

That said, it is good practice to provide inputs by name *and* in the expected order to make the code as readable as possible. We can check this for an unknown function using the **args** function.

```
args(newfunc2)
```

```
## function (a = 4, b = 10)
## NULL
```

We get even more power using **formals**, which actually allows us to manipulate the default arguments of defined functions without re-creating the whole function (this is often useful, and very powerful):

```
formals(newfunc2)
```

```
## $a
## [1] 4
##
## $b
## [1] 10
```

```
formals(newfunc2)$a=8
formals(newfunc2)
```

```
## $a
## [1] 8
##
## $b
## [1] 10
```

```
formals(newfunc2)[c('b','a')]=c(20,6)
formals(newfunc2)
```

```
## $a
## [1] 6
##
## $b
## [1] 20
```

A useful feature of functions is that you can natively send vector inputs and they will be utilised sensibly.

```
newfunc2(a=1, b=1:10)
```

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

```
newfunc2(a=1:10, b=1:10)
```

```
##  [1]   1   4   9  16  25  36  49  64  81 100
```

```
newfunc2(a=1:2, b=1:10)
```

```
##  [1]  1  4  3  8  5 12  7 16  9 20
```

Notice in the last case the shorter vector of a is recycled to match the length of b. For the most part vectors adhere to the principle of least surprise when being used in functions.

Returning Outputs

Notice above we returned the output by using the **return** function. If this is not supplied then the last completed operation becomes the default output and will be printed to screen *or* return to the assigned object.

```
noreturn = function(x){
  x*10
}
noreturn(4)
```

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

```
output = noreturn(4)
output
```

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

The reason to explicitly use return is that it is good practice (making exit point in code very clear), and allows you to return values conditionally in an easy manner without computing anything unnecessary:

```
noreturn2 = function(x){
  if(x < 10){
    return(x*10)
  }

  return(sqrt(x))
}

noreturn2(4)
```

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

```
noreturn2(16)
```

```
## [1] 4
```

Often you will not want the output of a function to be printed to screen (e.g. because it will generate a large object). Obviously given what I said above, this is not an issue if you are assigning the output of a function to an object (it will not be printed to screen anyway in this case). But it can still be a good idea if you do not want to accidentally print large volumes of data to screen. In this case we use **invisible**

```
invisfunc = function(x){
  invisible(x*10)
}

invisfunc(4)
```

To be safe, we should also wrap **invisible** with **return**, since **invisible** does not trigger a function return, rather it acts like we would expect when using no explicit **return**:

```
invisfunc2 = function(x){
  if(x < 10){
    invisible(x*10)
  }
  sqrt(x)
}

invisfunc2(4)
```

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

Perhaps surprisingly to some people, the above still returns 2 since the function continues even after it has evaluated the invisible statement. Many experienced **R** programmers use **invisible** in replace of

return (possibly not realising they are not exact behavioural synonyms), but when you have complex functions with multiple exit points you should *always* use **return** explicitly in combination:

```
invisfunc3 = function(x){
  if(x < 10){
    return(invisible(x*10))
  }

  return(sqrt(x))
}

invisfunc3(4)
invisfunc3(16)
```

```
## [1] 4
```

Infix Functions

R supports a special sort of function known as infix functions. They are a narrow range of functions that help create readable code, where they take exactly two inputs. The name is as opposed to prefix and postfix, where 'a+b' would be the infix version of the addition of a and b, '+a.b' the prefix, and 'a.b+' the postfix. The infix version should be the one that looks most normal to you, but others do get used (see Polish notation). Infix operators are defined such that in **R** code:

```
a %infix% b = function(a,b)
```

Where the named infix function always has to be surrounded by "%"s. **R** comes with a few of these such as %in%, %*% and %o%, and later we will see an example known as the pipe operator defined as %>%. Examples like %in% helps create readable code, since it checking wether elements of the LHS are *in* the RHS:

```
c(1,3,5,7) %in% c(2,3,4,5)
```

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

So above the second and third elements of the LHS exist in the RHS.

We can define our own infix functions easily (note the special back-tick: `):

```
`%longer%` = function(a,b){length(a) > length(b)}
```

This returns whether a is longer in length than b:

```
1:4 %longer% 2:4
```

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

```
1:4 %longer% 2:6
```

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

We can make simple infix head and tail routines:

```
`%head%` = function(a,b){a[1:b]}
`%tail%` = function(a,b){a[(length(a)-b+1):length(a)]}
```

```
1:10 %head% 4
```

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

```
1:10 %tail% 4
```

```
## [1]  7  8  9 10
```

Infix functions are not vital, but they often create simpler and more comprehensible code for no additional effort. They are also a way to cut down on **R**'s natural proliferation of brackets!

The Deeper Role of Functions

Notice above we used ticks to assign the function. This allows you to access reserved word objects. I mentioned earlier that '+' was a function- I was not kidding, it is actually an infix function we can access (although it is rarely referred to as an infix).

```
`+`
```

```
## function (e1, e2)  .Primitive("+")
```

So it actually executes a primitive (so very low level **C**) function that adds things. We can run this directly:

```
.Primitive("+")(1,2)
```

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

The '+' operator is actually much faster to execute (factor of 3 or so) due to less copying of data. It is also *way* easier to read, consider:

```
(1 + 2)*(3 + 4) # easy to read
```

```
## [1] 21
```

```
.Primitive("*")(.Primitive("+")(1,2),.Primitive("+")(3,4)) # what the...
```

```
## [1] 21
```

Somewhat dangerously, you can actually redefine '+' (or any function) locally in **R**:

```
`+` = .Primitive("-")
3 + 2
```

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

```
rm(`+`)
3 + 2
```

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

The fact **R** does this means it has a workaround for hardcore functional style programming (check out **Haskell** or **Erlang** for the logical conclusion of such an approach), and in practice you will never realise all this stuff going on in the background. This small part is mostly an interesting aside into the deeper workings of **R** (and if somebody is annoying you, just quietly change the behaviour of a BODMAS operator when they are not looking). Maybe even in their Rprofile...

