R Bootcamp - Course 1: Getting Started

James Baglin

Last updated: 13 July, 2020

Overview

Summary

This course will get you started with learning R and using the RStudio IDE. The course will cover installation of R and RStudio, accessing RStudio through RMIT, using RStudio's basic features and the basics of R programming.

Learning Objectives

By the end of this course, you will have covered the following:

- Installing R and RStudio
- Accessing RStudio from MyDesktop
- An overview of the RStudio interface
- Basic programming in R

Installion

Both R and RStudio run on Windows, Linux and Mac. Install R first and then install RStudio.

1. R: Windows, Linux and Mac: http://cran.ms.unimelb.edu.au/ (http://cran.ms.unimelb.edu.au/)
2. RStudio: Windows, Linux and Mac: http://www.rstudio.com/products/rstudio/download/ (http://www.rstudio.com/products/rstudio/download/)

Once installed, load RStudio. RStudio is an integrated development environment (IDE) for R. It will allow you to use R in a more efficient manner. RStudio provides the user with a streamlined user interface and access to many powerful tools to make working with R more efficient.

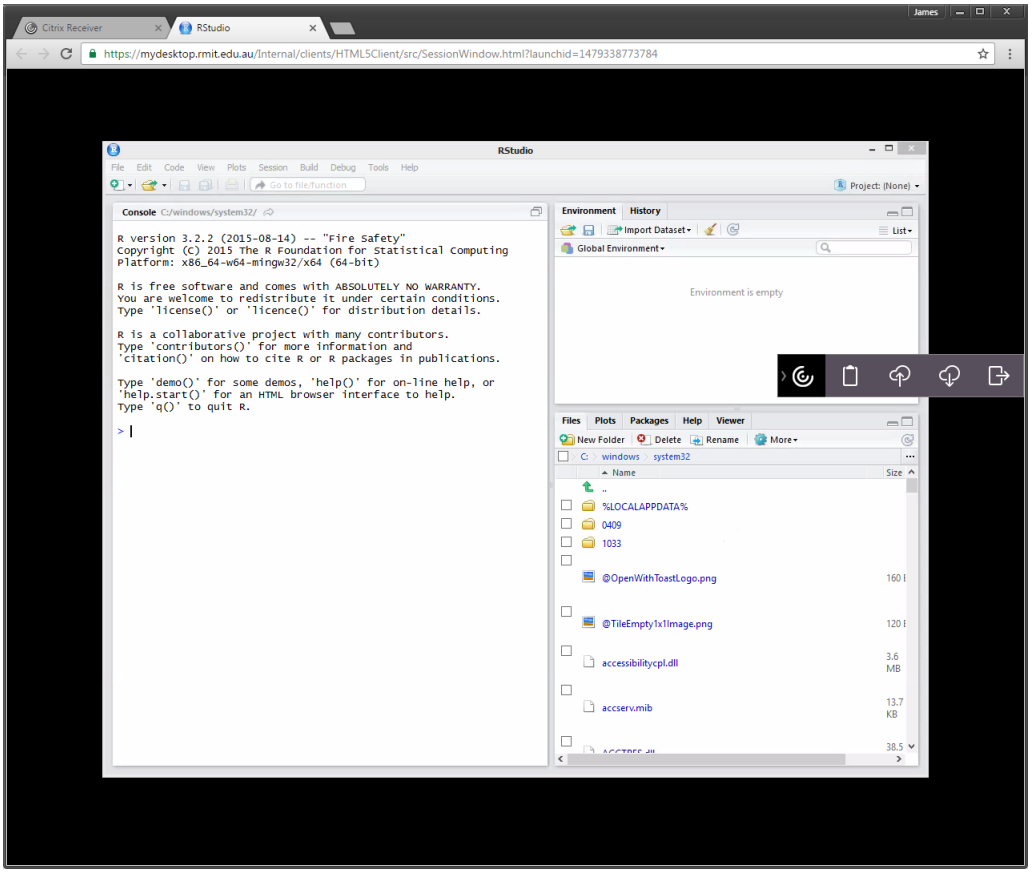
RStudio also offers a free server version (https://www.rstudio.com/products/rstudio/#Server), that allows you to install of RStudio on a cloud server. For example, here is a guide for using the Google Compute Engine to install and run an instance of RStudio on an Ubuntu server. There are many other useful guides for different cloud computing services and servers. The main advantage of setting up a server is not having your installation of R tied down to one physical machine. This allows you to work seamlessly across multiple computers and only needing to maintain and update one instance (as you will discover, R and Studio update on an annoyingly regular basis!). The RStudio Server Pro (https://www.rstudio.com/products/rstudio-server-pro/) version, which comes at a cost, adds many useful advanced features including real-time collaboration.

