Introduction to R

Mark Puttick (marknputtick@gmail.com)

23/04/2018

Contents

Introduction to R

R is a widely-used and accessible programming language. In recent years, it has become a major tool for in palaeobiology. In this course I will introduce the basic elements of using R.

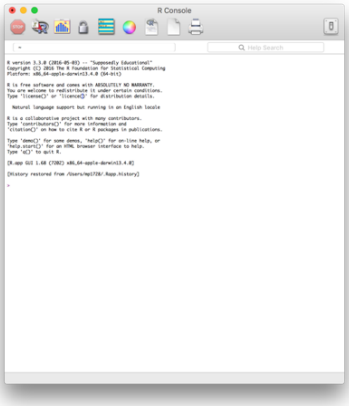
First thing first – installing R

(If you have R already skip this section)

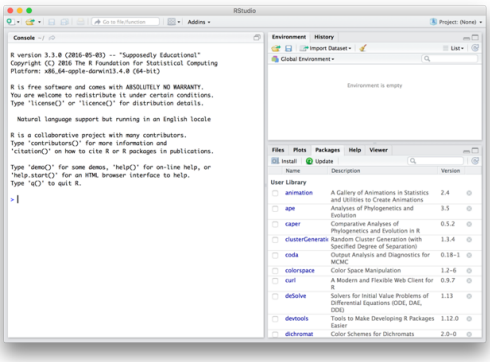
Visit https://cran.r-project.org and select your operating system (Windows, Mac, Linux). Simply follow the installation instructions Once you have R installed you may also wish to install RStudio https://www.rstudio.com/products/rstudio/download/ Using R or RStudio is largely a matter of personal preference

R screen

Open R or RStudio With R you will be presented with a screen similar to this:



R studio:



This is the R console. We are presented by the prompt '>' and we can type there. We can type the simple command 2 + 2 into the console and press Enter. Hopefully, we will see the following:

```
2 + 2
```

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

We have performed our first calculation in R and we will now look at some of the basics of the R language.

Objects

One of the most fundamental processes in R is the creation of Objects. An Object is simply a representation of single or multiple numbers and integers, characters, a logical (TRUE or FALSE), or complex mathematical numbers... To assign a value to an object we use the 'less than' sign < and a 'minus sign' –, so '<-' We can assign some objects

```
a <- 3
b <- 4
c <- 5
```

Here object a is given the value 3, b = 4, and c = 5

NOTE An equals sign = can be used instead of <- but it is generally better to use <- to avoid confusion with the use of = elsewhere in the R language, such as in logical arguments

So we can do simple addition

```
a + b
```

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

```
c + 3
```

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

This has NOT changed the value of c

```
c
```

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

As a calculator R uses familiar symbols for mathematical functions:

- addition `+`
- subtraction `-`
- multiplication `*`
- division `/`
- powers `^`
- square-root `sqrt` (also `^0.5`)

R uses the standard BODMAS rules for the order of calculations: Brackets, Operations (i.e powers, square roots), Division, Multiplication, Addition, Subtraction.

For example:

```
a - b
```

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

```
a * b
```

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

```
a / b
```

```
## [1] 0.75
```

```
a^2
```

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

```
sqrt(b)
```

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

```
b^0.5
```

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

```
a^2 * (a + b) - a
```

```
## [1] 60
```

Task One

- Create an object called `object_one` and assign it your favourite number.
- Raise this number to the power of 3 and save it as `object_cubed`
- Divide `object_cubed` by `object_one` and save this as `object_three`
- Divide `object_three` by `object_one` and overwrite the result as `object_one`
- Subtract the square-root of your least favourite number from `object_one`, save this as an object called `final`, and print the result on screen

Note

R is case-sensitive. Object `a` is NOT the same as object `A`. If we substitute `A` for R will print an error message

Vectors

In R we often encounter data structure called vectors which is an object containing multiple, independent values (in R jargon it is a data structure). For example, we may wish to represent species diversity in some formations. We do this by using the function `c` (more on functions soon) which combines (or concatenates) numbers into a vector

```
c(3, 4, 9, 2, 1, 7)
```

```
## [1] 3 4 9 2 1 7
```

If we perform an action it will apply to each element of the vector individually

```
c(3, 4, 9, 2, 1, 7) - 1
```

```
## [1] 2 3 8 1 0 6
```

Or we can perform a different action to each element by using another vector. Here we subtract the vector from itself

```
c(3, 4, 9, 2, 1, 7) - c(3, 4, 9, 2, 1, 7)
```

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

A calculation with a vector of unequal lengths will result in repetition of the smaller vector so that it equals the length of the longer vector. This is easier to understand with an example So this

```
c(3, 4, 9, 2, 1, 7) - c(1,2)
```

```
## [1] 2 2 8 0 0 5
```

Is the same as

```
c(3, 4, 9, 2, 1, 7) - c(1, 2, 1, 2, 1, 2)
```

```
## [1] 2 2 8 0 0 5
```

We can also create vectors using some base R language

```
1:10
```

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

```
10:1
```

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

Also, `seq` is very helpful when creating vectors

```
seq(from=100, to=10, by=-10)
```

