class: center, middle

# Data Analysis in R

Adam Rawles

---

## Recap

–

- What is R and RStudio?

–

- Basic arithmetic operators

–

- Variable assigment

–

- Data types

–

- Data structures (including subsetting)

–

- Functions

---

## Overview

–

- Installing packages

–

- Loading data

–

- Cleaning data

–

- Summary statistics

–

- Graphs and plots

---

## Installing packages

- Packages provide extra functions and/or data to R (for example, the `stringr` package provides functions to help with text cleaning)

–

- Installing packages with RStudio is easy to do

–

- RStudio has a built-in interface for installing packages...
- Or, you can use the R function `install.packages()`, and provide the name of the package(s) you want to install to the function

–

```
install.packages("ggplot2")
install.packages(c("ggplot2", "dplyr"))
```

–

- Packages must be loaded before they are available, via the `library()` function

–

```
library(ggplot2)
```

- Note: if you close RStudio, you'll need to reload your packages.
- Think of installing the package being like installing a program, and the `library()` call as like opening it - you wouldn't open a program without installing it and you wouldn't use it without opening it!

---

## Using packages

- If you've loaded a pacakge via `library()`, then you don't need to do anything else before you use it.

–

- However, when using a non-base R function or if you haven't loaded the package, it's always a good idea to make it explicit what package it came from.

–

- We do that using `::`

–

```
ggplot2::ggplot()
```

–

- This will mean that anyone reading your code will know exactly what package a function came from

–

- It'll also ensure that you're using the function you mean to in the event that two packages have functions with the same name

---

## Loading data

–

- Data can be loaded into R in many formats

–

- The easiest of which are .xlsx or .csv files

–

- These can be loaded into R two different ways

–

- The first is by using the `read.csv()`/`read.xlsx()` functions

–

- The `read.csv()` function takes a string of the file's path as it's only required input argument

–

- But there are a number of optional arguments (e.g. `header`, `stringsAsFactor`,...)

–

- The `read.xlsx()` function requires both a path, and the index of the sheet you want from the workbook

---

## Loading data

- The second is to use RStudio's built in "Import Dataset" interface

–

- This interface essentially just acts as a wrapper to the `read.csv()/xlsx()` functions
- You'll see that when you run it, the code to import the data is run in the console

---

## Loading data - example

---

## Loading data - example

```
test_data <- read.csv("test_data.csv",
                      header = TRUE,
                      stringsAsFactors = TRUE)

head(test_data, n = 2)
```

```
##   Firm LeadDivision Income Employees ReturnDate
## 1    X      Banking   1000        50 01/01/2018
## 2    B    Fiduciary   2000        60 01/06/2018
```

---

## Loading data - exercise

–

- Using RStudio's "Import Dataset" or the `read.csv()` function, load the test_data dataset

---

## Data cleaning

–

- After the data is loaded into R, we need to make sure that each column is in the correct format (e.g. character, factor, numeric, etc.)

–

- This can be done two different ways:

-
You
can
ei-
ther
click
on
the
dataframe
in
the
En-
vi-
ron-
ment
pane,
which
will
open
the
dataframe
in
the
Source
pane

- From here, you can hover over the column headers to see what type the data is stored as
- Or, you can use the is.xxxxx functions to check from the console.

–

```
- To do this, you choose the appropriate function for the type you want (e.g. `is.numeric()`), and
```

---

## Data cleaning

```r
is.numeric(test_data$Income)
```

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

```r
## we could also do is.numeric(test_data[,1]) as per our last session on subsetting

is.numeric(test_data$LeadDivsion)
```

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

```r
is.factor(test_data$LeadDivision)
```

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

---

## Data cleaning

- If you want to get the type of a column without comparing it to other types, you use the `is()` function

–

- The first value returned from this function will tell you the data type of the column

```r
is(test_data$Firm)
```

```
## [1] "factor"              "integer"             "oldClass"
## [4] "double"              "numeric"             "vector"
## [7] "data.frameRowLabels"
```

```r
is(test_data$Employees)
```

```
## [1] "integer"             "double"              "numeric"
## [4] "vector"              "data.frameRowLabels"
```

---

## Data cleaning

- If a column does not have the correct type, we can easily coerce the values into the type that we want

