Suffolk 2019

Contents

About this course **1**

1 Introduction to R and RStudio **1**
1.1 Learning objectives 1

2 R-syntax, data structures and types **4**
2.1 Seminar 4

3 Data import (from csv, txt, and excel) **9**
3.1 Seminar 9

4 R-syntax, data structures and types **11**
4.1 Seminar 11

5 Type coercion **15**
5.1 Seminar 15

6 Loops and conditions **17**
6.1 Seminar 17

7 Visualising data **20**
7.1 Seminar 20

8 Correlations and differences in means **30**
8.1 Seminar 30

9 Regression **30**
9.1 Seminar 30

10 Prediction and assessing prediction accuracy **30**
10.1 Seminar 30

About this course

This course is an introduction R, RStudio and statistics. Our primary aims are to get comfortable working with R and to be able to prepare, manipulate, analyse and visualise data..

---

1 Introduction to R and RStudio

1.1 Learning objectives

In this session, we will have a look at R and RStudio. We will interact with both and use the various components of RStudio.

1.1.1 What is R?

R is an environment for statistical computing and graphics. RStudio is an editor or integrated development environment (IDE) that makes working with R much more comfortable.

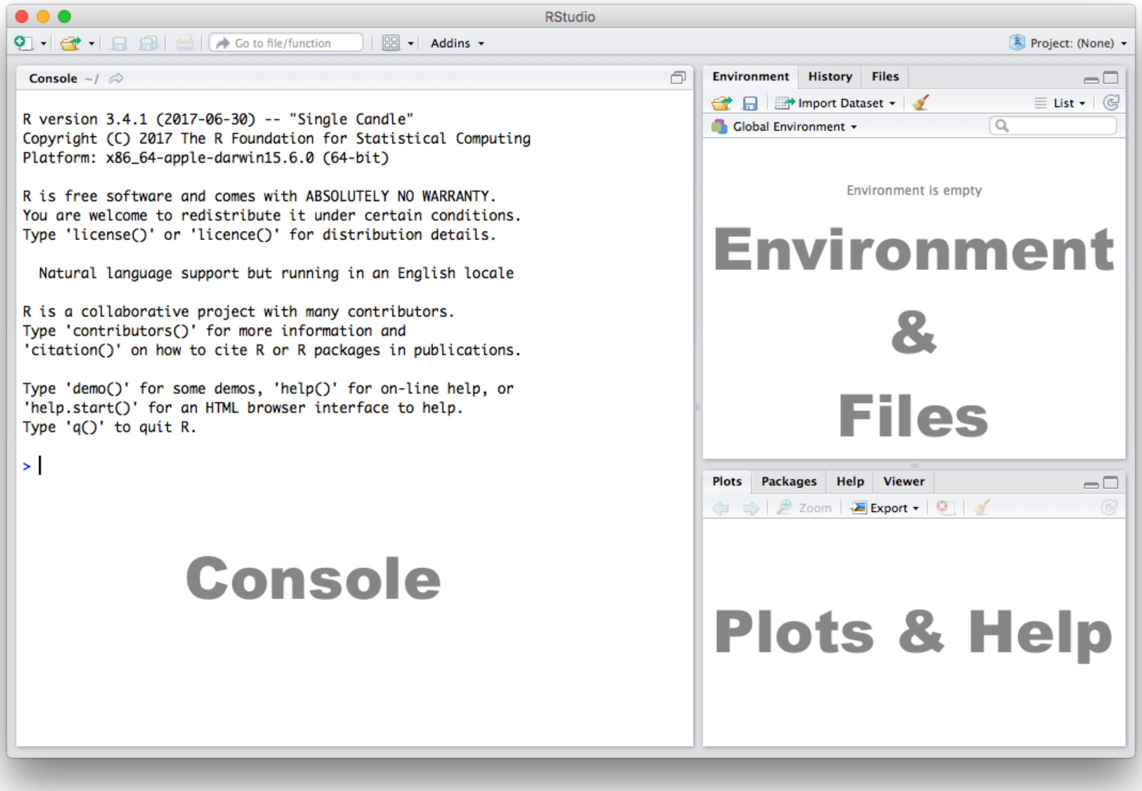
To install R and RStudio on your computer, download both from the following sources:

- Download R from The Comprehensive R Archive Network (CRAN)
- Download RStudio from RStudio.com

Keep both R and RStudio up to date. That means go online and check for newer versions. In case there are new versions, download those and re-install.

1.1.2 RStudio

Let's get acquainted with R. When you start RStudio for the first time, you'll see three panes:



```
R version 3.4.1 (2017-06-30) -- "Single Candle"
Copyright (C) 2017 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin15.6.0 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> |
```

1.1.3 Console

The Console in RStudio is the simplest way to interact with R. You can type some code at the Console and when you press ENTER, R will run that code. Depending on what you type, you may see some output in the Console or if you make a mistake, you may get a warning or an error message.

Let's familiarize ourselves with the console by using R as a simple calculator:

```
2 + 4
```

```
[1] 6
```

Now that we know how to use the + sign for addition, let's try some other mathematical operations such as subtraction (-), multiplication (*), and division (/).

```
10 - 4
```

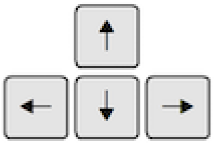
```
[1] 6
```

```
5 * 3
```

```
[1] 15
```

```
7 / 2
```

```
[1] 3.5
```



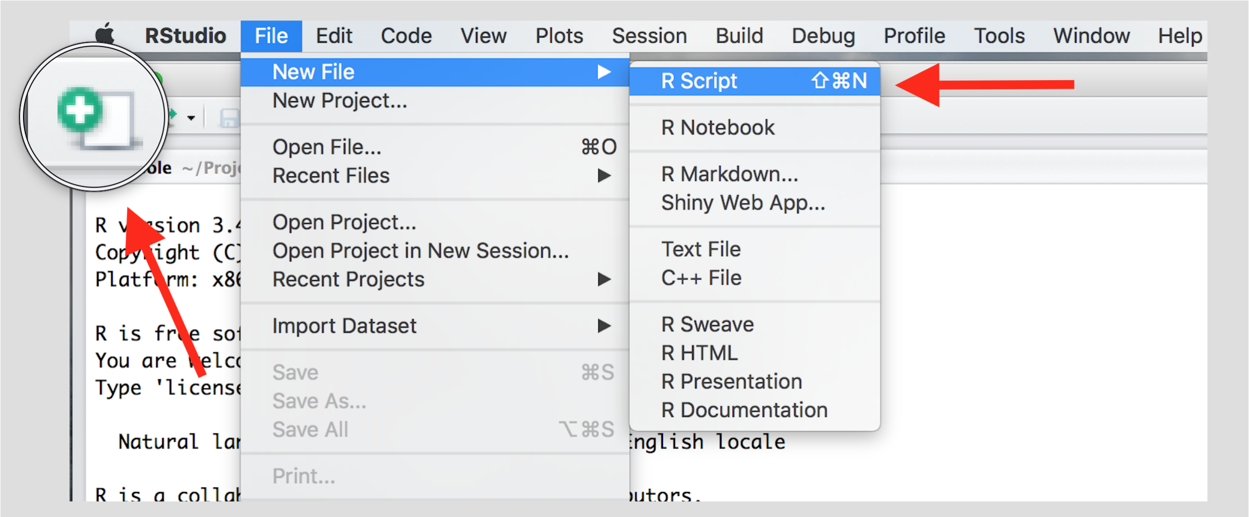
You can use the cursor or arrow keys on your keyboard to edit your code at the console:- Use the UP and DOWN keys to re-run something without typing it again- Use the LEFT and RIGHT keys to edit

Take a few minutes to play around at the console and try different things out. Don't worry if you make a mistake, you can't break anything easily!

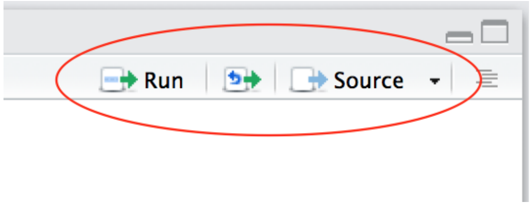
1.1.4 Scripts

The Console is great for simple tasks but if you're working on a project you would mostly likely want to save your work in some sort of a document or a file. Scripts in R are just plain text files that contain R code. You can edit a script just like you would edit a file in any word processing or note-taking application.

Create a new script using the menu or the toolbar button as shown below.



Once you've created a script, it is generally a good idea to give it a meaningful name and save it immediately. For our first session save your script as **seminar1.R**



Familiarize yourself with the script window in RStudio, and especially the two buttons labeled **Run** and **Source**

There are a few different ways to run your code from a script.

One line at a time	Place the cursor on the line you want to run and hit CTRL-ENTER or use the **Run** button
Multiple lines	Select the lines you want to run and hit CTRL-ENTER or use the **Run** button
Entire script	Use the **Source** button

2 R-syntax, data structures and types

2.1 Seminar

In this session we introduce R-syntax, and data types.

2.1.1 Functions

Functions are a set of instructions that carry out a specific task. Functions often require some input and generate some output. For example, instead of using the `+` operator for addition, we can use the `sum` function to add two or more numbers.

```
sum(1, 4, 10)
```

```
[1] 15
```

In the example above, `1, 4, 10` are the inputs and 15 is the output. A function always requires the use of parenthesis or round brackets `()`. Inputs to the function are called **arguments** and go inside the brackets. The output of a function is displayed on the screen but we can also have the option of saving the result of the output. More on this later.

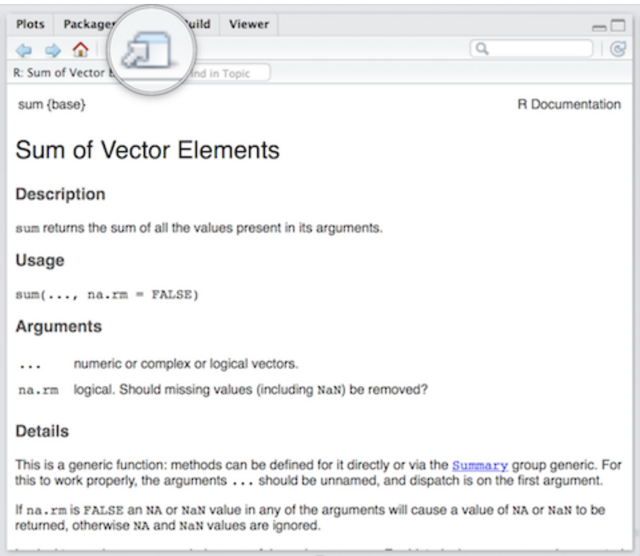
2.1.2 Getting Help

Another useful function in R is `help` which we can use to display online documentation. For example, if we wanted to know how to use the `sum` function, we could type `help(sum)` and look at the online documentation.

```
help(sum)
```

The question mark `?` can also be used as a shortcut to access online help.

```
?sum
```



sum {base}   R Documentation

Sum of Vector Elements

Description

`sum` returns the sum of all the values present in its arguments.