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

Task Two

Create vector called `task_two` with elements 4, 8, 32 and create a second vector that doubles the first number, multiplies the second number by 3, and halves the final number in one operation

Factors

Factors are similar to vectors except they specify a finite number of data values. That is they are a way of storing categorical data – this can be useful for statistics, for example when performing ANOVA So we create a simple vector:

```
1:3
```

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

And turn it into a factor

```
factor(1:3)
```

```
## [1] 1 2 3
## Levels: 1 2 3
```

We can create a simple factor with characters rather than numerical values:

```
factor(c("Triassic", "Jurassic", "Cretaceous"))
```

```
## [1] Triassic   Jurassic   Cretaceous
## Levels: Cretaceous Jurassic Triassic
```

Matrix

A very common way to store data in R is in the form of a matrix. This is very common in phylogenetics, for example, storing species' data values. As a toy example, we will create an object vector of numbers:

```
species_data <- 1:10
matrix(species_data)
```

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

We can specify a number of columns by adding the argument `ncol` in the `matrix` function

```
matrix(species_data, ncol=2)
```

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

By default R will matrices fill a column then move to the next. We can change this by changing `byrow` to equal `TRUE` so it fills across rows:

```
matrix(species_data, ncol=2, byrow=T)
```

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

Task Three

- Create `my_matrix` with 5 rows and 3 columns
- Fill `my_matrix` with integers from 1 to 15 in reverse order, and so the numbers decrease by one across the row (the first row should go: 15, 14, 13)
- Create a new matrix, `myMatrixTwo`, with 5 rows and 1 column that contains the outcome of `column one - column two - column three` from `my_matrix`

Functions

R is an excellent calculator, and can handle different ways to store data. However, the reason we use R is to actually perform some sort of analysis using our data and we do this using functions. We have already encountered functions above, so it is time for a formal introduction Functions in R take some input, perform some calculation using an algorithm, and present an output. That applies to everything we want to do in R: sorting data, calculations, and plotting data. The simplest functions require just input. Often we can also modify arguments in the function to modify the process the function performs. Many functions are part of the base of R. Later we will see how we can also obtain functions from specialised packages, and w ewill also see how to build our own. So we will specify our data

```
my_data <- c(15,3,6,2,8,7,9,3,5,7,1,3,6,90,2)
```

We often would like to know the `mean` of our data along with other common summary statistics

```
mean(my_data)
```

```
## [1] 11.13333
```

```
median(my_data)
```

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

```
min(my_data)
```

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

```
max(my_data)
```

```
## [1] 90
```

Minima and maxima can also be calculated using `range`

```
range(my_data)
```

```
## [1]  1 90
```

In fact, we can summarise all of this, along with 25% and 75% quartiles, using summary

```
summary(my_data)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    1.00    3.00    6.00   11.13    7.50   90.00
```

We can also examine a number of descriptive statistics, such as variance and standard deviation

```
var(my_data)
```

```
## [1] 488.6952
```

```
sd(my_data)
```

```
## [1] 22.10645
```

We can also perform multiple functions at once. So as standard deviation is the square root of the variance, we can make life difficult for ourselves by using the variance function `var` inside the square-root function `sqrt`

```
sqrt(var(my_data))
```

```
## [1] 22.10645
```

Often we will have incomplete data. In R these missing data are represented by `NA` signifying 'non-applicable'. We can add missing data to our data using the `c` function. We will overwritre the object `my_data`

```
my_data <- c(my_data, NA)
```

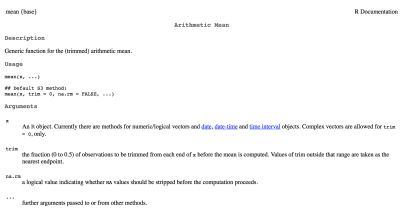
What is the mean now?

```
mean(my_data)
```

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

Oh dear, that's not very helpful. Is there something we could do? We could ask for help and we'll hopefully see a window like this

```
?mean
```



mean {base} R Documentation

Arithmetic Mean

Description

Generic function for the (trimmed) arithmetic mean.

Usage

```
mean(x, ...)

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

Arguments

x	An R object. Currently there are methods for numeric/logical vectors and date, date-time and time interval objects. Complex vectors are allowed for trim = 0, only.
trim	the fraction (0 to 0.5) of observations to be trimmed from each end of x before the mean is computed. Values of trim outside that range are taken as the nearest endpoint.
na.rm	a logical value indicating whether NA values should be stripped before the computation proceeds.
...	further arguments passed to or from other methods.

Within this screen we see the following: the name of function `mean`, its source `{base}`, description, usage, arguments. Is there something to help with our NA problem? Yes! `na.rm` "a logical value indicating whether NA values should be stripped before the computation proceeds" – i.e, ignore NA So we can change the argument `na.rm=TRUE`

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

