

Introduction to R

(with Tidyverse)

Version 2019-08

# Licence

This manual is © 2019, Simon Andrews.

This manual is distributed under the creative commons Attribution-Non-Commercial-Share Alike 2.0 licence. This means that you are free:

* to copy, distribute, display, and perform the work
* to make derivative works

Under the following conditions:

* Attribution. You must give the original author credit.
* Non-Commercial. You may not use this work for commercial purposes.
* Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under a licence identical to this one.

Please note that:

* For any reuse or distribution, you must make clear to others the licence terms of this work.
* Any of these conditions can be waived if you get permission from the copyright holder.
* Nothing in this license impairs or restricts the author's moral rights.

Full details of this licence can be found at

<http://creativecommons.org/licenses/by-nc-sa/2.0/uk/legalcode>

TABLE OF CONTENTS

[Getting Started with R 5](#_Toc16845155)

[What is R? 5](#_Toc16845156)

[Good things about R 5](#_Toc16845157)

[Bad things about R 5](#_Toc16845158)

[Installing R and RStudio 6](#_Toc16845159)

[Getting help with R 6](#_Toc16845160)

[Getting familiar with the R console 7](#_Toc16845161)

[Basic R Operations 9](#_Toc16845162)

[Storing and Retrieving Data in Variables 9](#_Toc16845163)

[Naming data structures 11](#_Toc16845164)

[Functions 11](#_Toc16845165)

[Storing Multiple Values in Vectors 13](#_Toc16845166)

[Data types in vectors 14](#_Toc16845167)

[Functions for making vectors 14](#_Toc16845168)

[Accessing Vector Subsets 15](#_Toc16845169)

[Vectorised Operations 16](#_Toc16845170)

[Beyond vectors – Lists, Data Frames and Tibbles 18](#_Toc16845171)

[Lists 18](#_Toc16845172)

[Data frames 20](#_Toc16845173)

[Tibbles 21](#_Toc16845174)

[Getting and Using the Tidyverse Packages 22](#_Toc16845175)

[Installing Tidyverse (or any other CRAN package) 22](#_Toc16845176)

[Using tidyverse in your R script 22](#_Toc16845177)

[Reading and Writing data from files 24](#_Toc16845178)

[Getting and setting the working directory 24](#_Toc16845179)

[Reading data from text files 24](#_Toc16845180)

[Writing data 26](#_Toc16845181)

[‘Tidy’ Data Format 26](#_Toc16845182)

[Wide Format 26](#_Toc16845183)

[Long Format 27](#_Toc16845184)

[Filtering and Subsetting your Tibbles 28](#_Toc16845185)

[Extracting Data Using Core R 28](#_Toc16845186)

[Fetching a single column using $ 28](#_Toc16845187)

[Fetching column and row positions using [ ] 28](#_Toc16845188)

[Manipulating Data using dplyr in tidyverse 29](#_Toc16845189)

[Selecting columns using select 29](#_Toc16845190)

[Functional selections using filter 30](#_Toc16845191)

[Combining Multiple Operations 31](#_Toc16845192)

[Drawing Graphs with ggplot 33](#_Toc16845193)

[Defining your data 33](#_Toc16845194)

[Geometries and Aesthetics 34](#_Toc16845195)

[How to specify aesthetics 35](#_Toc16845196)

[Putting it all together 35](#_Toc16845197)

[Other plot types 36](#_Toc16845198)

[Multiple Geometries 40](#_Toc16845199)

[Further information 41](#_Toc16845200)

[Most importantly..... 41](#_Toc16845201)

# Introduction

R is a popular language and environment that allows powerful and fast manipulation of data, offering many statistical and graphical options. One of the most attractive aspects of the language is the wide variety of additional packages which can provide extended functionality for the core language. One of the most popular of these are the ‘tidyverse’ packages which are a coordinated set of packages which supplement the core language’s functions for data manipulation and plotting.

In this course we are going to cover the basics of using R for data manipulation, analysis and plotting. We will start from the core language but will also extend into using the tidyverse to supplement this.

The aim of this course is to get you familiar enough with the R/Tidyverse environment to be able to start to use this for exploring your own data.

# Getting Started with R

## What is R?

R is a programming language – however it’s a slightly unusual language in that it has a specifically defined purpose, which is to facilitate the manipulation, analysis and plotting of data. As such it comes with a large amount of functionality relating to data manipulation, statistical analysis and graphics. This has made it a useful tool for many scientific disciplines.

Rs usefulness for science has also meant that it has been adopted as one of the most popular tools in bioinformatics. This in turn has meant that many groups who are developing new tools or methods often use R as the environment for these, so that if you want to access many of the latest developments you too will need to use R to do this.

### Good things about R

* It's free­
* It works on all platforms
* It can deal with much larger datasets than Excel for example
* Graphs can be produced to your own specification
* It can be used to perform powerful statistical analysis
* It is well supported and documented

### Bad things about R

* It can struggle to cope with extremely large datasets
* The environment can be daunting if you don't have any programming experience
* It has a rather unhelpful name when it comes to googling problems (though you can use [http://www.rseek.org/](http://www.rseek.org/%20) or google ‘R help forum’ and try that instead)

## Installing R and RStudio

Instructions for downloading and installing R can be found on the R project website <http://www.r-project.org/>. Versions are available for Windows, Linux and Mac.

RStudio is an integrated development environment for R, available for Windows, Linux and Mac OS and like R, is free software. It offers a neat and tidy environment to work in and also provides some help with importing datasets and installing packages etc. You must have R installed in order to run RStudio. More information and instructions for download can be found at <http://www.rstudio.org/>.

## Getting help with R

R has comprehensive help pages that are very useful once you have familiarised yourself with the layout. Information about a function (for example read.table) can be accessed by typing the following into the console:

help(read.table)

or

?read.table

This should include information about parameters that can be passed to the function, and at the bottom of the page should be examples that you can run which can be very useful.

If you don't know the function name that you're after, eg. for finding out the standard deviation, try

help.search("deviation")

or

??deviation

And you can always try searching the internet but remember that 'R' in a general search isn't always very good at returning relevant information so try and include as much information as possible, or go to [http://www.rseek.org/](http://www.rseek.org/%20) which will return more R specific information.