BODMAS

In case the above was not clear, **R** uses the BODMAS order of operator precedence. That is it first executes (from left to right):

- Brackets: (), {}
- Operators: ^, sin, sqrt, log, etc
- Division and Multiplication: / and *
- Addition and Subtraction: + and -

This is functionally the same as PEMDAS (the more common name in North America). Note division and multiplication are grouped together- this means groups of such expressions get evaluated with equal precedence from left to right. Misunderstanding this fact is why you see viral argument about simple arithmetic such 1 + 2 / 2 x 2 where the answer is correctly 3, but confused sometimes as being 1.5. It is for these sorts of situations that you should make a healthy use of disambiguating brackets. The more the merrier really, since there is no computational cost.

Code Formatting

To help my own coding, I often use the informal rule that a use a space between addition and subtraction, and none for division and multiplication. This makes something like 2/3 + 5x6/8 - 2x4/5 easy to visually comprehend than 2/3+5x6/8-2x4/5 or 2 / 3 + 5 x 6 / 8 - 2 x 4 / 5 since the high precedence operations touch each other. Do not bother pointing out when I do not do this though, it is more an ambition than a rule :-)

Likewise, it is generally helpful to put spaces around assignments and after commas, but I tend not to do this within function calls because it can get too spaced out. E.g.:

```
format_ex = function(a=1, b=2){
  temp = a + a/b
  return(temp)
}
```

There are a few different style guides out there Google and Tidyverse, where you can see I differ in a few ways (using "=" rather than "<-" in particular, and more spaces).

Environments

A result of **R** being primarily functional it that it has a lot of mechanisms for simple functional code and working with environments. In this context the environment is the scope available for searching for variable values and assigning them. Because **R** is by default functional it creates many instances of local environments; usually within functions, loops and pretty much anywhere you see a bracket. Probably the key significance of this default coding paradigm is shown in the following bit of code.

```
a = 4

newfunc3 = function(x){
  a = x
  a
}

newfunc3(2)
```

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

```
a
```

```
## [1] 4
```

Was that what you expected? In imperative languages (e.g. **C**) you would often expect the global value of a to be changed. In **R** functions are created in their own environment, and modifications of objects only happen within their own environment. This means it is very hard to create accidental side effects — for the most part all you get out is the return part at the end of the function (exceptions are things like reading/writing and plots, which we will get to later).

We can modify the global value of a by using the **assign** function.

```
a = 4

newfunc4 = function(x){
  assign('a', value=x, envir=.GlobalEnv)
  a
}

newfunc4(2)
```

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

```
a
```

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

Remember much earlier when we discussed the pros and mostly cons of using the '<-' operator. This is an example where it can actually do something interesting:

```
a = 4

newfunc5 = function(x){
  a <<- x
  a
}

newfunc5(2)
```

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

```
a
```

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

Here the '«-' does something unusual and a bit dangerous— it searches higher level environments until it finds the object on the LHS, and if it finds that it exists it assigns the RHS value to it. Effectively the assignment happens in all higher environments. If the variable does not exist then it is created in the global environment (which almost everything else in **R** sits under). If this sounds a bit odd, then it is because it is, and you almost certainly should never be using this syntax because it is creating unexpected side effects.

The typical safe and local nature of environment assignment means you might see code doing things like:

```
for(i in 1:3){
  print(i)
  for(i in 1:3){
    print(paste('    ', i))
  }
}
```

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

where even the inside of a **for** loop exists inside of its own environment. Whilst the above code does not break things, it is bad practice stylistically. If in doubt use different names for different depths of loop nesting, e.g. the usual i,j,k or n,m,o.

Environments can be thought of as nested, meaning if you create an environment within another one the lower environment can see the contents of the higher environment, but **not** modify it (easily anyway). E.g.:

```
highfunc = function(x=1){
  lowfunc = function(a=4, b=8){
    x = x * 2
    return(x*a*b)
  }

  a = 10
  b = 20
```

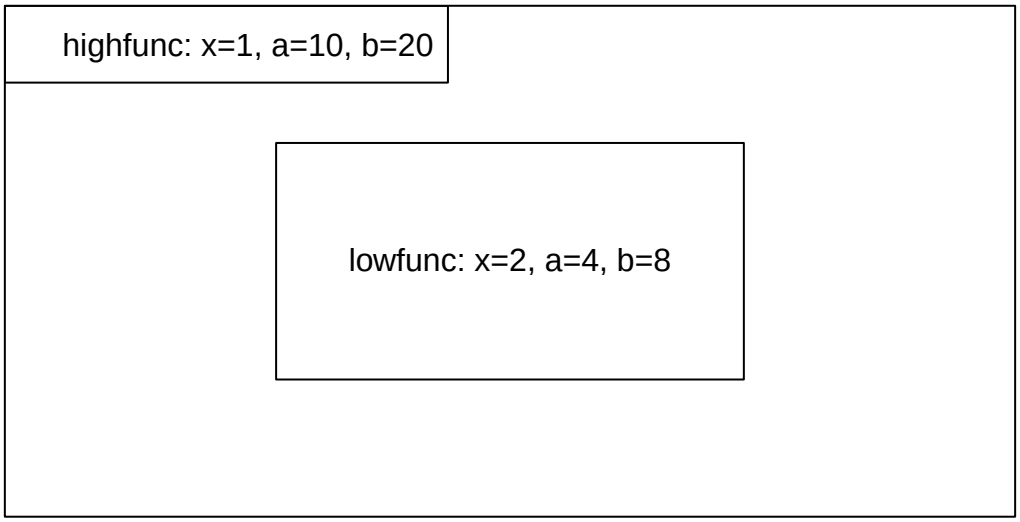
```
  temp = lowfunc()
  print(x)
  return(temp)
}

highfunc()
```

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

```
## [1] 64
```

Here **lowfunc** can see the value of x in **highfunc** and can create a local version of x where it is $\times 2$ larger. Since a and b are explicitly defined as parameters in **lowfunc** they do not inherit the modified values (10 and 20) in **highfunc**. Even though an environment hierarchy exists, objects that exist in the local environment always take precedence over identically named objects in different (even if higher) environments. Equally, after running **lowfunc** the value of x in **highfunc** is still what it was originally: 1 not 2. So after all of this, the output is $x_{lf} \times a_{lf} \times b_{lf} = 2 \times 4 \times 8 = 64$. We can visually show the environment state just before the function return in **lowfunc**.