Usage

```
sum(..., na.rm = FALSE)
```

Arguments

`...`	numeric or complex or logical vectors.
`na.rm`	logical. Should missing values (including `NaN`) be removed?

Details

This is a generic function: methods can be defined for it directly or via the Summary group generic. For this to work properly, the arguments `...` should be unnamed, and dispatch is on the first argument.

If `na.rm` is `FALSE` an `NA` or `NaN` value in any of the arguments will cause a value of `NA` or `NaN` to be returned, otherwise `NA` and `NaN` values are ignored.

Use the toolbar button shown in the picture above to expand and display the help in a new window.

Help pages for functions in R follow a consistent layout generally include these sections:

Description	A brief description of the function
Usage	The complete syntax or grammar including all arguments (inputs)
Arguments	Explanation of each argument
Details	Any relevant details about the function and its arguments
Value	The output value of the function
Examples	Example of how to use the function

2.1.3 The Assignment Operator

Now we know how to provide inputs to a function using parenthesis or round brackets `()`, but what about the output of a function?

We use the assignment operator `<-` for creating or updating objects. If we wanted to save the result of adding `sum(1, 4, 10)`, we would do the following:

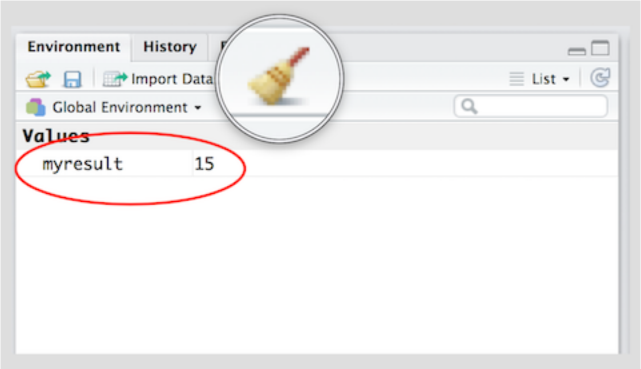
```
myresult <- sum(1, 4, 10)
```

The line above creates a new object called `myresult` in our environment and saves the result of the `sum(1, 4, 10)` in it. To see what's in `myresult`, just type it at the console:

```
myresult
```

```
[1] 15
```

Take a look at the **Environment** pane in RStudio and you'll see `myresult` there.



```
Values
myresult     15
```

To delete all objects from the environment, you can use the **broom** button as shown in the picture above.

We called our object `myresult` but we can call it anything as long as we follow a few simple rules. Object names can contain upper or lower case letters (`A-Z`, `a-z`), numbers (`0-9`), underscores (`_`) or a dot (`.`) but all object names must start with a letter. Choose names that are descriptive and easy to type.

Good Object Names	Bad Object Names
result	a
myresult	x1
my.result	this.name.is.just.too.long
my_result	
data1	

2.1.4 Vectors and subsetting

A vector is one dimensional. It can contain one element in which case it is also called a scalar or many elements. We can add and multiply vectors. Think of a vector as a row or column in your excel spreadsheet.

To create a vector, we use the `c()` function, where c stands for collect. We start by creating a numeric vector.

```r
vec1 <- c(10, 47, 99, 34, 21)
```

Creating a character vector works in the same way. We need to use quotation marks to indicate that the data type is textual data.

```r
vec2 <- c("Emilia", "Martin", "Agatha", "James", "Luke", "Jacques")
```

Let's see how many elements our vector contains using the `length()` function.

```r
length(vec1)
```

```
[1] 5
```

```r
length(vec2)
```

```
[1] 6
```

We need one coordinate to identify a unique element in a vector. For instance, we may be interested in the first element of the vector only. We use square brackets `[]` to access a specific element. The number in square brackets is the vector element that we wish to see.

```r
vec1[1]
```

```
[1] 10
```

To access all elements except the first element, we use the `-` operator

```
vec1[-1]
```

```
[1] 47 99 34 21
```

We can access elements 2 to 4 by using the colon `:` operator.

```
vec1[2:4]
```

```
[1] 47 99 34
```

We can access non-adjacent elements bu using the collect function `c()`.

```
vec1[c(2,5)]
```

```
[1] 47 21
```

Finally, we combine the `length()` function with the square brackets to access the last element in our vector.

```
vec1[ length(vec1) ]
```

```
[1] 21
```

2.1.5 Matrices

A matrix has two dimensions and stores data of the same type, e.g. numbers or text but never both. A matrix is always rectangular. Think of it as your excel spreadsheet - essentially, it is a data table.

We create a matrix using the `matrix()` function. We need to provide the following arguments:

```
mat1 <- matrix(
  data = c(99, 17, 19, 49, 88, 54),
  nrow = 2,
  ncol = 3,
  byrow = TRUE
)
```

Argument	Description
data	the data in the matrix
nrow	number of rows
ncol	number of columns
byrow	TRUE = matrix is filled rowwise

To display the matrix, we simply call the object by its name (in this case mat1).

```
mat1
```

```
     [,1] [,2] [,3]
[1,]   99   17   19
[2,]   49   88   54
```

To access a unique element in a matrix, we need 2 coordinates. First, a row coordinate and second, a column coordinate. We use square brackets and separate the coordinates with a comma `[ , ]`. The row coordinate goes before the comma and the column coordinate after.

We can access the the second row and third column like so:

```
mat1[2, 3]
```

```
[1] 54
```

To display an entire column, we specify the column we want to display and leave the row coordinate empty like so:

```
# display the 2nd column
mat1[ , 2]
```

```
[1] 17 88
```

Similarly, to display the entire second row, we specify the row coordinate but leave the column coordinate empty.

```
mat1[2, ]
```

```
[1] 49 88 54
```

2.1.6 Arrays

Arrays are similar to matrices but can contain more dimensions. You can think of an array as stacking multiple matrices. Generally, we refer to the rows, columns and layers in array. Let's create an array with 2 rows, 3 columns and 4 layers using the `array()` function.

```
arr1 <- array(
  data = c(1:24),
  dim = c(2, 3, 4)
)
```

To display the object, we call it by its name.

```
arr1
```

```
, , 1

     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

, , 2

     [,1] [,2] [,3]
[1,]    7    9   11
[2,]    8   10   12

, , 3

     [,1] [,2] [,3]
[1,]   13   15   17
[2,]   14   16   18

, , 4

     [,1] [,2] [,3]
[1,]   19   21   23
[2,]   20   22   24
```

We can subset an array using the square brackets `[]`. To access a single element we need as many coordinates as our object has dimensions. Let's check the number of dimensions in our object first.

```
dim(arr1)
```

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

The `dim()` function informs us that we have 3 dimensions. The first is of length 2, the second of length 3 and the fourth of length 4.

Access the second column of the third layer on your own.

```
arr1[ , 2, 3]
```

```
[1] 15 16
```

3 Data import (from csv, txt, and excel)

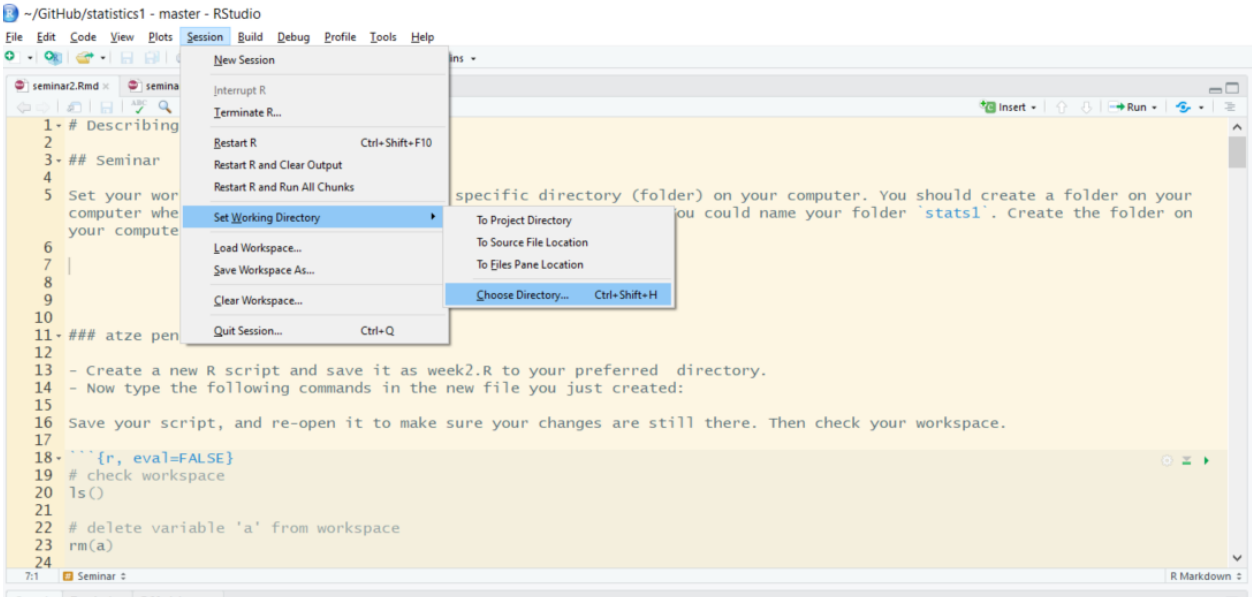
3.1 Seminar

In this section, we will load data in csv, txt, excel and R format. We will learn how to check and set our working directory.

3.1.1 Setting up

We set our working directory. R operates in specific directory (folder) on our computer. We create a folder on where we save our scripts. We name the folder `suffolk2019`. Let's create the folder on our computers now (in finder on Mac and explorer on Windows).