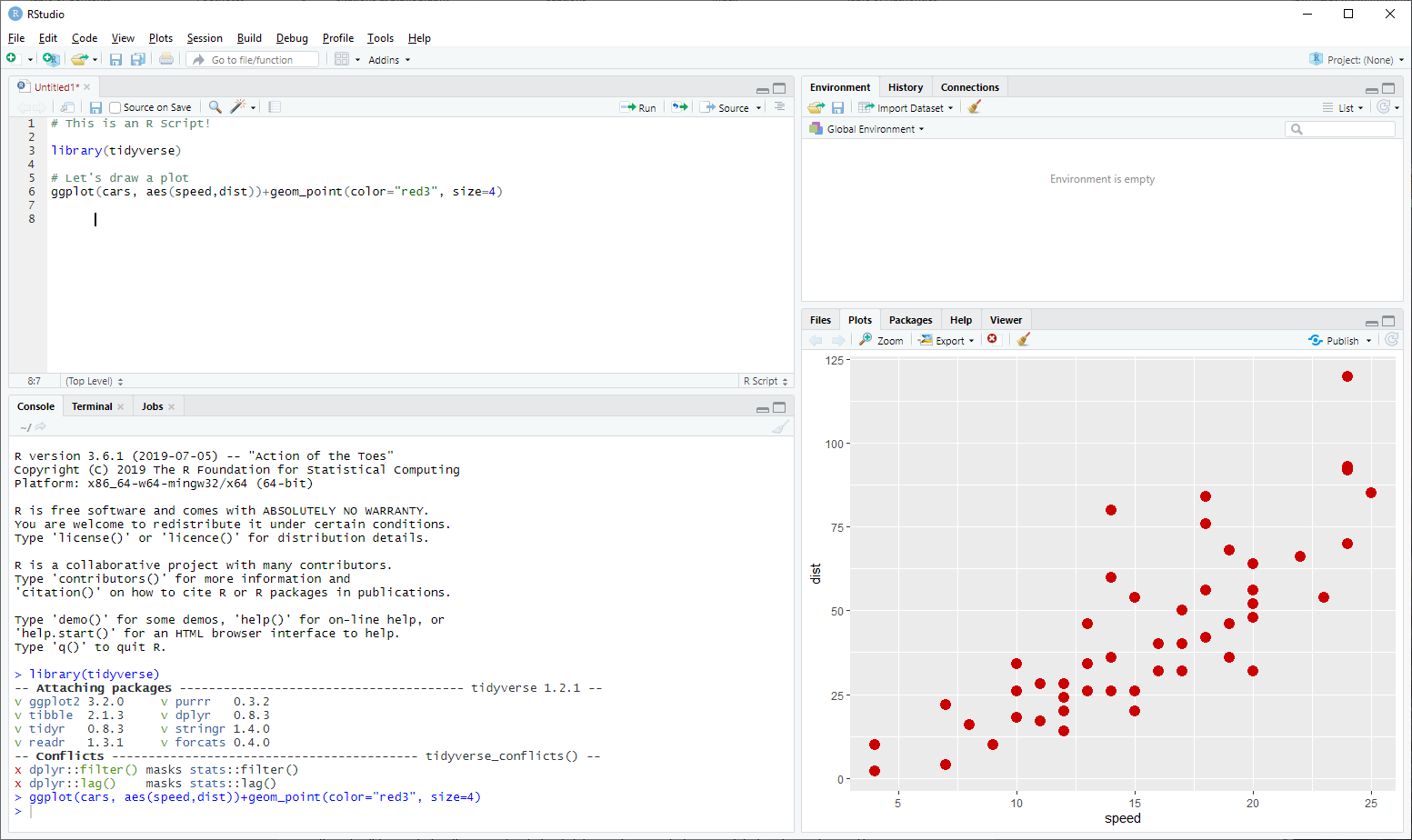
# Getting familiar with RStudio

R is a command line environment, this means that you type in an instruction and R interprets this and either stores the result or writes it back on the screen for you. You can run R in a very simple command shell window, but for all practical purposes it’s much easier to use a dedicated piece of software which makes it easy to work within the R environment.

There are several different R IDEs (integrated development environments) around, but by far the most common one is R studio and this is what we’re going to be using for this course.

Open RStudio. The default layout is shown below,

* Top left panel is the text editor. Commands can be sent into the console from here.
* Bottom left is the R console. You can type R directly into here.
* Top right is the workspace and history. History keeps a record of the last commands entered, this is searchable. The workspace tab shows all the R objects (data structures).
* Bottom right is where graphs are plotted and help topics are shown.



The actual R session is the console window at the bottom left of the R-studio window. This is where the computation is ultimately done in R. All of the other windows are either keeping records of what you’ve done or showing you the result of a previous command.

When you type a command into the console it is evaluated by the R interpreter. If the result of this is a value then it will be printed in the console. Lots of data can be imported, manipulated and saved in an R session and though it won't always be visible on the screen, there are various ways of viewing and manipulating it.

If you create graphs these will open in a new window within the IDE as shown in the following screenshot. This also shows a text editor in which you can write anything you like including notes, though it would generally be used to create a script (i.e. lines of R commands). You can start a new text editor by selecting File > New > RScript from the menu bar.

As well as issuing commands by typing directly into the console, you can also send commands to the R Console from the R Editor by selecting a command or a line of text and selecting Ctrl + Enter, or by copying and pasting.

In the console you can scroll through previous command that have been entered by using the up arrow ↑ on the keyboard.

The > symbol shows that R is ready for something to be entered. The console can work just like a calculator. Type 8+3 and press return. It doesn't matter whether there are spaces between the values or not.

> 8 + 3

[1] 11

The answer is printed in the console as above. We'll come on to what the [1] means at the end of this section.

> 27 / 5

[1] 5.4

These calculations have just produced output in the console - no values have been saved.

# Basic R Operations

## Storing and Retrieving Data in Variables

One of the most basic things you need to be able to do in any programming language is to store and retrieve data. To do this in R you create a ‘variable’ which is simply a piece of data which you have given a name to. Once you’ve done this then you can access the data elsewhere in your script by using the name instead of having to re-enter the data each time.

Creating a named variable is done by drawing an arrow, like <- (a less than sign followed by a minus sign). The arrow points from the data you want to store towards the name you want to store it under. For now we'll use x, y and z as names of data structures, though more informative names should really be used as we’ll discuss later in this section.

> x <- 8 + 3

If R has performed the command successfully you will not see any output, as the value of 8 + 3 has been saved to the data structure called x. You should immediately see x appear in your workspace tab within Rstudio. You can access and use this data structure at any time and can print the value of x into the console.

> x

[1] 11

Create another data structure called y. In this case we’ll draw the arrow the other way around. Functionally this is the same as the first example and it’s up to you whether you do data -> name or name <- data.

> 3 -> y

Now that values have been assigned to x and y they can be used in calculations.

> x + y

[1] 14

> x \* y

[1] 33

> x \* y -> z

> z

[1] 33

R is case sensitive so x and X are not the same. If you try to print the value of X out into the console an error will be returned as X has not been used so far in this session.

> X

Error: object 'X' not found

To check what variables you have created look at the ‘workspace’ tab in RStudio or enter ls() or objects() into the console.

If you use the same variable name as one that you have previously used then R will overwrite the previously stored data with the new value.

> y <- 3

> y

[1] 3

> y <- 12

> y

[1] 12

Using an equals sign also works in most situations, eg x = 8+3 but <- is generally preferred. You can also change the direction of the arrow if you want to calculate something and then send the result into a variable. The statements below are all functionally identical.

x<-3 x <-3 x<- 3 x <- 3

3->x 3 ->x 3-> x 3 -> x

However, if you enter a space between the less than and minus characters that make up the assignment operator then you would be asking R a question that has a logical answer i.e. is x less than -5.

> x < - 5

[1] FALSE

## Naming data structures

Data structures can be named anything you like (within reason), though they have to start with a letter. Having informative, descriptive names is useful and this often involves using more than one word. Providing there are no spaces between your words you can join them in various ways, using dots, underscores and capital letters though the Google R style guide recommends that names be joined with a full stop.

> mouse.age <- 112

> mouse.weight <- 25

> tail.length <- 54

These are all completely separate data structures that are not linked in any way.

Joining names by capitalising words is generally used for function names - don't worry about what these are for now - suffice to say it is not recommended to create names in the format mouseAge; use mouse\_age or preferably mouse.age.

These names can be as long as you like, it just becomes more of a chore to type them the longer they get. RStudio helps with this in that you can press the tab key to try to complete any variable name you’ve started typing and it will complete it for you. Numbers can be incorporated into the name as long as the name does not begin with a number. Although you may not wish to use such convoluted names, the following are perfectly valid:

> tail.length.mouse.1 <- 41

> tail.length.mouse2.condition4.KO <- 35

## Functions

Most of the work that you do in R will involve the use of functions. A function is simply a named set of code to allow you to manipulate your data in some way. There are many built in functions in R and you can write your own if there isn’t one which does exactly what you need.

Functions in R take the format *function.name(parameter 1, parameter 2 ... parameter n).* The brackets are always needed. Some functions are very simple and the only parameter you need to pass to the function is the data that you want the function to use.

For example, the function dim(*data.structure*) returns the dimensions (i.e. the numbers of rows and columns) of the data structure that you insert into the brackets, and it will not accept any additional arguments/parameters.