RStudio on RMIT MyDesktop

R and RStudio are both available via the RMIT MyDesktop. This allows you to use R on any portable computing device with a WiFi and Internet connection. It also allows you to run R without installing it on your computer.

http://www1.rmit.edu.au/students/mydesktop
(http://www1.rmit.edu.au/students/mydesktop)

After accessing MyDesktop, wait a few minutes and search for “RStudio” after clicking on the Windows button. If you’re having trouble accessing MyDesktop or RStudio through MyDesktop, contact RMIT IT Service Desk.



An Overview of the RStudio Interface

The videos provide a run-down of basic and most important features of the RStudio Interface.

Basic Scripts

Scripts are plain text files, with the .R extension, which acts as recipe book for your R code. You can type and edit your code in the script file and then issue parts or all of your code to the R console to be executed. You can create a new script in RStudio by going to File –> New File –> R Script. Ensure you save and name your script so you can come back to it later.

Projects

RStudio Projects associate all your files with a common directory (e.g. scripts and data). When you exit or change projects the environment variables are saved so you can come back to your project later without having to re-run all your code. Projects are the best way to organise your R files.

Other Views

The RStudio Environment window (top right) keeps track of all the objects in a particular session. Objects will be explained shortly. In this script we created an object labelled x, where x was a numeric vector (list) of 10,000 random and normally distributed values.

The bottom right corner of the screen has many useful tabs. The screenshot above shows the Plots view. We can see a histogram of the x values. The different tabs are summarised as follows:

- **Files**: You can browse your working directory for scripts and dataset files.
- **Plots**: This is where you can view and export plots produced in R.
- **Packages**: This is where you can view and load installed packages (packages are explained shortly).
- **Help**: Access R help files.
- **Viewer**: This is a web browser built into RStudio which is used to view web files generated in RStudio.

Programming Basics

In the following sections you will learn the basics of programming in R. If you haven't programmed before, now is a great opportunity to start. Fortunately, R is fairly intuitive to learn. We will start with the fundamentals for now and get heaps of practice throughout the semester. Its a great skill to possess and can even be a lot of fun if you like problem solving. Let's get started.

How to read these notes

The following code box refers to commands submitted to the console. You can copy this code directly and paste it into a script of run it directly from the RStudio console.

```
x <- c(1,2,3,4,5,6,7,8,9,10)
mean(x)
```

The following code box refers to output produced by the console:

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

Code, Commands and Syntax

We get R to do stuff by running commands from our script files or by typing directly into the R Console. Commands can be a whole range of different things, like basic arithmetic, loading data, producing plots or running statistical functions. To issue commands to R, we write code. The code we write is governed by "syntax" or a set of rules used by a programming language. If we don't use the right syntax, we will get an error. You will make heaps of errors when you're learning to use R. Don't worry about it. Its a normal part of learning. Let's start looking at the basics of R syntax.

Case Sensitivity

R is case sensitive so "A" and "a" are two different characters.

```
x <- "A" # Assign x as "A"
y <- "a" # Assign y as "a"

#Does x = y?
x == y
```

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

Command Separation and Grouping

Commands are separated using a new line:

```
x <- rnorm(10000,0,1) # Randomly generate 10,000 normally distributed val
  ues
mean(x) #Calculate the mean of x
```

```
## [1] -0.01520157
```

```
sd(x) #Calculate the standard deviation of x
```

```
## [1] 0.9937938
```

...or row, or by using a semi-colon ";"

```
mean(x); sd(x)
```

```
## [1] -0.01520157
```

```
## [1] 0.9937938
```

Comments

Use # to add comments to your code. Any code proceeding the # symbol is ignored by the console. Comments are useful to remind yourself and others what your code does. Get into the habit of commenting. Look back at the previous code chunks to see examples of commenting.