Now, we set our working directory to the folder, we just created like so:



~~~
# Describing

## Seminar

Set your wor                                         specific directory (folder) on your computer. You should create a folder on your
computer whe                                                                  ou could name your folder `stats1`. Create the folder on
your compute

### atze pen

- Create a new R script and save it as week2.R to your preferred  directory.
- Now type the following commands in the new file you just created:

Save your script, and re-open it to make sure your changes are still there. Then check your workspace.

```{r, eval=FALSE}
# check workspace
ls()

# delete variable 'a' from workspace
rm(a)
~~~

Create a new R script and save it as day1.R to your `suffolk2019` directory.

At the beginning of each new script, we want to clear the workspace. The workspace is stored in working memory on our computer. If we do not clear it for a new script, it becomes too full over time. Our computer will slow down and it will become difficult for us to know which objects are stored in working memory.

We check the contents of our workspace like so:

```
# check workspace
ls()
```

To remove a specific object, we use the `rm()` function which stands for remove. Within the round brackets, we put the name of the object we want to remove. We could remove the object `a` like so:

```r
# delete variable 'a' from workspace
rm(a)
```

At the beginning of each script, we should always clear the entire workspace. We can do so in the following way:

```r
# delete everything from workspace
rm( list = ls() )
```

You can also clear text from the console window. To do so press Crtl+l on Winowds or Command+l on Mac.

3.1.2 Loading data

Data comes in different file formats such as `.txt`, `.csv`, `.xlsx`, `.RData`, `.dta` and many more. To know the file type of a file right click on it and view preferences (in Windows explorer or Mac finder).

R can load files coming in many different file formats. To find out how to import a file coming in a specific format, it is usually a good idea to the google "R load file_format".

3.1.3 Importing a dataset in `.csv` format

One of the most common file types is `.csv` which means comma separated values. Columns are separated by commas and rows by line breaks.

The dataset's name is "non_western_immigrants.csv". To load it, we use the `read.csv()` function.

```r
dat1 <- read.csv("non_western_immigrants.csv")
```

3.1.4 Importing a dataset in Excel (xlsx) format

Another common file format is Microsoft's Excel `xlsx` format. We will load a dataset in this format now. To do so, we will need to install a package first. Packages are additional functions that we can add to R. A package is like an app on our phones.

We install the `readxl` package using `install.packages("readxl")`.

```r
install.packages("readxl")
```

We only need to install a package once. It does not hurt to do it more often though, because every time we install, it will install the most recent version of the package.

Once a package is installed, we need to load it using the `library()` function.

```r
library(readxl)
```

To load the excel file, we can now use the `read_excel()` function that is included in the `readxl` library. We need to provide the following arguments to the function:

Argument	Description
path	Filename of excel sheet
sheet	Sheet number to import

Now, let's load the file:

```r
dat2 <- read_excel("non_western_immigrants.xlsx", sheet = 1)
```

3.1.5 Importing a dataset in RData format

The native file format of R is called `.RData`. To load files saved in this format, we use the `load()` function like so:

```
load(file = "non_western_immigrants.RData")
```

Notice that we usually need to assign the object we load to using the `<-` operator. The `load()` function is an exception where we do not need to do this.

3.1.6 Importing a dataset in .txt format.

Loading a dataset that comes in .txt format requires some additional information. The format is a text format and we need to know how the columns are separated. Usually it is enough to open the file in a word processor such as notepad to see how this is done. The most common ways to separate columns is by using commas or tabs but other separators such as for instance semicolons are sometimes also used.

In our example, columns are separated by semicolons. We use the `read.table()` function and provide the following arguments:

Argument	Description
file	Filename of excel sheet
sep	the symbol that separates columns
header	whether the first row contains variable names or not

```
dat3 <- read.table(file = "non_western_immigrants.txt", sep =";", header = TRUE)
```

4 R-syntax, data structures and types

4.1 Seminar

4.1.1 Creating data frames

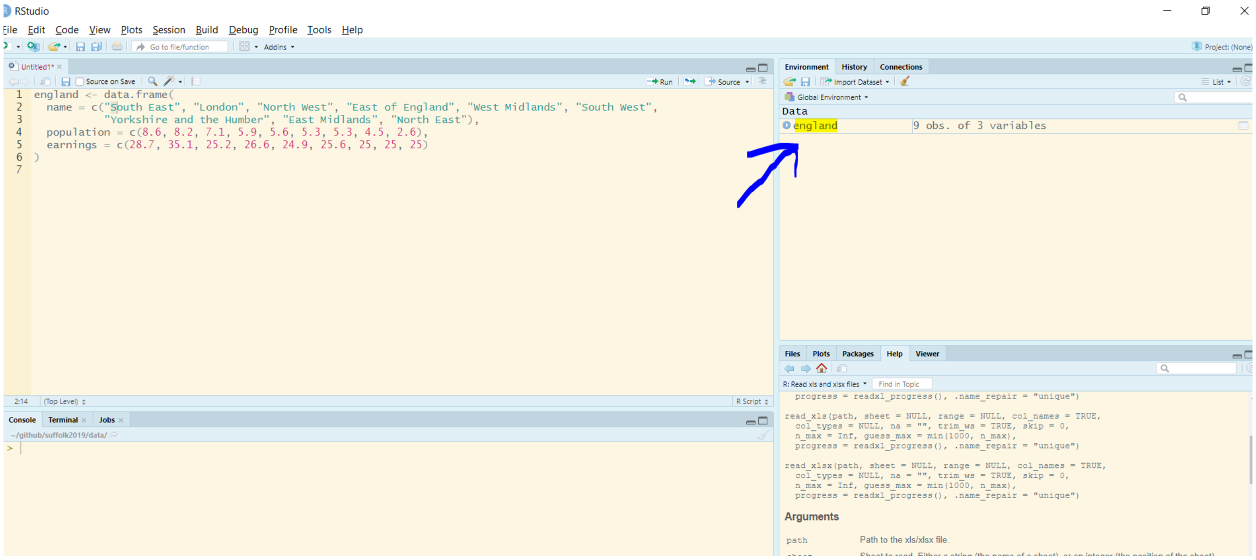
A data frame is an object that holds data in a tabular format similar to how spreadsheets work. Variables are generally kept in columns and observations are in rows. Data frames are similar to matrices but they can store vectors of different types (e.g. numbers and text).

We start by creating a data frame with the `data.frame()` function. We will give each column a name (a variable name) followed by the `=` operator and the respective vector of data that we want to assign to that column.

```
england <- data.frame(
  name = c("South East", "London", "North West", "East of England", "West Midlands", "South West",
           "Yorkshire and the Humber", "East Midlands", "North East"),
  population = c(8.6, 8.2, 7.1, 5.9, 5.6, 5.3, 5.3, 4.5, 2.6),
  earnings = c(28.7, 35.1, 25.2, 26.6, 24.9, 25.6, 25, 25, 25)
)
```

4.1.2 Working with data frames

we can display the entire dataset in spreadsheet view by clicking on the object name in the environment window.


RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function
Addins
Project: (None)
Untitled1*
Source on Save
Run
Source
Environment History Connections
Import Dataset
List
Global Environment
Console Terminal Jobs
~/github/suffolk2019/data/
Files Plots Packages Help Viewer
R: Read xls and xlsx files
Find in Topic
2:14
(Top Level)
R Script

```
england <- data.frame(
  name = c("South East", "London", "North West", "East of England", "West Midlands", "South West",
           "Yorkshire and the Humber", "East Midlands", "North East"),
  population = c(8.6, 8.2, 7.1, 5.9, 5.6, 5.3, 5.3, 4.5, 2.6),
  earnings = c(28.7, 35.1, 25.2, 26.6, 24.9, 25.6, 25, 25, 25)
)
```

Data

england 9 obs. of 3 variables

```
  progress = readxl_progress(), .name_repair = "unique")

read_xls(path, sheet = NULL, range = NULL, col_names = TRUE,
  col_types = NULL, na = "", trim_ws = TRUE, skip = 0,
  n_max = Inf, guess_max = min(1000, n_max),
  progress = readxl_progress(), .name_repair = "unique")

read_xlsx(path, sheet = NULL, range = NULL, col_names = TRUE,
  col_types = NULL, na = "", trim_ws = TRUE, skip = 0,
  n_max = Inf, guess_max = min(1000, n_max),
  progress = readxl_progress(), .name_repair = "unique")
```

Arguments

path Path to the xls/xlsx file.

Alternatively, you can call the object name to display the dataset in the console window. Let's do so:

```
england
```

```
                      name population earnings
1               South East        8.6     28.7
2                   London        8.2     35.1
3               North West        7.1     25.2
4          East of England        5.9     26.6
5            West Midlands        5.6     24.9
6               South West        5.3     25.6
7 Yorkshire and the Humber        5.3     25.0
8            East Midlands        4.5     25.0
9               North East        2.6     25.0
```

Often, datasets are too long to be viewed to in the console window. It is a good idea to look at the first couple of rows of a datasets to get an overview of its contents. We use the square brackets [] to view the first five rows and all columns.

```
england[1:5, ]
```

```
             name population earnings
1      South East        8.6     28.7
2          London        8.2     35.1
3      North West        7.1     25.2
4 East of England        5.9     26.6
5   West Midlands        5.6     24.9
```

v Columns in a dataframe have names. We will often need to know the name of a column/variable to access it. We use the `names()` function to view all variable names in a dataframe.

```
names(england)
```

```
[1] "name"       "population" "earnings"
```

We can access the earnings variable in multiple ways. First, we can use the $ operator. We write the name of the dataset object, followed by the $, followed by the variable name like so:

```
england$earnings
```

```
[1] 28.7 35.1 25.2 26.6 24.9 25.6 25.0 25.0 25.0
```

We can also use the square brackets to access the earnings column.

```
england[, "earnings" ]
```

```
[1] 28.7 35.1 25.2 26.6 24.9 25.6 25.0 25.0 25.0
```

The square brackets are sometimes preferred because we could access multiple columns at once like so:

```
england[, c("name", "earnings") ]
```

```
                      name earnings
1               South East     28.7
2                   London     35.1
3               North West     25.2
4          East of England     26.6
5            West Midlands     24.9
6               South West     25.6
7 Yorkshire and the Humber     25.0
8            East Midlands     25.0
9               North East     25.0
```

Variables come in different types such as numbers, text, logical (true/false). We need to know the type of a variable because the type affects statistical analysis. We use the `str()` function to check the type of each variable in our dataset.