The square root and log2 functions can accept just one parameter as input.

> sqrt(245)

[1] 15.65248

> log2(15.65248)

[1] 3.968319

To keep it tidier the result of the square root function can be assigned to a named variable.

> x <- sqrt(245)

> log2(x)

[1] 3.968319

Alternatively, the two calculations can be performed in one expression.

> log2(sqrt(245))

[1] 3.968319

Most other functions will accept multiple parameters, such as read.table() for importing data.

If you’re not sure of the parameters to pass to a function, or what additional parameters may be valid you can use the built in help to see this. For example to access the help for the read.table() function you could do:

?read.table

...and you would be taken to the help page for the function.

## Storing Multiple Values in Vectors

One thing which distinguishes R from other languages is that its most basic data structure is actually not a single value, but is an ordered set of values, called a vector. So, when you do:

> x <- 3

…you’re actually creating a vector with a length of 1. Vectors can hold many different types of data (text, numbers, true/false values etc), but all of the values held in an individual vector must be of the same type, so it can be all numbers, or all text but not a mix of the two.

To manually create a vector you can use the c() (short for combine) function which can take an arbitrary number of arguments and will combine them into a single vector. The only parameters that need to be passed to c() here are the data values that we want to combine.

> mouse.weights <- c(19, 22, 24, 18)

If words are used as data values they must be surrounded by quotes (either single or double - there's no difference in function, though pairs of quotes must be of the same type).

> mouse.strains <- c('castaneus', "black6", 'molossinus', '129sv')

If quotes are not used, R will try and find the data structure that you have referred to.

> mouse.strains <- c(castaneus, 'black6', 'molossinus', '129sv')

Error: object 'castaneus' not found

To access the whole data structure just type the name of it as before:

> mouse.weights

[1] 19 22 24 18

> mouse.strains

[1] "castaneus" "black6" "molossinus" "129sv"

Or use:

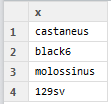
head(mouse.weights)

To see just the first few lines of the vector.

For a more graphical view you can also use the View function. In RStudio, the data is displayed in the text editor window. Both of these datasets are 4 element vectors.

> View(mouse.weights)

> View(mouse.strains)



### Data types in vectors

Within a vector all of the values must be of the same ‘type’. There are four basic data types in R:

* Numeric - An integer or floating point number
* Character - Any amount of text from single letter to whole essay
* Logical - TRUE or FALSE values
* Factor - A categorised set of character values

Internally R distinguishes between integers (whole numbers) and floating point numbers (fractional numbers) but they are stored the same way.

Factors are the default way which R stores many pieces of text. They are used when grouping data for statistical or plotting operations and in many cases are interchangeable with characters, but there are differences, especially when merging or sorting data which can cause problems if you use the wrong type for this kind of data.

To see what type of data you’re storing in a vector you can either look in the workspace tab of RStudio or you can use the class function.

> class(mixed.frame$numbers)

[1] "character"

> class(c(1,2,3))

[1] "numeric"

> class(c("a","b","c"))

[1] "character"

> class(c(TRUE,TRUE,FALSE))

[1] "logical"

### Functions for making vectors

Although you can make vectors manually using the c() function, there are also some specialised functions for making vectors. These provide a quick and easy way to make up commonly used series of values.

The seq() function can be used to make up arithmetic series of values. You can specify either a start, end and increment (by) value, or a start, increment (by) and length (length.out) and the function will make up an appropriate vector for you.

> seq(from=5,to=10,by=0.5)

[1] 5.0 5.5 6.0 6.5 7.0 7.5 8.0 8.5 9.0 9.5 10.0

> seq(from=1,by=2,length.out=10)

[1] 1 3 5 7 9 11 13 15 17 19

The rep() function simply repeats a value a specified number of times.

> rep("hello",5)

[1] "hello" "hello" "hello" "hello" "hello"

Finally, there is a special operator for creating vectors of sequential integers. You simply separate the lower and higher values by a colon to generate a vector of the intervening values.

> 10:20

[1] 10 11 12 13 14 15 16 17 18 19 20

You can also combine these functions with c() to make up more complicated vectors

c(rep(1,3),rep(2,3),rep(3,3))

[1] 1 1 1 2 2 2 3 3 3

### Accessing Vector Subsets

To access specific positions in a vector you can put square brackets after it, and then use a vector of the index positions you want to retrieve. Note that unlike most other programming languages index counts in R start at 1 and not 0. To view the 2nd value in the data structure:

> mouse.strains[2]

[1] "black6"

Note, that in this instance the number 2 in the above expression is actually just a shortcut for c(2), so what we’re pulling out are a vector of index positions. We can therefore also use the automated ways to make vectors of integers to easily pull out larger subsets. To view a range of values we could use the lower:higher notation we saw above:

> mouse.strains[2:4]

[1] "black6" "molossinus" "129sv"

We should think of the statement above as two separate operations. We use 2:4 to make a vector with 2,3,4 in it, and then we put that into square brackets to select the corresponding values from mouse.strains. To view or select non-adjacent values the c() function can be used again. To view the 2nd and the 4th values:

> mouse.strains[c(2,4)]

[1] "black6" "129sv"

Remember that to create a vector you need to use the c function. If you leave this out (which is really easy to do) you’ll get an error:

> mouse.strains[2,4]

Error in mouse.strains[2, 4] : incorrect number of dimensions

## Vectorised Operations

The other big difference between R and other programming languages is that normal operations are designed to be applied to whole vectors rather than individual values. This means that you can very quickly and easily apply changes to whole sets of data without having to write complex code to loop through individual values.

If we wanted to log transform a whole dataset for example then we can do this using a single operation.

> data.to.log <- c(1,10,100,1000,10000,100000)

> log10(data.to.log)

[1] 0 1 2 3 4 5

Here we passed a vector to the log10 function and we received back a vector of the log10 transformed versions of all of the elements in that vector without having to do anything else. We can use this for other operations too:

> data.to.log + 1

[1] 2 11 101 1001 10001 100001

> (data.to.log +1) \* 2

[1] 4 22 202 2002 20002 200002

You can also use two vectors in any mathematical operation and the calculation will be performed on the equivalent positions between the two vectors. If one vector is shorter than the other then the calculation will ‘wrap round’ and start again at the beginning (but you will get a warning if the longer vector’s length isn’t a multiple of the shorter vector’s length).

> x <- 1:10

> y <- 21:30

> x+y

[1] 22 24 26 28 30 32 34 36 38 40

> j <- c(1,2)

> k <- 1:20

> k\*j

[1] 1 4 3 8 5 12 7 16 9 20 11 24 13 28 15 32 17 36 19 40

It is important to understand how operations involving two vectors work since this ends up being a critical aspect of many parts of R. The basic rules are that if two vectors are the same length, then equivalent indices are paired together.



If they are not the same length then the shorter vector is recycled, but it is expected that the length of the longer vector will be a multiple of the length of the shorter vector (you’ll get a warning if it isn’t).



## Beyond vectors – Lists, Data Frames and Tibbles

Vectors are a core part of R and are extremely useful on their own, but they are limited by the fact that they can only hold a single set of values and that all values must be of the same type. To build more complex data structures we need to look at some of the other data structures R provides.

We are going to briefly look at three different data structures in R, but ultimately we are going to mostly focus on the ‘tibble’ which will be the data structure we will use for the rest of the course.

### Lists

The simplest level of organisation you can have in R above the vector is a list. A list is simply a collection of vectors - sort of like a vector of vectors but with some extra features. A list is created from a set of vectors, and whilst each vector is a fixed type, different vectors within the same list can hold different types of data.

Once a list is created the vectors in it can be accessed using the position at which they were added. You can also assign a name to each position in the list and then access the vector using that name rather than its position, which can make your code easier to read and more robust if you choose to add more vectors to the list at a later date.

