Introduction to R

Alan R. Pearse, QUT Mathematics Society

20 March, 2019

Preamble

R is a high-level statistical programming language. It is widely used for manipulating, graphing and modelling data in a range of fields, from ecology to finance and beyond.

This workshop is an introduction to the basics of R. You will learn:

1. How to navigate around the RStudio interface; a free Interactive Development Environment (IDE) for R;
2. How to install and load packages that provide extra functionality for R;
3. How to import data into R from spreadsheets;
4. How to manipulate and summarise data in R; and
5. How to produce high-quality visualisations of data in R.

The RStudio IDE

This section is hands-on. You will be taken through the RStudio environment by the workshop demonstrator.

Installing and loading packages

R has a wide range of functionality built-in, but, being an open-source software, members of the R community have built and published plug-in 'packages' that extend R's functionality to do more complicated tasks. In this section, you will learn how to install and load several packages that we will use throughout the workshop.

To install a package, make sure you have an internet connection. On the QUT computers, make sure you have external internet access. The workshop demonstrator can help you with this.

Once you're sure that you have internet access, run the following commands:

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

Once the installation commands have run successfully, try to avoid running them again. This may cause problems due to folder permission issues which arise when R tries to overwrite an existing instance of a package.

Installing a package *does not* mean you can immediately access the functionality in the package. You have to load them at the beginning of every R session to use them. To load packages, run the following commands:

```
library(dplyr)
```

```
##
## Attaching package: 'dplyr'

## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```r
library(tidyr)
library(ggplot2)
```

```
## Warning: package 'ggplot2' was built under R version 3.5.1
```

You now have access to the functionality of these packages!

Setting up a session in R

When people use R, they usually import their data from files that exist somewhere on their computer. These have to be brought into the R environment by running specific commands. However, we can simplify the data loading process by *setting the working directory*, which is the folder where R looks, by default, to find files which you refer to in your code.

There are at least three ways of setting the working directory.

1. By running the command `setwd()` with the path to a folder on your computer.
2. By running the command `setwd()` containing another command which allows you to select the relevant folder in Windows explorer (only on Windows OS),
3. By point-and-click methods inside RStudio.

The workshop demonstrator can help you with option (3). But for options (1) and (2), see the code blocks below:

```r
# setwd with path to folder written explicitly.
# You will need to change this to suit your own computer and needs.
setwd("D:/Dropbox/NRS")
# setwd with command to select a folder inside Windows explorer.
setwd(choose.dir())
```

This can be the difference between having to write

```r
# Without setwd
read.csv("D:/Dropbox/NRS/meuse.csv")
```

and

```r
# With setwd
read.csv("meuse.csv")
```

Try setting your working directory to the folder where you've unzipped the contents of the workshop pack.

Importing data from spreadsheets

Most data you will encounter lives in spreadsheets. You will have to access the spreadsheets from inside R if you want to use them in an R-based workflow.

For this workshop, we will assume that the data of interest lives in a comma-separated-values file (.csv). We will use the command `read.csv` to access these. However, there are functions in R for reading excel spreadsheets, text files, database tables, and even google forms. Since there is such a wide range of options, we can't deal with all of them. But .csv files are a good place to start, especially for scientific applications.

We assume you have set your working directory to the folder which contains meuse.csv. This file came with your workshop pack. To read this data into R, run the following command:

```
meuse <- read.csv(
  file = "meuse.csv",
  as.is = TRUE
)
```

Note, the `<-` is an assignment operator. It is equivalent to `=`. It just sets the value of the name on its left-hand side to the data/object/thing on its right-hand side.

If the code ran successfully, you won't see anything. To check that you got the result you expected, we have to run a simple command on the data. Let's use `names`; a function which displays all the column names of a dataset.

```
names(meuse)
```

```
##  [1] "x"       "y"       "cadmium" "copper"  "lead"    "zinc"    "elev"
##  [8] "dist"    "om"      "ffreq"   "soil"    "lime"    "landuse" "dist.m"
```

If you're seeing something like the above, everything is fine.

Writing out data

You can take an object inside R and write it to a spreadsheet-like file by using

```
write.csv(
  x = meuse,
  file = "a_new_csv.csv",
  row.names = FALSE
)
```

A more formal introduction to data structures in R

R is an object-oriented language, so most of its functionality is centred around 'classes' and 'objects'. A class is an abstraction, like when we refer to matrices and their properties, we refer to matrices in general and not necessarily to a particular matrix. An object comes from a class, just like

```
##           [,1]       [,2]
## [1,] 0.8288837 0.74256586
## [2,] 0.4470552 0.04158413
```

is an object of class `matrix`.

Classes define what sort of operations are possible with a particular data structure. In this workshop, we will be dealing with objects of class `data.frame`.

The `meuse` object we have created is a `data.frame`. We can query the class of objects using the command `class`.

```
class(meuse)
```

```
## [1] "data.frame"
```

The `data.frame` class is very versatile. Each `data.frame` contains a specific number of rows or observations

```
nrow(meuse)
```

```
## [1] 155
```

and columns or variables

```r
ncol(meuse)
```

```
## [1] 14
```

with set names for the variables

```r
names(meuse)
```

```
##  [1] "x"       "y"       "cadmium" "copper"  "lead"    "zinc"    "elev"
##  [8] "dist"    "om"      "ffreq"   "soil"    "lime"    "landuse" "dist.m"
```

Each column contains a different type of information. Some might contain discrete variables and others might contain numbers. But all these data types exist within the one object. If it's hard to take in, don't worry too much. Objects of class `data.frame` are so ubiquitous that most R functions can deal with them. The functions that we will see inside this workshop can all deal with `data.frame` objects.

Viewing data

You can view an entire object inside your console by simply running a command containing the name of the object, like this:

```r
meuse
```

But most datasets contain too many rows to display conveniently inside the console.

A more common approach is to look at the first and last six rows of a `data.frame` object using the `head` and `tail` commands.

```r
head(meuse) # first six
```

```
##        x      y cadmium copper lead zinc  elev       dist   om ffreq soil
## 1 181072 333611    11.7     85  299 1022 7.909 0.00135803 13.6     1    1
## 2 181025 333558     8.6     81  277 1141 6.983 0.01222430 14.0     1    1
## 3 181165 333537     6.5     68  199  640 7.800 0.10302900 13.0     1    1
## 4 181298 333484     2.6     81  116  257 7.655 0.19009400  8.0     1    2
## 5 181307 333330     2.8     48  117  269 7.480 0.27709000  8.7     1    2
## 6 181390 333260     3.0     61  137  281 7.791 0.36406700  7.8     1    2
##   lime landuse dist.m
## 1    1      Ah     50
## 2    1      Ah     30
## 3    1      Ah    150
## 4    0      Ga    270
## 5    0      Ah    380
## 6    0      Ga    470
```

```r
tail(meuse) # last six
```

```
##          x      y cadmium copper lead zinc  elev      dist  om ffreq soil
## 150 179030 330082     1.2     20   68  214 8.226 0.3749400 5.7     3    1
## 151 179184 330182     0.8     20   49  166 8.128 0.4238370 4.7     3    1
## 152 179085 330292     3.1     39  173  496 8.577 0.4238370 9.1     3    1
## 153 178875 330311     2.1     31  119  342 8.429 0.2770900 6.5     3    1
## 154 179466 330381     0.8     21   51  162 9.406 0.3586060 5.7     3    1
## 155 180627 330190     2.7     27  124  375 8.261 0.0122243 5.5     3    3
##     lime landuse dist.m
## 150    0      Ah    440
## 151    0      Am    540
## 152    0      Ah    520
```

```
## 153    0       Ah    350
## 154    0        W    460
## 155    0        W     40
```

These kinds of outputs are much more manageable for the console.

If you want to view your entire dataset in a scrollable window inside RStudio, you can run

```
View(meuse)
```

Accessing data and subsets of data

There are times when we only want or need one or two columns from a `data.frame` or a few rows. In these situations, you have to know how to access specific portions of your data.