Objects and Object Assignment

R objects can be variables, values, datasets, text or functions. R Studio lists the objects in the Environment window (top right). Stored objects are saved to the R session and can be recalled at any time by typing the object's name in the console. For example, the following codes assign the value of the mean of x to an object named "m". We use "<-" to make an assignment. You can also use the "=" symbol, but this should be avoided because it may be confused with the equals sign used as a logical operator. The RStudio keyboard shortcut for writing "<-" is to hit the ALT and "-" key.

```
m<-mean(x) #Assign the mean of x to variable m
#Print the mean of x
m
```

```
## [1] -0.01520157
```

Vectors

A vector is a simple sequence of stored data. We can store a simple dataset in a vector using the `c()` function. For example, let's say we have five random peoples' heights (cm): 166, 177, 164, 167, 177. We can store them in a vector, named "heights" to make it easy to do statistical calculations.

```
heights<-c(166, 177, 164, 167, 177)
mean(heights) #Calculate the mean height of the sample
```

```
## [1] 170.2
```

```
sd(heights) #Calculate the standard deviation of the sample's heights
```

```
## [1] 6.300794
```

Shortly you will learn to store datasets as data frame objects which is an even more powerful way to work with data.

Functions

Functions are the workhorse of R. R has many built in functions that do a whole host of different things. R's functionality can be greatly enhanced by installing packages (see below). You will learn to use a whole range of different functions as the course unfolds. This course has already introduced a number of very useful statistical functions. Recall the use of `mean()`, `sd()` and `hist()`.

Writing your own functions

Packages - Installation and Loading

Packages are compilations of functions. For example, the `ggplot2` packages provide access to functions designed to create beautiful graphs. Packages are why R is so powerful. Most statisticians develop packages to freely implement their cutting edge statistical methods that are not available in commercial packages. Others develop packages to fill the many gaps of commercial packages. There are an overwhelming number of packages available on the Comprehensive R Archive Network (CRAN) (https://cran.r-project.org/), as well as other unofficial packages (many available from GitHub. (https://github.com/) As of Nov 2016, the CRAN website listed 9,535 available packages!

Packages can be installed through RStudio by going to Tools –> Install Packages. You will need to know the name of the package you want to install. The following video shows an example of installing the `ggplot2` package which will be introduced in Course 2. Most packages require other packages to be installed to function properly. These are called dependencies. If these are missing when you try to install a new package, they will be installed by default. As you will discover, `ggplot2` has many dependencies.

Alternatively, you can use the `install.packages()` function:

```
install.packages("ggplot2")
```

After installing the package, you need to load it before you can access its functions. This is easy in RStudio. Select the Packages tab in the bottom right corner and tick the box next to the package name. RStudio will load the package. You need to load packages each time you start a new session. If you work from a different computer, you will need to install the package as well.

Alternatively, you can use either of the following functions to load an installed package:

```
library(ggplot2)
require(ggplot2)
```

Mathematical Operations

R can be used to do all types of mathematics. Try the following code to get an idea of basic operations.

```
# Addition
7 + 4
```

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

```
# Subtraction
7 - 4
```

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

```
# Multiplication
7 * 4
```

```
## [1] 28
```

```
# Division
4/7
```

```
## [1] 0.5714286
```

```
# Square root
sqrt(9)
```

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

```
# Power
9^2
```

```
## [1] 81
```

```
# Natural logarithm
log(2.718)
```

```
## [1] 0.9998963
```

```
# Exponent
exp(1)
```

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

Logical Operators

The following table explains the logical operators used in R. These operators are used for many different reasons. For example, sub-setting your data, filtering out particular values, or checking if certain conditions are true or false, or counting specific values. You will learn more as you progress.

```
# Less than
4 < 5
```

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

```
# Greater than
5 > 4
```

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

```
# Equal to or less than
5 <= 5
```

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

```
# Equal to or greater than
5 >= 4
```

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

```
# Equal  to
5 == 4
```

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

```
# Not equal to
5 != 4
```

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

```
# and
5 == 5 & 4 == 4
```

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

```
# or
5 == 5 | 4 == 3
```