–

- The exact method is different depending on what you are converting from and to

–

- Generally speaking however, the method for converting a column type is:

```
dataframe$column <- as.xxxxx(dataframe$column)
```

---

## Data cleaning - exercise

–

- Convert the the Firm column from a factor to a character, and check your conversion worked

---

## Date cleaning - answer

```
test_data$Firm <- as.character(test_data$Firm)

is.character(test_data$Firm)
```

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

---

## Data cleaning

- This method works well for:

–

- Numeric/integer to character

–

- Character to numeric/integer

–

- Integer to numeric

–

- Numeric to integer

–

- Character to factor

–

- Factor to character

–

- Numeric to factor

–

- For factor to numeric, there's an extra step:

–

- Before the factors levels can be converted, they need to be converted to characters first:

```
dataframe$column <- as.numeric(as.character(dataframe$column))
```

---

## Data cleaning - dates

- Dates in R can be tricky

–

- **R will not import date values in as dates unless you specify that it should**
  - But RStudio's "Import Dataset" feature can be handy here...

–

- If you don't, R will import them as characters (which means they'll be converted to factors unless you specify `stringsAsFactors = FALSE`)

–

- To convert a character to a date, use the `as.Date()` function...

---

## Data cleaning - dates (example)

```r
datetest <- "12/12/2018"
datetest <- as.Date(datetest, format = "%d/%m/%Y")
is(datetest)
```

```
## [1] "Date"     "oldClass"
```

–

- To convert from a factor to a date, first convert to a character...

---

## Data cleaning - dates (exercise)

- Our `test_data$ReturnDate` column is currently a factor, convert it to date

–

- Remember to convert the column to character first!

---

## Data cleaning - dates(answer)

```r
test_data$ReturnDate <- as.character(test_data$ReturnDate)
test_data$ReturnDate <- as.Date(test_data$ReturnDate,
                                format = "%d/%m/%Y")
```

```r
test_data$ReturnDate <- as.Date(as.character(test_data$ReturnDate),
                                format = "%d/%m/%Y")
```

---

## Data cleaning - dates

- Dates can also come in numeric form, calculated as the number of days from a particular origin

–

- For example, a numeric value of 1, with an origin of 12/12/2018 would correspond to a date value of 13/12/2018

–

- If your date values are in numeric format, you need to specify the origin in the `as.Date()` function...

---

## Data cleaning - dates (example)

```
datetest <- 17940
datetest <- as.Date(datetest,
                    origin = as.Date("01/01/1970",
                                     format = "%d/%m/%Y"))
datetest
```

```
## [1] "2019-02-13"
```

---

## Data cleaning - dates

- With dates, you'll often need to specify the format (format codes can be found online and are included in your help sheet)

–

- There's no `is.Date()` function in base R

–

- So to check whether a value is a date, use `is()`

---

## Data cleaning - conclusion

- To find out the type of a column, use the `is()` function

–

- Otherwise, you can (usually) test if a column is a specific type via the is.xxxxx functions

–

- Converting between datatypes (except from numeric -> factor) is easy with the as.xxxxx functions

–

- When converting numbers to factors, convert them to characters first

–

- When converting from characters or numeric to dates, `as.Date()` requires "format" or "origin" parameters respectively

---

## Summary statistics

- Before doing any in-depth analysis, it's always a good idea to get some descriptive statistics from your data

–

- This includes the mean, the median, the standard deviation, and the interquartile range, but the statistics you use will depend on your data

–

- Getting these values for each column is easy using the built-in functions R provides:

```
mean()
median()
sd()
quantile()
```

---

## Summary statistics - exercise

- Find the mean, median, standard deviation, and quantiles for our meter reading column

---

## Summary statistics - answers

```
mean(test_data$Income)
```

```
## [1] 4434.174
```

```
median(test_data$Income)
```

```
## [1] 3528
```

```
sd(test_data$Income)
```

```
## [1] 2574.724
```

```
quantile(test_data$Income)
```

```
##      0%     25%     50%     75%    100%
##  1000.0  2930.5  3528.0  5850.5 10000.0
```

---

## Summary statistics

- More often than not however, you'll want summary statistics for more than one column, or maybe broken down by group