Let's start with row and column indexing. You can refer to specific rows and columns of matrices and data frames like this:

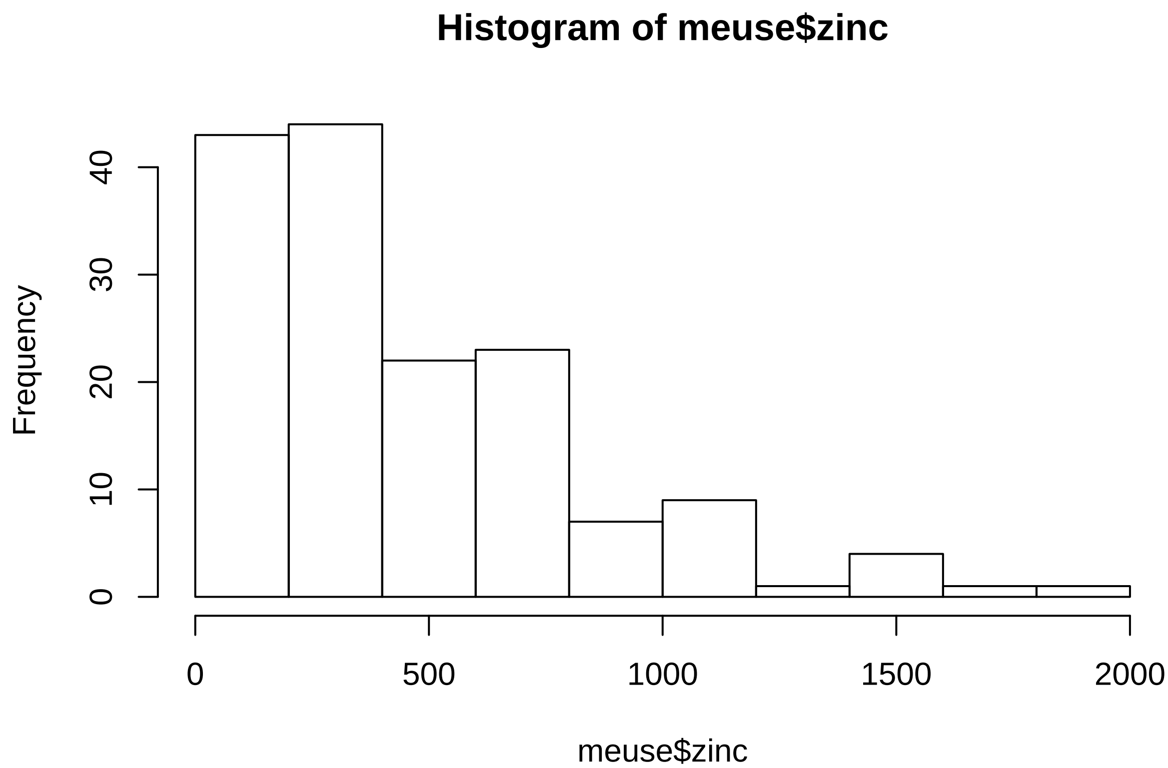
```
meuse[1,] # This gets the first row
meuse[,1] # This gets the first column
meuse[1,1] # This gets the first row of the first column
```

To access multiple rows or multiple columns, we extend the indices with vectors.

```
meuse[c(1,2,3), ]
meuse[1:3, ] # These get the first three rows
meuse[, c(1,2,3)]
meuse[, 1:3] # These get the first three columns
```

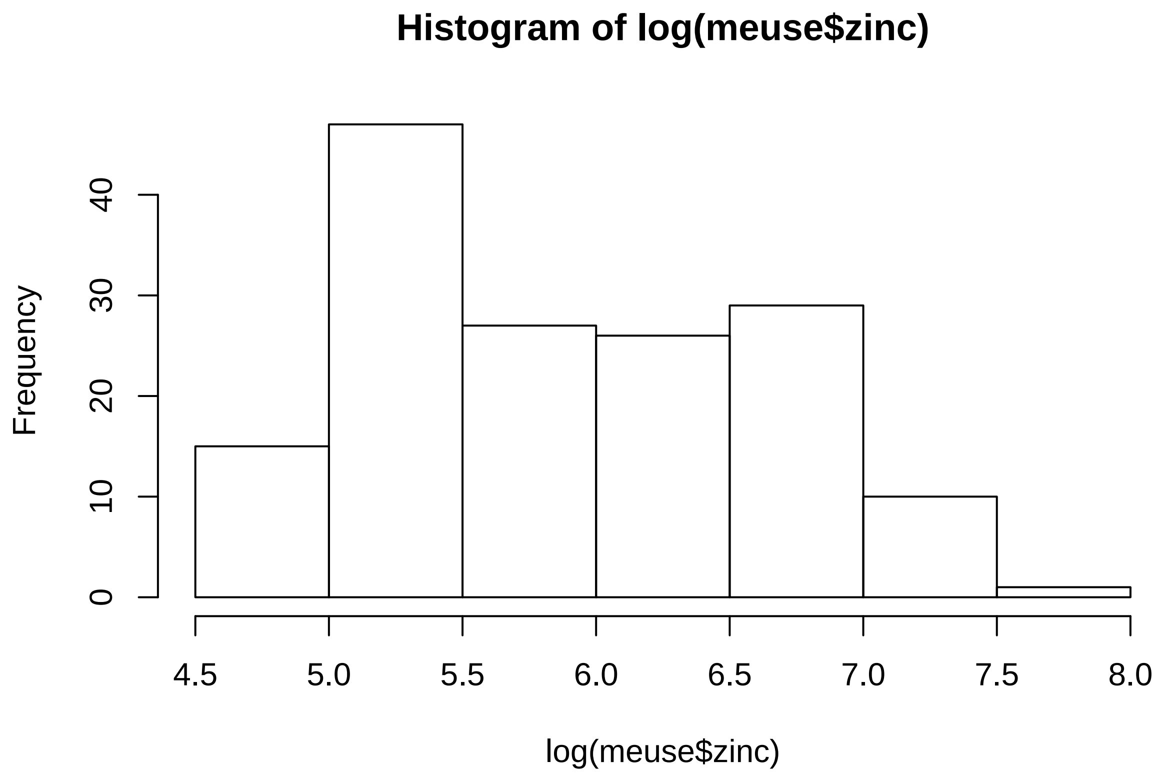
More often, it's useful to be able to access entire columns by name. Say, for example, you want to make a histogram of the zinc concentrations around the Meuse river. Then we write

```
hist(meuse$zinc)
```



We can do calculations on entire columns conveniently using this `$` notation. For example, for a histogram of log(zinc concentrations), we write

```
hist(log(meuse$zinc))
```



Another common situation is where you have to find some parts of your dataset which conform to a set of logical criteria. We can find these using the `subset` function. It is used like this:

```
meuse_sub <- subset(
  meuse, # the name of the dataset
  zinc < 200 # the logical condition
)

head(meuse_sub)
```

```
##         x      y cadmium copper lead zinc  elev     dist  om ffreq soil
## 10 181232 333168     1.6     24   80  183 9.049 0.309702 6.3     1    2
## 11 181191 333115     1.4     25   86  189 9.015 0.315116 6.4     1    2
## 26 181167 332778     1.5     22   76  194 8.973 0.429289 6.3     1    2
## 28 180973 332687     1.3     24   67  180 8.743 0.320574 4.4     1    2
## 30 181352 332946     1.5     21   65  180 9.043 0.489064 4.8     1    2
## 32 180878 332489     1.3     21   64  198 8.727 0.287957 1.0     1    2
##    lime landuse dist.m
## 10    0       W    420
## 11    0      Fh    400
## 26    0       W    530
## 28    0      Ag    400
## 30    0      Ag    630
## 32    0      Ag    390
```