As an example, the two data structures mouse.strains and mouse.weights can be combined into a list using the list() function.

> mouse.data <- list(weight=mouse.weights,strain=mouse.strains)

If you now view this list you will see that the two sets of data are now stored together and have the names weight and strain associated with them.

> mouse.data

$weight

[1] 19 22 24 18

$strain

[1] "castaneus" "black6" "molossinus" "129sv"

To access the data stored in a list you can use either the index position of each stored vector or its name. You can use the name either by putting it into square brackets, as you would with an index, or by using the special $ notation after the name of the list (which is generally the nicer way to do this).

> mouse.data[[1]]

[1] 19 22 24 18

> mouse.data[["weight"]]

[1] 19 22 24 18

> mouse.data$weight

[1] 19 22 24 18

In this case, because we’re accessing a position in a list rather than a vector we use double square brackets.

Once you’re retrieved the vector from the list you can use conventional square bracket notation to pull out individual values.

> mouse.data$weight[2:3]

[1] 22 24

Lists can be useful but they are really just a collection of vectors, and there is no linkage between positions in the different vectors stored within a list (mouse.data$weight[1] has no connection to mouse.data$strain[1]), in fact the vectors stored in a list don’t even have to be the same length. This is perfectly valid.

> names <- list(first=c("Bob","Dave"),last=c("Smith","Jones","Baker"))

> names

$first

[1] "Bob" "Dave"

$last

[1] "Smith" "Jones" "Baker"

Since most real data occurs in a more regular table structure we need a different data structure to represent this.

### Data frames

The most useful data structure in the core R language is the data frame. In essence, a data frame is just a list, but with one important difference which is that all of the vectors stored in a data frame are required to have the same length. By adding this constraint you can add the concept of a ‘row’, which is simply the set of equivalent positions in all of the different vectors in the data frame. This means you can create a 2D table type structure, where all of the columns store the same type of data.

Because of its 2D structure a data frame can then add some functionality to access data from both rows and columns in a convenient manner and makes it easy to perform operations which use the whole of the data structure.

Data frames are created in the same way as lists, with columns being either named or simply accessed by position.

> mouse.dataframe <- data.frame(weight=mouse.weights,strain=mouse.strains)

> mouse.dataframe

weight strain

1 19 castaneus

2 22 black6

3 24 molossinus

4 18 129sv

> mouse.dataframe[[1]]

[1] 19 22 24 18

> mouse.dataframe$weight

[1] 19 22 24 18

However, data frames have some extra functionality which allows you to extract subsets of data directly from within the 2D structure. Specifically the normal bracket notation for extracting data from a vector has been extended so that you can supply two values to specify which rows and columns you want to extract from a data frame. You can do something simple such as:

> mouse.dataframe[2,1]

[1] 22

You can choose to select all values from one dimension by leaving out the index value:

> mouse.dataframe[,1]

[1] 19 22 24 18

You can also use the same range selectors we saw before but in two dimensions:

> mouse.dataframe[2:4,1]

[1] 22 24 18

There are also some extra functions you can use with data frames. You can use the dim() function to find out how many rows and columns your data frame has, and you can use nrow() and ncol() to get these values individually.

> dim(mouse.dataframe)

[1] 4 2

> nrow(mouse.dataframe)

[1] 4

> ncol(mouse.dataframe)

[1] 2

### Tibbles

The data structure we’re actually going to use for most of our data is a tibble. A tibble is just the tidyverse version of a data frame. Everything we’ve shown you about how you create and use a data frame is exactly the same for a tibble.

Differences between data frames and tibbles are mostly behind the scenes (where you don’t need to know or care about them), or they are cosmetic – tibbles look a lot nicer when you print them out for example.

> head(as\_tibble(data))

# A tibble: 6 x 12

Probe Chromosome Start End `Probe Strand` Feature ID Description

*<chr>* *<dbl>* *<dbl>* *<dbl>* *<chr>* *<chr>* *<chr>* *<chr>*

1 AL64~ 1 9.11e5 9.15e5 + AL6456~ ENSG~ novel tran~

2 LINC~ 1 9.17e5 9.21e5 - LINC02~ ENSG~ long inter~

3 SAMD~ 1 9.24e5 9.45e5 + SAMD11 ENSG~ sterile al~

4 TMEM~ 1 1.51e7 1.52e7 - TMEM51~ ENSG~ TMEM51 ant~

5 TMEM~ 1 1.52e7 1.52e7 + TMEM51 ENSG~ transmembr~

6 FHAD1 1 1.52e7 1.54e7 + FHAD1 ENSG~ forkhead a~

# ... with 4 more variables: `Feature Strand` *<chr>*, Type *<chr>*, `Feature

# Orientation` *<chr>*, Distance *<dbl>*

> head(as.data.frame(data))

Probe Chromosome Start End Probe Strand Feature

1 AL645608.2 1 911435 914948 + AL645608.2

2 LINC02593 1 916865 921016 - LINC02593

3 SAMD11 1 923928 944581 + SAMD11

4 TMEM51-AS1 1 15111815 15153618 - TMEM51-AS1

5 TMEM51 1 15152532 15220478 + TMEM51

6 FHAD1 1 15247272 15400283 + FHAD1

Description

1 novel transcript

2 long intergenic non-protein coding RNA 2593 [Source:HGNC Symbol;Acc:HGNC:53933]

3 sterile alpha motif domain containing 11 [Source:HGNC Symbol;Acc:HGNC:28706]

4 TMEM51 antisense RNA 1 [Source:HGNC Symbol;Acc:HGNC:26301]

5 transmembrane protein 51 [Source:HGNC Symbol;Acc:HGNC:25488]

6 forkhead associated phosphopeptide binding domain 1 [Source:HGNC Symbol;Acc:HGN

The one major difference between data frames and tibbles which might affect you is that data frames have the concept of row names as well as column names, whereas tibbles do not support row names, so if you have a data frame with row names then these will need to be converted into a conventional column if it is transformed into a tibble.

Tibbles are part of the tidyverse package set, so let’s look at what that is and how to start using it.

# Getting and Using the Tidyverse Packages

Whilst there is a lot of functionality contained in the core R language, not everything you will ever need to do will be in there. R therefore supports an extension mechanism called ‘packages’ which are an easy way to add functionality to the language.

For convenience, packages are often stored and downloaded from package repositories. The largest one of these is CRAN, which is the package repository maintained by the people who write R. You may also hear of others, such as BioConductor (which is a bioinformatics focussed repository).

The ‘Tidyverse’ is a collective name for a bunch of R packages in CRAN. They are connected by a set of common design principles for how they work and the way in which data should be structured. Between them they supplement many of the core functions in R relating to data manipulation and plotting. The use of these packages is now very common so it’s a good idea to start out learning how they work.

## Installing Tidyverse (or any other CRAN package)

In order to use a package in your R code there are two steps you need to go through.

1. You need to download the package from the internet and copy it to the machine you’re working on. You only need to do this once on every machine you use.
2. You need to load the package into the R environment you’re working in. You need to do this in every script in which you want to use the package.

Because the tidyverse packages are in CRAN then there is a way built into the language to install them. This is through the use of the install.packages function.

install.packages("tidyverse")

Tidyverse isn’t actually a single package, but a whole collection of them so you will see loads of packages being download and installed. Remember though you only need to do this once per machine.

## Using tidyverse in your R script

Once you have the packages installed then you can bring them into your R environment by running:

library(tidyverse)

Running this will load in all of the core tidyverse packages and make the functions they provide available within your script. When they load you will see a brief summary of the packages which were loaded and the version numbers you have.