–

- R includes a function that will give you summary statistics for each column (the `summary()` function):

```
summary(test_data)
```

```
##      Firm              LeadDivision     Income        Employees
##  Length:23          Banking   :10    Min.   : 1000   Min.   : 40.00
##  Class :character   Fiduciary : 5    1st Qu.: 2930   1st Qu.: 48.00
##  Mode  :character   Insurance : 4    Median : 3528   Median : 59.00
##                     Investment: 4    Mean   : 4434   Mean   : 65.22
##                                      3rd Qu.: 5850   3rd Qu.: 67.50
##                                      Max.   :10000   Max.   :150.00
##    ReturnDate
##  Min.   :2018-01-01
##  1st Qu.:2018-01-01
##  Median :2018-06-01
##  Mean   :2018-05-03
##  3rd Qu.:2018-06-01
##  Max.   :2019-01-01
```

---

## Summary statistics by group

- Summary statistics by group are a little bit more complicated as they require us to "apply" the function for each group

–

- To do that, we use the `tapply()` function. This will take the test_data dataset, split it according to the grouping we give, and then use whatever summary function we provide

–

```
tapply(test_data$Income, test_data$LeadDivision, sd)
```

```
##    Banking  Fiduciary  Insurance Investment
##  3077.5284  3321.5053  1876.4645   870.7002
```

–

-

**So here we provide the Income column, say to split it up by Lead Division, and apply the `sd()` function**

## Summary statistics by group - exercise

- Produce the mean income for each division

---

## Summary statistics by group - answer

```
tapply(test_data$Income, test_data$LeadDivision, mean)
```

```
##    Banking  Fiduciary  Insurance Investment
##    4356.50    5124.00    4294.50    3905.75
```

---

## Summary statistics - conclusion

- Producing summary statistics for columns in a dataframe can be done through the `mean()`, `median()`, `sd()`, and `quan tile()` functions (among others)

–

- Producing summary statistics for multiple columns can be done with the `summary()` function

–

- Producing summary statistics by group requires us to `apply` the `summary()` function to each group

---

## Plotting

- After calculating your summary statistics, it can be useful to visualise the data to better understand any trends or differences

–

- For example, we may want to see if there's a difference between the incomes for different divisions

–

- We could use a plot to visualise that difference

–

- There are two good ways to create plots
  - The first uses R's built-in plotting functions
  - The second uses a package called "ggplot2"

–

We're going to focus on base R for now

---

## Plotting

- R's main plotting function is `plot()`

–

- This function takes the data you want to visualise as it's only required argument, with optional arguments to specify what kind of plot you want

–

- If you don't specify what kind of plot you want, `plot()` tries to guess the most appropriate type of plot for your data, and creates it