```
str(england)
```

```
'data.frame':   9 obs. of  3 variables:
 $ name      : Factor w/ 9 levels "East Midlands",..: 6 3 5 2 8 7 9 1 4
 $ population: num  8.6 8.2 7.1 5.9 5.6 5.3 5.3 4.5 2.6
 $ earnings  : num  28.7 35.1 25.2 26.6 24.9 25.6 25 25 25
```

The first variable in our dataset is a factor variable. Factors are categorical variables. Categories are mutually exclusive but they do not imply an ordering. For instance, “East of England” is not more or less than “West Midlands”. The variables population and earnings are both numeric variables.

```
str(england)
```

```
'data.frame':   9 obs. of  3 variables:
 $ name      : Factor w/ 9 levels "East Midlands",..: 6 3 5 2 8 7 9 1 4
 $ population: num  8.6 8.2 7.1 5.9 5.6 5.3 5.3 4.5 2.6
 $ earnings  : num  28.7 35.1 25.2 26.6 24.9 25.6 25 25 25
```

4.1.3 Amending data frames

Amending data sets usually involves adding rows or columns or removing rows or columns. We start by adding a new variable to our dataset which contains the percent of the population on income support.

To create a new variable we simply assign a new vector to the dataframe object like so:

```
england$pct_on_support <- c(3, 5.3, 5.3, 3.5, 5.1, 3.3, 5.2, 4.2, 6.1)
```

We call the dataframe object to view our changes.

```
england
```

```
                      name population earnings pct_on_support
1               South East        8.6     28.7            3.0
2                   London        8.2     35.1            5.3
```

```
3               North West        7.1     25.2            5.3
4          East of England        5.9     26.6            3.5
5            West Midlands        5.6     24.9            5.1
6               South West        5.3     25.6            3.3
7 Yorkshire and the Humber        5.3     25.0            5.2
8            East Midlands        4.5     25.0            4.2
9               North East        2.6     25.0            6.1
```

We can delete the variable we just created by assigning `NULL` to it.

```
england$pct_on_support <- NULL
```

Let's view our most recent changes.

```
england
```

```
                      name population earnings
1               South East        8.6     28.7
2                   London        8.2     35.1
3               North West        7.1     25.2
4          East of England        5.9     26.6
5            West Midlands        5.6     24.9
6               South West        5.3     25.6
7 Yorkshire and the Humber        5.3     25.0
8            East Midlands        4.5     25.0
9               North East        2.6     25.0
```

Adding a new row to a dataset means adding an observation. Let's add Brittany to our dataset. We need to fill in a value for each variable. If, we do not know a value, we declare it as missing. Missings are `NA` for numeric variables and `""` for character variables. We use the `rbind()` function (row bind) to add a row to our dataset.

```
england <- rbind(england,  c("Brittany", 4.5, NA) )
```

```
Warning in `[<-.factor`(`*tmp*`, ri, value = "Brittany"): invalid factor
level, NA generated
```

Let's examine our dataframe.

```
england
```

```
                       name population earnings
1                South East        8.6     28.7
2                    London        8.2     35.1
3                North West        7.1     25.2
4           East of England        5.9     26.6
5             West Midlands        5.6     24.9
6                South West        5.3     25.6
7  Yorkshire and the Humber        5.3       25
8             East Midlands        4.5       25
9                North East        2.6       25
10                     <NA>        4.5     <NA>
```

We were not allowed to enter "Brittany" as a value for the name variable. This is because the variable is a factor. While it is possible to add new levels to a factor (categories such as "Midlands" are called levels), it involves a bit more advanced programming. We will solve this problem later in the type coercion section.

4.1.4 Saving data frames

Datasets can be exported in many different file formats. We recommend exporting files as ".csv" files because csv is a very common file type. Such files can be handled by all statistical packages including Microsoft's Excel. We need to provide five arguments.

Argument	Description
x	The name of the object
file	The file name
sep	The symbol that separates columns
col.names	= TRUE saves the variable names (recommended)
row.names	= FALSE omits the row names (recommended)

```
write.table(x = england, file = "england.csv", sep = ",", col.names = TRUE, row.names = FALSE)
```

5 Type coercion

5.1 Seminar

We often need to change the type of a variable. This can be necessary to clean data or because we add a new level to a factor like in the previous section on amending data frames. We begin by loading the england dataset that we created previously.

```
england <- read.csv(file = "england.csv", sep = ",", header = TRUE)
```

Recall that we could not add "Brittany" as a name to our variable. This was because the variable name is stored as a factor and "Brittany" was a new category. The easiest way to add the name "Brittany" is to convert the name variable into a character variable.

5.1.1 Coerce a factor to character

Let's check our england dataset's variable types. .

```
str(england)
```

```
'data.frame':	10 obs. of  3 variables:
 $ name      : Factor w/ 9 levels "East Midlands",..: 6 3 5 2 8 7 9 1 4 NA
 $ population: num  8.6 8.2 7.1 5.9 5.6 5.3 5.3 4.5 2.6 4.5
 $ earnings  : num  28.7 35.1 25.2 26.6 24.9 25.6 25 25 25 NA
```

The variable "name" is a factor variable and needs to be converted to a character variable. We coerce the variable into a different type using the `as.character()` function.

```
england$name <- as.character(england$name)
```

Let's inspect the variable types of our dataset again:

```
str(england)
```

```
'data.frame':	10 obs. of  3 variables:
 $ name      : chr  "South East" "London" "North West" "East of England" ...
 $ population: num  8.6 8.2 7.1 5.9 5.6 5.3 5.3 4.5 2.6 4.5
 $ earnings  : num  28.7 35.1 25.2 26.6 24.9 25.6 25 25 25 NA
```

The variable name is now a character variable. We now change the last value of the name variable into Brittany.

```
england$name[ length(england$name) ] <- "Brittany"
```

Let's inspect our change.

```
england
```

```
                        name population earnings
1                 South East        8.6     28.7
2                     London        8.2     35.1
3                 North West        7.1     25.2
4            East of England        5.9     26.6
5              West Midlands        5.6     24.9
6                 South West        5.3     25.6
7   Yorkshire and the Humber        5.3     25.0
8              East Midlands        4.5     25.0
9                 North East        2.6     25.0
10                  Brittany        4.5       NA
```

5.1.2 Coerce a character variable into a factor variable

We can now easily convert the variable type back from character into a factor with the `as.factor()` function like so:

```
england$name <- as.factor(england$name)
str(england)
```

```
'data.frame':   10 obs. of  3 variables:
 $ name      : Factor w/ 10 levels "Brittany","East Midlands",..: 7 4 6 3 9 8 10 2 5 1
 $ population: num  8.6 8.2 7.1 5.9 5.6 5.3 5.3 4.5 2.6 4.5
 $ earnings  : num  28.7 35.1 25.2 26.6 24.9 25.6 25 25 25 NA
```

5.1.3 Coerce a character variable into a numeric variable

In R missing vlues are called `NA` for numeric types and `""` for character types. When we use load third party data, the coding often differs. This can happen, for instance, due to data entry errors.

We simulate a data entry error by changing the population value of Brittany to a character value.

```
england$population[ length(england$population) ] <- "mistake"
str(england)
```

```
'data.frame':   10 obs. of  3 variables:
 $ name      : Factor w/ 10 levels "Brittany","East Midlands",..: 7 4 6 3 9 8 10 2 5 1
 $ population: chr  "8.6" "8.2" "7.1" "5.9" ...
 $ earnings  : num  28.7 35.1 25.2 26.6 24.9 25.6 25 25 25 NA
```

Notice, that the variable population is now a character vector instead of a numeric vector. Whenever numbers and text are mixed, R will automatically treat the vector as a character vector.

We can convert the population variable back into a numeric variable using the `as.numeric()` function. All values that are not recognised as numbers will be changed to `NA`.

```
england$population <- as.numeric(england$population)
```

```
Warning: NAs introduced by coercion
```

```
str(england)
```

```
'data.frame':   10 obs. of  3 variables:
 $ name      : Factor w/ 10 levels "Brittany","East Midlands",..: 7 4 6 3 9 8 10 2 5 1
 $ population: num  8.6 8.2 7.1 5.9 5.6 5.3 5.3 4.5 2.6 NA
 $ earnings  : num  28.7 35.1 25.2 26.6 24.9 25.6 25 25 25 NA
```

```r
england
```

```
                       name population earnings
1                South East        8.6     28.7
2                    London        8.2     35.1
3                North West        7.1     25.2
4           East of England        5.9     26.6
5             West Midlands        5.6     24.9
6                South West        5.3     25.6
7  Yorkshire and the Humber        5.3     25.0
8             East Midlands        4.5     25.0
9                North East        2.6     25.0
10                 Brittany         NA       NA
```

6 Loops and conditions

6.1 Seminar

In this section, we introduce loops and conditional statements. Loops are generally useful, when we want to carry out the same operation over and over. Conditions are logical statements that are evaluated and if the statement is true a different operation is carried out than if the statement is false. We may, for instance, be interested in the average crime rate in our sample but only if the respondents are female. To do so, we need conditional statements.

6.1.1 For loops

Loops are useful when we need to carry out similar operations repeatedly. A for loop is an easy way to do this. We will create a simple “for loop” like so:

```r
for (idx in 1:7){
  print( idx )
}
```

```
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
```

In the above code, the `for()` function initiates the loop. We loop from 1 through 7 and `idx` takes the values from 1 to 7 iteratively. We can write code within the curly braces. Here, we simply print the current loop iteration.

We can create nested loops, i.e. loops within loops like so:

```r
# first loop
for (idx in 1:7){

  # second loop
  for (idx2 in 1: 7){

    if (idx < idx2) print("idx1 is smaller than idx2")
    else print("idx1 is larger than idx2")