We can combine multiple logical conditions. For example,

```r
meuse_sub <- subset(
  meuse,
  zinc < 200 & lead < 200
) # Must satisfy both conditions

head(meuse_sub)
```

```
##         x      y cadmium copper lead zinc  elev     dist  om ffreq soil
## 10 181232 333168     1.6     24   80  183 9.049 0.309702 6.3     1    2
## 11 181191 333115     1.4     25   86  189 9.015 0.315116 6.4     1    2
## 26 181167 332778     1.5     22   76  194 8.973 0.429289 6.3     1    2
## 28 180973 332687     1.3     24   67  180 8.743 0.320574 4.4     1    2
## 30 181352 332946     1.5     21   65  180 9.043 0.489064 4.8     1    2
## 32 180878 332489     1.3     21   64  198 8.727 0.287957 1.0     1    2
##    lime landuse dist.m
## 10    0       W    420
## 11    0      Fh    400
## 26    0       W    530
## 28    0      Ag    400
## 30    0      Ag    630
## 32    0      Ag    390
```

```r
meuse_sub <- subset(
  meuse,
  zinc < 200 | lead < 200
) # Can satistfy either condition

head(meuse_sub)
```

```
##        x      y cadmium copper lead zinc  elev      dist   om ffreq soil
## 3 181165 333537     6.5     68  199  640 7.800 0.1030290 13.0     1    1
## 4 181298 333484     2.6     81  116  257 7.655 0.1900940  8.0     1    2
## 5 181307 333330     2.8     48  117  269 7.480 0.2770900  8.7     1    2
## 6 181390 333260     3.0     61  137  281 7.791 0.3640670  7.8     1    2
## 7 181165 333370     3.2     31  132  346 8.217 0.1900940  9.2     1    2
## 8 181027 333363     2.8     29  150  406 8.490 0.0921516  9.5     1    1
##   lime landuse dist.m
## 3    1      Ah    150
## 4    0      Ga    270
## 5    0      Ah    380
## 6    0      Ga    470
## 7    0      Ah    240
## 8    0      Ab    120
```

Doing calculations with data

Often it is useful to be able to compute summary statistics of columns in your data. In situations like these, we can write

```r
mean(meuse$zinc)
```

```
## [1] 469.7161
```

```r
sd(meuse$zinc)
```

```
## [1] 367.0738
```

If we want to calculate entirely new columns of data based on existing columns of data, we can write things like

```
meuse$log_zinc <- log(meuse$zinc)
```

In this case, we refer to a column in the dataset (it doesn't have to exist already) and define it based on a calculation involving a column in the dataset which already exists.

Advanced summaries of data

Being able to compute summary statistics on columns of data is important. However, this isn't necessarily useful when your dataset contains distinct groups of observations and you need to be able to calculate statistics within each of those groups.

In the `meuse` dataset, there are three types of soil (just labelled 1, 2, 3) where concentrations of various heavy metals were observed. We might want to know the mean concentration of lead, cadmium and copper in each of the soil types.

To do this, we need to use the `group_by` and `summarise` functions inside the package `dplyr`.

In the first step, we identify the column which contains the grouping variable for our dataset. In this case, it's called 'soil'.

```
meuse_soil_groups <- group_by(
  meuse, # The dataset
  soil   # The grouping variable
)
```

Note that there is superficially nothing different about the new `data.frame` object. The changes have occurred under the hood, in the internal logical of R, so that when we perform the summaries on cadmium, lead and copper concentrations, we get the results we want.

The next step involves using the `summarise` function like so

```
summarise(
  meuse_soil_groups, # The grouped dataset
  Mean_Cd = mean(cadmium),
  Mean_Pb = mean(lead),
  Mean_Cu = mean(copper)
  # Format: name = calculation(column)
)
```

```
## # A tibble: 3 x 4
##    soil Mean_Cd Mean_Pb Mean_Cu
##   <int>   <dbl>   <dbl>   <dbl>
## 1     1   4.30    191.     47.3
## 2     2   1.73     98.9    30.1
## 3     3   0.558    58.2    23
```

Note that this is quite different to

```
summarise(
  meuse,
  Mean_Cd = mean(cadmium),
  Mean_Pb = mean(lead),
  Mean_Cu = mean(copper)
)
```

```
##    Mean_Cd  Mean_Pb  Mean_Cu
## 1 3.245806 153.3613 40.31613
```

We can do more than just calculate means. Virtually any function is allowed to be used inside `summarise`, as long as its output is a single value. To demonstrate, if we want the mean, median, standard deviation and sum of the lead concentrations in each soil group, we would write

```
summarise(
  meuse_soil_groups,
  Mean = mean(lead),
  Median = median(lead),
  StdDev = sd(lead),
  Sum = sum(lead)
)
```

```
## # A tibble: 3 x 5
##    soil  Mean Median StdDev   Sum
##   <int> <dbl>  <dbl>  <dbl> <int>
## 1     1 191.    170    119.  18522
## 2     2  98.9    80     58.7  4550
## 3     3  58.2    48.5   24.7   699
```

Note also that the names we give to the new columns are completely arbitrary, as long as they are valid column names; i.e. they don't contain special characters or start with numbers.

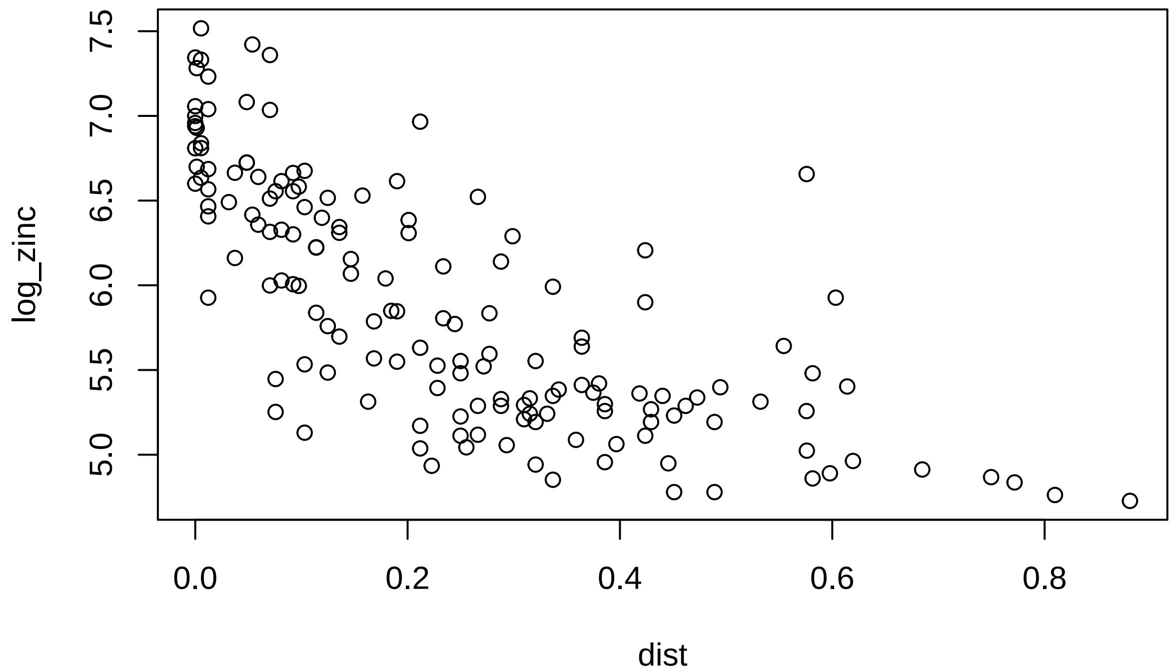
Graphical summaries of data

One of the main attractions of R is the ability to create publication-quality figures with ease. In this section, you will learn about two different ways to plot data in R:

1. `base` plot
2. `ggplot2`

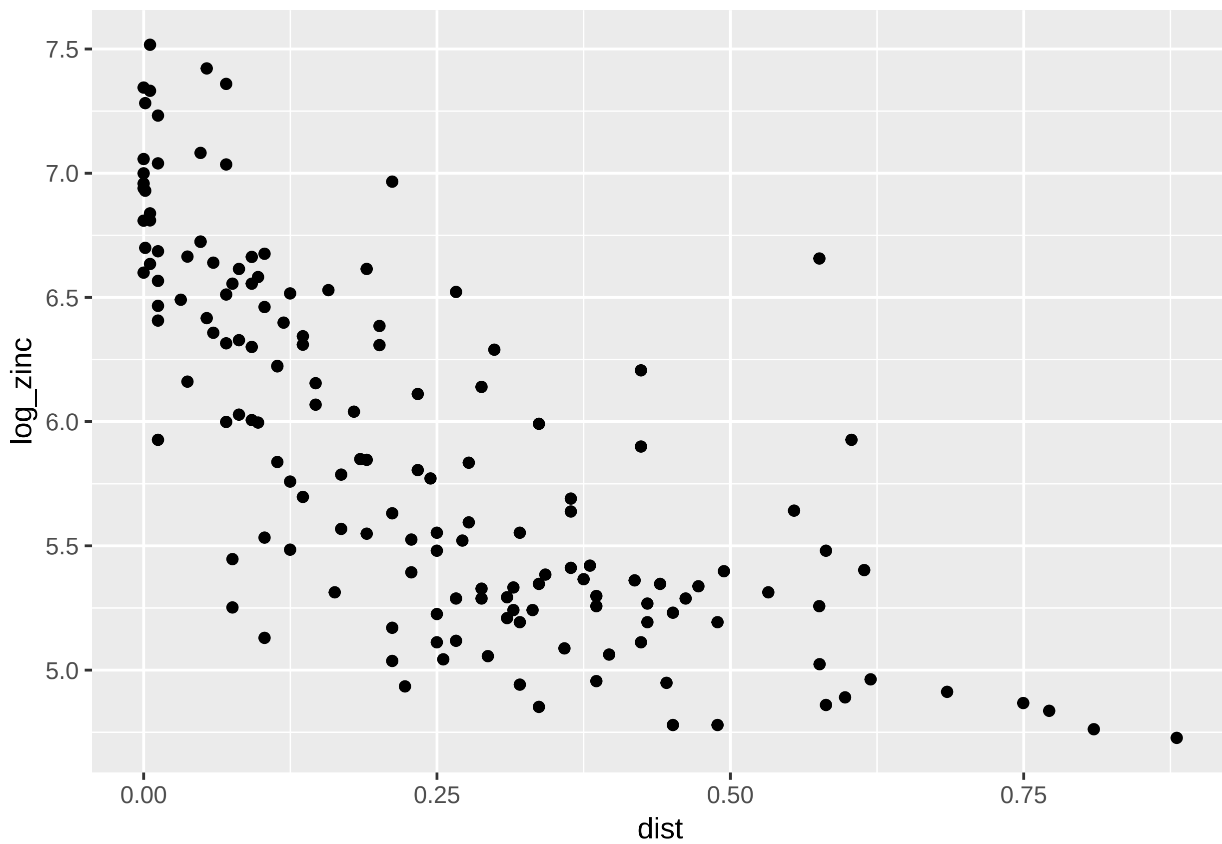
Base plot is the graphics device that comes pre-packaged with every installation of R. You can learn it quickly but it takes a long time to master, and for complicated tasks it is very difficult to use. It is characterised by the use of single-line commands like

```
plot(log_zinc ~ dist, data = meuse)
```



The package ggplot2 provides a different framework for plotting, which takes a bit of time to learn but, once you learn it, you can create beautiful plots in very little time. It is characterised by the use of longer, multi-line commands that are built up layer-by-layer to produce a plot.

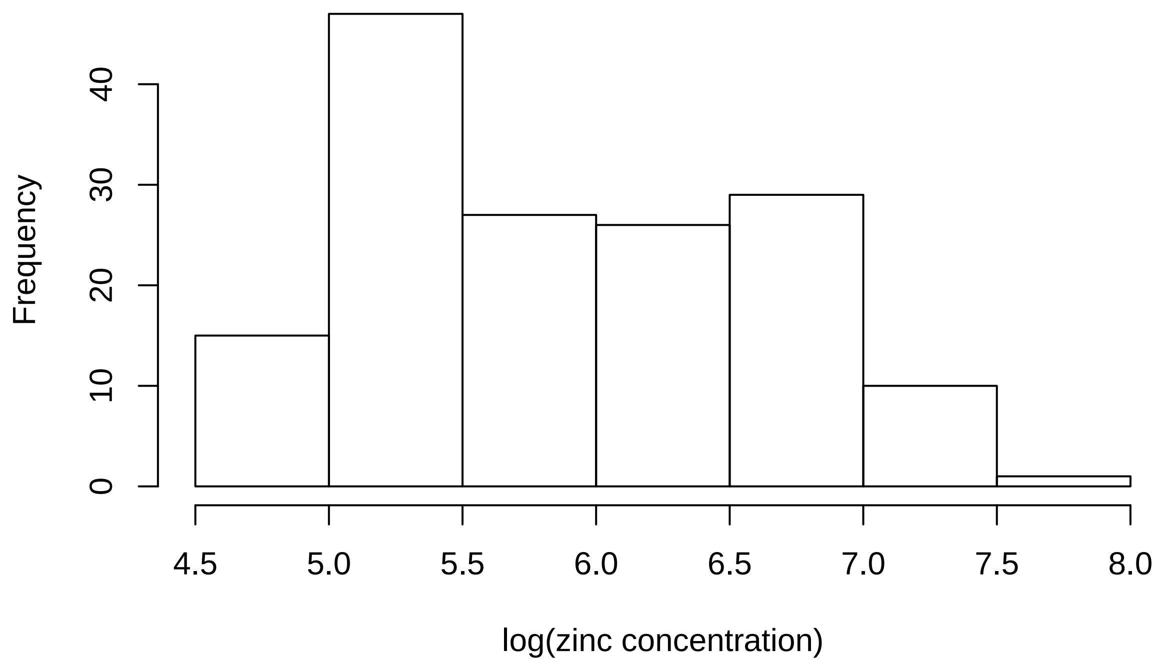
```
ggplot(
  data = meuse,
  aes(x = dist, y = log_zinc)
) +
  geom_point()
```



Histograms

You can make histograms in base plot by using `hist`.

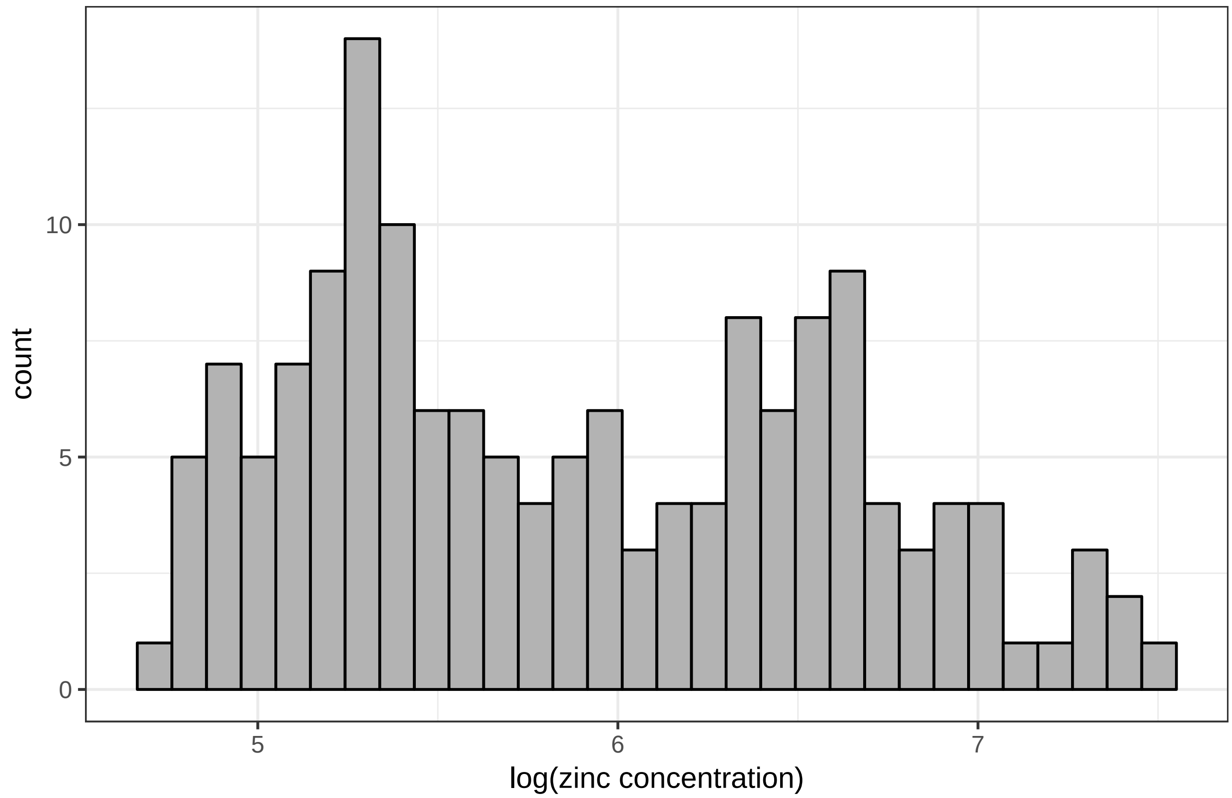
```
hist(
  meuse$log_zinc, # A single column of data
  xlab = "log(zinc concentration)",
  # xlab for x-axis label
  main = ""
  # main for plot title
)
```



