33INTRO TO R & R STUDIO WORKSHOP – SCI

```{r} ## SESSION 1 INTRODUCTION TO R

* This is going to be a very gentle and baby step intro to R. It’s intended for people who have never used R before or who have very little experience with R.
* If you are somewhat familiar with R or if you’re used to programming in other languages, let me know and I can speed things up/ tailor the workshop to your needs.
* R is a programming language and software environment that is mainly used for statistical computing and graphics. It is basically a very sophisticated calculator.

## Objectives:

By the end of this workshop, you should be familiar with how R works, and be able to do some basic data analysis and graphs.

## Getting started & R studio

* We are going to interact with R through a program called R Studio, which is a more friendly user interface in which you can basically talk to R through various commands which you can then save and edit. You do not have to use R studio and there are other interfaces for R like Notepad but most people use R Studio and if you have a Mac I think it can be complicated to get Notepad.
* Has everyone installed R? I sent a couple of links in a previous email.
* Assuming everyone has installed R and R Studio. Go ahead and open R Studio. You should see 4 windows
* TOP LEFT: Code editor/ script
* BOTTOM LEFT: R console where the commands actually get executed/ the space where you can directly speak to R.
* TOP RIGHT: workspace/ environment and history
* BOTTOM RIGHT: Plots and files
* You can change the layout if you’d like. I personally prefer to have my script and console side by side. To do that you can go to R Studio > Preferences > Pane layout and modify the layout on your menu.

## STEP1:R is a sophisticated calculator

* Let’s start by looking at the Bottom right screen the R console. -When you write code in the console. R executes your code when you press enter.
* You know that R is ready to receive your commands when you see a “>” cursor.
* So go ahead to the console and type. Then press enter:

3**+**5

## [1] 8

* STEP 2: R objects
* As you can see R works like a calculator. But note how R’s answer is preceded by a one inside brackets. That’s telling you that Rs answer is one component long.
* R can also hold objects in its environment/ its head if you will. For example, type in:

x <- 2

* Now look at the environment/ workspace, the window on the top left. an object x appears, with the current value of 2. That means R current has ‘in its mental workspace’, object x. And once it is there, you can do anything to it you could do to a number. So for example try a few calculations like:

x**+**5

## [1] 7

x**\***10

## [1] 20

**sqrt**(x)

## [1] 1.414214

x**^**2

## [1] 4

* The last two are the square root and the square of x. You should get the numbers you expect! And if you want R to report the current value of an object in its environment, then you just type the object’s name:

x

## [1] 2

* One main category of R objects are vectors. This means that they are not single numbers, but ordered series of numbers (our x so far is a vector of length 1, so this has not been evident).
* You make several numbers into a vector using the function c(), which stands for ‘combine’. So:

x <- **c**(2,1,5,4)

* Now you have a numeric vector in your environment of length 4. Repeat the calculations you did before, and note what the output is.

x**+**5

## [1] 7 6 10 9

x**\***10

## [1] 20 10 50 40

**sqrt**(x)

## [1] 1.414214 1.000000 2.236068 2.000000

x**^**2

## [1] 4 1 25 16

* You can also query or change any particular element of a vector using the square bracket notation [ ]. For example:

x[3]

## [1] 5

* In R, element numbering stars at 1, not 0. So x[3] indicates the third element of the vector x.
* Now let’s change one element:

x[3] <- 100  
x

## [1] 2 1 100 4

* Now enter x, see how the third element was changed
* Now let’s create another object, y, that holds the values of whether each element of x is equal to 5 or not:

y <- (x**==**5)

* Now look in your environment window. You now have another object, y. But note that whereas the object x is denoted ‘num’ (it’s a numeric vector), y is denoted ‘logi’, because it is a different type of vector called a logical vector that consists of TRUEs and FALSEs.

## CLASS OF OBJECTS

* Objects are the things you perform functions on.
* In R every object has a nature or in R language “class”
* The class determines the kind of object it is, and how any particular function can be applied to it.