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

```
# Not
x <- c(0,1,2,3,4,5)
!x==4
```

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

Missing Values

When missing values are present in datasets, R represents them using `NA` (not available). `NaN` represents impossible values, such as dividing 1/0. Sometimes missing values can cause errors. For example:

```
heights<-c(166, 177, 164, 167, NA) #Note missing value
mean(heights) #Calculate the mean height of the sample, ERROR!
```

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

Sometimes we need to tell R how to deal with the missing values...

```
mean(heights, na.rm=TRUE) #Calculate the mean height of the sample, remov
  ing missing values
```

```
## [1] 168.5
```

The `na.rm=TRUE` option tells R to remove missing values from the `mean()` function.

Characters

Character variables or "string" variables are usually associated with categorical or nominal variables, but may also refer to labels, plot titles and other types of textual input. When characters or text strings are used in R, they must be enclosed in quotation marks.

```
Gender<-c(male,female) #Create a character vector, ERROR!
```

```
## Error in eval(expr, envir, enclos): object 'male' not found
```

```
Gender<-c("male","female") #Create a character vector, fixed...
```

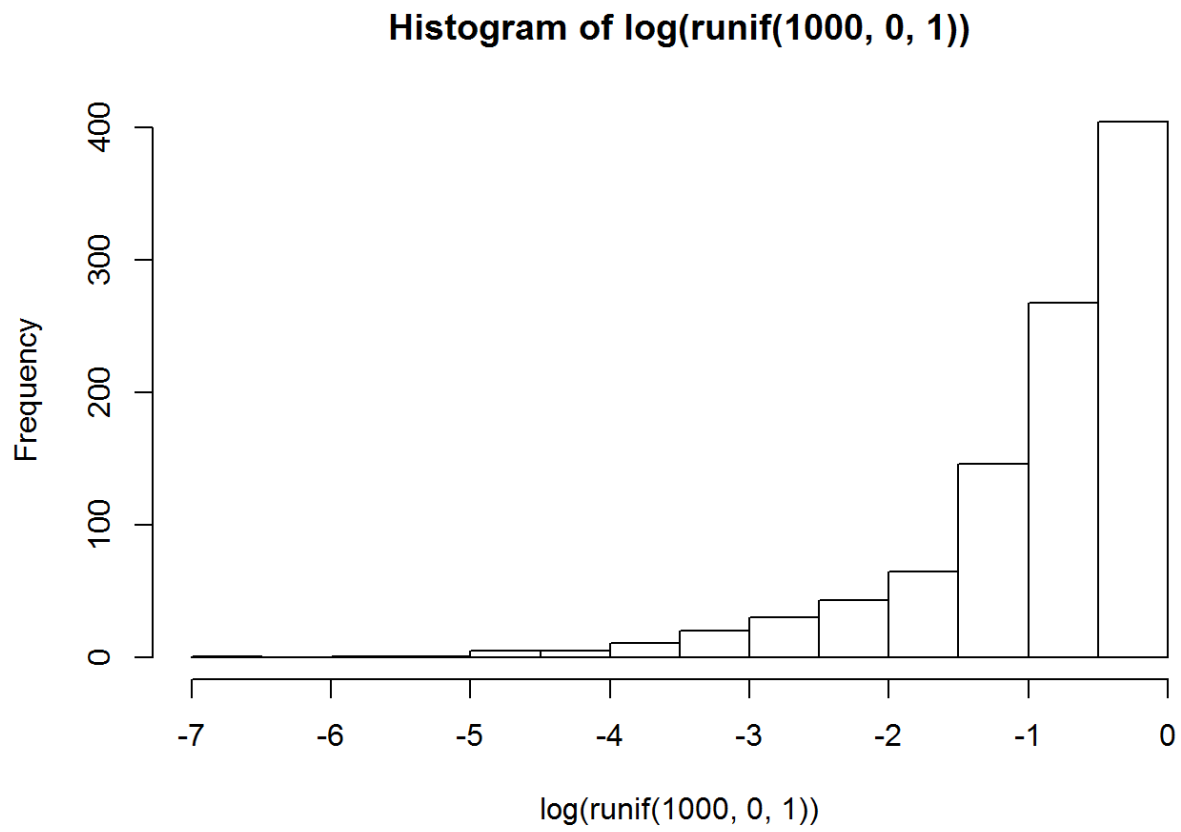
If you don't enclose in quotation marks, R thinks you're referring to an object.