Lexical Scoping

A key thing to realise in **R** is that it uses lexical scoping, which basically means it will search for a reference in the nearest environment in the hierarchy *when the function is defined*, **not** *when the function is run*. The latter would be dynamic scoping which some languages, like **S**, use instead (this is confusing to some, since **R** is based on **S**!). Try to figure out what the outcome of the following is in lexically scoped **R** and dynamically scoped **S**.

```
a = 1
b = 2

f = function(x){
  a*x + b
}

g = function(x){
  a = 2
  b = 1
  f(x)
}

g(2) #What comes out here?
```

```
a = 2
g(2) #And now?
```

The first answer for lexically scoped **R** is 4, and for dynamically scoped **S** is 5. The second answer for lexically scoped **R** is 6, and for dynamically scoped **S** is 5 (again). In **R**, because **f** was defined in the global environment and there is no local version of a and b it will use the values of a and b in the global environment. When these change (as in our second case when $a = 2$) so will the output for **f**. For a dynamically scoped language it will do this search dynamically when the function is called. In our example the first environment **f** sees outside its own is **g**, where a and b both have local definitions (2 and 1 respectively), so **f** would find these and stop looking any higher.

We can convert **f** to be dynamically scoped by defining it inside of **g**:

```
a = 1
b = 2

g = function(x){
  a = 2
  b = 1

  f = function(x){
    a*x + b
  }

  f(x)
}

g(2) #What comes out here?
a = 2
g(2) #And now?
```

Note in the above our **f** function does really have to be defined before the f(x) call. Some languages let you define functions after you call them because the compiler scans and compiles everything on an initial pass before going back to run things. Since **R** is interpreted it is executed literally in the order you read it.

Neither approach is *right* or *wrong*, but lexical scoping is considered to be more functional and predictable. This is because in practice the vast majority of codes define functions in the global environment, so people have a good sense of how to find the right values (just check the global environment). In **R**, defining functions within functions (whilst possible) is considered poor style and best avoided. In general you should aim to take the ambiguity out of the equation by writing the above more like:

```
g = function(x,a,b){
  a*x + b
}
g(2,1,2) # Earlier R solution
```

```
## [1] 4
```

```
g(2,2,1) # Earlier S solution
```

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

A good rule of thumb, if you find yourself worrying about the function scoping then you probably need to re-write your function and explicitly pass through more variables. I.e., it is best if the result is identical whether or not the language is lexically or dynamically scoped.

Joining the Dots

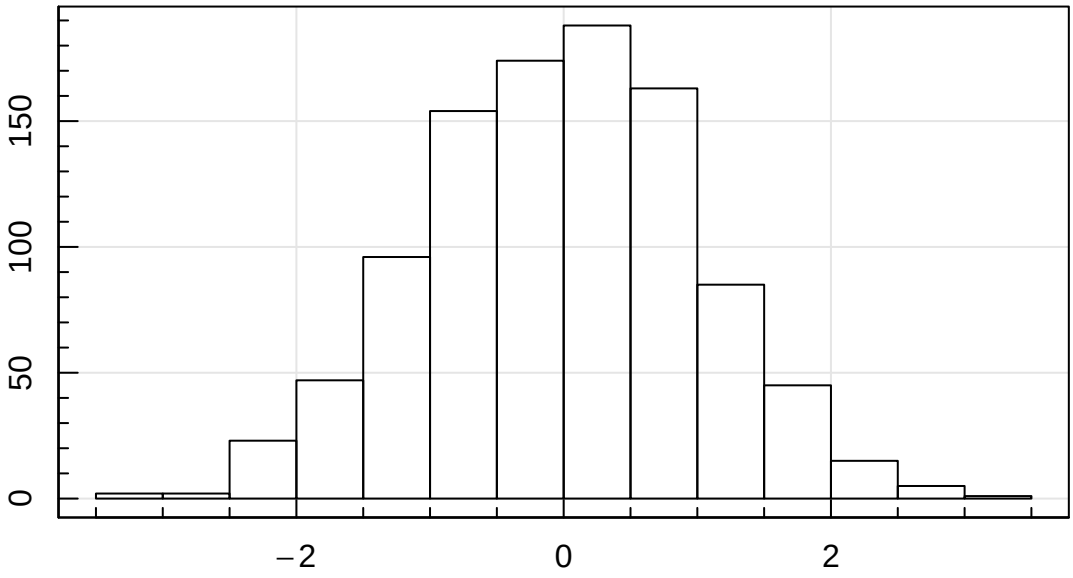
Related to how scoping works in **R** is the useful feature it has for passing down function arguments. Imagine you want to write a function that inside runs another function which might be able to take lots of arguments. Explicitly listing all of the those arguments in the higher level function might be a real pain. Instead we can use dots to denote an intention to pass down additional arguments.

```
plotnorm = function(n=1e3, mean=0, sd=1, ...){
  maghist(rnorm(n=n, mean=mean, sd=sd), ...)
}
```

The dots are a way of saying you might give some extra arguments (you do not have to), and if you do then pass them to the specified function. At the higher level any parameter provided that does not match the expected inputs is passed to the dots for use by a lower level function.

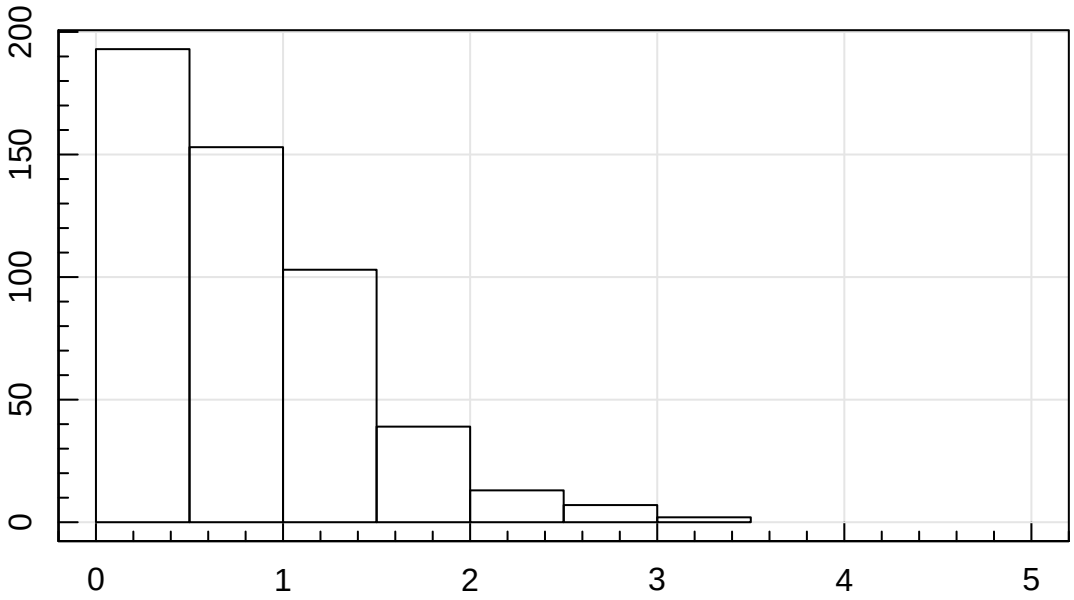
```
plotnorm()
```

```
## Summary of used sample:
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## -3.18994 -0.73391  0.00474 -0.01973  0.67629  3.12129
## Pop Std Dev: 1.0156
## MAD: 1.0516
## Half 16-84 Quan (1s): 1.0097
## Half 02-98 Quan (2s): 2.023
## Using 1000 out of 1000
```




```
plotnorm(xlim=c(0,5))
```

```
## Summary of used sample:
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## 0.003788 0.348477 0.674308 0.802404 1.124888 3.291711
## Pop Std Dev: 0.60197
## MAD: 0.57739
## Half 16-84 Quan (1s): 0.58037
## Half 02-98 Quan (2s): 1.1237
## Using 510 out of 1000 (51%) data points (490 < xlo & 0 > xhi)
```