```
## [1] 11.13333
```

Task Four

- Run the command `set.seed(109)`
- Create the object `random_numbers` that saves the output of the function `rnorm(100)`
- Find the mean, median, range, variance, standard deviation of the object `random_numbers`
- Find where the 95th quantile data of `random_numbers` (Hint – try `?quantile`)

Subsetting data

Vectors

Often we may like to take a subset of our data or identify where a specific value occurs in our data. To retrieve a specific element or elements we use the single square bracket operator `[ ]`

Again, we will take the object `my_data`

```
my_data <- c(15,3,6,2,8,7,9,3,5,7,1,3,6,90,2, NA)
my_data
```

```
##  [1] 15  3  6  2  8  7  9  3  5  7  1  3  6 90  2 NA
```

If we want to select the first element in the vector

```
my_data[1]
```

```
## [1] 15
```

Or the first five elements,

```
my_data[1:5]
```

```
## [1] 15  3  6  2  8
```

If we wish to select non-consecutive numbers we use the `c` function. So to select the 2nd and 10th element,

```
my_data[c(2, 10)]
```

```
## [1] 3 7
```

The `head` and `tail` functions also select the n^{th} first or last elements on a vector. By default n = 5, but we can modify this

```
head(my_data, n=1)
```

```
## [1] 15
```

```
tail(my_data, n=1)
```

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

We will add some names to the data. As this is fake data, we can be lazy and use some R tricks. First we will count the number of elements in our vector using `length`

```
length(my_data)
```

```
## [1] 16
```

In R letters are built-in so we can obtain some simple names from the alphabet

```
letters
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q"
## [18] "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

So we have some letters and know how many we need to provide names. Here we subset letters to match the length of `my_data`

```
letters[1:length(my_data)]
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p"
```

```r
names(my_data) <- letters[1:length(my_data)]
```

Removing NAs

Often in R we want to get rid of missing data values. As we encountered above, we can deal with NAs but sometimes we wish to remove them entirely. Where do NAs occur in our vector? Let's utilise the `is.na` which does as its name suggests – across each element it asks if that element is NA and returns a logical vector

```
is.na(my_data)
```

```
##     a     b     c     d     e     f     g     h     i     j     k     l
## FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##     m     n     o     p
## FALSE FALSE FALSE  TRUE
```

We can see that that last element is NA as it is marked TRUE. We can verify this is NA by first using the `which` function that will take the logical vector as input and return the value that is marked TRUE

```
which(is.na(my_data))
```

```
##  p
## 16
```

The 16th element of `my_data` is NA. Finally, we can remove these elements missing data. If we use – in a [ ] such as [-16] in a vector

```
my_data[-16]
```

```
##  a  b  c  d  e  f  g  h  i  j  k  l  m  n  o
## 15  3  6  2  8  7  9  3  5  7  1  3  6 90  2
```

This has removed the 16th element and retained everything else

We can make this more reproducible by using the `which` function

```
my_data[-which(is.na(my_data))]
```

```
##  a  b  c  d  e  f  g  h  i  j  k  l  m  n  o
## 15  3  6  2  8  7  9  3  5  7  1  3  6 90  2
```

Let's overwrite `my_data` to remove the NA

```
my_data <- my_data[-which(is.na(my_data))]
```

Outliers

So which element is the maximum value? Obviously as our data is only 15 elements long so we could simply look at the data. However, we may be dealing with elements with 100s or 1000s of entries or wish to automate the process. In which case, we'll get R to tell us.

We can utilise one of two hugely useful functions to do this. The first is which that takes a vector and assesses it against another value. For example, we ask which element of `my_data` is equal to the maximum value in `my_data`

```
which(my_data == max(my_data))
```

```
##  n
## 14
```

This tells us the 14th element in the vector, named n, is the maximum value.

Alternatively, we could use the function match to get the same result (a truism in R is there are multiple ways to do the same thing)

```
match(max(my_data), my_data)
```

```
## [1] 14
```

So far we only know the where in the vector the maximum element occurs and its name, not the value of that element – we need to subset the data. Using `which`

```
my_data[which(my_data == max(my_data))]
```

```
##  n
## 90
```

Or using `match`

```
my_data[match(max(my_data), my_data)]
```

```
##  n
## 90
```

Matrix

Again, we will set up some data

```
species_data <- 1:40
species_matrix <- matrix(species_data, ncol=4, byrow=T)
```

Subsetting matrices is very similar to vectors except we need to give two values – the row number and column number. We do this by in square brackets [row, column] To select the entire first row:

```
species_matrix[1,]
```

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

Or the entire first column:

```
species_matrix[,1]
```

```
##  [1]  1  5  9 13 17 21 25 29 33 37
```

Select the 4^{th} element of the 2^{nd} column:

```
species_matrix[4,2]
```

```
## [1] 14
```

Task Five

Run the commands:

- `randMat <- matrix(rnorm(1000), ncol=50, nrow=20)`
- `randMat[sample(1:1000, 10)] <- NA`
- Find where missing data occurs in `randMat`
- Create a vector of the complete data from `randMat` and find the mean
- Find the mean of each row of randMat (Hint – search `?apply`)

Plots and plotting plots

One of the key features of R is its ability to produce publication quality and flexible plots. We will utilise some in-built R data. Specifically we will use the data mtcars which comprises of "data from the 1974 Motor Trend US magazine, and comprises fuel consumption and 10 aspects of automobile design and performance for 32 automobiles (1973–74 models)". Fascinating stuff.