* To find out the class of an object in your environment, you use the class() function. (I assume you still have objects x and y in your environment from earlier.). Let’s try that:

**class**(x)

## [1] "numeric"

**class**(y)

## [1] "logical"

* So x is a numeric vector (it consists of data values), and z is a logical one (it consists of TRUEs and FALSEs).
* RStudio is great because it shows this information all the time in your environment window to your right.

-There are also other kinds of objects. For example you might have some categorical data for which you need quotation marks.Let’s create a vector for eye colors.

eye\_colors<- **c**("blue", "brown","green","hazel")  
**class**(eye\_colors)

## [1] "character"

* You see that “eye\_colors” is an object of the class ‘character’.
* An object’s class may determine whether a given function can be properly applied to it. For example try:

**length** (eye\_colors)

## [1] 4

* R tells you the vector “eye\_colors” is of length 4 meaning it contains 4 elements.
* In R your statistical tests are also objects (with their own classes).
* And your graphs may also be objects or saved as objects.
* This is useful because once they are defined in your environment, you can apply all kinds of functions to them like printing them, combining them, comparing them, extracting values from them, and so forth.
* So the thing you have to get used to in R is that everything is an object, with a class proper to the kind of object that it is.
* There are other kinds of objects in R that we are going to learn about in this workshop but first I want to introduce you to some basic function in R, and show you how to write and save script in R Studio.

## FUNCTIONS

* Functions are commands followed by () brackets, and they basically say to R: “do the operation defined by the function to the objects listed in the brackets (these objects are called the arguments of the function)”. For example:

**mean**(x)

## [1] 26.75

* This means do the function arithmetic mean to the argument vector x.
* If you don’t say otherwise, R will return the outcome of doing the function to the console.
* You can also store the outcome in an object instead. For instance:

my\_mean <- **mean**(x)

* Now the object “my\_mean” appears in your environment.
* This is useful for all sorts of reasons
* For example if you want to know which of your data points are below the mean. You can use the function which() and the < operator

**which** ( x **<** my\_mean)

## [1] 1 2 4

* Here are is telling you that the first, second and fourth elements of vector x are below the mean.
* What if we wanted to know which elements of x were equal to the mean? In R the mathematical equal sign is signified by 2 double signs “==”. So let’s try:

**which**(x**==**my\_mean)

## integer(0)

* It returns integer(0) because none of the elements of vector x are equal to the mean “3”. Lets try instead:

**which**(x **==** 1)

## [1] 2

## STEP 3: BASIC STATISTICAL FUNCTIONS

* R has a bunch of built-in statistical functions in addition to the mean. You can apply those functions to your x vector from earlier .Let’s try the following:

**sd**(x)

## [1] 48.84926

**median** (x)

## [1] 3

**max**(x)

## [1] 100

**min**(x)

## [1] 1

**length**(x)

## [1] 4

## FACTORS

* The term factor refers to a statistical data type used to store categorical variables. The difference between a categorical variable and a continuous variable is that a categorical variable can belong to a limited number of categories. A continuous variable, on the other hand, can correspond to an infinite number of values.
* It is important that R knows whether it is dealing with a continuous or a categorical variable, as the statistical models you will develop in the future treat both types differently. (You will see later why this is the case.)
* A good example of a categorical variable is sex. In many circumstances you can limit the sex categories to “Male” or “Female”. (Sometimes you may need different categories. For example, you may need to consider chromosomal variation, hermaphroditic animals, or different cultural norms, but you will always have a finite number of categories.)

sex\_vector <- **c** ("male","female","female","female","male","male","female")  
**class**(sex\_vector)

## [1] "character"

sex\_vector <- **factor** (sex\_vector)  
**class**(sex\_vector)

## [1] "factor"

sex\_vector

## [1] male female female female male male female  
## Levels: female male

* R tells you this factor has 2 levels female and male
* There are two types of categorical variables: a nominal categorical variable and an ordinal categorical variable.
* A nominal variable is a categorical variable without an implied order. This means that it is impossible to say that ‘one is worth more than the other’. For example, if you had a vector with categories of animals like “Elephant”, “Giraffe”, “Donkey” and “Horse”. Here, it is impossible to say that one stands above or below the other. (Note that some of you might disagree ;-) ).