In ggplot2, the plot is opened using `ggplot` and then the histogram is drawn using `geom_histogram`.

```
ggplot(
  data = meuse, # The dataset
  aes(
    x = log_zinc # Name of column to draw in histogram
  )
) +
  geom_histogram(
    col = "black", # for bar outline
    fill = "grey70" # for bar filling
  ) + # Draws a histogram
  theme_bw() + # Controls background display
  labs(
    x = "log(zinc concentration)",
    # x for x-axis label
    title = ""
    # title for plot title
  )
```

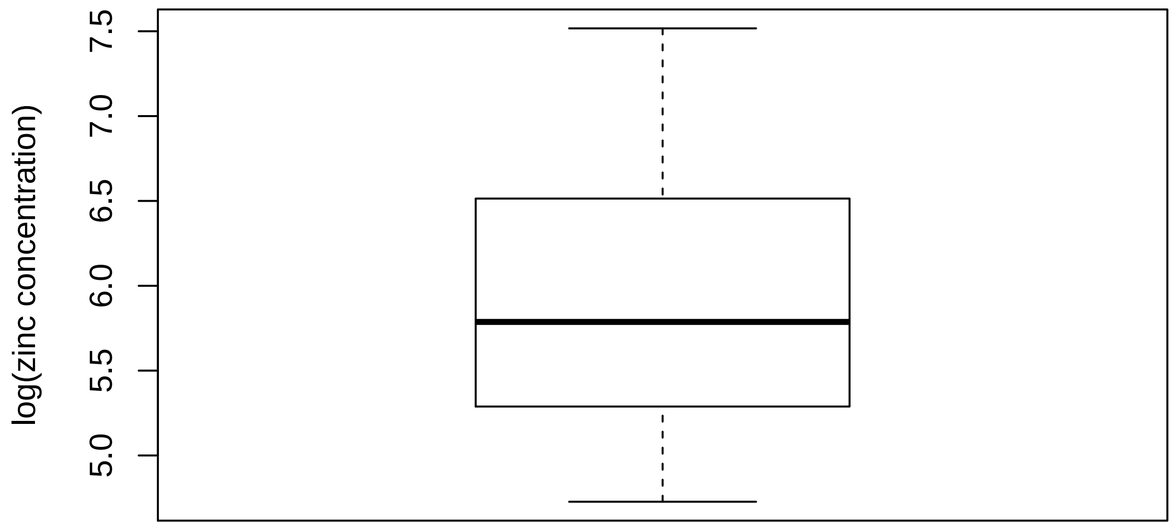
```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



Boxplots

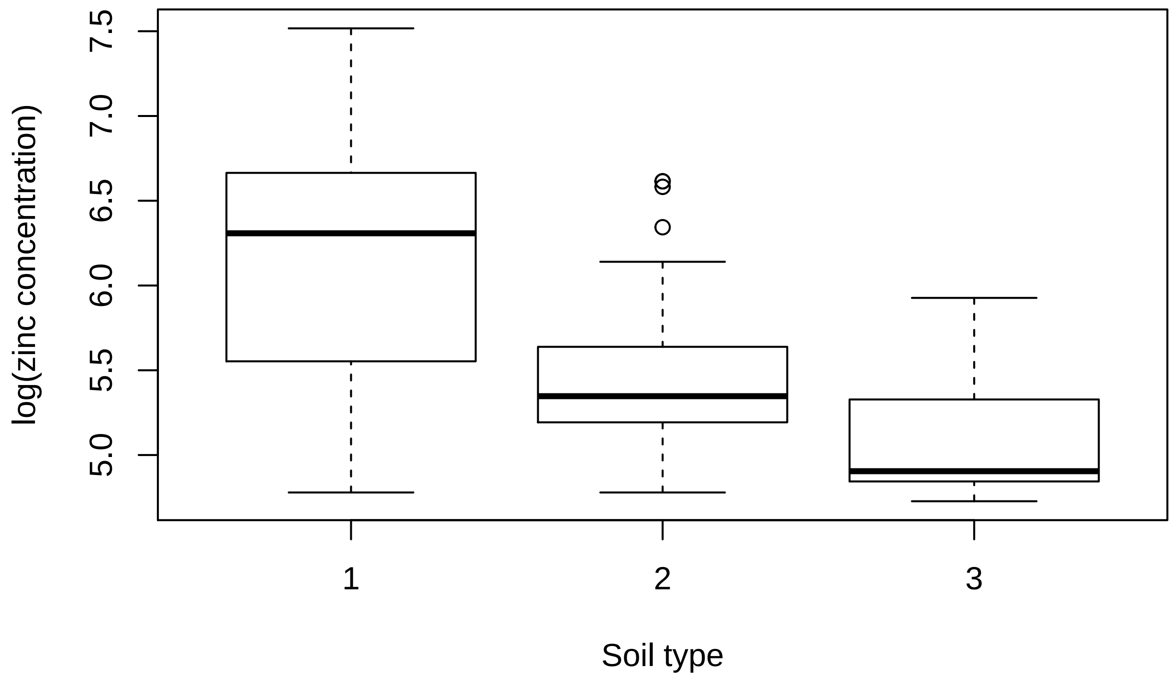
Boxplots for a single variable can be drawn using `boxplot`.

```
boxplot(
  meuse$log_zinc,
  main = "",
  xlab = "",
  ylab = "log(zinc concentration)"
)
```



We can split the boxplots out by a discrete variable, also.