Pipes

In recent years, the clarity of R coding has been substantially improved with the introduction of the `magrittr` package. This package overcomes the cumbersome nested coding typically required to join multiple functions. For example, the following "traditional" code performs the following computations:

- Generate 1000 data data points from a uniform distribution between 0 and 1
- Log transform the data
- Plot histogram of resulting distribution

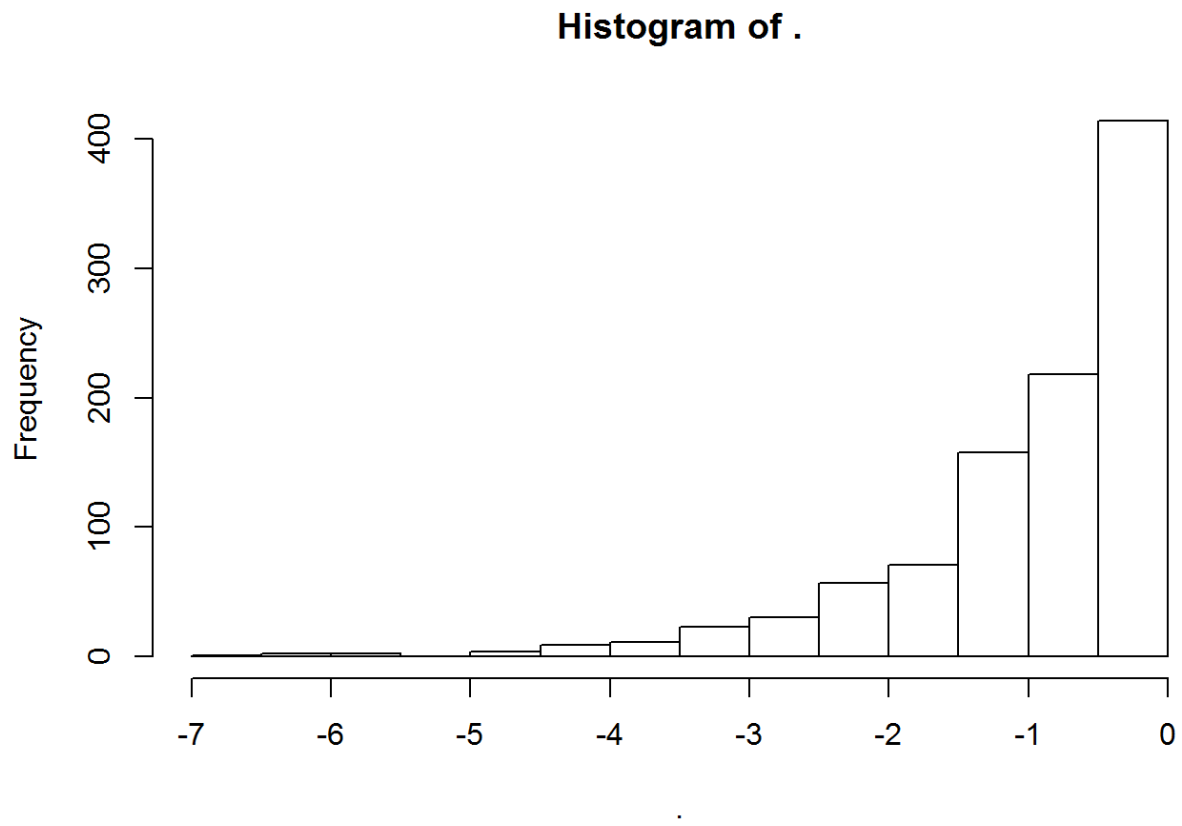
```
hist(log(runif(1000,0,1)))
```



As you can see, the functions `hist()`, `log()` and `runif()` are nested within each other. There are lots of side-by-side parentheses, which give you a headache when trying to identify errors. You have to read the code backwards to understand the order in which these functions are applied. This might be OK for three functions, but statistical computations gets a lot more complex.