* In contrast, ordinal variables do have a natural ordering. For example if youre dealing with a categorical variable for temperature and you had : “Low”, “Medium” and “High” levels. Here it is obvious that “Medium” stands above “Low”, and “High” stands above “Medium”.

animals\_vector <- **c**("elephant", "giraffe", "donkey", "horse")  
temperature\_vector <- **c**("high", "low", "high","low", "medium")  
animals\_vector <- **factor** (animals\_vector)  
temperature\_vector <- **factor** (temperature\_vector)

* Your can use the function levels () to take a look at the levels of your factor or to change them. For example.

**levels** (sex\_vector)

## [1] "female" "male"

**levels** (sex\_vector) <- **c**("F","M")  
**levels** (sex\_vector)

## [1] "F" "M"

* The summary function gives you a quick look of the contents of any object as well

**summary**(sex\_vector)

## F M   
## 4 3

* Sex doesn’t have a natural ordering between its categories. But sometimes you may deal with factors that do have a natural ordering between its categories. If this is the case, we have to make sure that we pass this information to R…

Let’ us’s say that you are leading a research team of five data analysts and that you want to evaluate their performance. To do this, you track their speed, evaluate each analyst as “slow”, “medium” or “fast”, and save the results in speed\_vector

*#Let's create a speed vector*  
speed\_vector <- **c**("fast", "slow","fast","medium", "medium","medium")  
  
*#Now let's transform into factor and order them*  
speed\_vector <- **factor** (speed\_vector, ordered = TRUE, levels=**c**("slow","medium","fast"))  
  
*#Print*  
speed\_vector

## [1] fast slow fast medium medium medium  
## Levels: slow < medium < fast

## WRITING SCRIPTS

* So far we have been working directly into the console, writing one command at a time for R to execute.
* But when you do data analysis, what you really want to do is to write a whole series of commands, edit and perfect them, then send them to R for execution as a block.
* We do that by writing what is called a script.
* A script is just a text file with a series of lines of code that will get sent to R when you are ready to execute them.
* R Studio, you open a new script using the ‘File > New File > R Script’ menu commands at the top.
* That’s your blank script. The first thing you might want to do is save it somewhere with an appropriate name (save icon on the top of the script window, or ‘File > Save’ via the menus. Or Control+s.).
* The file extension should be .r.
* Done that? Ok. Let’s write something in our script. Try writing out:

*# This is my first script*  
  
a <- 2  
**print** (a)

## [1] 2

a <- a **\*** 3  
**print** (a)

## [1] 6

b <- **rep** ("R is awesome", a )

## THE WORKING DIRECTORY - We are getting on to using a more interesting script, but first, a bit about working directories.

* The working directory is the place on your computer R is going to look first for files (like scripts and data files) to read in,and also the default place it is going to save the scripts you’re working on and files you’re working with.
* It’s worth setting the working directory right before you begin so you know where everything needs to be. You find your current working directory with the function:

**getwd**()

## [1] "/Users/sarahalami/Google Drive/R resources"

* And you set the working directory with the setwd() function: setwd(), where you can type in the name of your working directory.
* I usually don’t really like to do that because you can mispell etc.
* Instead RStudio allows you to set the working directory via a menu by browsing your computer: ‘Session > Set Working Directory > Choose Directory’.

## OTHER R OBJECTS2: matrices and matrix calculations

* In R, a matrix is a collection of elements of the exact same class (numeric, character or logical) arranged into a fixed number of rows and columns. Since you are only working with rows and columns, a matrix is called two-dimensional.
* You can construct a matrix in R with the matrix() function. Consider the following example:

**matrix**(1**:**9, byrow = TRUE, nrow = 3)

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

* The first argument is the collection of elements that R will arrange into the rows and columns of the matrix. Here, we use 1:9 which is a shortcut for

**c**(1, 2, 3, 4, 5, 6, 7, 8, 9)

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

* The argument byrow indicates that the matrix is filled by the rows. If we want the matrix to be filled by the columns, we just place byrow = FALSE.
* The third argument nrow indicates that the matrix should have three rows.
* Now, let’s make and analyze a matrice.
* We are going to analyze the box office numbers of the Star Wars franchise. Those numbers are avaible from Wikipedia.