  } # end of second loop
} # end of first loop
```

```
[1] "idx1 is larger than idx2"
[1] "idx1 is smaller than idx2"
[1] "idx1 is smaller than idx2"
[1] "idx1 is smaller than idx2"
[1] "idx1 is smaller than idx2"
[1] "idx1 is smaller than idx2"
[1] "idx1 is smaller than idx2"
[1] "idx1 is larger than idx2"
[1] "idx1 is larger than idx2"
[1] "idx1 is smaller than idx2"
[1] "idx1 is smaller than idx2"
[1] "idx1 is smaller than idx2"
[1] "idx1 is smaller than idx2"
[1] "idx1 is smaller than idx2"
[1] "idx1 is larger than idx2"
[1] "idx1 is larger than idx2"
[1] "idx1 is larger than idx2"
[1] "idx1 is smaller than idx2"
[1] "idx1 is smaller than idx2"
[1] "idx1 is smaller than idx2"
[1] "idx1 is smaller than idx2"
[1] "idx1 is larger than idx2"
[1] "idx1 is larger than idx2"
[1] "idx1 is larger than idx2"
[1] "idx1 is larger than idx2"
[1] "idx1 is smaller than idx2"
[1] "idx1 is smaller than idx2"
[1] "idx1 is smaller than idx2"
[1] "idx1 is larger than idx2"
[1] "idx1 is larger than idx2"
[1] "idx1 is larger than idx2"
[1] "idx1 is larger than idx2"
[1] "idx1 is larger than idx2"
[1] "idx1 is smaller than idx2"
[1] "idx1 is smaller than idx2"
[1] "idx1 is larger than idx2"
[1] "idx1 is larger than idx2"
[1] "idx1 is larger than idx2"
[1] "idx1 is larger than idx2"
[1] "idx1 is larger than idx2"
[1] "idx1 is larger than idx2"
[1] "idx1 is smaller than idx2"
[1] "idx1 is larger than idx2"
[1] "idx1 is larger than idx2"
[1] "idx1 is larger than idx2"
[1] "idx1 is larger than idx2"
[1] "idx1 is larger than idx2"
[1] "idx1 is larger than idx2"
[1] "idx1 is larger than idx2"
```

While this is a toy example, it illustrates how we could do pairwise comparisons between observations in our dataset. We also added an if/else condition using the functions `if()` and `else`. If the if statement is TRUE the first statement is printed and if not the second statement is printed.

6.1.2 Conditions

To illustrate the use of conditions we load the non-western foreigners dataset.

```
dat1 <- read.csv("non_western_immigrants.csv")
```

Variable Name	Description
IMMBRIT	Out of every 100 people in Britain, how many do you think are immigrants from Non-western countries?
over.estimate	1 if estimate is higher than 10.7%.
RSex	1 = male, 2 = female
RAge	Age of respondent
Househld	Number of people living in respondent's household
Cons, Lab, SNP, Ukip, BNP, GP, party.other	Party self-identification
paper	Do you normally read any daily morning newspaper 3+ times/week?
WWWhourspW	How many hours WWW per week?
religious	Do you regard yourself as belonging to any particular religion?
employMonths	How many mnths w. present employer?
urban	Population density, 4 categories (highest density is 4, lowest is 1)
health.good	How is your health in general for someone of your age? (0: bad, 1: fair, 2: fairly good, 3: good)
HHInc	Income bands for household, high number = high HH income

The variable `over.estimate` is equal to 1 if respondents over estimate the number of non-western immigrants in Britain. We will evaluate whether women are more or less likely to over estimate. First, we use the `mean()` function to assess the overall average value.

```
mean(dat1$over.estimate)
```

```
[1] 0.7235462
```

A mean of 0.72 indicates that 72% of the 1049 respondents in the dataset over estimate the number of non-western immigrants. To assess whether the number is larger among men than women, we need conditional statements.

We first take the mean of "over.estimate" for men:

```
mean( dat1$over.estimate[ dat1$RSex==1 ] )
```

```
[1] 0.6527197
```

Here, we used square brackets to subset the data. The subset that we evaluate is described by the logical statement `dat1$RSex==1`. The `==` operator is a logical equal that is true if a condition is fulfilled and false otherwise. In this case, it is true if the variable "RSex" is 1 which stands for men.

Take the mean of over.estimate for women on your own.

```
mean( dat1$over.estimate[ dat1$RSex== 2] )
```

```
[1] 0.7828371
```

It turns out, that females in our sample over estimate the number of non-western immigrants more. Whether the difference in our sample is systematic, i.e. whether it would hold in the population as well is a matter that we will return to.

Here we have taken two conditional means and compared them. Doing so is the first step towards statistical inference.

6.1.3 The ifelse() function

Categorical variables such as "RSex" are usually coded 0/1 and the variable name usually refers to the category that is 1. "RSex" is a bad variable name because it is not clear whehter the values 1 and 2 refer to males or females.

We will create a new variable called "female" that is equal to 1 if the respondent is female and 0 otherwise. We do so using the `ifelse()` function. The function first evaluates a logical condition and subsequently carries out one operation if the statement is true (yes) and another if the statement is false (no).

```
dat1$female <- ifelse( dat1$RSex == 2, yes = 1, no = 0 )
```

Let's check whetehr we correctly converted the variable using the `table()` function which produces a frequency table.

```
table(dat1$RSex)
```

```

  1   2 
478 571 
```

```
table(dat1$female)
```

```

  0   1 
478 571 
```

7 Visualising data

7.1 Seminar

In this section, we will learn how to visualise data which is an important step towards understanding relationships better.

The non-western foreingers data is about the subjective perception of immigrants from non-western countries. The perception of immigrants from a context that is not similar to the one's own ,is often used as a proxy for racism. Whether this is a fair measure or not is debatable but let's examine the data from a survey carried out in Britain.

Let's check the codebook of our data.

Variable	Description
IMMBRIT	Out of every 100 people in Britain, how many do you think are immigrants from non-western countr
over.estimate	1 if estimate is higher than 10.7%.
RSex	1 = male, 2 = female
RAge	Age of respondent
Househld	Number of people living in respondent's household
party identification	1 = Conservatives, 2 = Labour, 3 = SNP, 4 = Greens, 5 = Ukip, 6 = BNP, 7 = other
paper	Do you normally read any daily morning newspaper 3+ times/week?
WWWhourspW	How many hours WWW per week?
religious	Do you regard yourself as belonging to any particular religion?
employMonths	How many mnths w. present employer?
urban	Population density, 4 categories (highest density is 4, lowest is 1)
health.good	How is your health in general for someone of your age? (0: bad, 1: fair, 2: fairly good, 3: good)
HHInc	Income bands for household, high number = high HH income

Let's load the dataset.

```
dat1 <- read.csv("non_western_immigrants.csv")
```

We can look at the variable names in our data with the `names()` function.

The `dim()` function can be used to find out the dimensions of the dataset (dimension 1 = rows, dimension 2 = columns).

```
dim(dat1)
```

```
[1] 1049   13
```

So, the `dim()` function tells us that we have data from 1049 respondents with 13 variables for each respondent.

Let's take a quick peek at the first 10 observations to see what the dataset looks like. By default the `head()` function returns the first 6 rows, but let's tell it to return the first 10 rows instead.

```
head(dat1, n = 10)
```

```
   IMMBRIT over.estimate RSex RAge Househld paper WWWhourspW religious
1        1             0    1   50        2     0          1         0
2       50             1    2   18        3     0          4         0
3       50             1    2   60        1     0          1         0
4       15             1    2   77        2     1          2         1
5       20             1    2   67        1     0          1         1
6       30             1    1   30        4     1         14         0
7       60             1    2   56        2     0          5         1
8        7             0    1   49        1     1          8         0
9       30             1    1   40        4     0          3         1
10       2             0    1   61        3     1          0         1
   employMonths urban health.good HHInc party_self
1            72     4           1    13          2
2            72     4           2     3          7
3           456     3           3     9          7
4            72     1           3     8          7
5            72     3           3     9          7
6            72     1           2     9          7
7           180     1           2    13          3
8           156     4           2    14          7
9           264     2           2    11          3
10           72     1           3     8          1
```

Finally, let's look at summary statistics of our dataset.

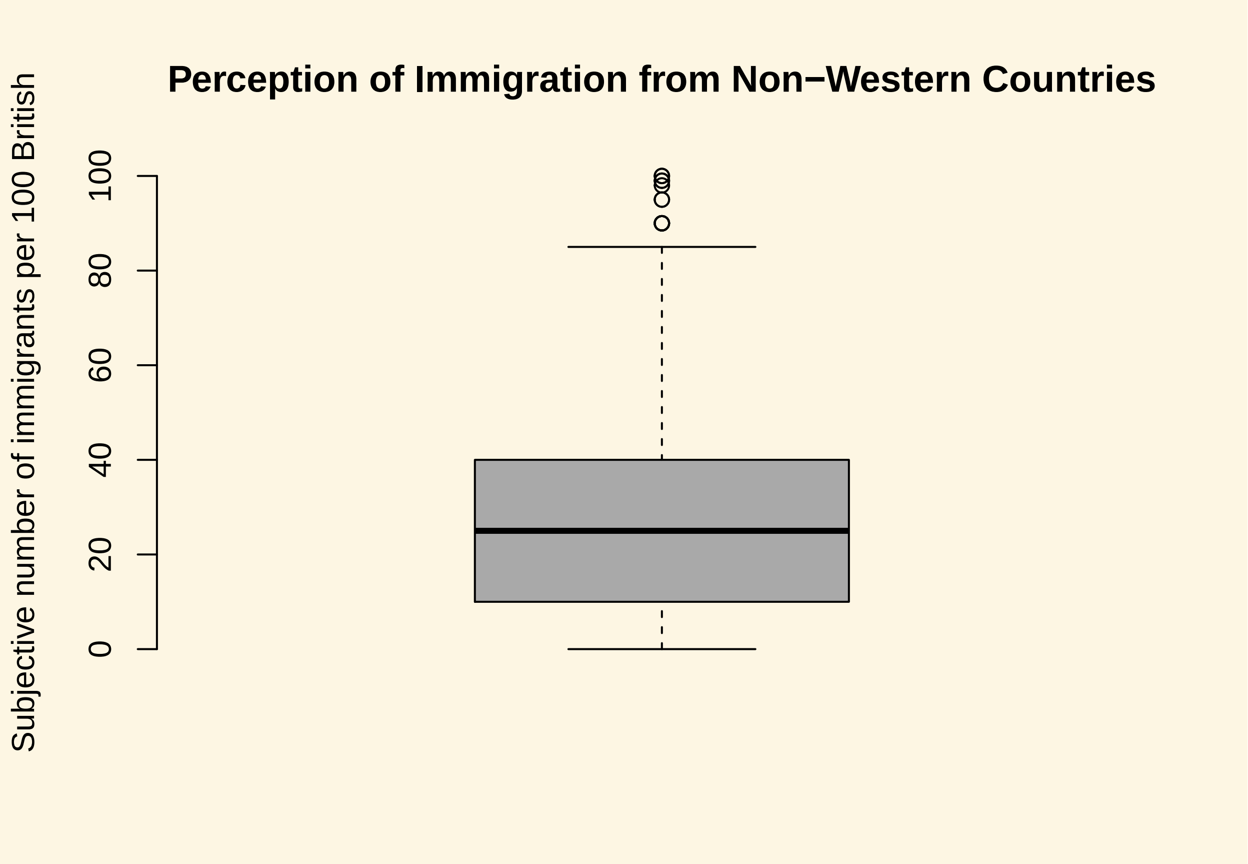
```r
summary(dat1)
```

