

# Programming for E&BI

Christoph Walsh

# What is R and RStudio?

- R is a *programming language* which specializes in statistical computing and graphics.
  - ▶ A programming language is a way to instruct a computer to perform operations via written text.
  - ▶ When programming, we need to be very exact. Otherwise the computer will throw an error or do something we didn't intend it to do!
- RStudio is a desktop application where you can write R code, execute R programs, and view plots created by R.

# Why Learn R?

R has many benefits over some alternatives:

- R is free and open source.
- Large active community creating packages and providing support.
- Easier to learn than some alternatives.
- Availability of RStudio for free.
- R is extremely versatile. For example, these slides were made in RStudio!

# Installing R

- To download and install R, go to <https://mirror.lyrahosting.com/CRAN>.
- To download and install RStudio, go to <https://posit.co/download/rstudio-desktop/>.

# The R Console

- We can perform calculations in the Console tab in RStudio.
- At the most basic level, we can use it as a simple calculator:
  - ▶ Add:  $2 + 3$
  - ▶ Subtract:  $5 - 3$
  - ▶ Multiply:  $2 * 3$
  - ▶ Divide:  $3 / 2$
  - ▶ Exponentiation:  $2^3$
  - ▶ Combining operations:  $(2 + 4) / (4 * 2)$

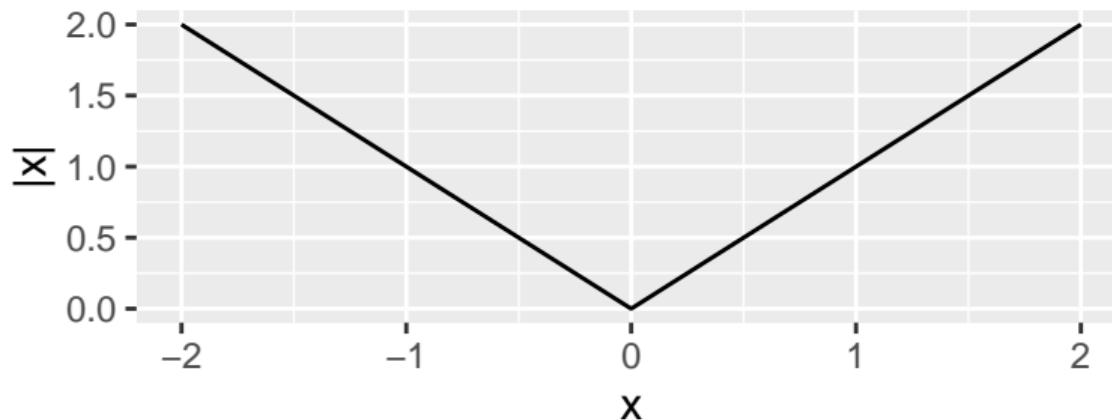
# R Functions

- Just like Excel, R has functions.
- These functions work in a very similar way:
  - ▶ Functions have names, and we provide *arguments* to functions inside parentheses.
- In the next few slides we will see some examples of mathematical functions and how to evaluate them in R.

## Absolute Value

- The absolute value function turns negative numbers into positive ones and has no effect on zero or positive numbers.

$$|x| = \begin{cases} x & \text{if } x \geq 0 \\ -x & \text{otherwise} \end{cases}$$



## Absolute Value in R

- The absolute value function in R is called `abs`. We can use it as follows:

```
abs(-2)
```

```
[1] 2
```

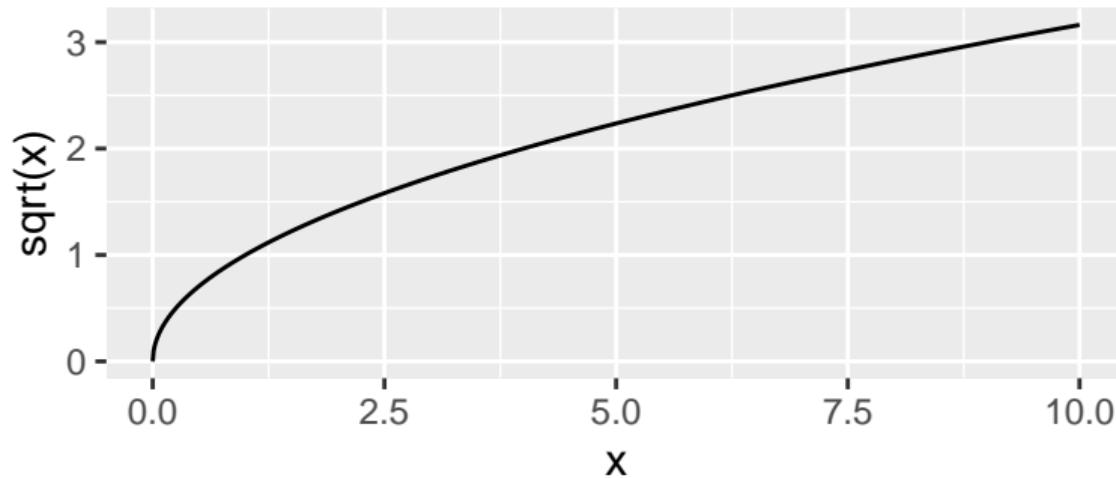
```
abs(3)
```

```
[1] 3
```

- We can see help about a function using `help(abs)` or `?abs`

# Principal Square Roots

- The square root of a number is the  $y$  that solves  $y^2 = x$ .
- If  $x = 4$ , both  $y = -2$  and  $y = 2$  solve  $y^2 = x$ .
- The principal square root is the positive  $y$  solving this.



## Principal Square Roots in R

- We can use the `sqrt( )` function or take to the power of  $\frac{1}{2}$  to take the square root.

```
sqrt(9)
```

```
[1] 3
```

```
9^(1/2)
```

```
[1] 3
```

- The cubed root is the  $y$  that solves  $y = x^3$ . We can calculate this in R by taking the power of  $\frac{1}{3}$ .

```
8^(1/3)
```