* We’re gonna define 3 vectors. Each one represents the box office numbers from the first three Star Wars movies. The first element of each vector indicates the US box office revenue, the second element refers to the Non-US box office. Yo dont have to enter those exact numbers you can make up simpler number. I have also sent you the code over chat.

*# Box office Star Wars (in millions!)*  
new\_hope <- **c**(460.998, 314.4)  
empire\_strikes <- **c**(290.475, 247.900)  
return\_jedi <- **c**(309.306, 165.8)

* Now we are gonna combine all these figures into a single vector. And then, we’re gonna build a matrix from this vector.

-Use c(new\_hope, empire\_strikes, return\_jedi) to combine the three vectors into one vector. Call this vector box\_office.

-Construct a matrix with 3 rows, where each row represents a movie. Use the matrix() function to do this. The first argument is the vector box\_office, containing all box office figures. Next, you’ll have to specify nrow = 3 and byrow = TRUE. Name the resulting matrix star\_wars\_matrix.

*# Create box\_office*  
box\_office <- **c**(new\_hope, empire\_strikes, return\_jedi)  
  
*# Construct star\_wars\_matrix*  
star\_wars\_matrix <- **matrix**(**c**(new\_hope,empire\_strikes,return\_jedi), byrow=TRUE, nrow=3)  
star\_wars\_matrix

## [,1] [,2]  
## [1,] 460.998 314.4  
## [2,] 290.475 247.9  
## [3,] 309.306 165.8

*#same as*  
star\_wars\_matrix <- **matrix**(box\_office, byrow=TRUE, nrow=3)  
star\_wars\_matrix

## [,1] [,2]  
## [1,] 460.998 314.4  
## [2,] 290.475 247.9  
## [3,] 309.306 165.8

* Now let’s name our rows and columns

**rownames** (star\_wars\_matrix) <- **c**( "A new hope","The empire strikes back","Return of the Jedi")  
**colnames** (star\_wars\_matrix) <- **c**("US revenue", "Overseas revenue")

* You can do calculations on a matrix. For example let’s calculate the total revenue which is basically the revenue from US + the revenue from overseas.We’re gonna save it in a vector called

total\_revenue<- **rowSums**(star\_wars\_matrix)

* Now let’s add a column for that total revenue in the matrix. We’re gonna do that using a function called cbind() which merges matrices and/or vectors together by column so c stands for column and bind.

star\_wars\_super\_matrix<- **cbind**(star\_wars\_matrix, total\_revenue)  
star\_wars\_super\_matrix

## US revenue Overseas revenue total\_revenue  
## A new hope 460.998 314.4 775.398  
## The empire strikes back 290.475 247.9 538.375  
## Return of the Jedi 309.306 165.8 475.106

* Now let’s add a row. First we’re gonna create another matrix with similar data for the prequels trilogy.

phantom\_menace <- **c**(474.5,552.5)  
attack\_of\_clones <- **c**(310.7,338.7)  
revenge\_of\_sith <- **c**(380.3, 468.5)  
  
prequel\_matrix <- **matrix**(**c**(phantom\_menace,attack\_of\_clones,revenge\_of\_sith), byrow=TRUE, nrow=3)  
prequel\_matrix

## [,1] [,2]  
## [1,] 474.5 552.5  
## [2,] 310.7 338.7  
## [3,] 380.3 468.5

**rownames** (prequel\_matrix) <-**c**("The phantom menace","Attack of the clones","Revenge of the Sith")  
*#no need to name the columns because they are staying the same*  
  
all\_star\_wars\_matrix <- **rbind**(star\_wars\_matrix, prequel\_matrix)  
all\_star\_wars\_matrix

## US revenue Overseas revenue  
## A new hope 460.998 314.4  
## The empire strikes back 290.475 247.9  
## Return of the Jedi 309.306 165.8  
## The phantom menace 474.500 552.5  
## Attack of the clones 310.700 338.7  
## Revenge of the Sith 380.300 468.5

* Just like cbind() has rbind(), colSums() has rowSums(). Now by summing rows we can calculate the total box office revenue for the entire saga.