---

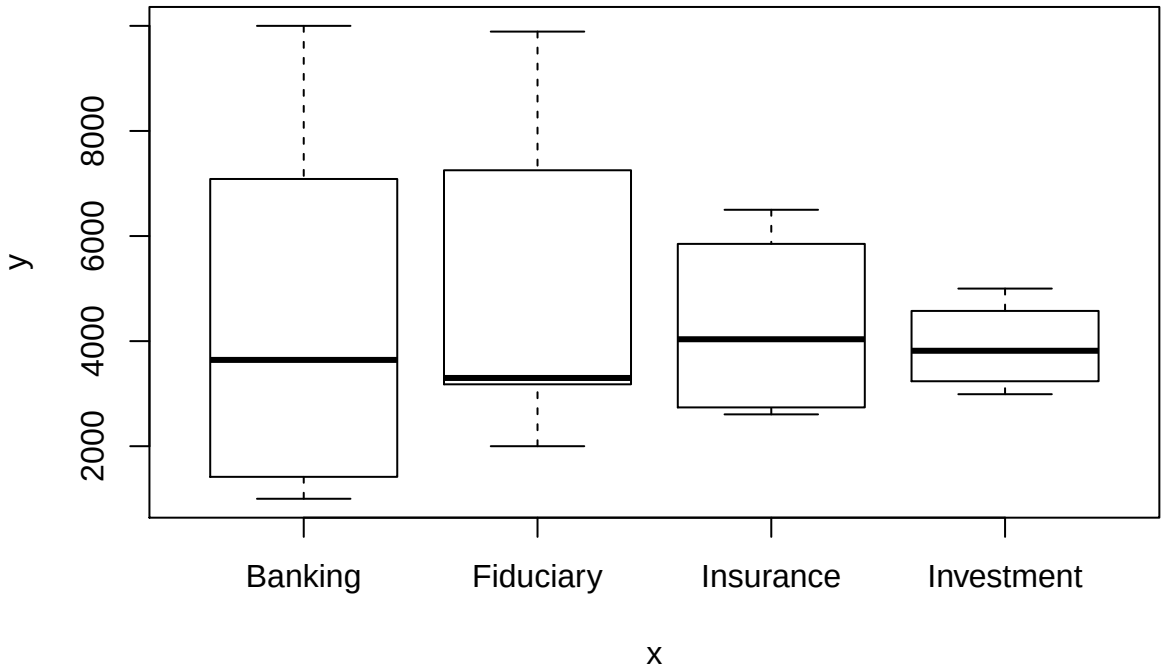
## Plotting - example

- First off, let's plot the incomes of the divisions to see if there's a difference:

```
plot(x = test_data$LeadDivision, y = test_data$Income)
```

---

## Plotting - example




---

## Plotting - example

- As you can see, we've specified our x and y axis, but not what type of plot we want

–

- But the `plot()` function has guessed that we probably want a boxplot

–

- Note: you can also force R to create a boxplot using the `boxplot()` function

---

## Plotting - example

- Next, we might want to see if there's a correlation between the income of a firm and the number of employees it has

–

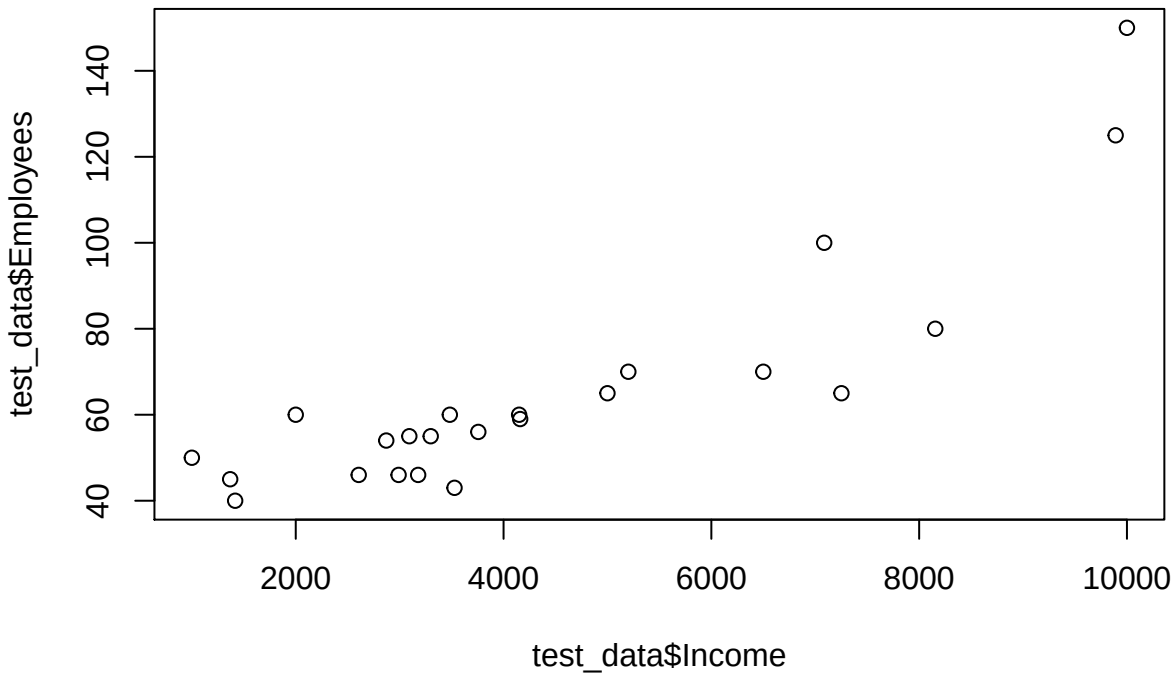
- Again, we can use the plot function, specify our x and y axis, and R will guess what the best plot is

–

```
plot(x = test_data$Income, y = test_data$Employees)
```

---

## Plotting - example




---

## Plotting - example

- R has created a scatter plot by default, but we can change the type if we wish using the option “type” argument
- See your help sheet for the different types