```
boxplot(
  log_zinc ~ soil,
  data = meuse, # the dataset
  main = "",
  xlab = "Soil type",
  ylab = "log(zinc concentration)"
)
```

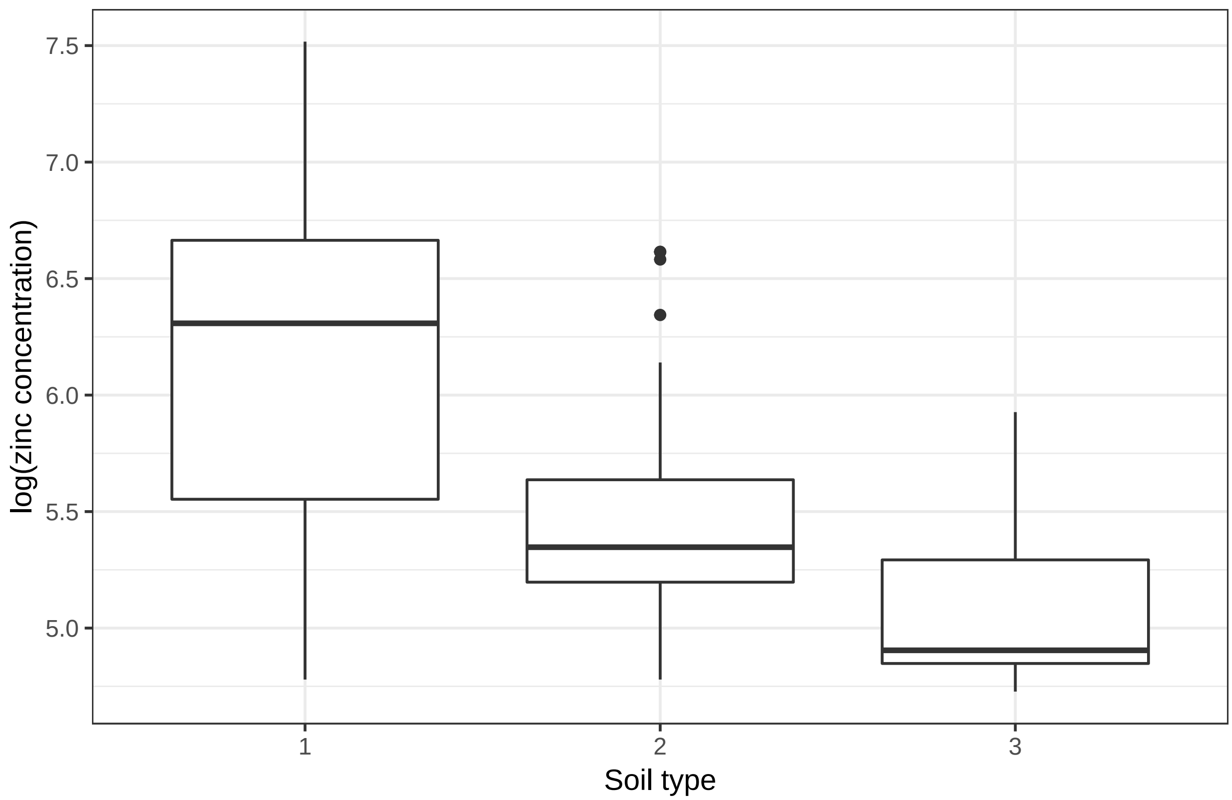


In ggplot2, we can do the same things but the single-variable boxplot requires more code to create.

```
ggplot(
  data = meuse,
  aes(x = 0, y = log_zinc)
) +
  geom_boxplot() +
  theme_bw() +
  labs(
    x = "",
    y = "log(zinc concentration)",
    title = ""
  ) +
  theme(
    axis.ticks.x = element_blank(),
    axis.text.x = element_blank()
  ) # Removes text from x axis
```

However, it is slightly more elegant for the boxplot split out by a factor.

```
ggplot(
  data = meuse,
  aes(x = as.factor(soil), y = log_zinc)
) +
  geom_boxplot() +
  theme_bw() +
  labs(
    x = "Soil type",
    y = "log(zinc concentration)",
    title = ""
  )
```



Scatter plots

Simple scatter plots are easy to create in base plot and ggplot2. We have already seen examples of these.

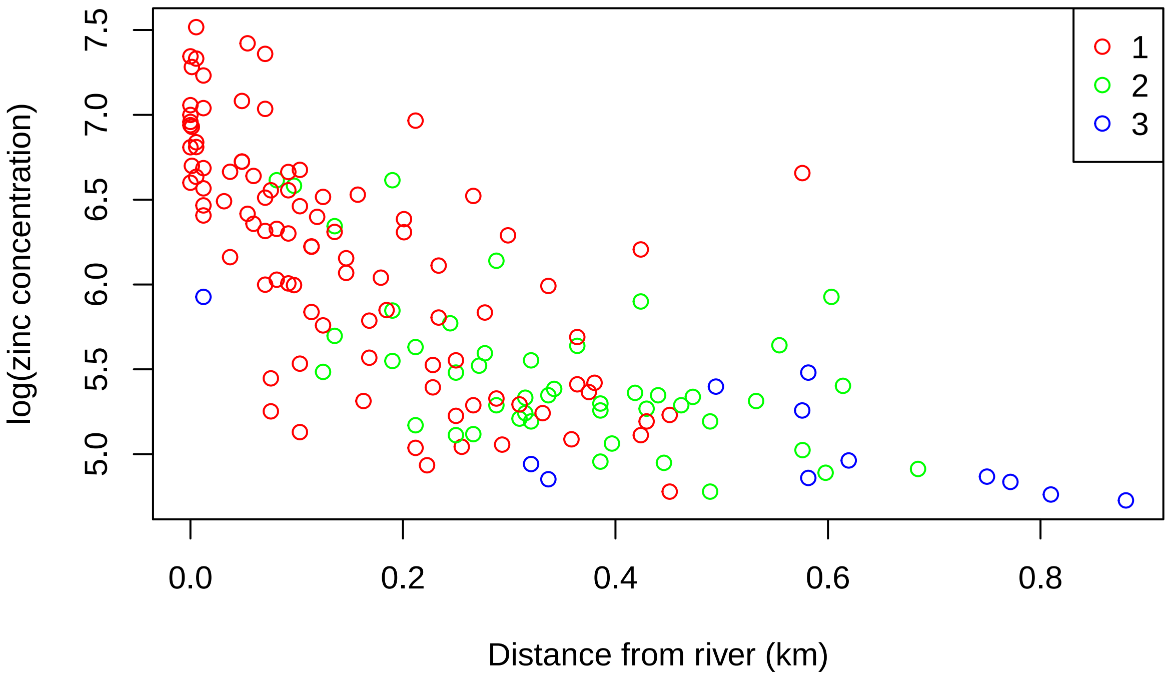
However, when it comes to more advanced scatter plots, with points coloured by a third variable, and with a legend, ggplot2 is much more elegant than base plot.

```
# base plot

meuse$col <- NA # must create separate column with colour names
meuse$col[meuse$soil == 1] <- "red"
meuse$col[meuse$soil == 2] <- "green"
meuse$col[meuse$soil == 3] <- "blue"


plot(
  log_zinc ~ dist,
  data = meuse,
  col = meuse$col,
  xlab = "Distance from river (km)",
  ylab = "log(zinc concentration)",
  main = ""
)
legend(
  x = "topright",
  col = c("red", "green", "blue"),
```

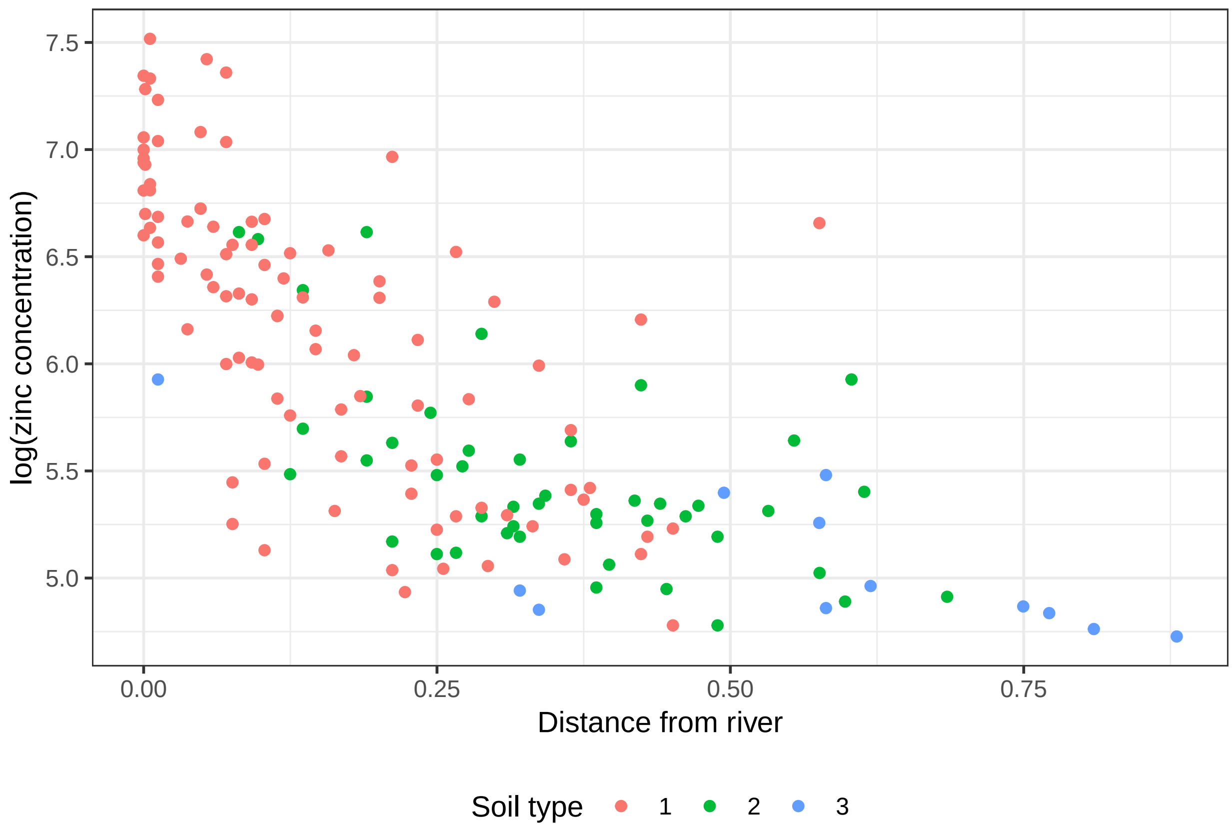
```
  pch = c(1, 1, 1),
  legend = c("1", "2", "3")
) # Legend must be plotted separately
```