> library(tidyverse)

-- **Attaching packages** --------------------------------------- tidyverse 1.2.1 --

v ggplot2 3.2.0 v purrr 0.3.2

v tibble 2.1.3 v dplyr 0.8.1

v tidyr 0.8.3 v stringr 1.4.0

v readr 1.3.1 v forcats 0.4.0

-- **Conflicts** ------------------------------------------ tidyverse\_conflicts() --

x dplyr::filter() masks stats::filter()

x dplyr::lag() masks stats::lag()

Once you’ve done that then you can freely use functions coming from within tidyverse the same way as core R functions.

# Reading and Writing data from files

Most of the time you are not going to manually enter data into your script but will load it from files. We will be using some tidyverse functions to read from text files directly into tibbles. When we do this we will obviously need to specify the location of the file. We could do this by specifying the full location of the file to load, for example something like;

O:/Training/R/Introduction/CouseData/myfile.txt

..but typing out the whole of that location is a pain and we’re likely to make a mistake. To make life easier what we can do is to set a location in our filesystem, called the “working directory” which is where R will look for files by default. That means that once this is set then we can load files by just using their name.

myfile.txt

## Getting and setting the working directory

To check which directory you have set as your working directory at the moment you can use:

> getwd()

[1] "m:/"

To set your working directory:

setwd*(“the/place/I/want/to/read/and/write/files”)*

Once you have started to type the path you can again use the tab key to get RStudio to show you possible completions so you don’t have to type the whole thing.

Alternatively, RStudio allows you to set the working directory through the menu options.

Session > Set Working Directory > Choose Directory

## Reading data from text files

Most of the time, the format you want to use for storing data you’re going to read into R is a ‘delimited text file’. This is a plain text file where each row of the file is one row of your data, and the colums within a row are indicated by a special character, called a delimiter.

The two most common delimiters you will see are:

* The tab character, which means the file is a ‘tab delimited file’ or a ‘tab separated values file’ – a TSV file.
* A comma, which means the file is a ‘comma separated values file’ – a CSV file.

Tidyverse defines a few very similar functions for reading in data. The function you use will depend on the format of your file.

* Tab delimited data uses read\_tsv
* Comma separated data uses read\_csv

By default each of these functions will expect that the first line of your data contains the names of the columns and that every other line is data.

They will return you a tibble containing your data, but they won’t save it for you. It is up to you to save the data into a variable in the same way as you did before, using an arrow (->)

For example – we provided you with a data file called “trumpton.txt” this is a tab delimited file so we can read it with read\_tsv.

> read\_tsv("trumpton.txt")

Parsed with column specification:

cols(

LastName = col\_character(),

FirstName = col\_character(),

Age = col\_double(),

Weight = col\_double(),

Height = col\_double()

)

# A tibble: 7 x 5

LastName FirstName Age Weight Height

*<chr>* *<chr>* *<dbl>* *<dbl>* *<dbl>*

1 Hugh Chris 26 90 175

2 Pew Adam 32 102 183

3 Barney Daniel 18 88 168

4 McGrew Chris 48 97 155

5 Cuthbert Carl 28 91 188

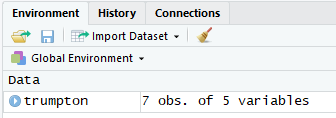
6 Dibble Liam 35 94 145

7 Grub Doug 31 89 164

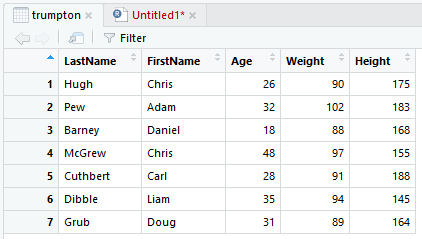
When we load the file we will get a bit of information telling us how read\_tsv interpreted the data in the file. It will list the columns it found and will tell us what type of data it thought they contained. It will then return the tibble containing the data. In this example we didn’t save the data anywhere so it just gets printed to the console. This can be a good sanity check, but finally we will want to save the data to an R variable so we can use it.

read\_tsv("trumpton.txt") -> trumpton

This will create a variable called trumpton which should show up in the environment tab of RStudio.



We can also click on the name of the data in this tab (not the blue circle) to open a graphical view of it (it actually runs the View function on it to do this).



## Writing data

In the same way as you can use read\_tsv or read\_csv to read data from a file into a tibble, you can use the corresponding write\_tsv or write\_csv functions to write the data from a tibble back into a text file.

> write\_csv(trumpton, "trumpton.csv")

## ‘Tidy’ Data Format

What we have seen is that the tibble data structure gives us a 2-dimensional table in which we can store data. Within this basic structure though there is still some flexibility about how to represent your data. The tidyverse, as well as providing code to manipulate your data also comes with some guidelines about how your data should be structured. Let us look at an example where we have a small amount of data.

In the data below we have measured two genes (ABC1 and DEF1), for two conditions (WT and KO). Each gene has been measured 3 times so we have 12 data points all together. There are two major ways you can represent the data.

### Wide Format

In wide format each gene would be one row in the table and each column would be a sample:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Gene** | **WT\_1** | **WT\_2** | **WT\_3** | **KO\_1** | **KO\_2** | **KO\_3** |
| **ABC1** | 8.86 | 4.18 | 8.90 | 4.00 | 14.52 | 13.39 |
| **DEF1** | 29.60 | 41.22 | 36.15 | 11.18 | 16.68 | 1.64 |

This structure has the advantage that it’s fairly compact and it makes it easy to see all of the data associated with a particular gene. However it has some disadvantages in that the genotype of the samples isn’t actually specifically recorded anywhere – it’s just included as part of the sample name. The actual expression values are also split over different rows or columns such that if you wanted to calculate some parameter for all of the measurements you’d have to collate them from different rows and columns.

### Long Format

The alternative structure you can use is long format and would look like this:

|  |  |  |  |
| --- | --- | --- | --- |
| **Gene** | **Genotype** | **Replicate** | **Value** |
| ABC1 | WT | 1 | 8.86 |
| ABC1 | WT | 2 | 4.18 |
| ABC1 | WT | 3 | 8.90 |
| ABC1 | KO | 1 | 4.00 |
| ABC1 | KO | 2 | 14.52 |
| ABC1 | KO | 3 | 13.39 |
| DEF1 | WT | 1 | 29.60 |
| DEF1 | WT | 2 | 41.22 |
| DEF1 | WT | 3 | 36.15 |
| DEF1 | KO | 1 | 11.18 |
| DEF1 | KO | 2 | 16.68 |
| DEF1 | KO | 3 | 1.64 |

This table is slightly harder for humans to interpret since the values are all laid out in a single column, and it’s a little larger because some of the values are now duplicated (each gene name is present 6 times for example), but now we have explicit annotation for each of the measures where we have a piece of data to say exactly which genotype and replicate it represents. For the actual measures, they all existing in a single column (and therefore vector) so it’s easy to refer to them all.

‘Tidy’ data uses the long format. The rationale for this comes down to how easy it is to filter, summarise and analyse so you should aim to have your data in this form. The rules for tidy data are that.

1. Each variable (type of measurement) has its own column
2. Each observation is a row
3. Each experiment is a tibble

In this context a variable is one of the measures which you explicitly set out to categorise in your experimental design. Sometimes the same type of measurement can be more than one variable – for example you could measure birth weight and adult weight, and although both of those are weight measures they would be different variables and would each have their own column, but normally it’s easiest to start by thinking that each type of data gets its own column.

# Filtering and Subsetting your Tibbles

Now that we have our data loaded into a tibble we can start working with it. We are going to use a mixture of selection methods from both core R, and the tidyverse package “dplyr” to do this manipulation of the data.

## Extracting Data Using Core R

There are a couple of quick and useful things which we can do to extract subsets of data using core R. The main things we may want to do are:

### Fetching a single column using $

You can retrieve a single column (vector) of data using the $ symbol. If I wanted to fetch the set of first names from the trumpton data we imported earlier then we can use:

> trumpton$FirstName

[1] "Chris" "Adam" "Daniel" "Chris" "Carl" "Liam" "Doug"

### Fetching column and row positions using [ ]

We saw earlier that we can retrieve specific data from a vector by putting square brackets after it and then putting a vector of index positions into it.