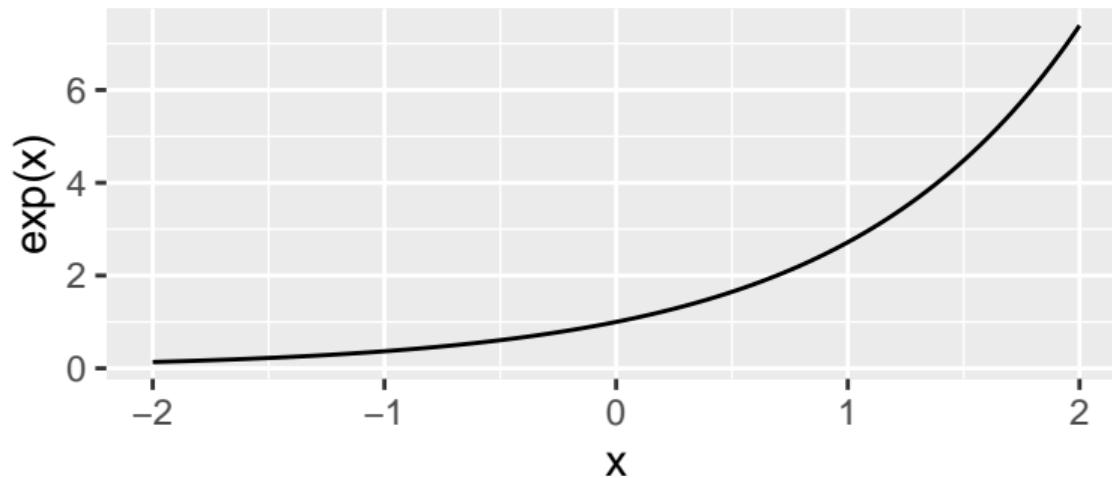
```
[1] 2
```

# The Exponential Function

The exponential function is a very common function in statistics:

$$e^x = \lim_{n \rightarrow \infty} \left(1 + \frac{x}{n}\right)^n$$

Note: you don't need to know this function for the exam.



# The Exponential Function in R

- In R we use the `exp()` function to calculate the exponential of a number:

```
exp(0)
```

```
[1] 1
```

```
exp(1)
```

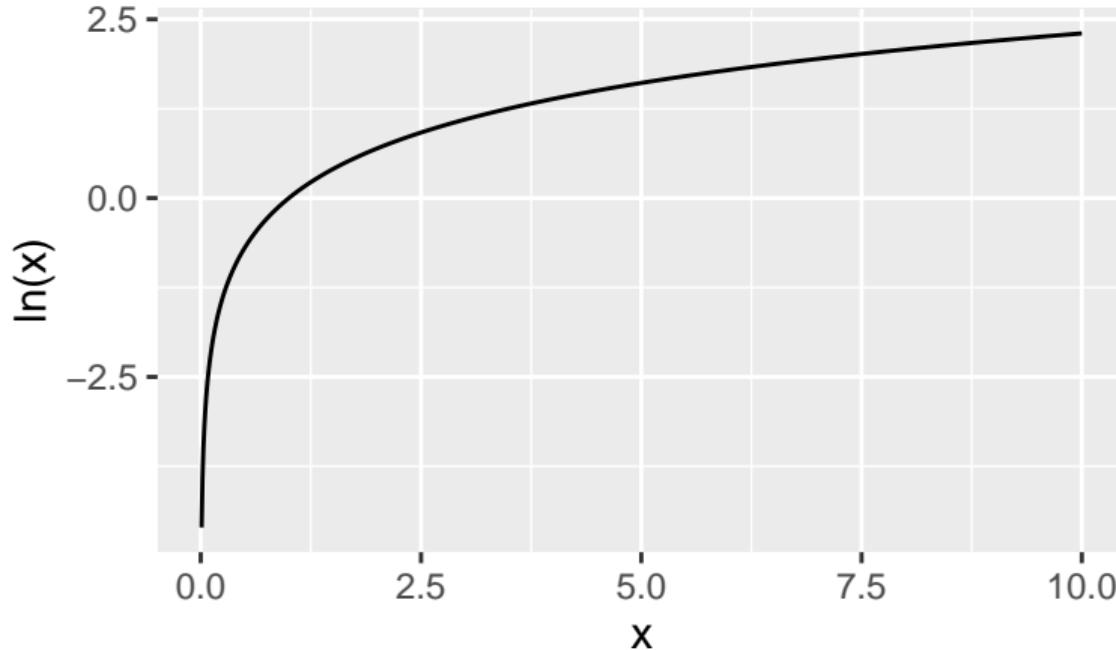
```
[1] 2.718282
```

# The Logarithm

- Another common mathematical function is the logarithm, which is like the reverse of exponentiation.
- The log of a number  $x$  to a base  $b$ , denoted  $\log_b(x)$ , is the number of times we need to multiply  $b$  by itself to get  $x$ .
- For example,  $\log_{10}(100) = 2$ , because  $10 \times 10 = 100$ . We need to multiply the base  $b = 10$  by itself twice to get to  $x = 100$ .

# The Natural Logarithm

- A special logarithm is the natural logarithm,  $\log_e(x)$ , which is the logarithm to the base  $\exp(1) = e^1 \approx 2.7183$ . This is also written as  $\ln(x)$ .



# The Logarithm in R

- In R we use the `log()` function to calculate the natural logarithm:

```
log(1)
```

```
[1] 0
```

- We can calculate the logarithm of a number to a different base using the `base` argument. For example, for  $\log_{10}(100)$ :

```
log(100, base = 10)
```

```
[1] 2
```

## The Assignment Operator: <-

- We can store objects using the *assignment operator*, <-.
- For example:

```
a <- 2  
b <- 3
```

- $a$  and  $b$  are then visible in the **Environment** tab in RStudio.
- We can then use  $a$  and  $b$  for calculations:

```
a + b  
[1] 5
```

# Common Object Types: Numerical, Logical and Character

- Numerical vectors (list of numbers):

```
a <- c(1, 3, 7, 2)
```

- Logical vectors (list of Yes/No responses):

```
a <- c(TRUE, FALSE, TRUE, TRUE)
```

- Character vectors (list of letters/words):

```
a <- c("abc", "def", "ghi")
```

## Common Object Types: Factors

- Categorical variables are stored in R as “factors”.
- For example, imagine a survey asking how long it took for people to get to campus (in minutes) and their travel mode (one of “cycle”, “train”, or “walk”).
- You could store these variables as a numerical and a character variable:

```
time <- c(25, 20, 15, 10, 17, 30)
travel_mode <- c("train", "train", "walk",
                 "cycle", "walk", "train")
```

- But if we tell R that this travel mode variable is a categorical variable, it will be useful for operations that we will learn later. We can convert this to a factor using the `factor` function:

```
travel_mode <- factor(travel_mode)
```

## Common Object Types: Data Frames and Lists

- A `data.frame` collects vectors of the same length into a single dataset.
- We can convert the `time` and `travel_mode` variables into a `data.frame` as follows:

```
df <- data.frame(travel_mode, time)
```

- We can also “View” it in RStudio by clicking on it in the Environment tab.
- A `data.frame` is actually a special type of list:

```
my_list <- list(x = 1:3, y = TRUE, z = c("a", "b"))
```

- While vectors must have all elements of the same type (numeric/logical/character/factor), lists can have elements of any type *and any length*.
- Dataframes can have columns of different types, but all columns must have the same length.