```
# ggplot2

ggplot(
  data = meuse,
  aes(
    x = dist,
    y = log_zinc
  )
) +
  geom_point(
    aes(
      col = as.factor(soil) # Simply map variable to colour
    )
  ) +
  theme_bw() +
  labs(
    x = "Distance from river",
    y = "log(zinc concentration)",
    col = "Soil type" # Label legend
  ) +
  theme(
    legend.position = "bottom" # legend placement
```

```
)
```



Simple modelling of data

Statistical modelling is a huge area and a can of worms, so we won't go too far into it. But here, we will do some linear modelling of data.

Linear models are

- A general case of a few very old methods (T-tests, ANOVA)
- Useful for statistical modelling of general trends
- Very easy to compute
- Make assumptions which can be restrictive

In R, linear models are computed using the `lm` function.

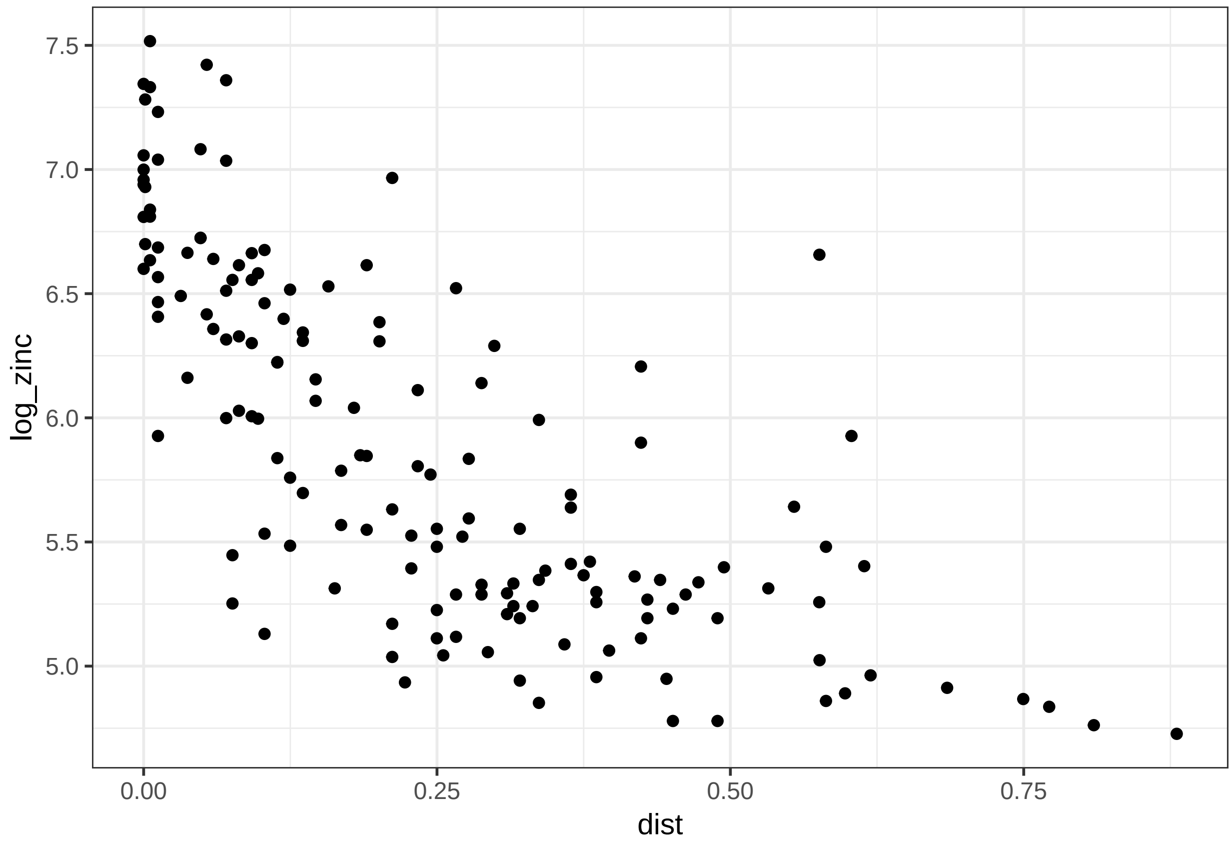
Running a model

Let's say we are interested in the correlation between log(zinc) concentrations in soil and distance away from the river. We expect it to decrease with distance because zinc is a heavy metal that settles quickly from the water column during floods.

It is usually a good idea to make a plot first, to see whether the relationship is clear.

```
meuse_plot <- ggplot(data = meuse, aes(x = dist, y = log_zinc)) +
  geom_point() +
```

```
  theme_bw()
meuse_plot
```

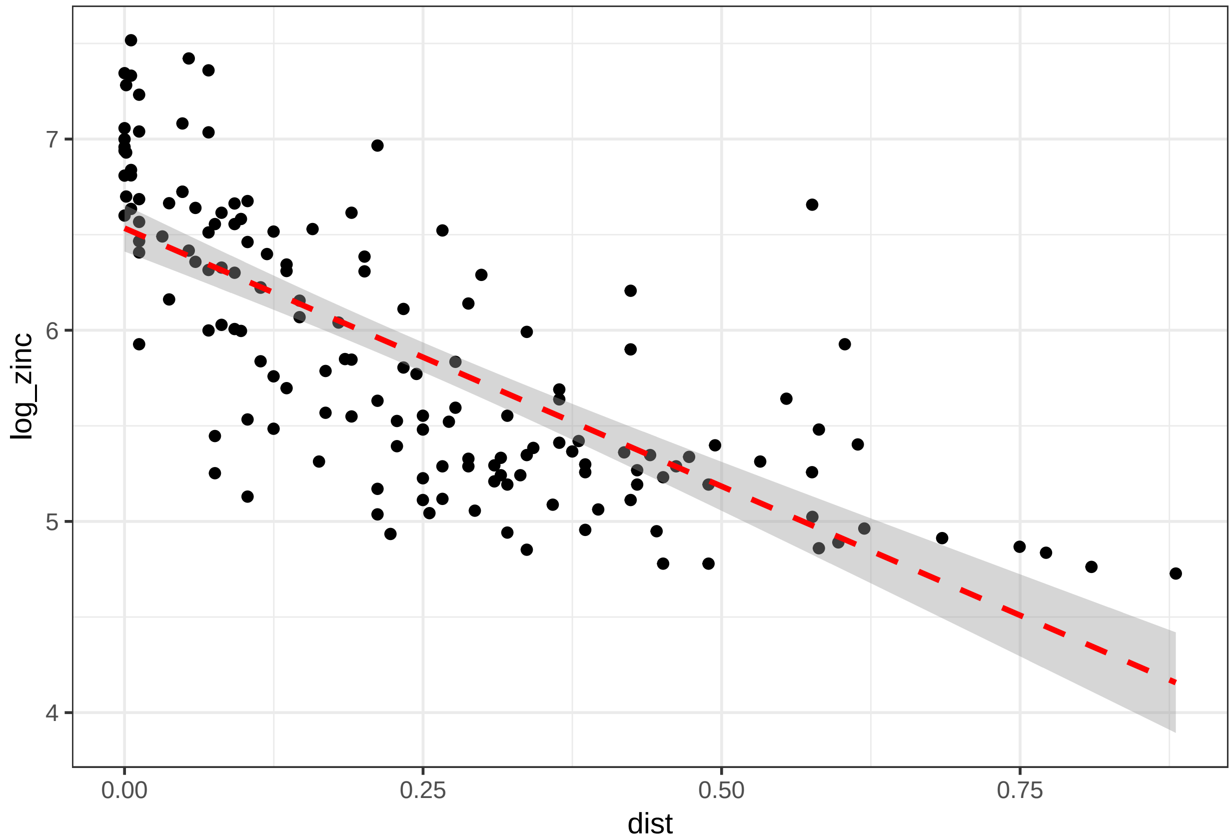


This plot confirms our general ideas. So now we can run a model using this code:

```
model <- lm(
  log_zinc ~ dist, # Formula: y ~ x1 + x2 + x3 + ...
  data = meuse      # The data set
)
```

The model looks like this:

```
meuse_plot +
  geom_smooth(method = lm, lty = 2, col = "red")
```



If we want a statistical summary of our model, we can run the following code. There is a lot going on here and it will be explained in the workshop.

```
summary(model)
```

```
##
## Call:
## lm(formula = log_zinc ~ dist, data = meuse)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -1.12573 -0.36442 -0.00161  0.31932  1.67774
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  6.53380    0.06172  105.87   <2e-16 ***
## dist        -2.69991    0.19874  -13.59   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.4876 on 153 degrees of freedom
## Multiple R-squared:  0.5468, Adjusted R-squared:  0.5438
## F-statistic: 184.6 on 1 and 153 DF,  p-value: < 2.2e-16
```

Model diagnostics

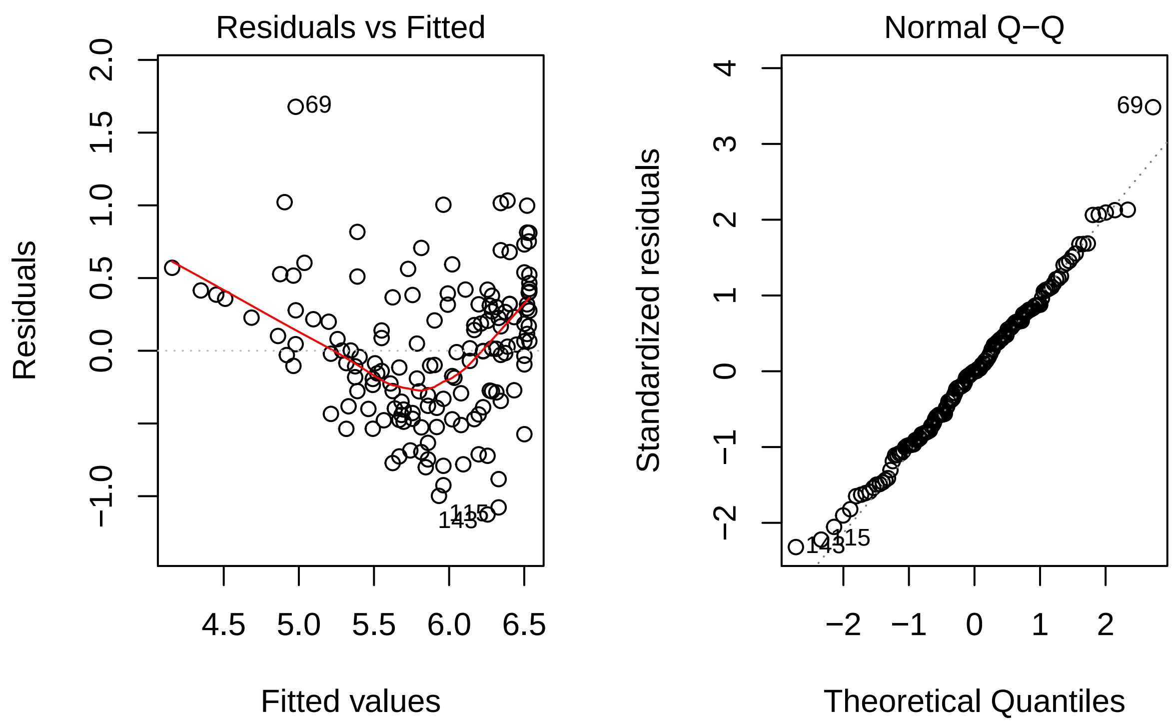
Linear modelling makes a few assumptions, namely that:

- The errors (residuals) are normally distributed
- The errors are homoscedastic

We need to check these to see whether a linear model really is appropriate for our data. This is usually done graphically using a normal qq-plot and a residuals vs fitted plot.

Thankfully, we can get these really easily for our linear models with

```
par(mfrow = c(1, 2))
plot(model, which = 1:2)
```



Programming elements in R

R is a programming language, so it comes with all the basic programming elements such as loops.

Ifs

Sometimes you want code to execute one way if a certain logical condition is true and another if it is not. Here, you need an if statement. In R, these look like

```
x <- runif(1)
x # show what x is
```

```
## [1] 0.04198193
```

```r
logical_condition <- x > 0.5
logical_condition # is x greater than .5?
```

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

```r
if(logical_condition){
  print("Yee") # executes if true
} else {
  print("Haw") # executes if false
}
```

```
## [1] "Haw"
```

Writing a loop

The most common type of loop is a for loop. In R, they are set up like this:

```r
for(i in 1:10){
  print((1:i)^2)
}
```

```
## [1] 1
## [1] 1 4
## [1] 1 4 9
## [1]  1  4  9 16
## [1]  1  4  9 16 25
## [1]  1  4  9 16 25 36
## [1]  1  4  9 16 25 36 49
## [1]  1  4  9 16 25 36 49 64
## [1]  1  4  9 16 25 36 49 64 81
##  [1]   1   4   9  16  25  36  49  64  81 100
```

Pretty simple right? In this case, there's an index i which takes values from 1 through to 10 and in each iteration, we display all the squares of the numbers from 1 through to i.

These can be used to accomplish great things.

Writing a function

If we have a process that we need to do repeatedly, we can set up a function; a set of code that accepts some inputs and returns an output after performing a fixed operation.

In R, the anatomy of a function is

```r
do_something_useless <- function(x, y, z){
  # This function takes three numbers x, y, z and computes
  # x + y * z^2 - y^2 * z
  return(x + y * z^2 - y^2 * z)
}
```

Once we define a function, we can use it by writing down its name and giving it its inputs.

```r
do_something_useless(1, 2, 3)
```

```
## [1] 7
```

Writing our own cumsum function

Say we want to write a function that takes a vector of numbers and computes the cumulative sum at each element of that vector. This is how we would do it in R:

```
cumsum_ <- function(
  vec # Our vector of numbers
){

  # Compute how long the vector is
  number_elements <- length(vec)

  # Assign an empty vector to store results
  cumulative_sum <- vector("numeric", number_elements)

  # Run a for loop
  for(i in 1:number_elements){
    # Assign i'th cumulative sum
    cumulative_sum[i] <- sum(vec[1:i])
  }

  # Spit out result
  return(cumulative_sum)

}
```

Now we want to test our function. R has its own inbuilt `cumsum` so we will check our function's answers against that.

```
# Test 1:
cumsum_(1:3) # our function
```

```
## [1] 1 3 6
```

```
cumsum(1:3) # inbuilt
```

```
## [1] 1 3 6
```

```
# Test 2:
cumsum_(c(608, 1, 999))
```

```
## [1]  608  609 1608
```

```
cumsum(c(608, 1, 999))
```

```
## [1]  608  609 1608
```

```
# Test 3:
cumsum_(c(-1, 1, 2))
```

```
## [1] -1  0  2
```

```
cumsum(c(-1, 1, 2))
```

```
## [1] -1  0  2
```