Here *xlim* is not a named argument in **plotnorm**, and so is passed into **maghist** where it does have a meaning (as you might guess, it sets the x-axis limits).

For advanced function calling you can use the **do.call** function (often used in more complicated **R** packages). We will not discuss it further here, but check the relevant help pages if you are worrying about advanced function behaviour that cannot be solved with dots (e.g. deeply nested functions all requiring varying numbers of arguments).

Object Orientated Programing in R

As mentioned above, **R** does give uses access to a number of object orientated programming (OOP) systems: S3, S4, RC/R5 and R6. The first three are formally part of **R** and are increasingly complex and decreasingly popular. The last (R6) is actually a separate package that provides a lightweight and popular OOP system that is used for many popular concurrent systems like data base access and GUIs (**R-Studio** and **Shiny**).

A classic example of what OOP offers that functional programming does not is in state modification, where you both want to modify an object and return the modified component. Imagine we want to remove the top element of a vector and return both that value and update the vector state. To do this functionally we have to do something like:

```
pop = function(x){return(list(top=x[1], x=x[2:length(x)]))}

vector = 1:10
vector = pop(vector)
top = vector$top
vector = vector$x

print(top)
```

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

```
print(vector)
```

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

In OOP world we would instead be able to do something like (we will not go into the details of the syntax here, but this is actually using R6 style OOP):

```
top = vector$pop()
```

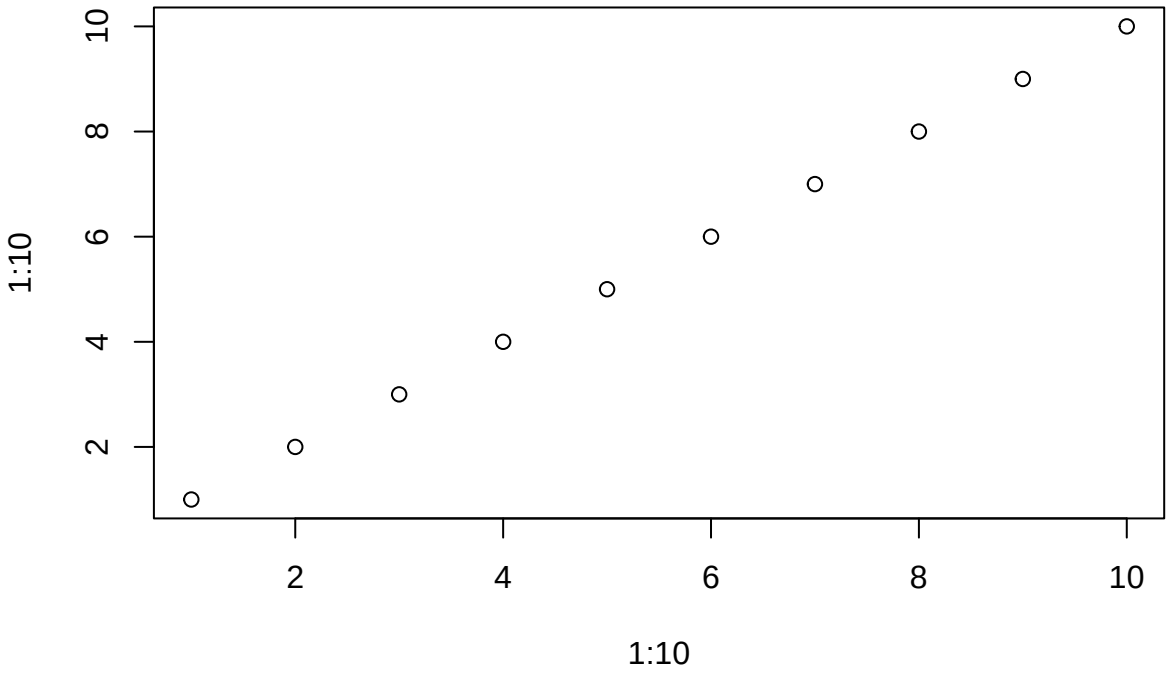
Where *top* would be the scalar value 1 (as before) and the state of the vector object would be modified in place. This can have obvious advantages in terms of code simplicity and even speed (much less unnecessary copying of data between objects). The down side is your code is potentially harder to read (certainly to fellow **R** users, who are not used to using OOP overall), and you are implementing side effects everywhere. This is very much against the functional paradigm we have been espousing, and for data science work I would argue it is a real concern.

A more complete introduction to OOP is beyond the scope of this section, but there are plenty of online resources that can be Googled for the very keen. S3 is still very popular since it is very close to functional in nature, after that R6 is probably the OOP mechanism of choice these days (full OOP, but easy to pick up, well documented and light-weight). For our purposes, everything from here on out will be using functional style programming bar some simple examples of S3 classes.