# Indexing Vectors with Numbers

- The elements of a vector like:

```
a <- c(1, 2, 4, 3, 2)
```

are indexed 1-5.

- We can extract elements from this vector using these indices:

```
a[3]
```

```
[1] 4
```

```
a[c(1, 3, 4)]
```

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

## Indexing Vectors with Logical Vectors

- We can also index vectors using an equal-length logical vector, where we extract only the elements that are TRUE:

```
a <- c(1, 2, 4, 3, 2)
a[c(TRUE, FALSE, TRUE, TRUE, FALSE)]
```

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

# Sequences

We can create vectors that are sequences of numbers in different ways:

```
1:10
```

```
[1] 1 2 3 4 5 6 7 8 9 10
```

```
10:1
```

```
[1] 10 9 8 7 6 5 4 3 2 1
```

```
seq(from = 10, to = 100, by = 10)
```

```
[1] 10 20 30 40 50 60 70 80 90 100
```

```
seq(from = 0, to = 1, length.out = 5)
```

```
[1] 0.00 0.25 0.50 0.75 1.00
```

## Repeating Numbers and Vectors

To save time typing, we can repeat numbers and vectors with the `rep()` function:

```
rep(1, times = 20)
```

```
[1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

```
rep(1:3, times = 4)
```

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

```
rep(1:3, each = 4)
```

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

## Summary Statistics for Vectors

```
a <- 1:10
```

```
a
```

```
[1] 1 2 3 4 5 6 7 8 9 10
```

```
length(a)
```

```
[1] 10
```

```
min(a)
```

```
[1] 1
```

```
max(a)
```

```
[1] 10
```

## Summary Statistics for Vectors

```
mean(a)
```

```
[1] 5.5
```

```
median(a)
```

```
[1] 5.5
```

```
sum(a)
```

```
[1] 55
```

```
summary(a)
```

| Min. | 1st Qu. | Median | Mean | 3rd Qu. | Max.  |
|------|---------|--------|------|---------|-------|
| 1.00 | 3.25    | 5.50   | 5.50 | 7.75    | 10.00 |

## Tabulate a Vector

```
a <- c(1, 3, 2, 4, 4, 2, 4)  
table(a)
```

a

1 2 3 4

1 2 1 3

## Example: A Linear Mortgage

- You take out a linear mortgage for €150,000 with a 15-year term and an interest rate of 7%. How much interest will you pay on this loan?
- You pay  $\text{€}150,000/15 = \text{€}10,000$  per year off the loan principal.
- The debt follows a sequence 150,000, 140,000, 130,000, ..., 10,000, 0.
- The interest of 7% each period is on the remaining debt.
- We can get the total interest payments with R using:

```
debt <- seq(from = 150000, to = 0, by = -10000)
interest <- debt * 0.07
sum(interest)
```

```
[1] 84000
```

## Example: Sum of Geometric Sequences

- The parents of a child choose to invest €1,250 at 5% interest each year from when the child is born until (and including) their 20th birthday. On the 20th birthday, how much will the investment be worth?
- The first investment is worth:  $1250 \times (1.05)^{20} = 3316.62$ .
- The second investment is worth:  $1250 \times (1.05)^{19} = 3158.69$ .
- The last investment won't have received interest and is worth  $1250 \times (1.05)^0 = 1,250$ .

```
sum(1250 * 1.05^(20:0))
```

```
[1] 44649.06
```

This is the same as using the solution to the sum of a geometric sequence:

```
1250 * (1.05^21 - 1) / 0.05
```

```
[1] 44649.06
```

# Comparing Numerical Vectors

```
a <- 1:5  
b <- 5:1  
a
```

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

```
b
```

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

```
a < b
```

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

```
a <= b
```

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

# Comparing Numerical Vectors

```
a == b
```

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

```
a != b
```

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

```
a >= b
```

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

```
a > b
```

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

## Comparing Logical Vectors

```
a <- c(TRUE, TRUE, FALSE, FALSE)  
b <- c(TRUE, FALSE, TRUE, FALSE)
```

We use `&` for *logical AND*, `|` for *logical OR* and `!` for *logical NOT*.

```
a & b
```

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

```
a | b
```

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

```
!a
```

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

# R Scripts

- When working on a project, it's often better to write all the commands you want to run in an *R script* instead of directly into the console:
  - ▶ It documents and saves your work.
  - ▶ It makes your work shareable and reproduceable.
  - ▶ You can edit earlier commands in a chain of commands.
  - ▶ It's easier to spot mistakes.
- There are different ways of “running” an R script:
  - ▶ Selecting all or a subset of lines and hitting “Run”.
  - ▶ Sourcing “with echo” and “without echo”.

## Commenting in R Scripts

- When writing code it's good practice to write "comments" to help others (and you!) understand your code.
- Anything written after a # symbol is ignored by the R console. So we precede all comments with a #.

```
# Set values of a and b:  
a <- 2  
b <- 3  
print(a + b) # Compute the sum of a and b and print:
```

```
[1] 5
```

- Commenting is also useful if you want to temporarily disable a certain part of your script. You just need to put a # before the commands you want to disable.
  - This is called "commenting out".
  - Select the lines you want to comment out, then go to Code → Comment/Uncomment Lines.

## CSV files

- The most common way we read data into R is through CSV (comma-separated values) files.
- These are plain text files with a .csv extension.
- We would store our travel mode survey data in a CSV file like this:

```
travel_mode,time
train,25
train,20
walk,15
cycle,10
walk,17
train,30
```

## CSV files: Commas in data points

- Each line needs to have the same number of commas.
- If data points contain commas themselves (e.g. suppose a category was train, cycle), we need to wrap the data points with quotes.
- So often you will see CSV files where the text is wrapped in quotes:

```
"travel_mode","time"  
"train",25  
"train",20  
"walk",15  
"cycle",10  
"walk",17  
"train",30
```

## CSV files: Comma decimal separators

- In continental Europe, commas are used as decimal separators: “one and a half” is written as 1,5.
- Clearly this will cause problems with CSV files!
- So sometimes you might see files with ; delimiting variables instead of ,.
- An example of that would be:

```
"travel_mode";"time"  
"train";25,0  
"train";20,0  
"walk";15,0  
"cycle";10,0  
"walk";17,0  
"train";30,0
```

- When this happens, we need to tell R that the file is using a ; separator.
- In the exam, however, we'll deal exclusively with more standard CSV files.

# Reading CSV files into R

- In order to read a CSV file into R, we need to tell R where the file is located on our computer.
- There are three different approaches to do this:
  - 1 The absolute path method
  - 2 The relative path method
  - 3 The RStudio Projects method (my recommendation!)
- For this we save the travel mode data in a file called `test.csv`.

## Reading CSV files into R: The Absolute Path Method

- This approach involves giving R the full path to the file.
- On Windows, the full path would look something like C:\Users\username\Documents\test.csv.
- However, the backslash has a special purpose in R so we can't use this.
- We need to use either forward slashes ("/") or *double-backslashes* ("\"):
  - ▶ C:/Users/username/Documents/test.csv
  - ▶ C:\\Users\\\\username\\\\Documents\\\\test.csv
- The fastest way to get the full path to a file is with the `file.choose()` command.
- Run the command, navigate to the file, and then the full file path will appear in the console. You can then copy this to your clipboard.
- You can then paste this as an argument into the `read.csv()` command:

```
df <- read.csv("C:\\Users\\\\username\\\\Documents\\\\test.csv")
```

- When we use this approach, it doesn't matter what the current working directory of the R process is.

## Reading CSV files into R: The Relative Path Method

- We can find out what the current working directory is with `getwd()`.
- To change it to the Documents folder, we can use the `setwd()` command.
- When the R process is in the working directory with the data, we only need to provide the name of the file in the `read.csv()` command.
- The steps would then be:

```
setwd("C:\\Users\\username\\Documents\\")
df <- read.csv("test.csv")
```

- Suppose the full path of the file was actually  
`"C:\\Users\\username\\Documents\\data\\test.csv"`
- We could then read in the data from the Documents folder by only giving the *relative path* to the file, which is `"data\\test.csv"`:

```
setwd("C:\\Users\\username\\Documents\\")
df <- read.csv("data\\test.csv")
```

# Reading CSV files into R: The RStudio Projects Method

- If you share your code with someone, they will have to edit these lines that read in the data, or change the working directory. This is not ideal!
- A better way to deal with file paths is by using the RStudio Project feature.
- Suppose you saved your data in a folder called PQS on your computer.
- To go `File → New Project`, choose “Existing Directory”, and navigate to the PQS folder.
- When you are in the PQS project, RStudio automatically changes the current working directory to that folder. Then you don't need to provide the full file path, or use the `setwd( )` command.
- Therefore I recommend this approach over the previous methods.

# R Packages: Installing a Package

- Up to now, all the functions we have been using come by default with R.
  - ▶ We call the default functionality “base R”.
- But people have written *packages* that expand the functionality of R to do more things.
- For example, base R is not able to read in data from an Excel file.
  - ▶ But there are several packages that can do this.
  - ▶ We'll learn how to do this using the `readxl` package.
- To install a package, you can use the command:

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

- Alternatively, you can install the package using Tools → Install Packages... in RStudio.

# R Packages: Loading and Using a Package

- R doesn't load up the functions of all the installed packages by default. We also need to *load* a package after we install it.
- We can do this with the `library()` command:

```
library(readxl)
```

- If a package is not installed, the `library` function will return an error.
- To read in a file called `test.xlsx` in the current working directory, we do:

```
df <- read_excel("test.xlsx")
```

- The `read_excel()` function loads the data as a `tibble`, which is like a `data.frame` but with a few extra features. We can force the data to be a plain `data.frame` with:

```
df <- data.frame(read_excel("test.xlsx"))
```

## Dataframes: Eredivisie Data

|    | team             | wins | draws | losses | goals_for | goals_against |
|----|------------------|------|-------|--------|-----------|---------------|
| 1  | AZ               | 20   | 7     | 7      | 68        | 35            |
| 2  | Ajax             | 20   | 9     | 5      | 86        | 38            |
| 3  | Excelsior        | 9    | 5     | 20     | 32        | 71            |
| 4  | FC Emmen         | 6    | 10    | 18     | 33        | 65            |
| 5  | FC Groningen     | 4    | 6     | 24     | 31        | 75            |
| 6  | FC Twente        | 18   | 10    | 6      | 66        | 27            |
| 7  | FC Utrecht       | 15   | 9     | 10     | 55        | 50            |
| 8  | FC Volendam      | 10   | 6     | 18     | 42        | 71            |
| 9  | Feyenoord        | 25   | 7     | 2      | 81        | 30            |
| 10 | Fortuna Sittard  | 10   | 6     | 18     | 39        | 62            |
| 11 | Go Ahead Eagles  | 10   | 10    | 14     | 46        | 56            |
| 12 | NEC              | 8    | 15    | 11     | 42        | 45            |
| 13 | PSV              | 23   | 6     | 5      | 89        | 40            |
| 14 | RKC Waalwijk     | 11   | 8     | 15     | 50        | 64            |
| 15 | SC Cambuur       | 5    | 4     | 25     | 26        | 69            |
| 16 | Sparta Rotterdam | 17   | 8     | 9      | 60        | 37            |
| 17 | Vitesse          | 10   | 10    | 14     | 45        | 50            |
| 18 | sc Heerenveen    | 12   | 10    | 12     | 44        | 50            |

## Dataframes: Indexing

- We can get the 2nd row and 3rd column of df with `df[2, 3]`.
- We can get the team name and number of wins variables for Ajax, Feyenoord and PSV with `df[c(2, 9, 13), c(1, 2)]`.
- We can extract all variables for Ajax with `df[2, ]`.
- We can extract all values for “goals for” with `df[, 5]`.
- We can also extract all values for a column with a variable name:
  - ▶ `df$goals_for`.
  - ▶ `df[, "goals_for"]`.
  - ▶ `df[["goals_for"]]`.
- We can extract several columns with:
  - ▶ `df[, c("team", "goals_for")]`
- To get the rows for all teams with at least 20 wins: `df[df$wins >= 20, ]`.

## Dataframes: Creating Variables - Goal Difference

```
df$goal_diff <- df$goals_for - df$goals_against  
head(df)
```

|   | team         | wins | draws | losses | goals_for | goals_against | goal_diff |
|---|--------------|------|-------|--------|-----------|---------------|-----------|
| 1 | AZ           | 20   | 7     | 7      | 68        | 35            | 33        |
| 2 | Ajax         | 20   | 9     | 5      | 86        | 38            | 48        |
| 3 | Excelsior    | 9    | 5     | 20     | 32        | 71            | -39       |
| 4 | FC Emmen     | 6    | 10    | 18     | 33        | 65            | -32       |
| 5 | FC Groningen | 4    | 6     | 24     | 31        | 75            | -44       |
| 6 | FC Twente    | 18   | 10    | 6      | 66        | 27            | 39        |

## Dataframes: Creating Variables - Total Points

$$\text{Points} = 3 \times \text{Wins} + 1 \times \text{Draws} + 0 \times \text{Losses}$$