The same thing is true for tibbles (and data frames) but here, because we have a 2D dataset, we need to put two vectors of indices in the brackets for the rows and columns to keep.

> trumpton[3:6,c(2,4,5)]

# A tibble: 4 x 3

FirstName Weight Height

*<chr>* *<dbl>* *<dbl>*

1 Daniel 88 168

2 Chris 97 155

3 Carl 91 188

4 Liam 94 145

Here we are fetching rows 3 to 6 and columns 2, 4 and 5.

## Manipulating Data using dplyr in tidyverse

We are also going to look at some options to manipulate your data using functions from the tidyverse package dplyr.

One of the design principles for the tidyverse is that all of the functions it defines to manipulate data operate in the same way. The rules are:

1. All manipulation functions expect their data to in tibbles (or data frames)
2. The tibble must be the first argument to the function
3. The function must return a modified version of the tibble (rather than changing the original data)

### Selecting columns using select

The simplest transformation in dplyr is the select function. This allows you to extract specific columns from the tibble. As with all other tidyverse functions the first argument is always the tibble, you can then follow this by either several arguments saying which columns you want (usually their names), or a single vector of column names.

> select(trumpton,FirstName,LastName, Weight)

# A tibble: 7 x 3

FirstName LastName Weight

*<chr>* *<chr>* *<dbl>*

1 Chris Hugh 90

2 Adam Pew 102

3 Daniel Barney 88

4 Chris McGrew 97

5 Carl Cuthbert 91

6 Liam Dibble 94

7 Doug Grub 89

We could have also put the names into a single vector and got the same answer:

> select(trumpton,**c**(FirstName,LastName, Weight))

# A tibble: 7 x 3

FirstName LastName Weight

*<chr>* *<chr>* *<dbl>*

1 Chris Hugh 90

2 Adam Pew 102

3 Daniel Barney 88

4 Chris McGrew 97

5 Carl Cuthbert 91

6 Liam Dibble 94

7 Doug Grub 89

We can also use index positions instead of column names.

> select(trumpton,2,4,5)

# A tibble: 7 x 3

FirstName Weight Height

*<chr>* *<dbl>* *<dbl>*

1 Chris 90 175

2 Adam 102 183

3 Daniel 88 168

4 Chris 97 155

5 Carl 91 188

6 Liam 94 145

7 Doug 89 164

Sometimes it’s easier to say which columns you ***don’t*** want instead of which you do. To do this you just put a minus in front of the name (or number) of the columns you want to exclude.

> select(trumpton, -LastName)

# A tibble: 7 x 4

FirstName Age Weight Height

*<chr>* *<dbl>* *<dbl>* *<dbl>*

1 Chris 26 90 175

2 Adam 32 102 183

3 Daniel 18 88 168

4 Chris 48 97 155

5 Carl 28 91 188

6 Liam 35 94 145

7 Doug 31 89 164

> select(trumpton, -1, -2)

# A tibble: 7 x 3

Age Weight Height

*<dbl>* *<dbl>* *<dbl>*

1 26 90 175

2 32 102 183

3 18 88 168

4 48 97 155

5 28 91 188

6 35 94 145

7 31 89 164

### Functional selections using filter

Beyond just pulling data by position we will often want to filter our data based on a functional test. Functional tests will be of the form x>5 or y<=10 or z==”Hello” where x, y or z would be the names of columns in your tibble.

For example if we wanted to get only the people in trumpton who were taller than 170cm we would do:

> filter(trumpton, Height>=170)

# A tibble: 3 x 5

LastName FirstName Age Weight Height

*<chr>* *<chr>* *<dbl>* *<dbl>* *<dbl>*

1 Hugh Chris 26 90 175

2 Pew Adam 32 102 183

3 Cuthbert Carl 28 91 188

If we wanted to see the people who were called Chris we could do.

> filter(trumpton, FirstName == "Chris")

# A tibble: 2 x 5

LastName FirstName Age Weight Height

*<chr>* *<chr>* *<dbl>* *<dbl>* *<dbl>*

1 Hugh Chris 26 90 175

2 McGrew Chris 48 97 155

Note that for the functional test here you need to use two equals signs. A double equals sign is a functional test, whereas a single equals is an assignment and will give you an error.

> filter(trumpton, FirstName = "Chris")

Error: `FirstName` (`FirstName = "Chris"`) must not be named, do you need `==`?

### Combining Multiple Operations

We’ve seen a few examples of ways in which we can select and filter from a tibble. Each of the operations we’ve done so far has been a single change to the data, but often we want to make multiple changes.

For example, if we wanted to select only people in trumpton who were more than 170cm tall, and were called Chris and we only wanted to know their age and weight we’d have to do that in three separate steps. In the example below I’m running the three steps separately and I’m saving intermediate results into new tibbles and then passing them to the next operation in the chain.

> filter(trumpton, Height >= 170) -> answer1

> filter(answer1, FirstName == "Chris") -> answer2

> select(answer2, Age, Weight)

# A tibble: 1 x 2

Age Weight

*<dbl>* *<dbl>*

1 26 90

Whilst this works, it’s also a bit wasteful since it creates a new temporary named tibble at each step. We can find a nicer way to combine them.

One of the rules of tidyverse is that all functions take a tibble as their first argument, and they return a tibble. This means that we can combine functions together by passing the output of one operation as the input to the next one.

To do this tidyverse defines a new operator (like + or -) called the pipe operator, written as %>%

The %>% operator takes whatever is on its left hand side and makes it become the first argument to a function on its right hand side. So the two statements below are exactly the same.

> select(trumpton,-LastName)

# A tibble: 7 x 4

FirstName Age Weight Height

*<chr>* *<dbl>* *<dbl>* *<dbl>*

1 Chris 26 90 175

2 Adam 32 102 183

3 Daniel 18 88 168

4 Chris 48 97 155

5 Carl 28 91 188

6 Liam 35 94 145

7 Doug 31 89 164

> trumpton %>% select(-LastName)

# A tibble: 7 x 4

FirstName Age Weight Height

*<chr>* *<dbl>* *<dbl>* *<dbl>*

1 Chris 26 90 175

2 Adam 32 102 183

3 Daniel 18 88 168

4 Chris 48 97 155

5 Carl 28 91 188

6 Liam 35 94 145

7 Doug 31 89 164

Going one step further we can therefore chain multiple operations in tidyverse together by combining multiple functions joined by %>% operators. The tibbles are passed seamlessly from one to the other.

With our original filter of finding the age and weight of people called Chris who are 170cm tall we can rewrite out original solution more compactly using pipes.

> trumpton %>% filter(Height>=170) %>% filter(FirstName=="Chris") %>% select(Age,Weight)

# A tibble: 1 x 2

Age Weight

*<dbl>* *<dbl>*

1 26 90

Because the lines can get quite long when doing this we normally put a newline in after each pipe, and indent the next line so the structure of the operation is clearer.

trumpton %>%

filter(Height >= 170) %>%

filter(FirstName == "Chris") %>%

select(Age,Weight)

# Drawing Graphs with ggplot

One of the other big attractions of R is being able to use it to plot figures and graphs. Core R comes with a set of plotting functions for different graph types (boxplot, barplot etc), but as with data manipulation there is a tidyverse alternative to the core functionality.

The tidyverse plotting package is called ggplot and this is one of the most popular packages in R. We will therefore use ggplot to plot out our data.

Here we are going to look at some simple plots we can perform using ggplot. We will do a little bit of customisation, but there is obviously much more that you can do with these graphs.