---

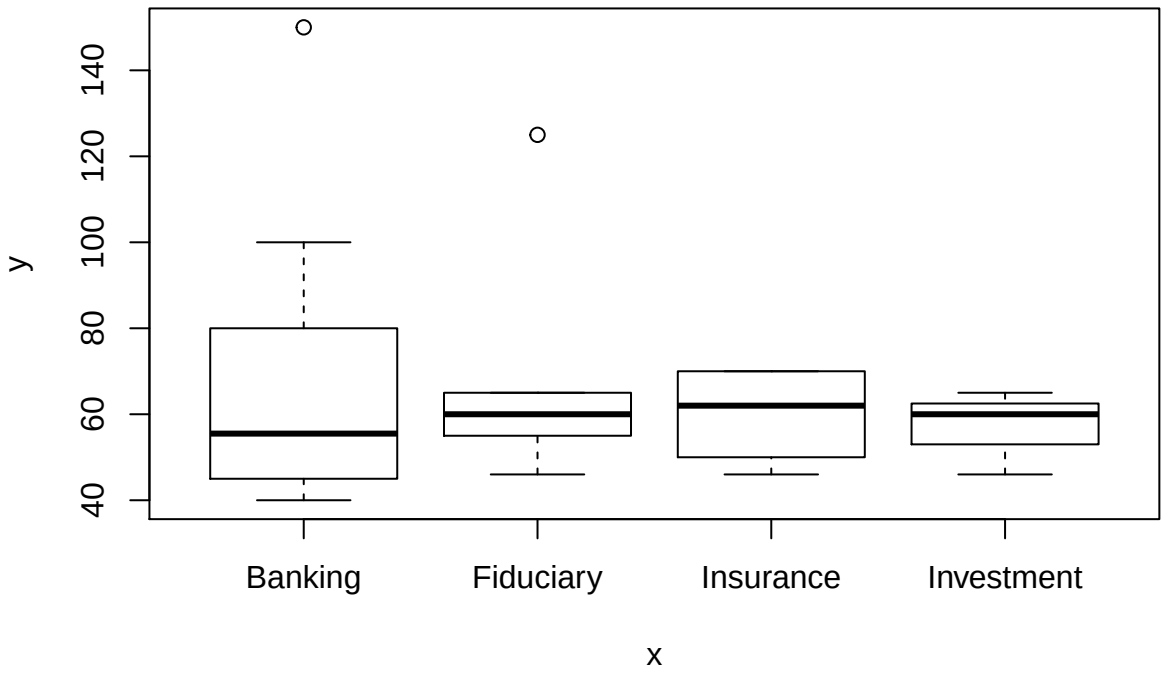
## Plotting - exercise

- Create a plot of employees against Lead Divisions...

---

## Plotting - answer

```
plot(x = test_data$LeadDivision, y = test_data$Employees)
```




---

## Histograms - example

- One of the best features of the plotting system in R is how easy it is to make histograms