Plots and Graphics

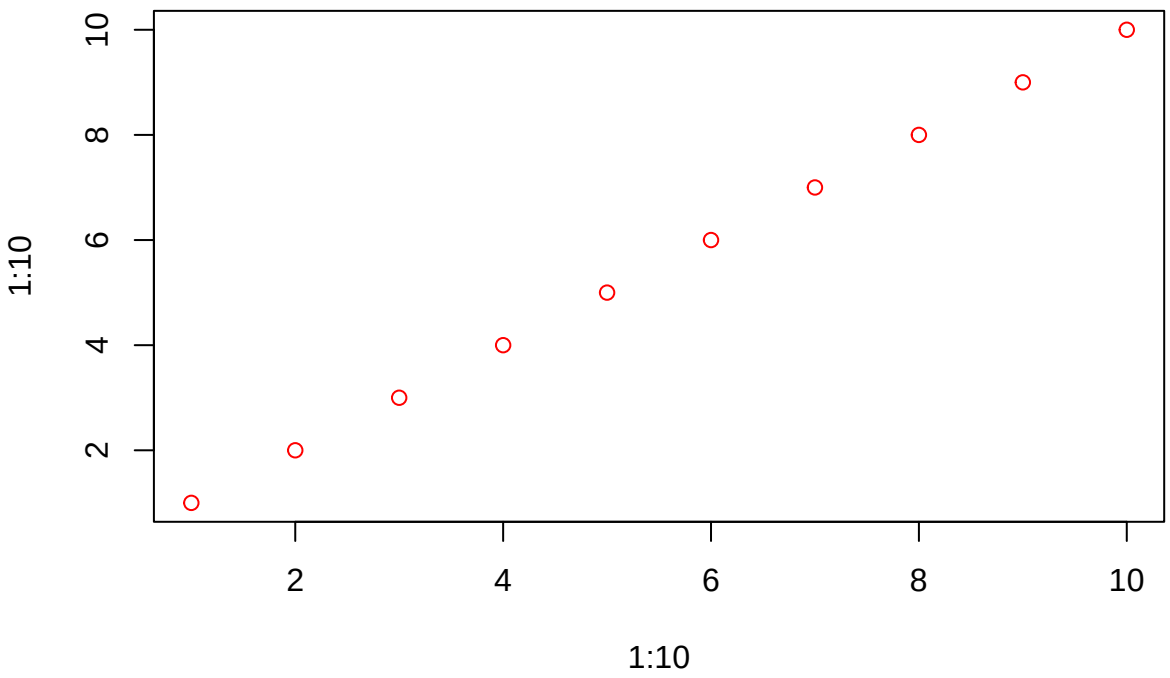
R is well geared to making simple plots straight out of the box using **base** graphics.

```
plot(x=1:10, y=1:10)
```

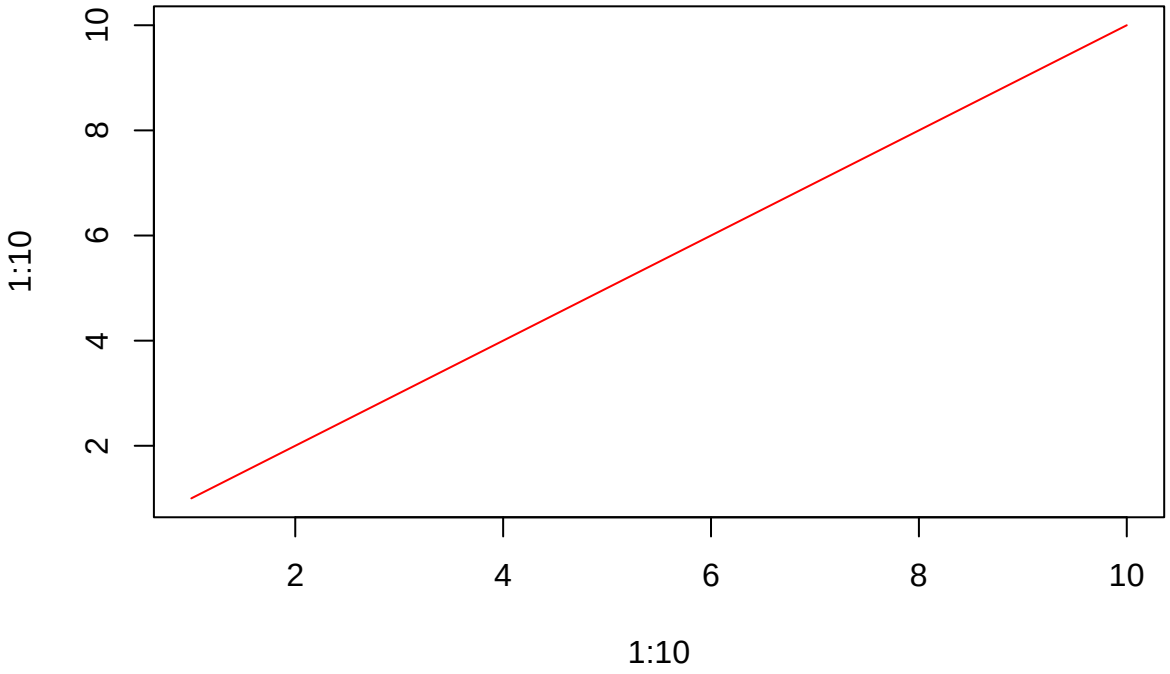


Pretty dull, let's add some colour (and since x and y are the first 2 arguments, these usually are not explicitly stated).

```
plot(1:10, 1:10, col='red')
```