## Defining your data

Since ggplot is part of the tidyverse it expects that its data will arrive in the form of a tibble (or data frame), and that it will be structured in ‘tidy’ format. All plots start with a call to the ggplot function, and the first argument will be the tibble containing the data you want to plot.

Here we’re going to use a small dataset showing the expression of a WT and a KO where we also know the names of the genes and the p-value of the significance of the difference.

> read\_tsv("expr\_data.txt") -> expression

Parsed with column specification:

cols(

Gene = col\_character(),

WT = col\_double(),

KO = col\_double(),

pValue = col\_double()

)

> expression

# A tibble: 12 x 4

Gene WT KO pValue

*<chr>* *<dbl>* *<dbl>* *<dbl>*

1 Mia1 5.83 3.24 0.1

2 Snrpa 8.59 5.02 0.001

3 Itpkc 8.49 6.16 0.04

4 Adck4 7.69 6.41 0.2

5 Numbl 8.37 6.81 0.1

6 Ltbp4 6.96 10.4 0.001

7 Shkbp1 7.57 5.83 0.1

8 Spnb4 10.7 9.38 0.2

9 Blvrb 7.32 5.29 0.05

10 Pgam1 0 0.285 0.5

11 Sertad3 8.13 3.02 0.0001

12 Sertad1 7.69 4.34 0.01

Calling ggplot at this stage will work, but won’t give us anything more than a blank plot. That’s OK though because we’ll fix that in a second.

> ggplot(expression)

## Geometries and Aesthetics

The reason the call to ggplot didn’t actually draw anything is that you haven’t told it what sort of graphical representation you want to use, and which bits of your data you want to plot. This information is provided by providing a geometry and defining the ‘aesthetics’ of that geometry.

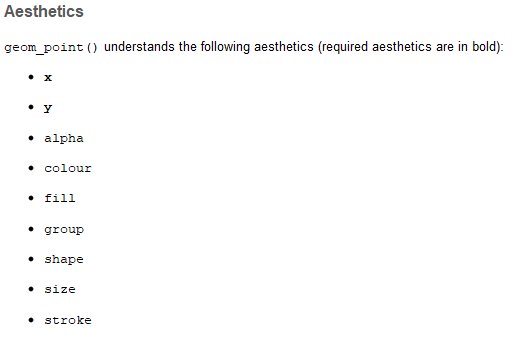
Geometries in ggplot say what sort of plot you want to draw. They are all functions, but you’re going to use them in a slightly non-standard way.

* geom\_point() Point geometry, (x/y plots, stripcharts etc)
* geom\_line() Line graphs
* geom\_boxplot() Box plots
* geom\_bar() Barplots
* geom\_histogram() Histogram plots

There are plenty of other geometries, but these should give you an idea to get started.

Every geometry in ggplot then has a number of aesthetics associates with it. An aesthetic is simply a graphical parameter which can be adjusted to change how the plot looks. You can look up the help for any of the geometry functions to see what aesthetics it has and how they can be altered.

For example the help page for geom\_point says this:



So that shows us the full list of things which we can change in this type of plot, but that at the minimum we must supply x and y, otherwise the plot can’t be drawn.

## How to specify aesthetics

You can specify aesthetics in one of two ways;

1. You can provide a fixed value for the aesthetic, eg color="red" if you want everything to be red
2. You can provide an “aesthetic mapping” which will be the name of a column in your data which will be used to dynamically modify that aesthetic

Aesthetic mappings are generated by using the aes() function. Inside that the argument are name=value pairs where the name is the name of the aesthetic and the value is the name of the column in your tibble which you want to use to provide the data for that aesthetic.

Fixed aesthetic values (color="red") have to be supplied in the call to the geometry function:

geom\_point(color="red")

In complex plots you will often want to use the same aesthetics for multiple geometries – eg using geom\_point() to draw a scatterplot and using geom\_text() to label the points. The x and y aesthetics will therefore be the same for both of these (because the data is the same). For dynamic aesthetic plots therefore you can specify the aes() mappings either in the individual geometries, or in the original ggplot() function.

ggplot(expression, mapping=aes(x=WT, y=KO)

## Putting it all together

We have now seen that a ggplot plot will use an initial call to the ggplot function, where we can add specific aesthetic mappings, followed by calls to one or more geometry functions which will actually draw on the graph, but we haven’t said how these will be combined.

The answer to this is that the call to ggplot generates a value which we can store in a variable. We can then add (literally using a +) the output of calls to geometry functions to add plotting layers to the variable. The plot is finally drawn when we print the value.

So for our simple scatterplot of WT vs KO we would do an initial call to ggplot, and specify the aesthetic mappings;

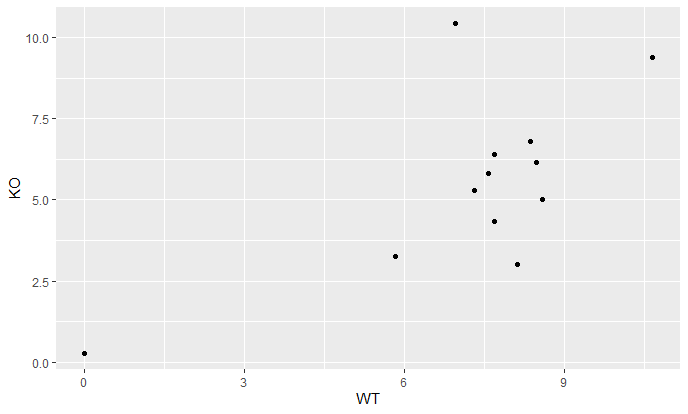
ggplot(expression, aes(x=WT, y=KO)) -> my.plot

We can then add on a geometry to say how it should be drawn. In this case a point geometry

my.plot + geom\_point() -> my.plot

Finally we can print the value to draw the plot

my.plot



Again, for convenience, we will often not actually store the graph in a variable, but will just build it up and write it out in a single (somewhat large) statement. For clarity it’s a good idea to use a new line every time you add on a geometry – much like starting a new line after every pipe when filtering.

expression %>%

ggplot(aes(x=WT, y=KO)) +

geom\_point()

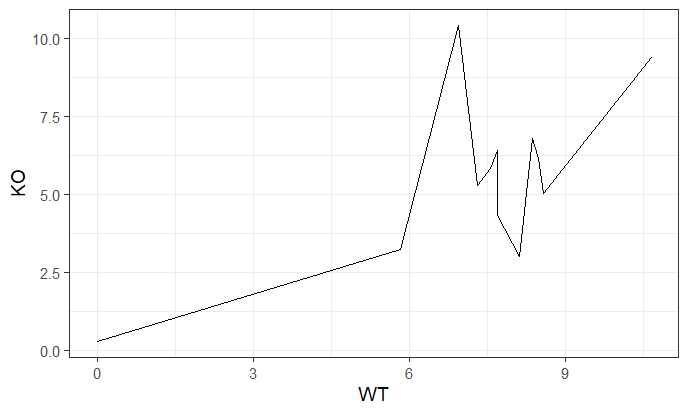
## Other plot types

You can change the type of plot you’re drawing by simply using a different geometry. For example plotting the same data as above as a line graph would just require you switching to geom\_line().

expression %>%

ggplot(aes(x=WT, y=KO)) +

geom\_line()

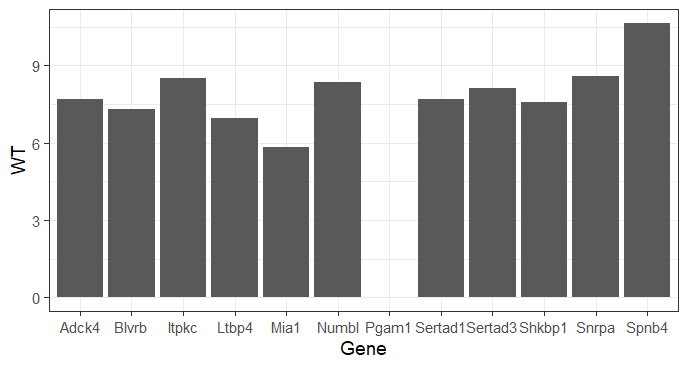


If we want to do a barplot then we still need to supply x and y, but this time x will be a categorical column (gene names) instead of a quantitative one. For geom\_bar we also need to say what we actually want to plot by setting the stat= option. This is because we can either supply a set of quantitative bar heights (stat=”identity”) or we can provide a set of values for each group and plot out the number of values (stat=”count”), or make it calculate some kinds of summary (mean, median, mode etc.) with stat=”summary”.

expression %>%

ggplot(aes(x=Gene, y=WT)) +

geom\_bar(stat="identity")



There are also some plots which will summarise larger distributions of values. Here we’re looking at a set of measurements split between two genotypes.