In order to improve the readability of this code, we can use the pipeline or "pipe"" operator `%>%` from the `magrittr` package, hit Ctrl + shift + m in RStudio as a shortcut, to link these functions together in the order of their computation. When you see `%>%` think of the word "then". For example, `runif()` then `log()` then `hist()`:

```
library(magrittr)
runif(1000, 0, 1) %>% log() %>% hist()
```



I think most people will agree that this code is easier to read. However, the "traditional" nested code is absolutely acceptable and you will come across both styles.

Pipes are really powerful when paired with compatible packages such as `dplyr` (covered in Course 2).

So what is really going on? Essentially, the pipe operating passes a preceding function as the first argument to the following function. Therefore, the following two lines of code are equivalent:

```
log(4)
```

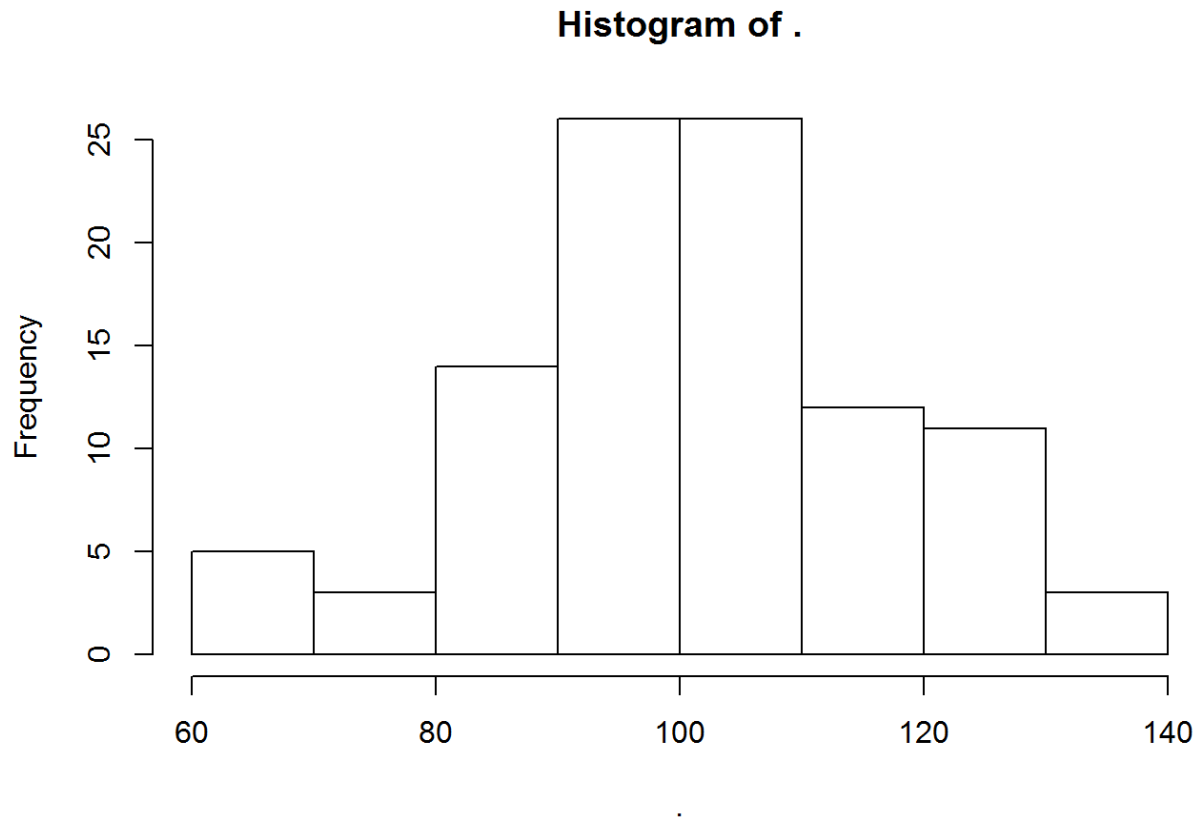
```
## [1] 1.386294
```

```
4 %>% log()
```

```
## [1] 1.386294
```