```
df$total_points <- 3 * df$wins + df$draws  
head(df[, c("team", "wins", "draws", "losses", "total_points")])
```

|   | team         | wins | draws | losses | total_points |
|---|--------------|------|-------|--------|--------------|
| 1 | AZ           | 20   | 7     | 7      | 67           |
| 2 | Ajax         | 20   | 9     | 5      | 69           |
| 3 | Excelsior    | 9    | 5     | 20     | 32           |
| 4 | FC Emmen     | 6    | 10    | 18     | 28           |
| 5 | FC Groningen | 4    | 6     | 24     | 18           |
| 6 | FC Twente    | 18   | 10    | 6      | 64           |

## Dataframes: Creating Variables - Team Ranking

```
df <- df[order(df$total_points, df$goal_diff, decreasing = TRUE), ]  
df$ranking <- 1:nrow(df)  
head(df[, c("team", "total_points", "goal_diff", "ranking")])
```

|    | team             | total_points | goal_diff | ranking |
|----|------------------|--------------|-----------|---------|
| 9  | Feyenoord        | 82           | 51        | 1       |
| 13 | PSV              | 75           | 49        | 2       |
| 2  | Ajax             | 69           | 48        | 3       |
| 1  | AZ               | 67           | 33        | 4       |
| 6  | FC Twente        | 64           | 39        | 5       |
| 16 | Sparta Rotterdam | 59           | 23        | 6       |

# Dataframes: Creating Variables - Relegation Status

```
df$relegation_status <- ""  
df$relegation_status[df$ranking < 16] <- "No relegation"  
df$relegation_status[df$ranking == 16] <- "Relegation playoffs"  
df$relegation_status[df$ranking %in% 17:18] <- "Automatic relegation"  
  
df[, c("team", "ranking", "relegation_status")]
```

|    | team             | ranking | relegation_status    |
|----|------------------|---------|----------------------|
| 9  | Feyenoord        | 1       | No relegation        |
| 13 | PSV              | 2       | No relegation        |
| 2  | Ajax             | 3       | No relegation        |
| 1  | AZ               | 4       | No relegation        |
| 6  | FC Twente        | 5       | No relegation        |
| 16 | Sparta Rotterdam | 6       | No relegation        |
| 7  | FC Utrecht       | 7       | No relegation        |
| 18 | sc Heerenveen    | 8       | No relegation        |
| 14 | RKC Waalwijk     | 9       | No relegation        |
| 17 | Vitesse          | 10      | No relegation        |
| 11 | Go Ahead Eagles  | 11      | No relegation        |
| 12 | NEC              | 12      | No relegation        |
| 10 | Fortuna Sittard  | 13      | No relegation        |
| 8  | FC Volendam      | 14      | No relegation        |
| 3  | Excelsior        | 15      | No relegation        |
| 4  | FC Emmen         | 16      | Relegation playoffs  |
| 15 | SC Cambuur       | 17      | Automatic relegation |
| 5  | FC Groningen     | 18      | Automatic relegation |

# The %in% Operator

- When we write a `%in%` b we are checking for each element in a if there is a *matching element somewhere* in b.

```
a <- 1:6  
b <- c(3, 5, 7)  
a %in% b
```

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

An equivalent but longer way of doing the same thing would be:

```
a == b[1] | a == b[2] | a == b[3]
```

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

## Dataframes: Summarizing a Dataframe

The `summary()` command gives the summary statistics of all variables in `df`:

```
summary(df[, c("team", "wins", "draws", "losses")])
```

|         | team      | wins          | draws          | losses        |
|---------|-----------|---------------|----------------|---------------|
| Length: | 18        | Min. : 4.00   | Min. : 4.000   | Min. : 2.00   |
| Class : | character | 1st Qu.: 9.25 | 1st Qu.: 6.000 | 1st Qu.: 7.50 |
| Mode :  | character | Median :10.50 | Median : 8.000 | Median :13.00 |
|         |           | Mean :12.94   | Mean : 8.111   | Mean :12.94   |
|         |           | 3rd Qu.:17.75 | 3rd Qu.:10.000 | 3rd Qu.:18.00 |
|         |           | Max. :25.00   | Max. :15.000   | Max. :25.00   |

## Dataframes: Peaking at your data

- We use `head(df, n = 4)` to see the first 4 rows of df. `head(df)` on its own shows 6 rows by default.
- We use `tail(df, n = 2)` to see the last 2 rows of df.

```
nrow(df) # see the number of rows in df
```

```
[1] 18
```

```
ncol(df) # see the number of columns in df
```

```
[1] 10
```

```
dim(df) # see both the number of rows and number of columns in df
```

```
[1] 18 10
```

## Dataframes: See all variable names

- `names(df)` shows the variable names of all variables in `df`.

```
names(df)
```

```
[1] "team"                  "wins"                 "draws"  
[4] "losses"                "goals_for"             "goals_against"  
[7] "goal_diff"              "total_points"          "ranking"  
[10] "relegation_status"
```

- Later we will learn how to use this command to change the names of variables in `df`.

# Data Cleaning

Very often when we have a spreadsheet, we need to do some “cleaning” before we can work with it in R.

This can happen when:

- The data don't start at the top of the file because the first few rows contain some other information.
- The dates are not formatted correctly.
- Numbers are interpreted as characters.
- The data contain extra columns that we don't want.
- There are rows with missing data that we want to omit.
- The variable names are not what we want them to be.

# ASML Stock Price Data

We will learn how to clean data with the `asml-trades.csv` data.

The variable names are:

- Date: The date the data from that row are from.
- Open: The opening price of the stock on that day.
- High: The highest price the stock traded at on that day.
- Low: The lowest price the stock traded at on that day.
- Last: The price of the last-traded stock at on that day.
- Close: The closing price of the stock on that day.
- Number.of.Shares: The number of shares traded that day.
- Number.of.Trades: The number of trades made that day.

## Skipping Rows

If the data don't begin at the top of a file, you can tell R to ignore the empty rows with the skip option:

```
df <- read.csv("asml-trades.csv", skip = 3)
```

```
summary(df)
```

```
Date           Open          High          Low
Length:521      Min. :394.7    Min. :408.2    Min. :375.8
Class :character 1st Qu.:535.5   1st Qu.:545.5   1st Qu.:525.8
Mode  :character Median :592.1   Median :597.4   Median :582.5
                  Mean  :589.2   Mean  :597.5   Mean  :579.6
                  3rd Qu.:645.5  3rd Qu.:652.5  3rd Qu.:636.1
                  Max. :770.5   Max. :777.5   Max. :764.2
                  NA's  :6       NA's  :6       NA's  :6
Last            Close         Number.of.Shares Number.of.Trades
Min.  :397.4     Min.  :397.4     Length:521      Length:521
1st Qu.:535.9    1st Qu.:535.9    Class :character Class :character
Median :589.4    Median :589.4    Mode  :character  Mode  :character
Mean   :588.4    Mean   :588.4
3rd Qu.:644.0    3rd Qu.:644.0
Max.  :770.5    Max.  :770.5
NA's  :6       NA's  :6
Print.table
Mode:logical
NA's:521
```

## Formatting Dates

The date variable was read in as a character instead of a date.