> many.values

# A tibble: 100,000 x 2

values genotype

*<dbl>* *<chr>*

1 1.90 KO

2 2.39 WT

3 4.32 KO

4 2.94 KO

5 0.728 WT

6 -0.280 WT

7 0.337 WT

8 -1.31 WT

9 1.55 WT

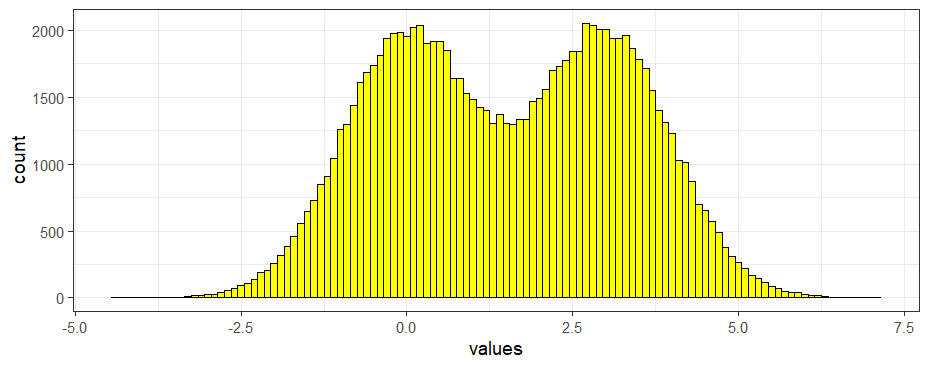
10 1.86 KO

We can plot out the whole distribution of values in a couple of ways.

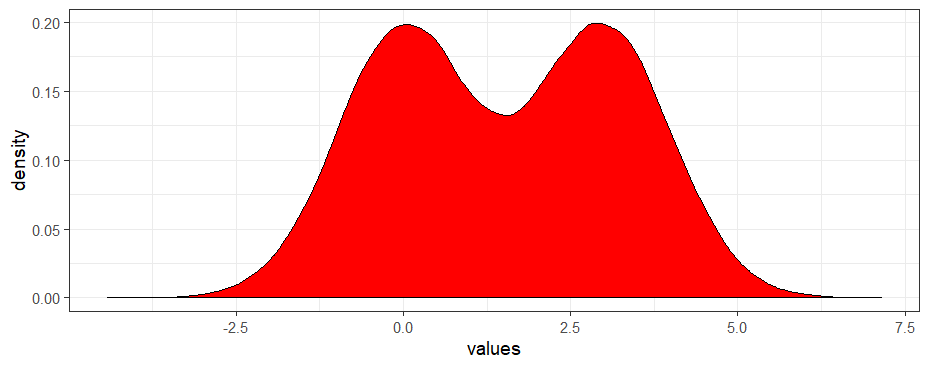
many.values %>%

ggplot(aes(values))+

geom\_histogram(binwidth = 0.1, fill="yellow", color="black")

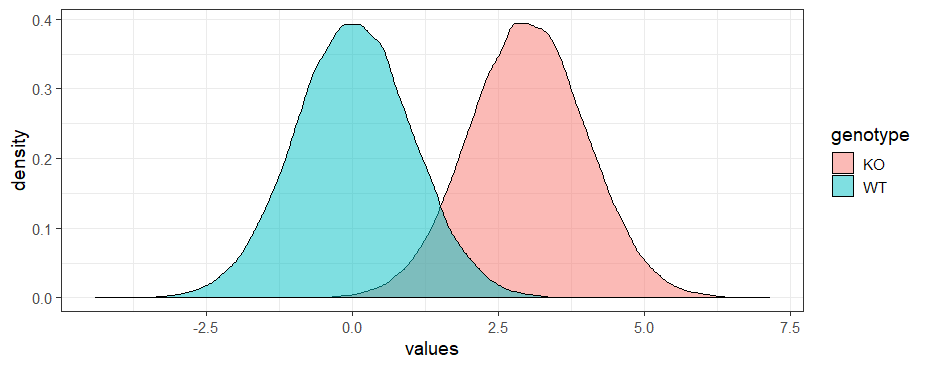


many.values %>% ggplot(aes(values))+geom\_density(fill="red")



Because we have a categorical value on which we can split the data (genotype), we can easily get separate distributions for the WT and KO by simply setting the colour to have a mapping aesthetic of the genotype. Because our distributions will overlap we’re also going to specify the alpha (transparency) value to be 0.5 so we can see both distributions where the do overlap.

many.values %>% ggplot(aes(values))+geom\_density(aes(fill=genotype), alpha=0.5)



## Multiple Geometries

If we want to build up more complex plots then we can keep adding geometries. We can also use different geometry functions to add other annotations such as plot titles and labels for the x and y axes, for example:

* ggtitle() adds a plot title
* xlab() adds an x axis label
* ylab() adds a y axis label

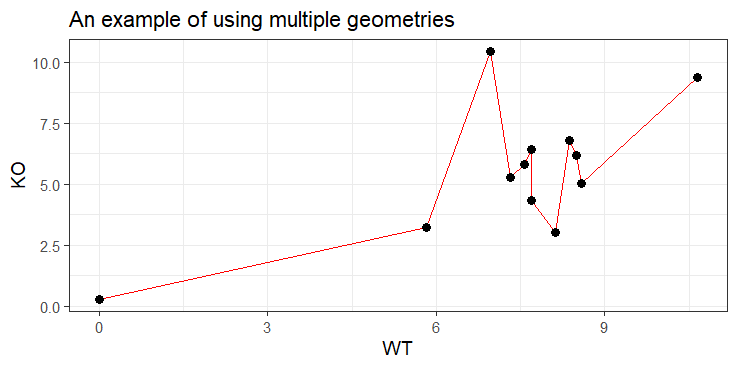
expression %>%

ggplot(aes(x=WT, y=KO)) +

geom\_line(color="red") +

geom\_point(size=3)+

ggtitle("An example of using multiple geometries")



# Further information

This course has only touched upon the many functions of R. For further information see the R project website <http://www.r-project.org/> which has various manuals available; these are useful for understanding R in more detail.

Following on from this course we have a number of other R courses which delve into more detail on different aspects of R. All of the course material can be found at <https://www.bioinformatics.babraham.ac.uk/training.html>

Follow on courses which might be of interest are:

* Advanced R – looks into more aspects of the core language focussing on robustly developing larger scripts and using automation and looping.
* Plotting with R – covers the usage of the core R plotting libraries to generate complex figures
* Using ggplot – looks at an add-in graphics package for R which supplements the graphical capabilities in the core language.
* Statistics with R – looks at how you can use the statistical capabilities within R to analyse data.

## Most importantly.....

If you have a specific problem don't spend hours getting frustrated. There are many additional online resources available (try googling ‘R help forum’) and you can also contact us at [babraham.bioinformatics@babraham.ac.uk](mailto:babraham.bioinformatics@babraham.ac.uk) and we will be happy to try to help out.