```
    IMMBRIT       over.estimate         RSex           RAge      
 Min.   :  0.00   Min.   :0.0000   Min.   :1.000   Min.   :17.00  
 1st Qu.: 10.00   1st Qu.:0.0000   1st Qu.:1.000   1st Qu.:36.00  
 Median : 25.00   Median :1.0000   Median :2.000   Median :49.00  
 Mean   : 29.03   Mean   :0.7235   Mean   :1.544   Mean   :49.75  
 3rd Qu.: 40.00   3rd Qu.:1.0000   3rd Qu.:2.000   3rd Qu.:62.00  
 Max.   :100.00   Max.   :1.0000   Max.   :2.000   Max.   :99.00  
    Househld        paper         WWWhourspW       religious     
 Min.   :1.000   Min.   :0.0000   Min.   :  0.000   Min.   :0.0000  
 1st Qu.:1.000   1st Qu.:0.0000   1st Qu.:  0.000   1st Qu.:0.0000  
 Median :2.000   Median :0.0000   Median :  2.000   Median :0.0000  
 Mean   :2.392   Mean   :0.4538   Mean   :  5.251   Mean   :0.4929  
 3rd Qu.:3.000   3rd Qu.:1.0000   3rd Qu.:  7.000   3rd Qu.:1.0000  
 Max.   :8.000   Max.   :1.0000   Max.   :100.000   Max.   :1.0000  
  employMonths       urban        health.good        HHInc       
 Min.   :  1.00   Min.   :1.000   Min.   :0.000   Min.   : 1.000  
 1st Qu.: 72.00   1st Qu.:2.000   1st Qu.:2.000   1st Qu.: 6.000  
 Median : 72.00   Median :3.000   Median :2.000   Median : 9.000  
 Mean   : 86.56   Mean   :2.568   Mean   :2.044   Mean   : 9.586  
 3rd Qu.: 72.00   3rd Qu.:3.000   3rd Qu.:3.000   3rd Qu.:13.000  
 Max.   :600.00   Max.   :4.000   Max.   :3.000   Max.   :17.000  
   party_self   
 Min.   :1.000  
 1st Qu.:1.000  
 Median :2.000  
 Mean   :3.825  
 3rd Qu.:7.000  
 Max.   :7.000  
```

7.1.1 Plots

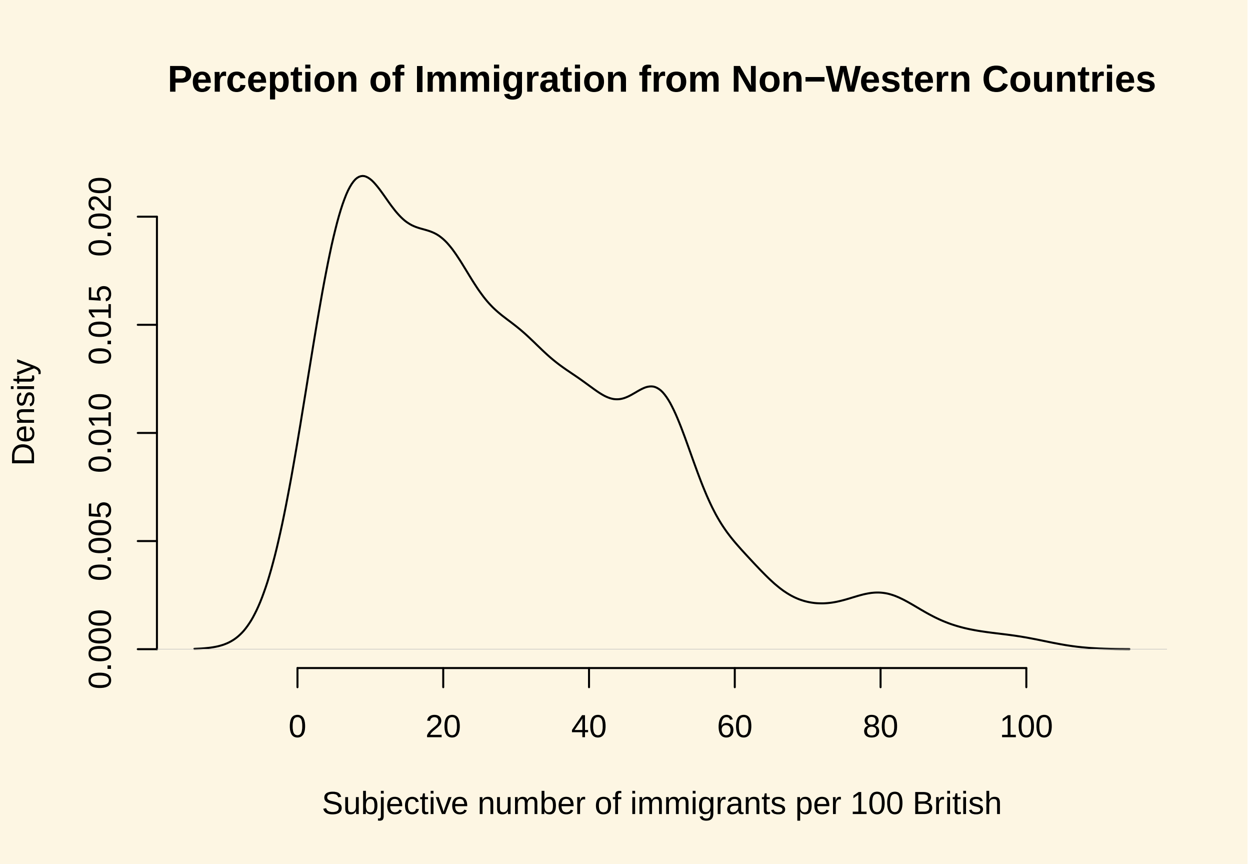
We can visualize the data with the help of a boxplot, so let's see how the perception of the number of immigrants is distributed.

```r
# how good are we at guessing immigration
boxplot(
  dat1$IMMBRIT,
  main = "Perception of Immigration from Non-Western Countries",
  ylab = "Subjective number of immigrants per 100 British",
  frame.plot = FALSE, col = "darkgray"
  )
```



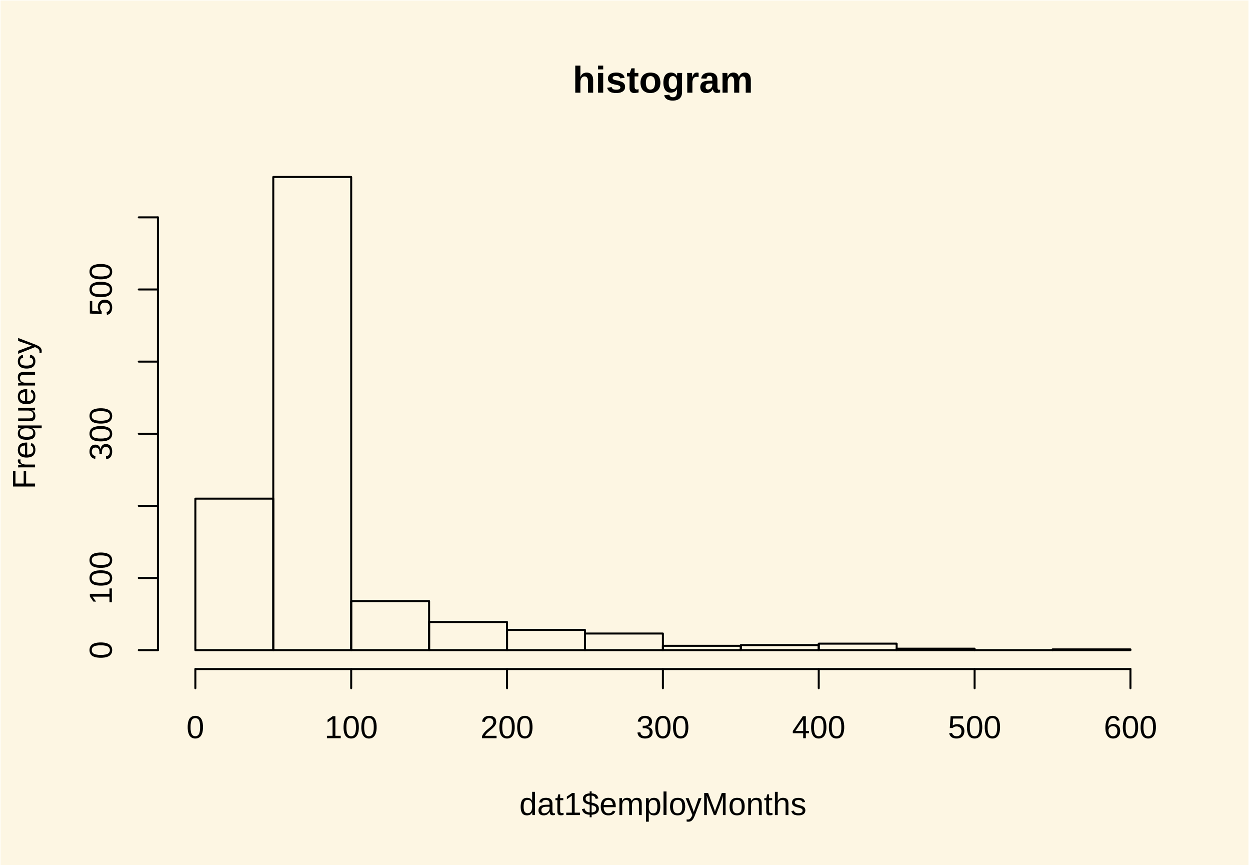
Notice how the lower whisker is much shorter than the upper one. The distribution is right skewed. The right tail (higher values) is a lot longer. We can see this beter using a density plot. We combine R's `denisty()` function with the `plot()` function.