_

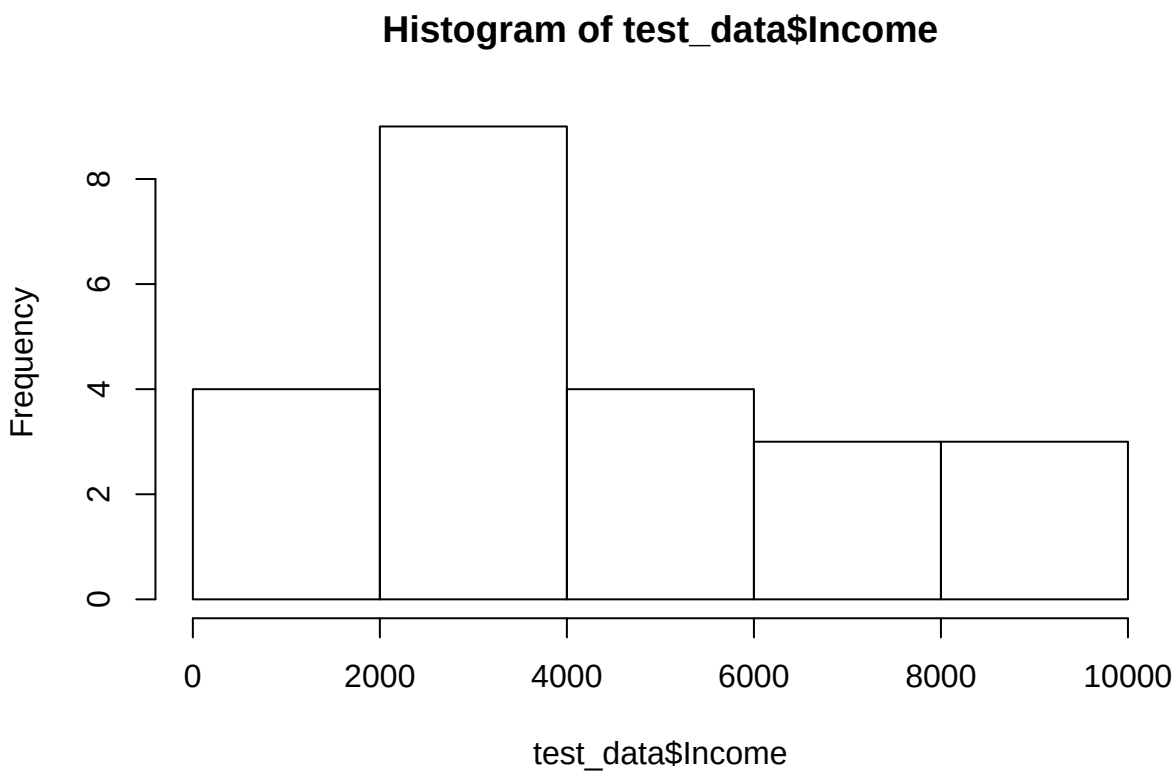
- You can either use the `plot()` function and specify "h" as the type, or we can use the `hist()` function and provide one variable to produce a simple histogram...

_

```r
hist(test_data$Income)
```

---

### Histograms - example




---

### Customizing your graphs

- To make your graph easier to understand, you may want to...
  - Change the title
  - Change the axis labels
  - Change the points
  - Change the size of the bins for our histogram

–

- Note: These are a few of the customization options, but there are many, many more

---

## Customizing your graphs

- Change the title
  - To change the title, all we need to do is specify a "main" parameter in our plot function...

–

```
plot(test_data$Income, test_data$Employees,
     main = "Correlation between Income and # of Employees")
```

–

- Change the axis labels
  - To change the labels on the axes, we use the "xlab"/"ylab"" parameters...

–

```
plot(test_data$Income, test_data$Employees,
     main = "Correlation between Income and # of Employees",
     xlab = "Income (£)",
     ylab = "Number of Employees")
```

---

## Customizing your graphs

- Changing the graph points
  - To change the points, we use the "pch" paremeter, and specify a value between 0 and 25...

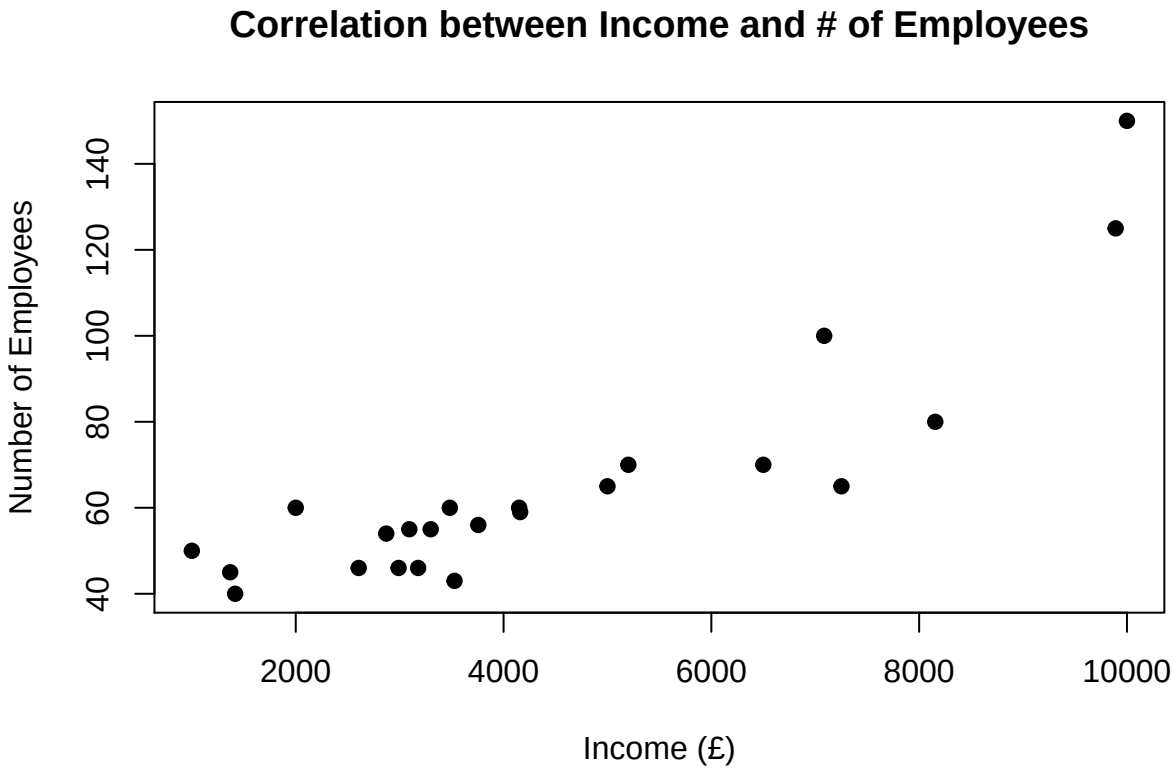
–

```
plot(test_data$Income, test_data$Employees,
     main = "Correlation between Income and # of Employees",
     xlab = "Income (£)",
     ylab = "Number of Employees",
     pch = 19)
```