```
head(df$Date, n = 3)
```

```
[1] "31/8/2021" "1/9/2021" "2/9/2021"
```

The dates are in the format “dd/mm/yyyy”. We can convert these to dates using the `as.Date()` function, telling R the format the dates are in:

```
df$Date <- as.Date(df$Date, format = "%d/%m/%Y")
```

```
summary(df$Date)
```

|  | Min.         | 1st Qu.      | Median       | Mean         | 3rd Qu.      | Max.         |
|--|--------------|--------------|--------------|--------------|--------------|--------------|
|  | "2021-08-31" | "2022-03-01" | "2022-08-30" | "2022-08-29" | "2023-02-28" | "2023-08-29" |

## Formatting Dates: More Examples

```
as.Date("12/31/2023", format = "%m/%d/%Y")
```

```
[1] "2023-12-31"
```

```
as.Date("31-12-2023", format = "%d-%m-%Y")
```

```
[1] "2023-12-31"
```

```
as.Date("31/12/23", format = "%d/%m/%y")
```

```
[1] "2023-12-31"
```

```
as.Date("31 Dec 2023", format = "%d %b %Y")
```

```
[1] "2023-12-31"
```

```
as.Date("31 December 2023", format = "%d %B %Y")
```

```
[1] "2023-12-31"
```

## Converting Characters to Numbers

- Although most values are numbers, there are some elements with "None" in the variables Number.of.Shares and Number.of.Trades.
- These character elements force the entire variable to be character, because all elements in a vector must have the same type.
- If we convert the character elements to NA (missing values), we can convert the variable to a numeric vector:

```
df$Number.of.Shares[df$Number.of.Shares == "None"] <- NA  
df$Number.of.Shares <- as.numeric(df$Number.of.Shares)
```

- We could skip the step of converting character elements to NA, but then R would warn us: NAs introduced by coercion.

# Deleting Variables

We can delete variables by assigning NULL to that variable.

```
df$Print.table <- NULL
```

Alternatively we can drop variables using the column index of the variables we want to drop. We can drop the 9th variable (Print.table) with:

```
df <- df[, -9]
```

## Dropping Rows with Missing Data

- For all variables apart from the date we have missing data on 6 rows. Let's take a look at which dates these are:

```
df$Date[is.na(df$Open)]
```

```
[1] "2022-04-15" "2022-04-18" "2022-12-26" "2023-04-07" "2023-04-10"  
[6] "2023-05-01"
```

- These are dates around Easter and Christmas when the stock market is closed.
- We can drop rows with any missing variables with the `na.omit()` function.

```
nrow(df)
```

```
[1] 521
```

```
df <- na.omit(df)  
nrow(df)
```

```
[1] 515
```

## Renaming Variables

We can change a variable name using its column index as follows:

```
names(df)[7] <- "num_shares"
```

We can change a variable name using its current name as follows:

```
names(df)[names(df) == "Number.of.Trades"] <- "num_trades"
```

We can change multiple variable names at once with:

```
names(df)[2:5] <- c("open", "high", "low", "last")
```

## Renaming Variables: Converting to Lower Case

We can convert characters to lower case with the `tolower()` function:

```
test <- c("hello!", "HELLO!", "Hello!", "HeLlO!")  
tolower(test)
```

```
[1] "hello!" "hello!" "hello!" "hello!"
```

We can convert all variable names to lower case with:

```
names(df) <- tolower(names(df))  
names(df)
```

```
[1] "date"          "open"         "high"        "low"         "last"  
[6] "close"        "num_shares"   "num_trades"
```

# Introduction to Plotting

- We will now learn how to plot data with R.
- We will learn how to make:
  - ▶ Histograms: these display the distribution of a numeric variable.
  - ▶ Bar charts: these display frequencies of values for numeric or categorical data.
  - ▶ Scatter plots: these display the relationship between two numeric variables.
- There are two ways to make these plots:
  - ① The “base R” approach (using built-in plotting functions in R)  
These are quick and easy to make, but don’t look very nice.
  - ② The “ggplot” approach (using the ggplot2 package).  
These require more code, but are prettier and more customizable.

# The Palmer Penguins Dataset

- We will use this famous dataset to demonstrate plotting.
- It contains the weight, gender, flipper length, and bill length and depth for 3 types of penguins: the adelie, the chinstrap and gentoo.



We can load the dataset with:

```
install.packages("palmerpenguins")
library(palmerpenguins)
data(penguins)
```

## Summarizing the Data

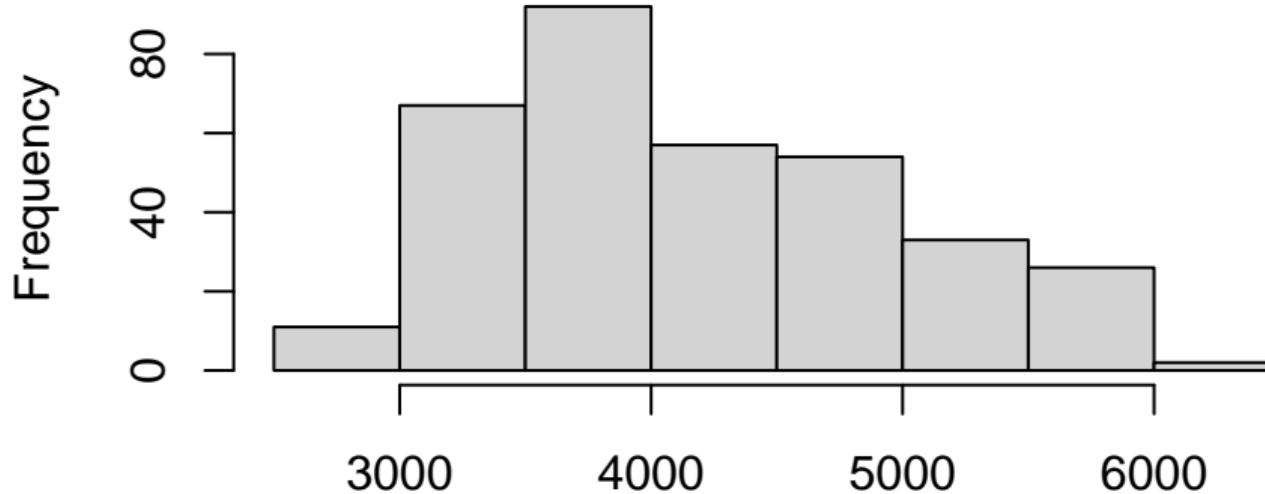
```
summary(penguins)
```

| species           | island        | bill_length_mm | bill_depth_mm |
|-------------------|---------------|----------------|---------------|
| Adelie :152       | Biscoe :168   | Min. :32.10    | Min. :13.10   |
| Chinstrap: 68     | Dream :124    | 1st Qu.:39.23  | 1st Qu.:15.60 |
| Gentoo :124       | Torgersen: 52 | Median :44.45  | Median :17.30 |
|                   |               | Mean :43.92    | Mean :17.15   |
|                   |               | 3rd Qu.:48.50  | 3rd Qu.:18.70 |
|                   |               | Max. :59.60    | Max. :21.50   |
|                   |               | NA's :2        | NA's :2       |
| flipper_length_mm | body_mass_g   | sex            | year          |
| Min. :172.0       | Min. :2700    | female:165     | Min. :2007    |
| 1st Qu.:190.0     | 1st Qu.:3550  | male :168      | 1st Qu.:2007  |
| Median :197.0     | Median :4050  | NA's : 11      | Median :2008  |
| Mean :200.9       | Mean :4202    |                | Mean :2008    |
| 3rd Qu.:213.0     | 3rd Qu.:4750  |                | 3rd Qu.:2009  |
| Max. :231.0       | Max. :6300    |                | Max. :2009    |
| NA's :2           | NA's :2       |                |               |

## Histograms in Base R