revenue\_entire\_saga <- **colSums**(all\_star\_wars\_matrix)  
revenue\_entire\_saga

## US revenue Overseas revenue   
## 2226.279 2087.800

* You can also select matrix elements similar to what we did with vectors:

all\_star\_wars\_matrix[1,2] *# selects the element at the 1st row and second column*

## [1] 314.4

If you want to select all elements of a row or a column, no number is needed before or after the comma, respectively:

all\_star\_wars\_matrix[,1]*#selects all elements of the first column.*

## A new hope The empire strikes back Return of the Jedi   
## 460.998 290.475 309.306   
## The phantom menace Attack of the clones Revenge of the Sith   
## 474.500 310.700 380.300

all\_star\_wars\_matrix[1,]*#selects all elements of the first row.*

## US revenue Overseas revenue   
## 460.998 314.400

* Similar to what we learned h with vectors, the standard operators like +, -, /, \*, etc. work in an element-wise way on matrices in R. For example:

2**\***all\_star\_wars\_matrix*#multiplies each element of the matrix by 2*

## US revenue Overseas revenue  
## A new hope 921.996 628.8  
## The empire strikes back 580.950 495.8  
## Return of the Jedi 618.612 331.6  
## The phantom menace 949.000 1105.0  
## Attack of the clones 621.400 677.4  
## Revenge of the Sith 760.600 937.0

* For example:

new\_hope <- **c**(460.998, 314.4)  
empire\_strikes <- **c**(290.475, 247.900)  
return\_jedi <- **c**(309.306, 165.8)  
star\_wars\_matrix <- **matrix**(**c**(new\_hope,empire\_strikes,return\_jedi), byrow=TRUE, nrow=3)  
  
phantom\_menace <- **c**(474.5,552.5)  
attack\_of\_clones <- **c**(310.7,338.7)  
revenge\_of\_sith <- **c**(380.3, 468.5)  
prequel\_matrix <- **matrix**(**c**(phantom\_menace,attack\_of\_clones,revenge\_of\_sith), byrow=TRUE, nrow=3)  
  
  
star\_wars\_matrix **\*** prequel\_matrix *# creates a matrix where each element is the product of the corresponding elements in the 2 matrices.*

## [,1] [,2]  
## [1,] 218743.55 173706.00  
## [2,] 90250.58 83963.73  
## [3,] 117629.07 77677.30

* Those who are familiar with matrices should note that this is not the standard matrix multiplication for which you should use %\*% in R (it wouldn’t work with these anyway because the two matrices are the same number f rows and cols)

## DATA FRAMES & SOME BASIC GRAPHS AND STATISTICAL TESTS IN R

-The main issue with matrices is that all the elements that you put in a matrix should be of the same type or class. The data set on Star Wars only contained numeric elements for example.

In most datasets you usually have different kinds or classes of data: categorical, numerical etc.

For example you can have questions such as:

‘Are you married?’ or ‘yes/no’ questions (logical) ‘How old are you?’ (numeric) ‘What’s your favorite movie?’ or other ‘open-ended’ questions (character) … The output, namely the respondents’ answers to the questions formulated above, is a data set of different data types. You will often find yourself working with data sets that contain different data types instead of only one.

A data frame has the variables of a data set as columns and the observations as rows. This will be a familiar concept for those coming from different statistical software packages such as SAS or SPSS.

* We are going to read in some data and do a simple analysis or two.
* In R you can import your data from various format which I will show you how to do later. For now we’re gonna work with a dataset thats built-in R. Go ahead and load the following dataset.

**data**("iris")

* To summarise, the data set consists of four measurements (length and width of the petals and sepals in cm) of one hundred and fifty Iris flowers from three species:
* Since this is a buit-in dataset you can find that info in R’s help menu. On order to look up anything that’s built into R you can either search directly in R’s help menu or type a question mark followed by the name of whichever object you want to know more about.

?iris

* First, let’s find out what class this dataset is

**class**(iris)

## [1] "data.frame"

* Yes, it’s a special class of object called a data frame that consists of multiple vectors organized in columns. Not let’s see what it looks like. We are gonna use the head() function which gives you the first 6 rows of a dataset to look at. You can also look at the tail which gives you the last 6 rows.