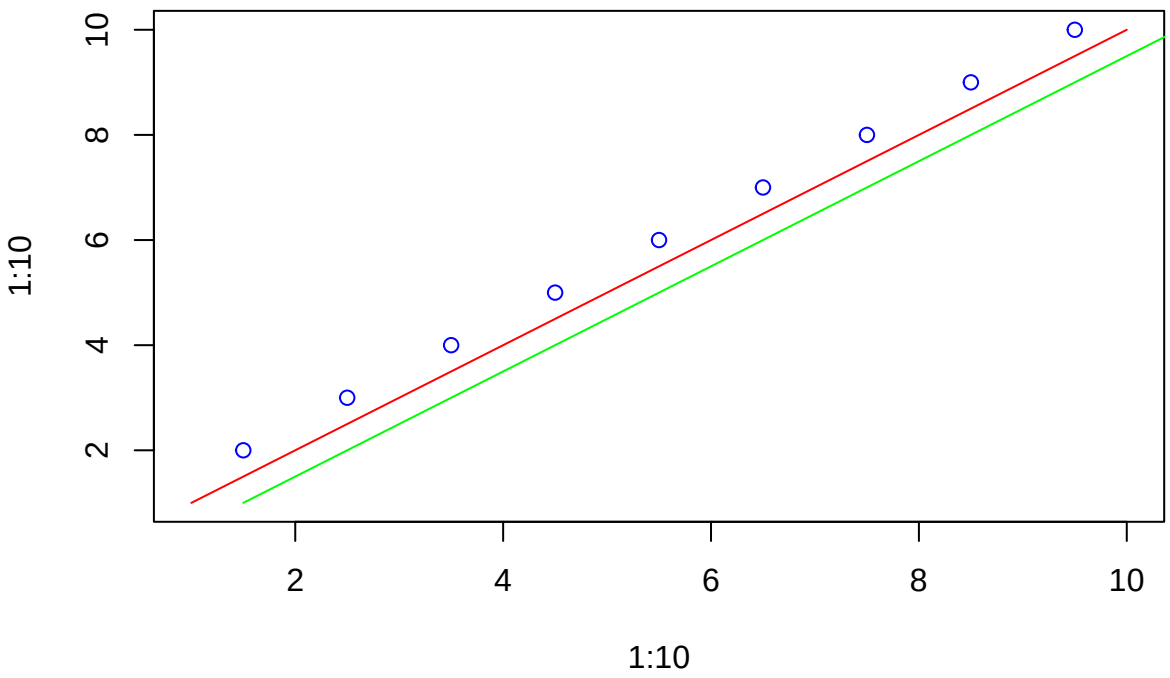



```
plot(1:10, 1:10, type='l', col='red')
```



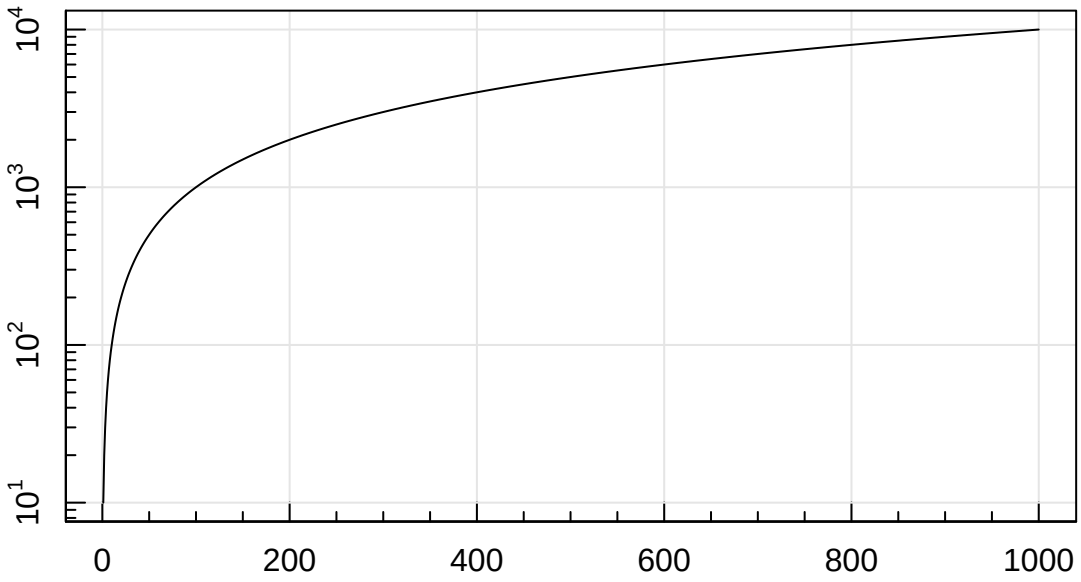
We can add objects to a plot with a variety of functions. **points** adds points and **lines** adds lines.

```
plot(1:10, 1:10, type='l', col='red')
points(seq(0.5,9.5,by=1), 1:10, col='blue')
lines(seq(1.5,10.5,by=1), 1:10, col='green')
```



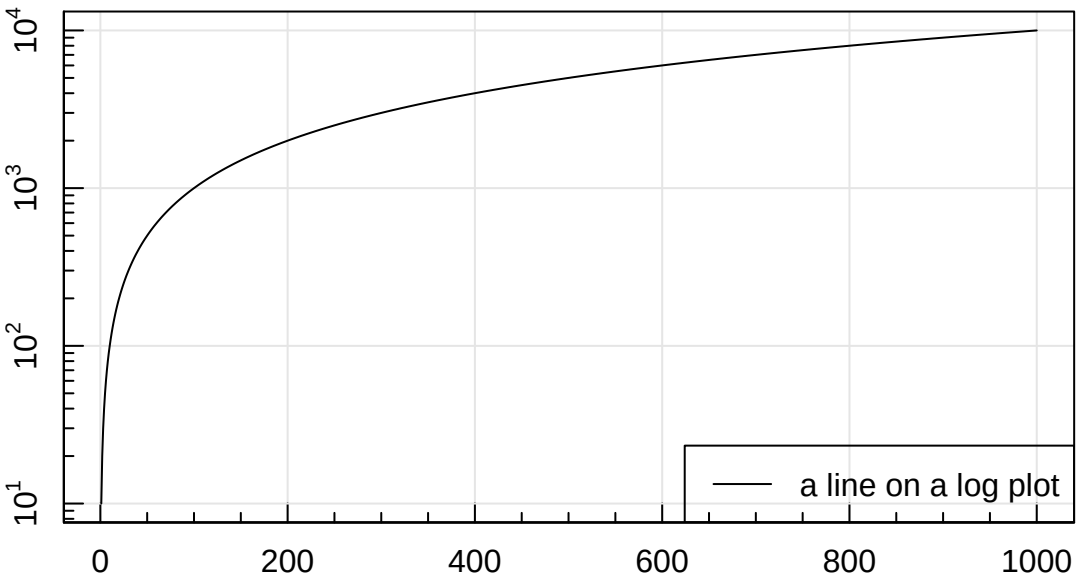
We can make log axes by using log='x' or log='xy' in the arguments. The **base** plotting default is not very pretty, so we will use the **magplot** function (part of the **magicaxis** package) instead:

```
library(magicaxis, quietly=TRUE) # Load but do not print out messages
magplot(1:1e3, newfunc(1:1e3), log='y', type='l')
```



And we can add a legend with the **legend** function:

```
magplot(1:1e3, newfunc(1:1e3), log='y', type='l')
legend('bottomright',legend='a line on a log plot',lty=1)
```



There are lots of other pretty and useful plot types available in **magicaxis**, have a look at ?magicaxis.