To call up these data we will use `data`

```
data(mtcars)
```

And then see how these data are presented using the is function

```
is(mtcars)
```

```
## [1] "data.frame" "list"       "oldClass"   "vector"
```

The first element here is important. We can see the data are stored as a `data.frame`. A `data.frame` is a matrix, but unlike an R `matrix` it allows for the storage of different types of values (i.e, a `data.frame` can store numerical and character data simultaneously, but a 'matrix can not)

To call up these data we will use `head` to print the first six rows

```
head(mtcars)
```

```
##                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

As we know these data are in `matrix` form we can use the `dim` function to analyse how many rows and columns are contained in these data

```
dim(mtcars)
```

```
## [1] 32 11
```

This tells us there are 32 rows and 11 columns

As mtcars is a data.frame we can use the functions `names` and `attach` to tell us the column names of data, and store them as individual objects

```
names(mtcars)
```

```
##  [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"   "gear"
## [11] "carb"
```

```
attach(mtcars)
```

for example, `mpg`

```
mpg
```

```
##  [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2
## [15] 10.4 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4
## [29] 15.8 19.7 15.0 21.4
```

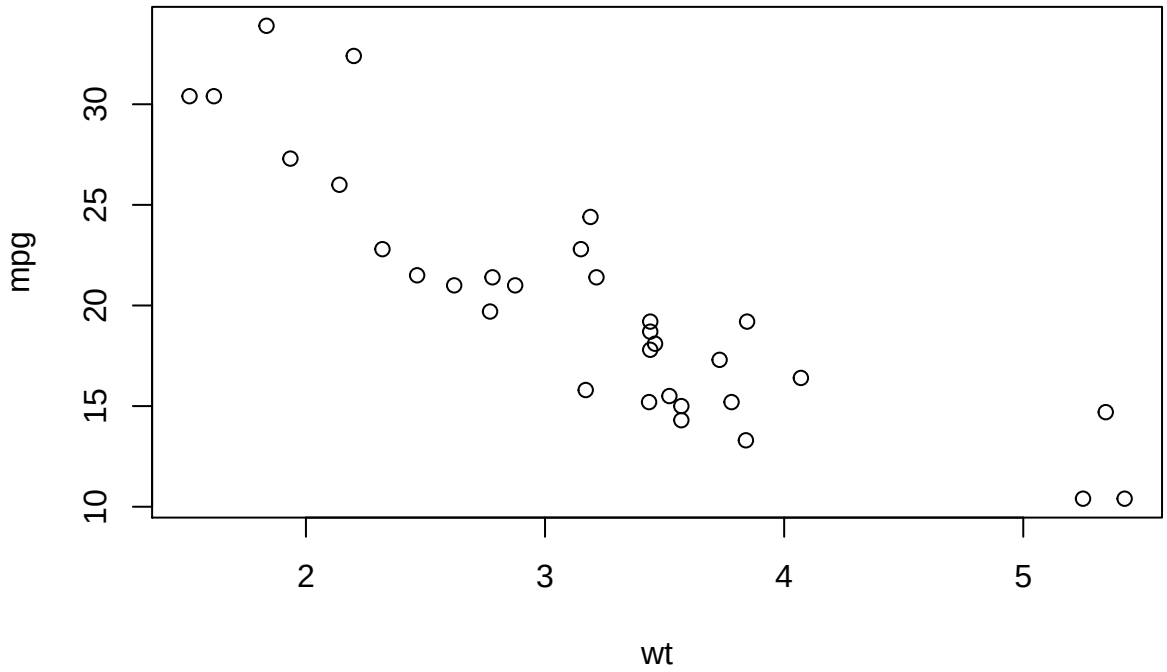
we can use various functions on these data, such as `summary`

```
summary(mpg)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   10.40   15.43   19.20   20.09   22.80   33.90
```

We can plot scatter plots using 'plot' to plot the plot of miles per gallon (mpg) against weight (wt)