```
hist(penguins$body_mass_g)
```

**Histogram of penguins\$body\_mass\_g**



## Bar Plots in Base R

The `table()` function applied to a vector shows the number of times each value appears.

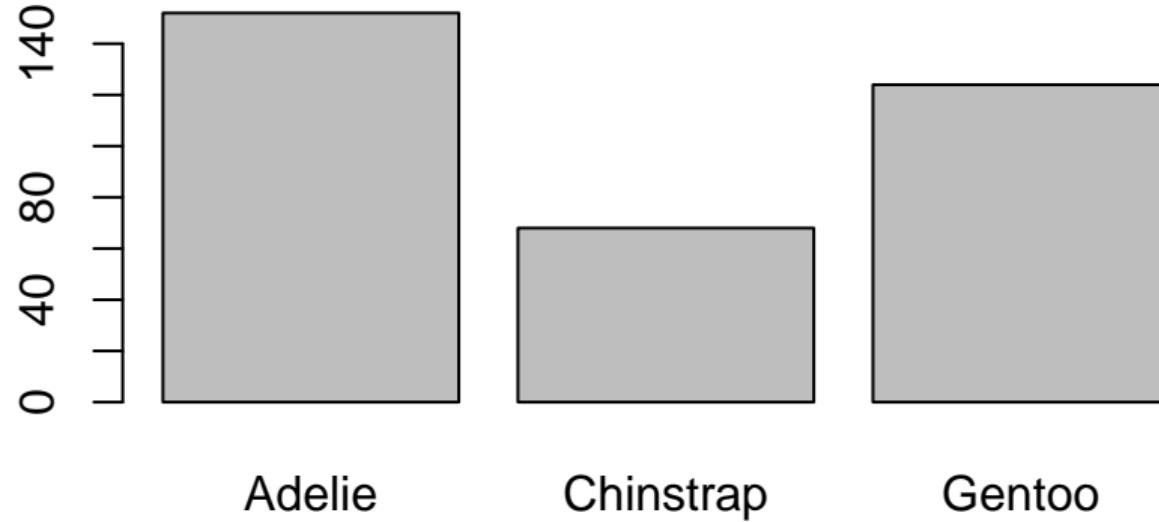
```
table(penguins$species)
```

|        |           |        |
|--------|-----------|--------|
| Adelie | Chinstrap | Gentoo |
| 152    | 68        | 124    |

We can use a bar plot to visualize these relative frequencies.

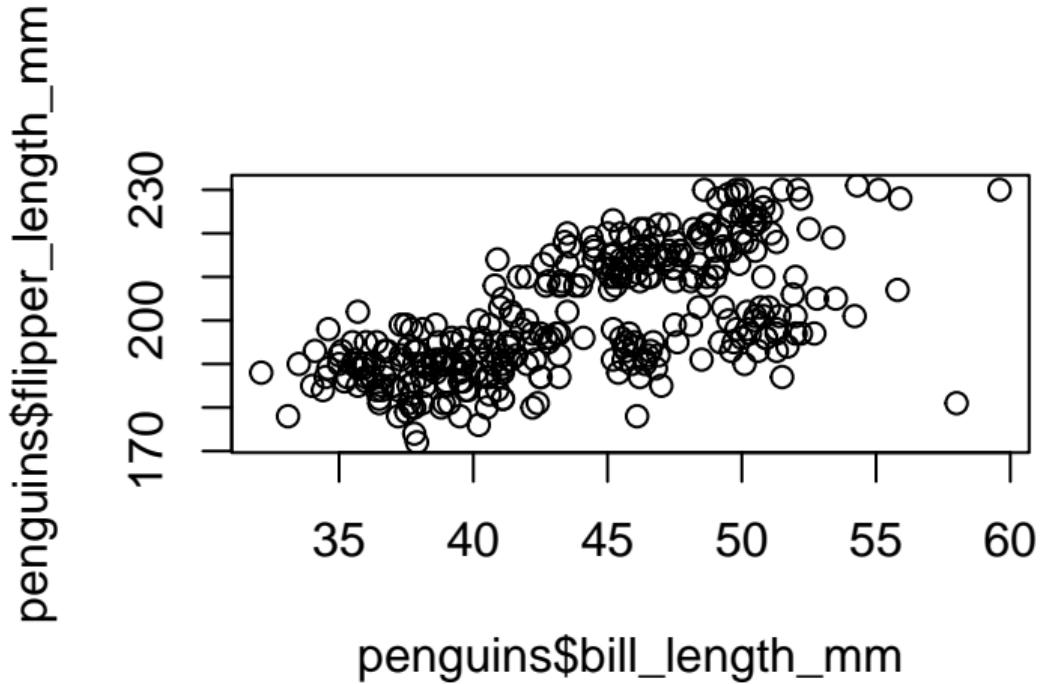
## Bar Plots in Base R