**head**(iris)

## Sepal.Length Sepal.Width Petal.Length Petal.Width Species  
## 1 5.1 3.5 1.4 0.2 setosa  
## 2 4.9 3.0 1.4 0.2 setosa  
## 3 4.7 3.2 1.3 0.2 setosa  
## 4 4.6 3.1 1.5 0.2 setosa  
## 5 5.0 3.6 1.4 0.2 setosa  
## 6 5.4 3.9 1.7 0.4 setosa

**tail** (iris)

## Sepal.Length Sepal.Width Petal.Length Petal.Width Species  
## 145 6.7 3.3 5.7 2.5 virginica  
## 146 6.7 3.0 5.2 2.3 virginica  
## 147 6.3 2.5 5.0 1.9 virginica  
## 148 6.5 3.0 5.2 2.0 virginica  
## 149 6.2 3.4 5.4 2.3 virginica  
## 150 5.9 3.0 5.1 1.8 virginica

* Another useful function is str(), which gives you a lot of info about the structure of your dataset.

**str**(iris)

## 'data.frame': 150 obs. of 5 variables:  
## $ Sepal.Length: num 5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...  
## $ Sepal.Width : num 3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...  
## $ Petal.Length: num 1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...  
## $ Petal.Width : num 0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...  
## $ Species : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...

* This function gives you a bunch of useful information. It tells you that you dataset is made up of 150 observations of 5 variables:Sepal length, sepal width, petal length, petal width and the species of flower. (You can also get this information in the Environment window, by clicking on the little expansion arrow next to the object d.)
* Another useful function is colnames(), which returns the column names of a data frame.

**colnames**(iris)

## [1] "Sepal.Length" "Sepal.Width" "Petal.Length" "Petal.Width"   
## [5] "Species"

* You can access the sub-objects of an object (usually) using the $ notation. So, for example:

iris**$**Species

## [1] setosa setosa setosa setosa setosa setosa   
## [7] setosa setosa setosa setosa setosa setosa   
## [13] setosa setosa setosa setosa setosa setosa   
## [19] setosa setosa setosa setosa setosa setosa   
## [25] setosa setosa setosa setosa setosa setosa   
## [31] setosa setosa setosa setosa setosa setosa   
## [37] setosa setosa setosa setosa setosa setosa   
## [43] setosa setosa setosa setosa setosa setosa   
## [49] setosa setosa versicolor versicolor versicolor versicolor  
## [55] versicolor versicolor versicolor versicolor versicolor versicolor  
## [61] versicolor versicolor versicolor versicolor versicolor versicolor  
## [67] versicolor versicolor versicolor versicolor versicolor versicolor  
## [73] versicolor versicolor versicolor versicolor versicolor versicolor  
## [79] versicolor versicolor versicolor versicolor versicolor versicolor  
## [85] versicolor versicolor versicolor versicolor versicolor versicolor  
## [91] versicolor versicolor versicolor versicolor versicolor versicolor  
## [97] versicolor versicolor versicolor versicolor virginica virginica   
## [103] virginica virginica virginica virginica virginica virginica   
## [109] virginica virginica virginica virginica virginica virginica   
## [115] virginica virginica virginica virginica virginica virginica   
## [121] virginica virginica virginica virginica virginica virginica   
## [127] virginica virginica virginica virginica virginica virginica   
## [133] virginica virginica virginica virginica virginica virginica   
## [139] virginica virginica virginica virginica virginica virginica   
## [145] virginica virginica virginica virginica virginica virginica   
## Levels: setosa versicolor virginica

* you can also interrogate a data frame using row and column positions:

iris[2,3]

## [1] 1.4

* So the 3rd variable of the second case of these data is 1.4.
* Sometimes there can be issues/ you need to write some extra code to run certain functions if your data is missing values. So let’s see if there are any missing values in this dataset.

*#is.na (iris)*  
**sum**(**is.na**(iris))

## [1] 0

50 50 50

## PIPES

* You can also pass arguments to functions using pipes, which allow objects to always be first arguments.
* In base R, pipes look like this: |>

> table(iris$Species)

setosa versicolor virginica

50 50 50

> iris$Species |> table()

setosa versicolor virginica

* What’s the point? When you have a lot of nested functions it becomes harder to see what’s going on.