R has some great extra packages (instead of **base** and **magicaxis**) for plotting:

- **grid** (Element based plotting, rather than "pen and paper" like base.)
- **lattice** (Built on top of grid— popular for multivariate data.)
- **ggplot2** ('Grammar of graphics' implementation in **R**. This take some getting used to, but basically you create plots like you would describe them verbally. Many people *love* it, I personally don't quite click with it.)
- **rgl** (Interface to **OpenGL** interactive 3D graphics, making 3D interactive plots easy.)

Other Routes

You might come across guides using packages from the so-called 'tidyverse' suite of packages. Things like **dplyr** and **ggplot2**. We will not being using those here. I personally do not find them intuitive and often needlessly high level, obfuscating the code in many cases.

One notable exception is the **magrittr** package. This can simplify complex chains of functions with an additional infix operator called the pipe: '%>%'. It allows you to do this:

```
library(magrittr, quietly=TRUE)
```

```
rnorm(1000) %>% sd -> sd_out
sd_out
```

```
## [1] 0.9781876
```

Here **rnorm** is an **R** function that generates randomly Normal samples (1000 in this case), by default with a mean of 0 and standard deviation of 1 (we will look into various **R** distributions later). This is piped to the **sd** function that computes standard deviations of vectors. By the very definition of the original parameters of **rnorm** this should be ~ 1, which it is. Notice we use the right arrow operator '->' to do the final RHS assignment (I did say it would be useful for pipes). Compare this to the base **R** way of achieving the same outcome:

```
sd_out2 = sd(rnorm(1000))
sd_out2
```

```
## [1] 0.9749941
```

The reason some people like the former is you basically read it left to right: "randomly sample from the Normal, calculate the standard deviation of this, and pass it to sd_out". The latter reads more like:

"we have a variable called sd_out2, which will be the standard deviation of random samples from the Normal". Which you prefer probably comes down to whether you like to write in the active voice (we used **magrittr** pipes) or passive voice (base **R** was used by us).

Even though it can create quite elegant code, for the purposes of this course we will stick as close to base **R** as possible, so we will not use **magrittr** pipes in future examples. Feel free to give them a go though!

Update: Another Leaky Pipe

Just to confuse the issue, as of mid 2021 **R** version 4.1.0 has its own native pipe written '|>' (one whole less character!). It works a tiny bit differently, where function calls require explicit brackets, e.g.:

```
rnorm(1000) |> sd() -> sd_out
sd_out
```

```
## [1] 0.980781
```

It also has fewer features regarding where the argument is passed- it is basically **always** the first argument (but see update below). To get around this limitation they also introduced a more compact way of creating anonymous functions, e.g. instead of writing this:

```
(function(x){x^2})(2)
```

```
## [1] 4
```

you can write this lambda style function, where the word 'function' is represented by a '\':

```
(\(x){x^2})(2)
```

```
## [1] 4
```

Which means you can easily do this:

```
temp_pipe = function(a=2, b=4){
  return(a*b + b)
}

4 |> (\(x) temp_pipe(b=x))()
```

```
## [1] 12
```

The reason base **R** uses this clunkier version is for reasons of speed (it is much faster) and debugging ease. This is a bit clunker than the **magrittr** pipe solution though, where you can use a '.' as a placement wildcard:

```
4 %>% temp_pipe(b=.)
```

```
## [1] 12
```

As of 2022 you can now use '_' for a native wildcard!

```
4 |> temp_pipe(b=_)
```

```
## [1] 12
```

The only downside is you can only use one wildcard per function (whereas **magrittr** allows multiple), but that covers most use cases in practice. It seems the native pipe is pretty elegant to use finally :-D

So after all of that, we **still** will not use pipes in this course, even the base **R** pipes. You are welcome to use them in your work though (just do not ask us for help).

Rmarkdown

These lectures are all written in pure **Rmarkdown**, which is an extremely simple language heavily based on the open standard **markdown**. Students are required to use **Rmarkdown** for doing their

assignments. I also recommend it when working through the material in this course in general, it should make the workflow much easier to manage and to update etc.

We will not cover the syntax in detail here, but please have a look at the basics here:

rmarkdown.rstudio.com/articles_intro.html

Once you learn the basics, it is very easy to work with, and you will naturally pick up the more advanced features as you go.

The company that makes **R-Studio** (**Posit**) has recently started a new project called **Quarto**. This is a standalone or embedded (in R-Studio and other IDEs) software product that can compile **Rmarkdown**, but it also natively supports other languages (e.g. Python, Julia) and engines (e.g. Jupyter). I've not tried it out personally, but feel free to give it a try. Note all assignments will still need to be provided as **Rmarkdown** (.Rmd files) and as compiled PDFs.

Summary

Hopefully this brief introduction has given the keen student plenty of grist for the mill. From this point you should be confident of reading most vanilla base **R** code, and certainly able to do basic operations and sub-setting. For convenience here we include two pages of the useful **R** cheat sheet made available by **R-Studio**. Note these will not compile for other people unless you put the appropriate figures in an accessible path for your Rmd file (these are in the figures folder at the course tinyurl).