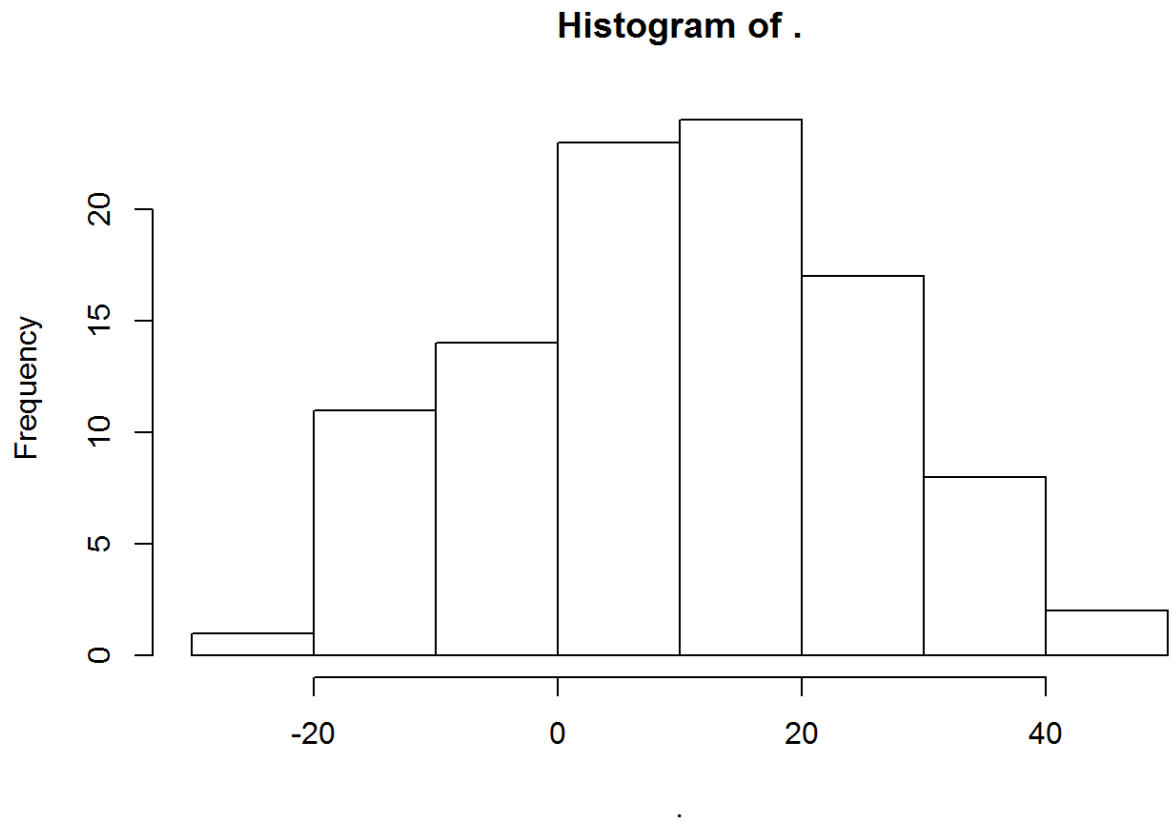
Pipes work best when functions are optimised to take the first argument from a preceding function. However, preceding functions can still be passed to second, third or fourth arguments quite easily using a `.`. For example, the `rnorm()` function can be used to generate normally distributed data. The first argument is always the number of values to generate and the second and third values are the population mean and standard deviation, `rnorm(x, mu, sd)`. Let's say we want to pass a value of 15 to the third argument using a pipe. This is how we would do it:

```
sd <- 15
sd %>% rnorm(100,100,.) %>% hist()
```



Or, if we want to pass a value of 10 as the mean:

```
x <- 10
x %>% rnorm(100,.,15) %>% hist()
```



The `.` pipes the preceding function to the required argument.

As the use of pipes has spread like a virus for R coding, this course will predominantly use pipes. However, you will also come across the more traditional nested code from time to time. A proficient use of R will need to understand both styles.

Getting Help

If you ever forget how to use a function, you can bring up the help file in RStudio using the help() function,

```
help(mean)
```

This will display the help files associated with the `mean()` function. You can also browse the help files in the RStudio Help tab (Bottom right window).

Sometimes the help files aren't much help and what you really need to see are some examples. Use Google instead.