```
plot(x=wt, y=mpg)
```



other common plotting options:

- Histogram: `hist`
- Barplot: `barplot`
- Boxplot: `boxplot`
- Pie: `pie`
- Contour: `contour`

common plot options

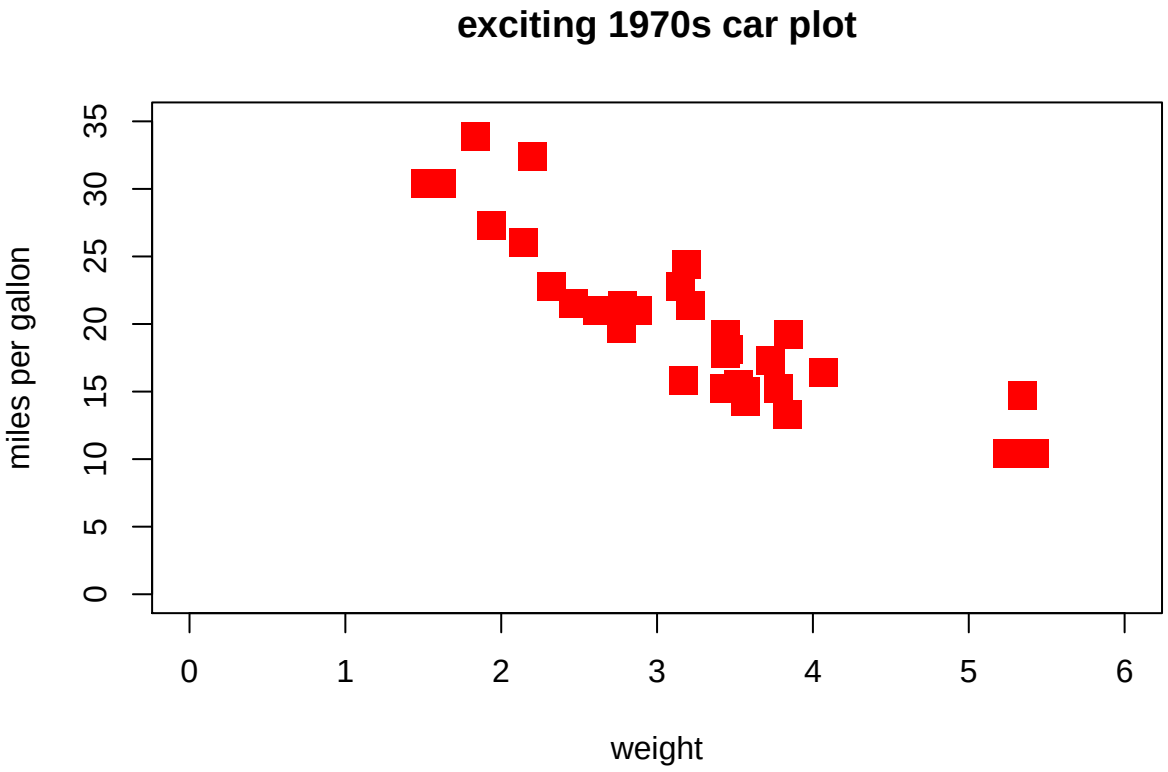
- colour of plot labels `col="red"`
- linetype `lty=2`
- plot character – i.e, symbols `pch=3`
- character expansion – symbol size `cex=2`
- axis labels `xlab = "x", ylab = "y"`
- plot title `main = "title"`
- range of the axis `xlim=c(min,max)`

There are many, many more options. You can explore many of the options by using

```
?par
```

We saw that a lot of plot options are default as in the plot above but we can implement some of the new arguments

```
plot(wt, mpg, pch=15, cex=2, col="red", xlab="weight",
ylab="miles per gallon", xlim=c(0, 6), ylim=c(0, 35),
main="exciting 1970s car plot")
```



Task Six

- Plot `qsec` as a function of `wt` with red triangles
- Then try to label each of the points with name of the car it represents (Hint: try the function `text` after plotting)

Loops

Loops are very common in R. Basically it is a way of repeating a process x number of times without having the do it manually.

Suppose we want to print the following statement 1000 times

"All R and no play makes Jack a dull boy"

Unless you're planning to spend the winter in a empty, haunted hotel I recommend automating the process with R.

One way is a `for` loop. A `for` loop has the syntax

```
for (i in 1:5) { expression }
```

This will take the counter i and from 1 to 5 use i in an expression. So we could just print i to screen to show what a `for` loop does

```
for(i in 1:5) print(i)
```

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

The i counter can also be used as input in functions. For example, we could take the function `rep` and tell it to repeat an input i times

```
for(i in 1:5) print(rep("loopy", i))
```

```
## [1] "loopy"
## [1] "loopy" "loopy"
## [1] "loopy" "loopy" "loopy"
## [1] "loopy" "loopy" "loopy" "loopy"
## [1] "loopy" "loopy" "loopy" "loopy" "loopy"
```

Here the function `rep("loopy")` has been repeated five times, but each time the number of repetitions has changed according to the value of i.

We can use them for more sensible purposes. For example, we could estimate Euler's number e which is equal to about 2.718282 according to the following infinite series: $e = 1 + 1/2! + 1/3! + 1/4!...$

! stands for factorial (i.e, 3! is 3 * 2 * 1). In R we can use the function `factorial` to estimate e

We'll start with one

```
euler <- 1
```

We can then add to this in a loop

```
for(i in 1:100) euler <- euler + 1 / factorial(i)
```

And what to we get?