```r
plot(
  density(dat1$IMMBRIT),
  bty = "n",
  main = "Perception of Immigration from Non-Western Countries",
  xlab = "Subjective number of immigrants per 100 British"
  )
```



We can also plot histograms using the `hist()` function.

```r
# histogram
hist( dat1$employMonths, main = "histogram")
```



It is plausible that perception of immigration from Non-Western countries is related to party affiliation. In our dataset, we have a some party affiliation dummies (binary variables). We can use square brackets to subset our data such that we produce a boxplot only for members of the Conservative Party. We first create the binary variables *Cons* and *Lab* for (Conservatives and Labour respectively) using the `ifelse()` function.

```
dat1$Cons <- ifelse(dat1$party_self == 1, yes = 1, no = 0)
dat1$Lab <- ifelse(dat1$party_self == 2, yes = 1, no = 0)
```

We have a look at the variable *Cons* using the `table()` function first.

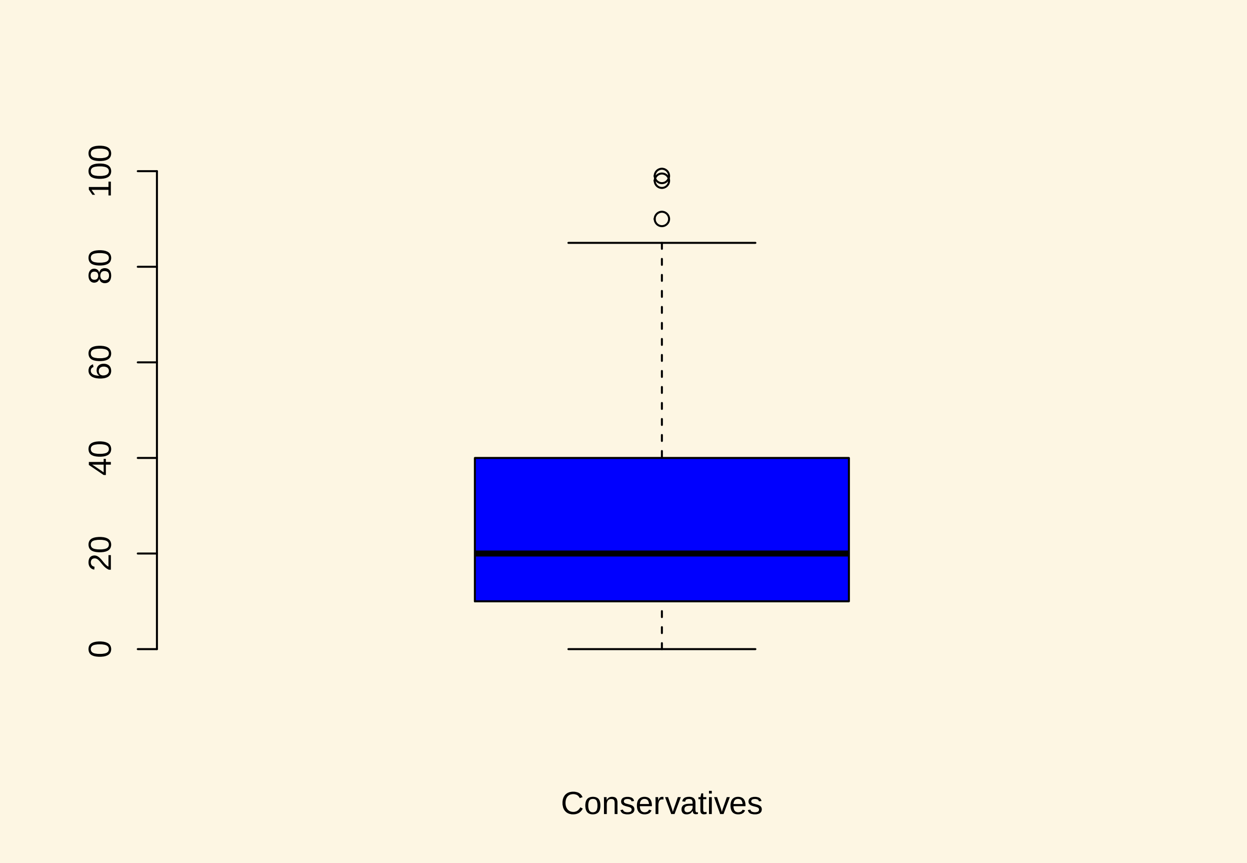
```
table(dat1$Cons)
```

```

  0   1 
765 284 
```

In our data, 284 respondents associate with the Conservative party and 765 do not. We create a boxplot of *IMMBRIT* but only for members of the Conservative Party. We do so by using the square brackets to subset our data.

```
# boxplot of immbrit for those observations where Cons is 1
boxplot(
  dat1$IMMBRIT[dat1$Cons==1],
  frame.plot = FALSE,
  xlab = "Conservatives",
  col = "blue"
  )
```

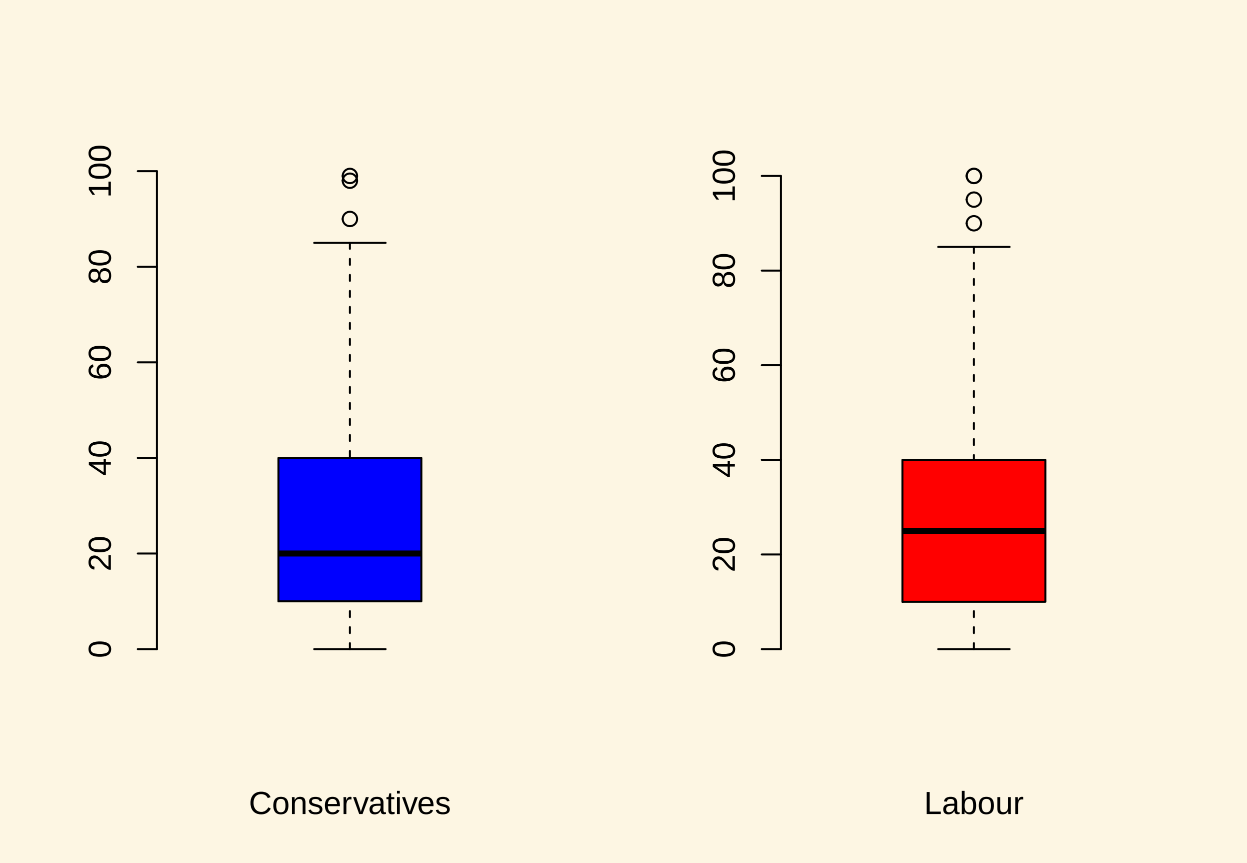


We would now like to compare the distribution of the perception fo Conservatives to the distribution among Labour respondents. We can subset the data just like we did for the Conservative Party. In addtion, we want to plot the two plots next to each other, i.e., they should be in the same plot. We can achieve this with the `par()` function and the `mfrow` argument. This will spilt the plot window into rows and columns. We want 2 columns to plot 2 boxplots next to each other.