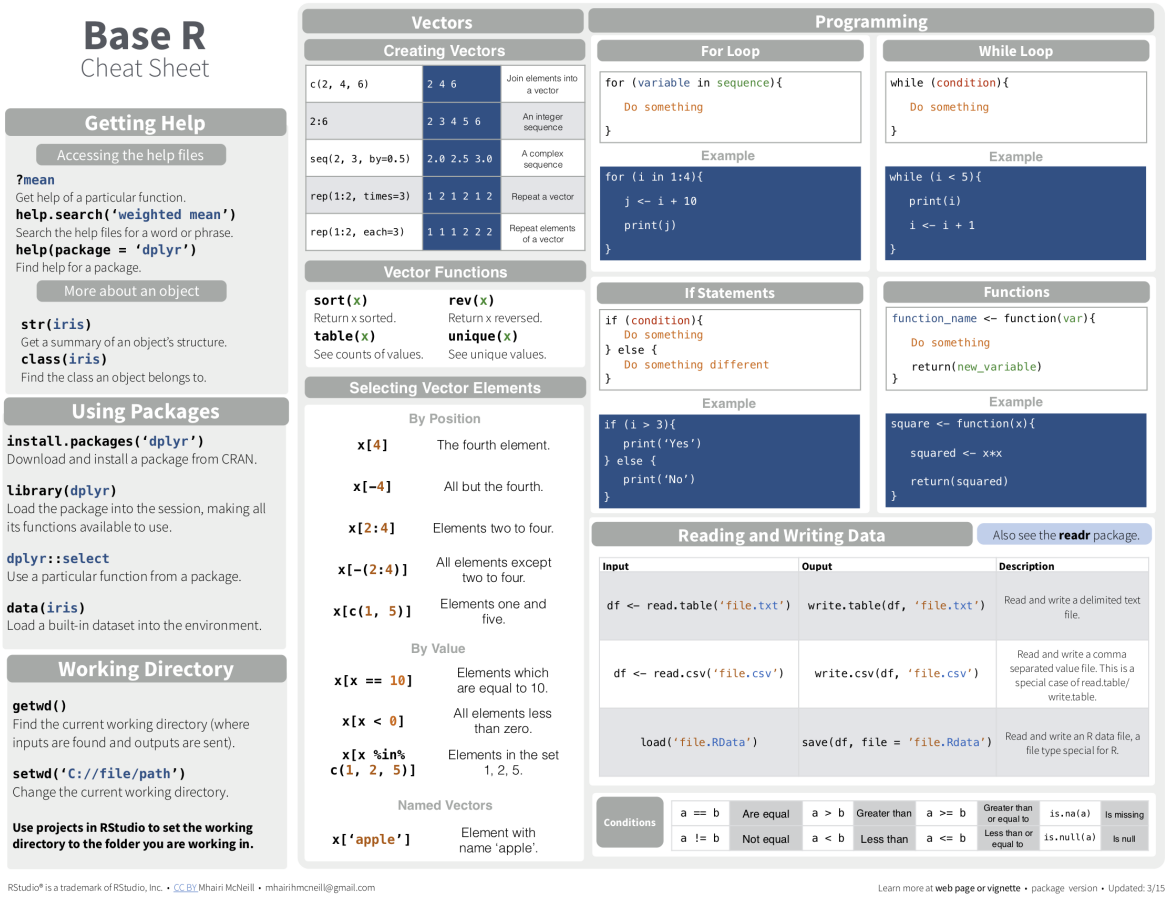


Figure 1: R cheet sheet page 1 from https://www.rstudio.com/resources/cheatsheets/

There is a huge amount of online material available, so it should not be necessary to buy a particular text to answer a question you might have. Stack-Exchange is an excellent avenue to find answers for

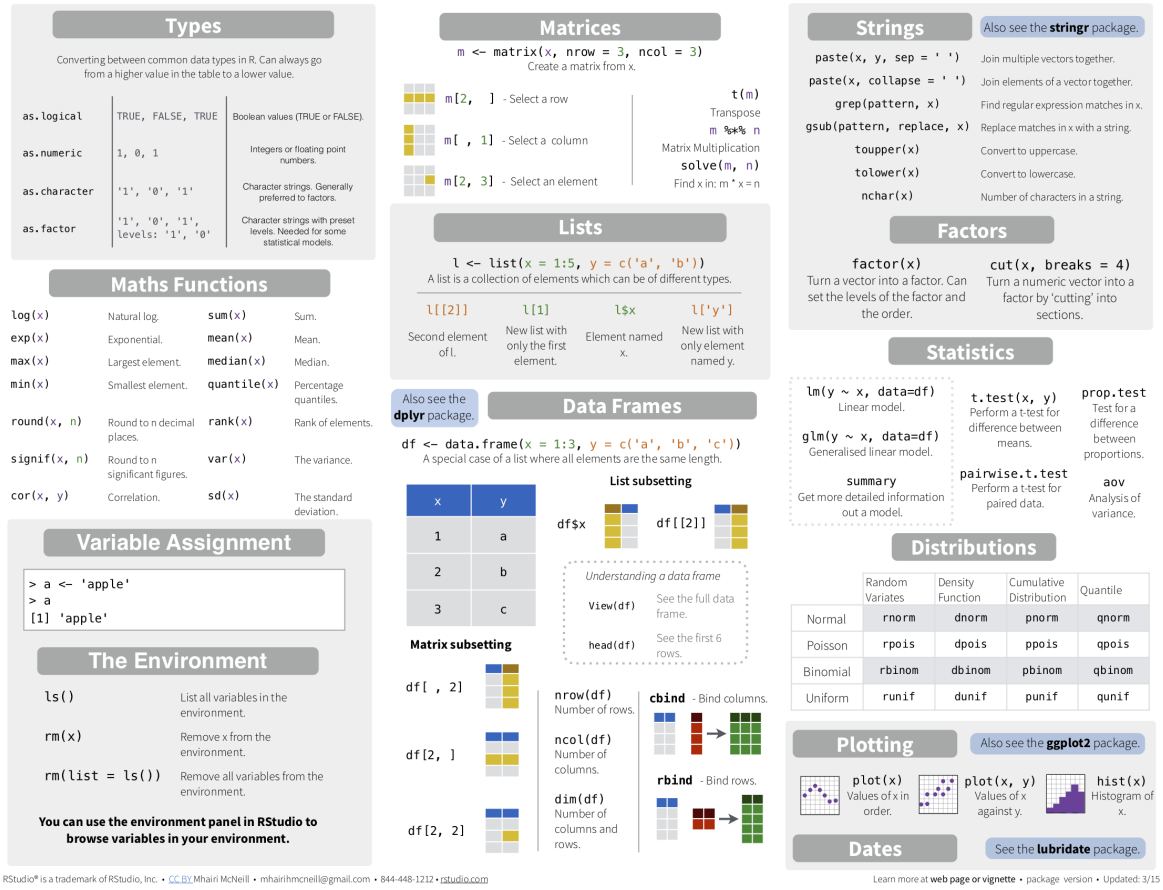


Figure 2: R cheet sheet page 2 from https://www.rstudio.com/resources/cheatsheets/

'dumb' questions (you will never be the first person to ask any **R** question), and a bit friendlier than the notoriously spiky **R** mailing list (and mailing lists are very '90s in any case).