```
euler
```

```
## [1] 2.718282
```

Or we can use loops for other ridiculous things

```
for (i in 1:10) {
print("All R and no play makes Jack a dull boy")
}
```

```
## [1] "All R and no play makes Jack a dull boy"
## [1] "All R and no play makes Jack a dull boy"
## [1] "All R and no play makes Jack a dull boy"
## [1] "All R and no play makes Jack a dull boy"
## [1] "All R and no play makes Jack a dull boy"
## [1] "All R and no play makes Jack a dull boy"
## [1] "All R and no play makes Jack a dull boy"
## [1] "All R and no play makes Jack a dull boy"
## [1] "All R and no play makes Jack a dull boy"
## [1] "All R and no play makes Jack a dull boy"
```

Probably the most common use of loops is to iteratively loop over an input, and store the output in some form. For example we will generate 1000 numbers from a normal distribution using `rnorm`

```
set.seed(453)
randomVar <- rnorm(1000)
```

Note

`set.seed` is used here to make the code reproducible as it sets the random number generator starting value

Suppose, for some reason, we wanted subtract each value from its previous value in the randomVar vector. For the second occurrence we would do something like

```
randomVar[2] - randomVar[1]
```

```
## [1] 2.311621
```

And we can build this into a loop. We will go from 2:1000 and we will end up with n-1 values (we can not compare entry 1 to entry 0 as entry 0 doesn't exist). We will also want to store our output. So we will create a new empty vector `loopResults`. We can subscript both our input vectors and the growing output vector to store all of these results during the loop using `[ ]`

```
loopResults <- c()
```

In our loop will we take the i^{th} occurrence and subtract the i^{th}*-1* occurrence and store this as i^{th}*-1* result. Sounds complicated?

Probably easier to look at our loop set up

```
for(i in 2:1000) {
  loopResults[i-1] <- randomVar[i] - randomVar[i-1]
}
```

The focal number is i and we will subtract *i-1*. This requires us to start the loop at 2 not 1. i.e for the 2^{nd} value of i we will go from

```
loopResults[2-1] <- randomVar[2] - randomVar[2-1]
```

to

```
loopResults[1000-1] <- randomVar[1000] - randomVar[1000-1]
```

We our now ready to iterate over all the numbers

```
for(i in 2:1000) {
  loopResults[i-1] <- randomVar[i] - randomVar[i-1]
```

```
}
```

This will give you some numbers as a result

```
head(loopResults)
```

```
## [1]  2.3116210  1.4773187 -0.4772367 -1.8036887 -2.1779413  3.6623927
```

Loops crop up all the time in R. However, some people frown upon them because of their inefficiency and clunkiness. There are sometimes other ways to do things. For example, earlier you we may have noticed we could have simply done:

```
# rep("All R and no play makes Jack a dull boy", 1000)
# randomVar[2:1000] - randomVar[1:999]
```

The `apply` family of functions can be helpful when trying to avoid loops. But generally if it works for you and doesn't take too long, then it's fine to use loops.

Task Seven

- Create an object called `orig_sample` that samples 1000 elements from a uniform distribution between 1 and 1000
- Create an empty vector called `res`
- Run a `for` loop that iterates 1000 times over `orig_sample`. For each iteration, sample 3 random elements from orig_sample, find the mean of this random sample and save to object `res`
- Compare the histograms of `res` and `orig_sample`

Setting working directory

When we read our own data in R it will be from a specific folder, and by default data will be saved to that folder. There are a number of ways to set the working directory. The most reproducible is using the command `setwd`. The command `getwd()` will display the current working directory

```
setwd("~/Documents/RCourse/RCourse_Data")
getwd()
```

```
## [1] "/Users/mp1728/Documents/RCourse/RCourse_Data"
```

Alternatively you can set it using interactive menus. On a Mac, the easiest way is to go to the R menu bar. Find Tools/change working directory or Misc/change working directory (depending on your version of R) and navigate to your folder. On a PC, drop down the File menu and navigate to the folder.

Reading data into R

Ok let's read in some data First we can set our working directory to our files. You can download the data here

Save this as 'rainfall.txt' in your working directory

```
setwd("~/Documents/RCourse/RCourse_Data")
```

'rainfall.txt' can be opened using the function `read.table`

```
rainfall <- read.table("~/Documents/RCourse/RCourse_Data/rainfall.txt")
head(rainfall)
```

```
##         V1       V2
## 1 northern southern
## 2    111.1     97.6
## 3     86.4     80.9
## 4    101.1     80.2
## 5    130.7      115
## 6     77.1       75
```

Success! But the first row looks a bit odd. In fact it looks more like column headers have been read in as data in the rows. We tell R that column names are in the file

```
rainfall <- read.table("~/Documents/RCourse/RCourse_Data/rainfall.txt", header=T)
head(rainfall)
```

```
##   northern southern
## 1    111.1     97.6
## 2     86.4     80.9
## 3    101.1     80.2
## 4    130.7    115.0
## 5     77.1     75.0
## 6     96.8    114.7
```

That looks better. By default R will assume there are no rownames or colnames so will give defaults (1,2,3...) for rownames and (V1, V2, V3....) for colnames. By specifying the `header` and `row.names` arguments we can overwrite these defaults

As well as `read.table` we can use `read.csv` to read in 'comma separated values' files from Excel. R can read in .xlsx documents from excel but it is more comfortable with .txt or .csv files We can read the file 'tannin' using read.csv and this gives colnames automatically

We can read the 'tannin' file using read.csv and this gives colnames automatically