```r
# split plot window into 1 row and 2 columns
par(mfrow = c(1,2))

# plot 1
boxplot(
  dat1$IMMBRIT[dat1$Cons==1],
  frame.plot = FALSE,
  xlab = "Conservatives",
  col = "blue"
  )

# plot 2
boxplot(
  dat1$IMMBRIT[dat1$Lab==1],
  frame.plot = FALSE,
  xlab = "Labour",
  col = "red"
  )
```



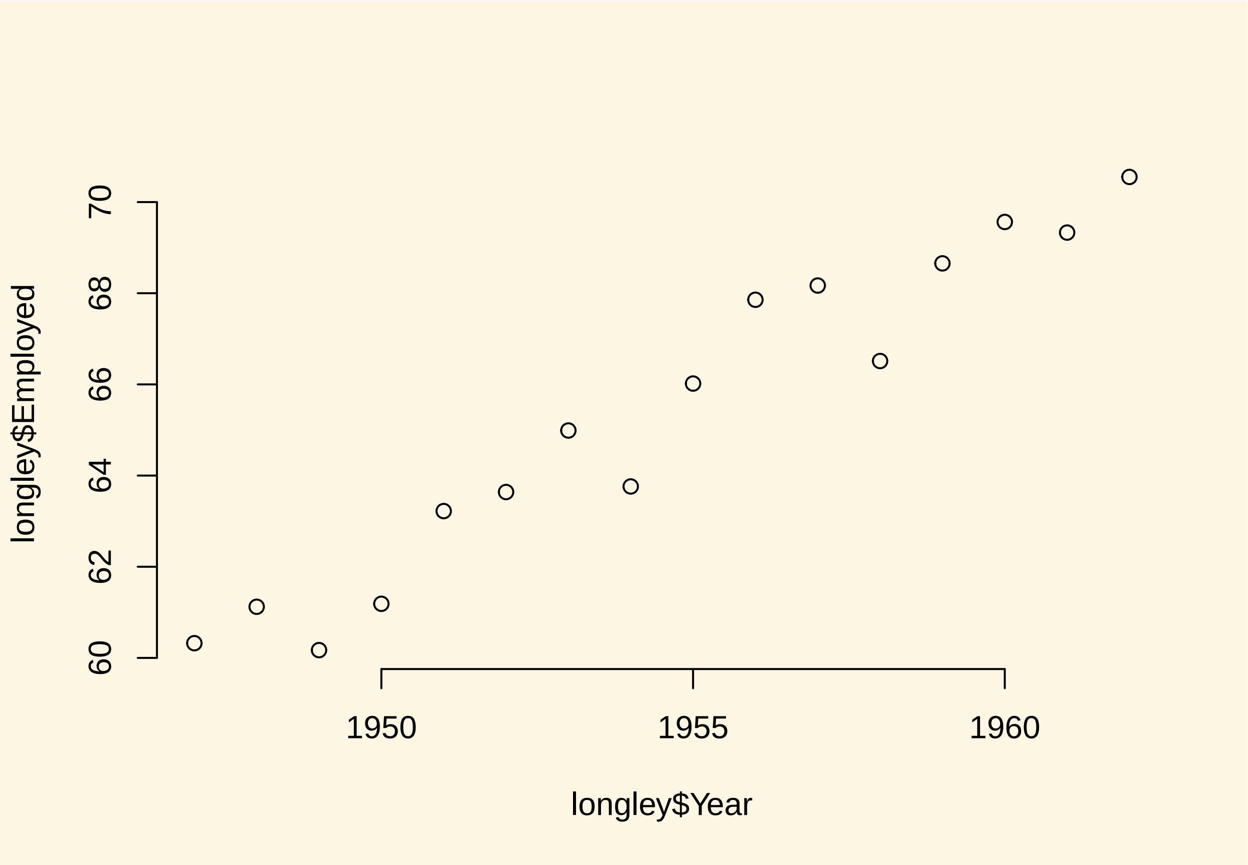
It is very hard to spot differences. The distributions are similar. The median for Labour respondents is larger which mean that the central Labour respondent over-estimates immigration more than the central Conservative respondent.

You can play around with the non-western foreigners data on your own time. We now turn to a dataset that is integrated in R already. It is called `longley`. Use the `help()` function to see what this dataset is about.

```
help(longley)
```

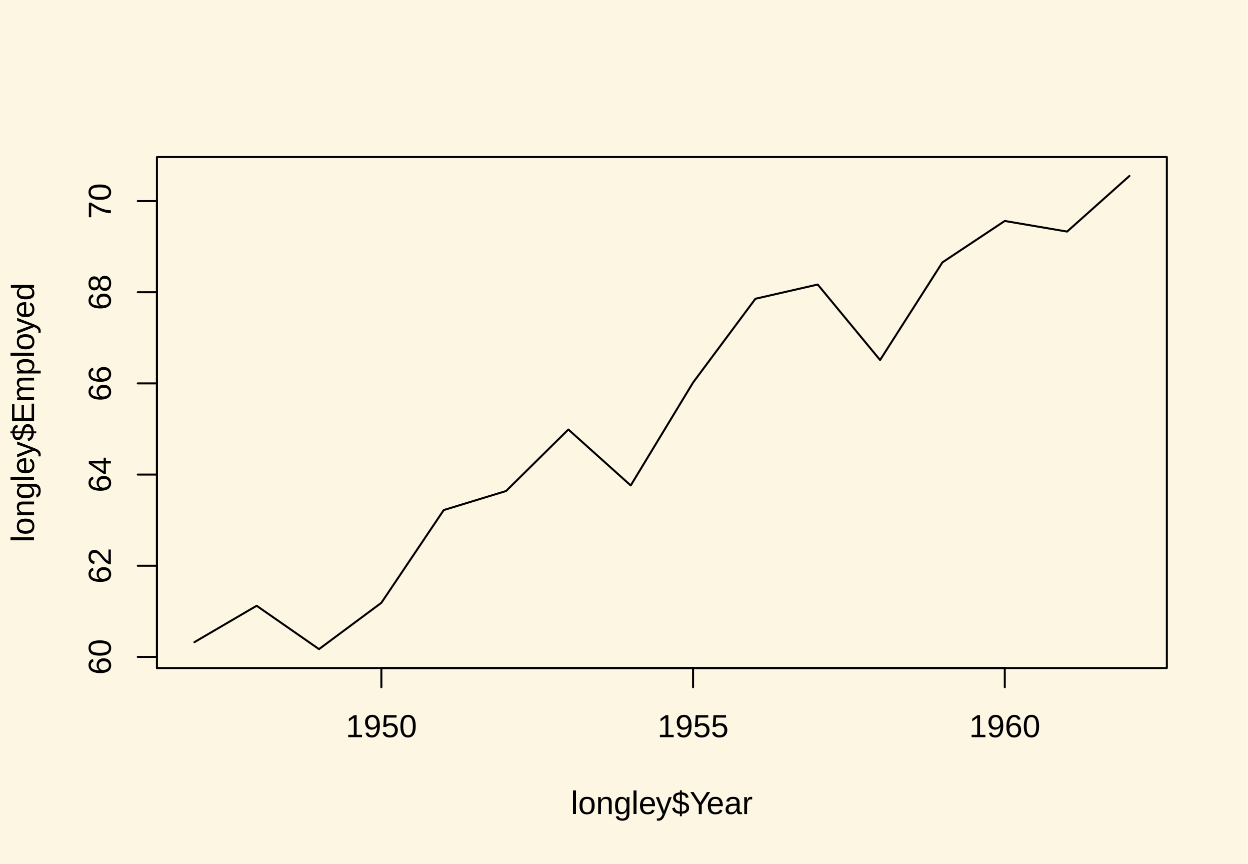
Let's create a scatterplot with the `Year` variable on the x-axis and `Employed` on the y-axis.

```
plot(x = longley$Year, # x-axis variable
     y = longley$Employed, # y-axis variable
     bty = "n" # no box around the plot
     )
```



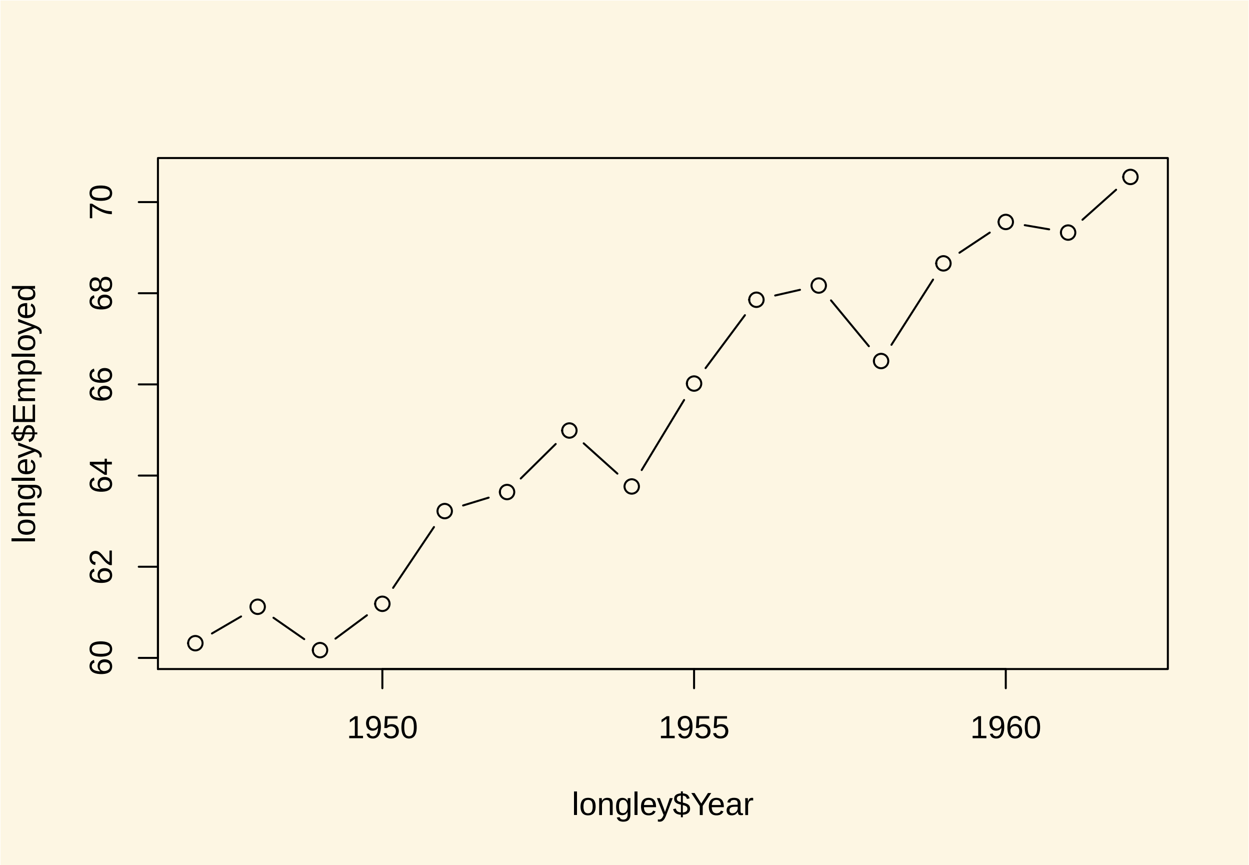
To create a line plot instead, we use the same function with one additional argument `type = "l"`.

```
plot(longley$Year, longley$Employed, type = "l")
```



Create a plot that includes both points and lines.

```
plot(longley$Year, longley$Employed, type = "b")
```



8 Correlations and differences in means

8.1 Seminar

Will be uploaded tomorrow

9 Regression

9.1 Seminar

Will be uploaded tomorrow

10 Prediction and assessing prediction accuracy

10.1 Seminar

Will be uploaded tomorrow