barplot(prop.table(table(iris$Species)), main="species of flowers")

iris$Species |> table() |> prop.table () |> barplot(main="species of flowers")

* Pipes were initially invented in the context of a package called tidyverse, which is a sublanguage of R based on pipes.
* Let’s take advantage of that to learn how to install and read packages in R …
* In tidyverse, all functions take an object as their first argument and have functions that take of query that object. Pipes look differently than base R (a more recent innovation) : %>% but it’s the same logic.

> iris$Species %>% table()

.

setosa versicolor virginica

50 50 50

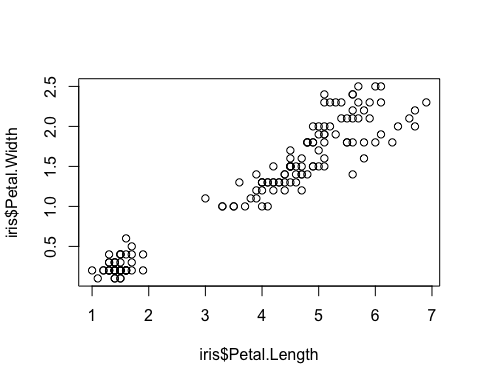
## SIMPLE PLOTS

* Now let’s go over some exploratory plots. First let’s check out the function plot.

?plot

It tells you that this function has two “necessary” arguments x and y and then there is a third extra argument, called type that allows you to change the graphic details of the plot (e.g. plotting dots vs squares). FOr the fun to work you have to enter the two main arguments, but the 3rd one is set to a default which you can change but you don’t have to for the plot to work. Let’s have the petal length variable on the X axis and the petal width the Y axis.

**plot**(iris**$**Petal.Length, iris**$**Petal.Width)



* You should have a plot in the bottom right window, showing that the bigger the pethal length the larger the petal width. Let’s try to see if that’s actually the case when calculating the correlation coefficient (Pearson’s R) between these 2 vars. For that we’re gonna use a function called cor.test() that does exactly that.

**cor.test**(iris**$**Petal.Length, iris**$**Petal.Width)

##   
## Pearson's product-moment correlation  
##   
## data: iris$Petal.Length and iris$Petal.Width  
## t = 43.387, df = 148, p-value < 2.2e-16  
## alternative hypothesis: true correlation is not equal to 0  
## 95 percent confidence interval:  
## 0.9490525 0.9729853  
## sample estimates:  
## cor   
## 0.9628654

* Yes they are almost perfectly coorelated
* There are lots of ways of making your plot nicer, labelling the axes and so forth, but we won’t go into these today. I want to focus on how R works and once you get some bases you can then explore on your own.
* In the plot() command you just wrote, we encountered for the first time something we will often meet in R:the formula.
* An R formula is something on the left hand side, then a tilde symbol, then something on the right hand side, as in:

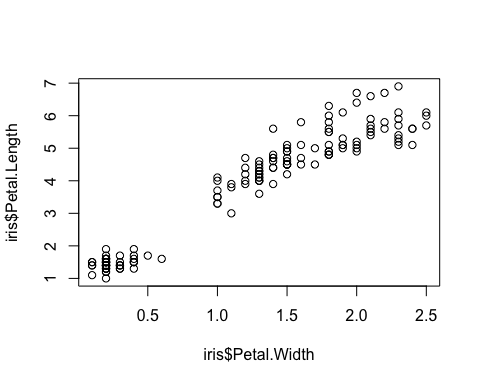
iris**$**Petal.Length **~** iris**$**Petal.Width

## iris$Petal.Length ~ iris$Petal.Width

* You will use formulas of this kinda lot in R, both for plots and for statistical tests.
* The left hand side is what you want as the outcome variable / vertical axis; the right hand side is what you want as the predictor variable(s) / horizontal axis.

Try:

**plot**(iris**$**Petal.Length **~** iris**$**Petal.Width)



* Right you get the same plot.

## saving & reading a db into R and R studio

**write.csv**(iris, "db\_iris.csv", row.names = F)  
db <- **read.csv**("db\_iris.csv", as.is=T)

```