```
tannins <- read.csv("https://puttickbiology.files.wordpress.com/2018/04/tannin.odt")
head(tannins)
```

```
##   growth tannin
## 1     12      0
## 2     10      1
## 3      8      2
## 4     11      3
## 5      6      4
## 6      7      5
```

Saving scripts

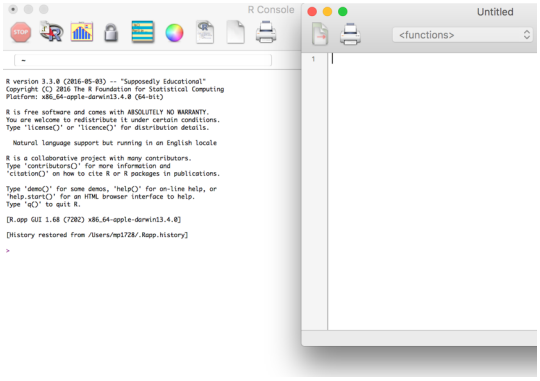
This can't be recommended enough

Thus far, we've been typing commands directly into the R console. However, this is not the most sensible plan. We want to make our R sessions reproducible and easy to save.

The best way to do this is to write scripts using the R Editor. This allows us to write commands into the R Editor, run them, and save them as '.R' files. If we need to repeat the process, we can open the file and repeat.

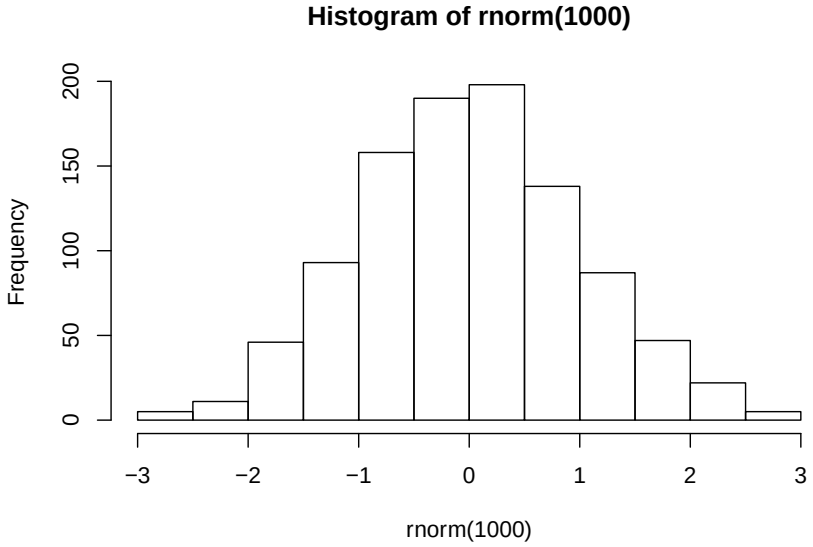
We can open an Editor by clicking File > New or by typing Cmd + N on Mac(Ctrl + N on Windows).

Hopefully we will see something like this



We can now type commands into this window. We can plot a histogram of a normal distribution

```
hist(rnorm(1000))
```



With the mouse, we can select the we have typed this and press Cmd + Enter on a Mac or Ctrl + R on Windows or Ctrl + Enter on Linux. Voilà! A plot! This makes any R code we write easy to save (File > Save)

The R Library and installing packages

We have seen the power of R so far by just using the functions built in base. However, we will probably want to use R for more specialised purposes. This is where we make use of the R library which contains a huge and growing number of packages. Knowing which library we want will probably come from prior knowledge or an internet search.

"As far back as I can remember, I always wanted to be a phylogeneticist. To me, being a phylogeneticist was better than being President of the United States."

This being true, I will start with the APE library which is the foundation of phylogenetics in R (it is not a necessity, but most R packages have ridiculous names: APE stands for "Analysis of Phylogenetics and Evolution with R").

So let's install APE onto our machine. There are a number of ways to do this. My preferred method is to simply type

```
install.packages("ape", repos="https://www.stats.bris.ac.uk/R/")
```

```
## Installing package into '/Users/mp1728/Library/R/3.4/library'
## (as 'lib' is unspecified)
```

```
##
## The downloaded binary packages are in
##  /var/folders/lb/grtql71j7d1d8wyln9l7x8prddz3_y/T//Rtmp2JMUGx/downloaded_packages
```

Once we install a package we don't need to install it again.

Now the package is downloaded we need to load it into our R session

```
library(ape)
```

Note – we will have to do this in every session of R so it is good practice to write this at the top of any script. Ok – let's checked that's worked. We can get a description of APE