- Putting those all together...

---

### Customizing your graphs




---

### Customizing your graphs (histograms)

- In addition to the previous customization options, for histograms, we can change the size of the bins

–

- To do this, we specify the "breaks" parameter in our `hist()` function

–

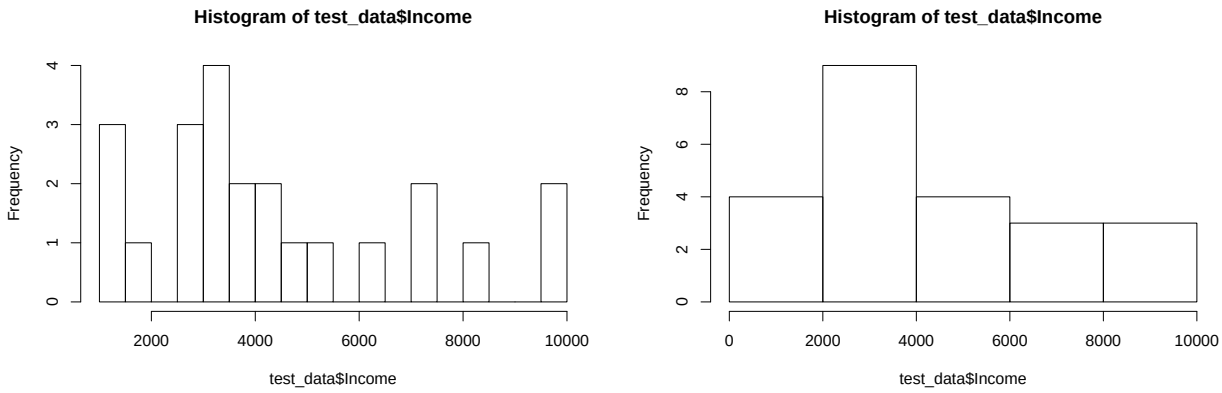
- This splits up out our data into x number of bins of equal size

–

```
hist(test_data$Income, breaks = 30)
```

---

## Customizing your graphs (histograms)




---

## Plotting - final exercise

- Create a plot of your choice, and change the title and axis labels.

---

## Plotting - conclusion

- In short, R has lots of in-built tools to make quick, simple graphs and plots

–

- Most types of graph only require a few input parameters, but there's lots of customization options

–

- In a later module, we'll look at a package called ggplot2, which we'll use to create more complex graphics

---

## Conclusion

- Installing packages
  - `install.packages()`, `library()`
- Loading data
  - `read.csv()`/`read.xlsx()`
- Cleaning data

  - Data type checks
  - Data type conversion
    * Look out for dates and factors!

- Plotting
  - `plot()`/`hist()`

- Customization
  - "main", "xlab", "ylab", "pch", "breaks"

---

## Next time...

- Creating functions
- For loops
- If else statements