```
?ape
```

And a list of all the functions in the library

```
library(help = ape)
```

And the example datasets

```
data(package = "ape")
```

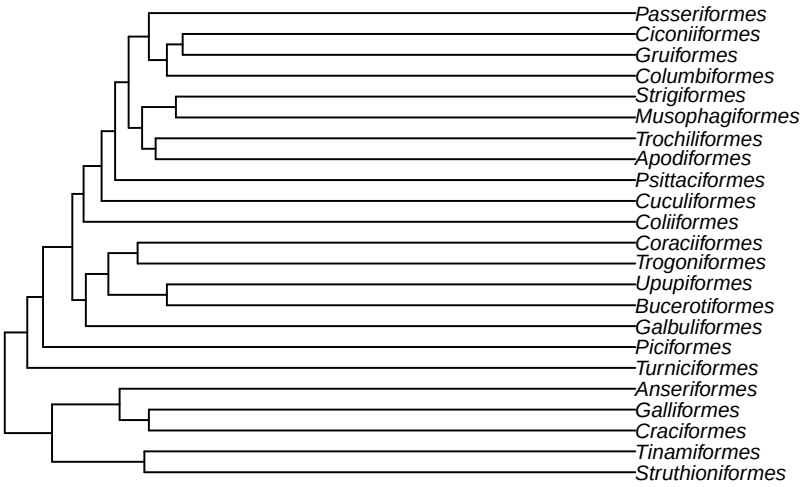
So I have already looked through the functions and have decided to plot a phylogeny of birds

So I will call up the data

```
data(bird.orders)
```

We can now use the function in APE called plot.phylo to plot the phylogeny of bird.orders

```
plot.phylo(bird.orders, cex=0.7)
```



Great. We have installed, loaded, and used our library Sporadically, authors will update their R packages. We can check for these updates and, erm, update them using `update.packages()`

Writing R functions

Writing customised functions in R is a powerful tool. We can tell R what we do to do that is specific to our data. This can be used to speed-up or automate code, or to perform brand new calculations in our research.

Earlier we took the `my_data` vector and calculated the mean and median

```
my_data <- c(15,3,6,2,8,7,9,3,5,7,1,3,6,90,2)
mean(my_data)
```

```
## [1] 11.13333
```

```
median(my_data)
```

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

But what if we would like to know the mode? Ah, it appears R has no in-built functions for calculating the mode. So what can we do? We can build our own function!

We know the mode is the 'most common' number occurring in our data. So we can utilise table to tabulate each number and how often it occurs in our data

```
table(my_data)
```

```
## my_data
##  1  2  3  5  6  7  8  9 15 90
##  1  2  3  1  2  2  1  1  1  1
```

```
my_data
```

```
##  [1] 15  3  6  2  8  7  9  3  5  7  1  3  6 90  2
```

So this returns a data object with names of the numbers that occur in our vector and below each the number of times they occur. We could just read the answer from this, but what if we have a massive dataset or need to automate the process? Let's continue building our function.

Which of our numbers occurs the most often? We can take the occurrences for the second row and find the number that occurs most often using which and max. First we will create the object occurrences and take this object to find the number that occurs the most often

```
occurrences <- table(my_data)
mostCommon <- which(occurrences == max(occurrences))
mostCommon
```

```
## 3
## 3
```

Here we have the number of occurrences (the bottom 3) and the actual value in our original dataset (the name 3 above). By coincidence both numbers are the same in this dataset. So we want to take the names of the object `mostCommon`

```
as.numeric(names(mostCommon))
```

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

There is our answer – it's 3! Let's bring everything together

```
occurrences <- table(my_data)
mostCommon <- which(occurrences == max(occurrences))
mode_output <- names(mostCommon)
as.numeric(mode_output)
```

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

We have built the expression algortithm for our function. Now we can put it into an R function called `calculateMode`. We need to employ the R function function to build this function. It has the following syntax

```
functionName <- function(input, arguments) { expression }
```

This means we create an object of class `function`. In this function we set the structure for the input and arguments that it needs in the `function(input, arguments)`. Then the actual calculations are placed in the `{ expression }`

```
calculateMode <- function(input, arguments) { expression
}
```

What shall our input be? Well, it'll be a vector like `my_data` but we won't call it `my_data` as that will be confusing. We can just go for a generic name like `input_vector`. Although we may need them in the future, our current function doesn't need any tweaking arguments so we can leave them out. So we now have:

```
calculateMode <- function(input_vector) { expression }
```

This means the expression part of the function will process an object called `input_vector`. So we need to slightly update our code from above. We need to replace `my_data` with `input_vector`

```
calculateMode <- function(input_vector) {
  occurrences <- table(input_vector)
  mostCommon <- which(occurrences == max(occurrences))
  mode_output <- names(mostCommon)
  return(as.numeric(mode_output))
}
```

Notice we have added `return` to the function. This means while all the other calculations will remain hidden in the background, this will printed to screen and can be saved as an object at the functions completion. So we can now run our function in the R console. I will select it all and press cmd + Enter (or ctrl + R).

Nothing has happened. Nothing is good, which in this case, means everything is good. By which I mean, running the function successfully will print no output or errors, and that is exactly what we want it to do.

We can now test run our function by using `my_data` as input

```
calculateMode(my_data)
```

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

Success! We have completed our first R function.

Task Eight

- Write a function to calculate the mean. Test it on my_data and compare it to the result from the R function `mean`
- BONUS TASK. Write a function to calculate the median. Again test it on `my_data` and compare it to the result from the R function `median`