```
barplot(table(penguins$species))
```



## Scatter Plots in Base R

```
plot(penguins$bill_length_mm, penguins$flipper_length_mm)
```



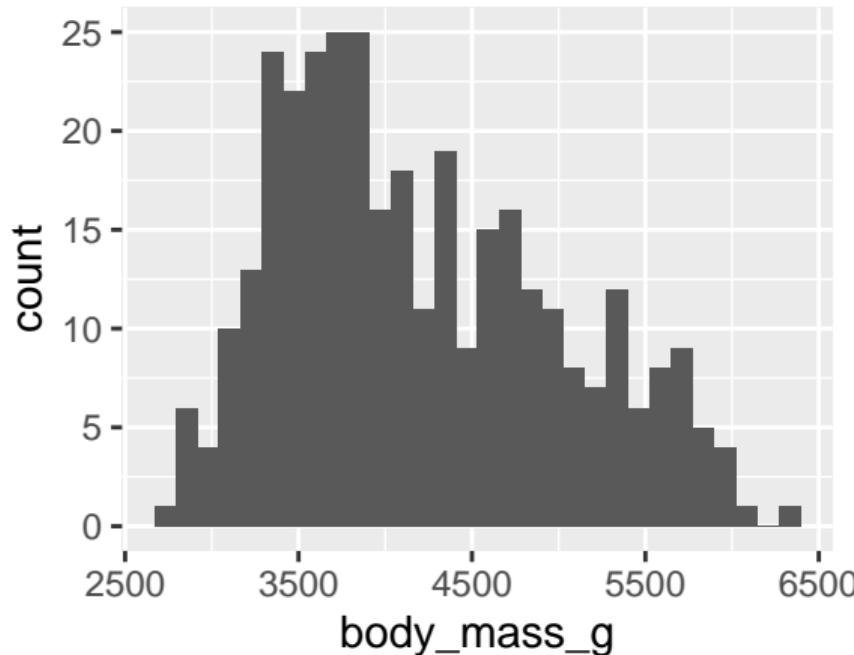
# ggplot: The Grammar of Graphics

- Although it's possible to customize the plots from base R, we will instead learn how to produce nicer plots using the `ggplot2` package.
- The “gg” in `ggplot` stands for “Grammar of Graphics”, which is a scheme to *layer* elements in a plot.
- With the `ggplot2` package, we create plots by adding *layers*.
- Install and load the `ggplot2` package:

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

## Basic Histogram:

```
ggplot(penguins, aes(body_mass_g)) +  
  geom_histogram()
```



# Customizing Histograms

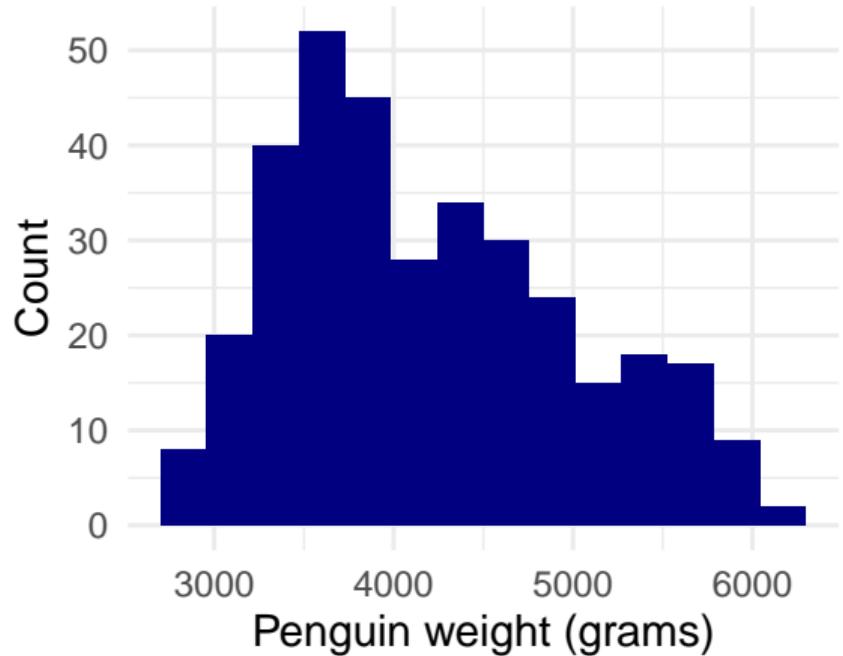
We can customize this with options and by adding *layers*:

- Choosing the number of bins.
- Changing the color of the bins.
- Specifying the axis labels.
- Changing the plot theme (`theme_minimal()` for removing the background colors).

```
ggplot(penguins, aes(body_mass_g)) +
  geom_histogram(bins = 15, fill = "navy") +
  xlab("Penguin weight (grams)") +
  ylab("Count") +
  theme_minimal()
```

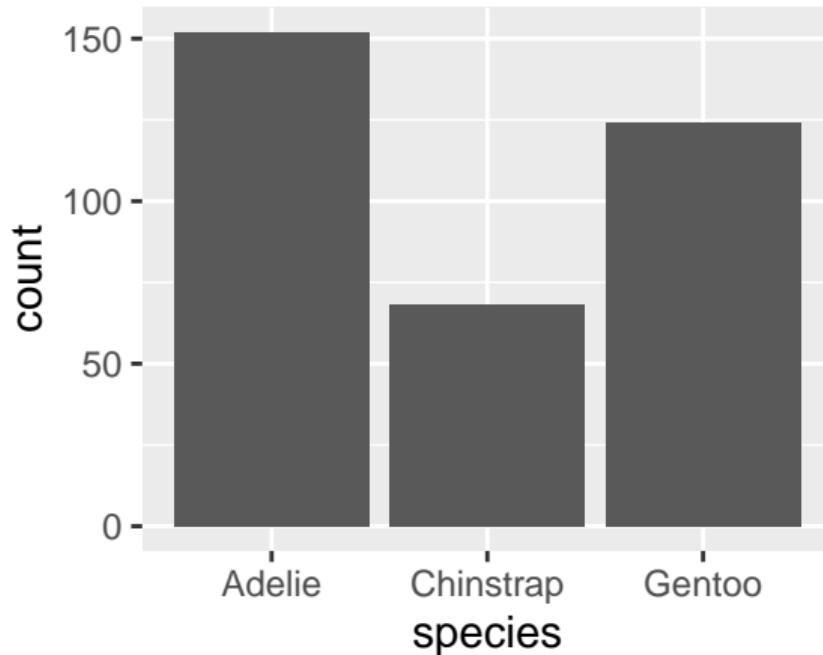
(Output on the next slide)

# Customizing Histograms



## Basic Bar Plot

```
ggplot(penguins, aes(species)) +  
  geom_bar()
```



## Cross-Tabulation

- We can also create a bar plot with a cross-tabulation.
- When we put 2 variables x and y in the `table()` function with `table(x, y)`, it shows us how often each combination of the values in x and y appear together in the data:

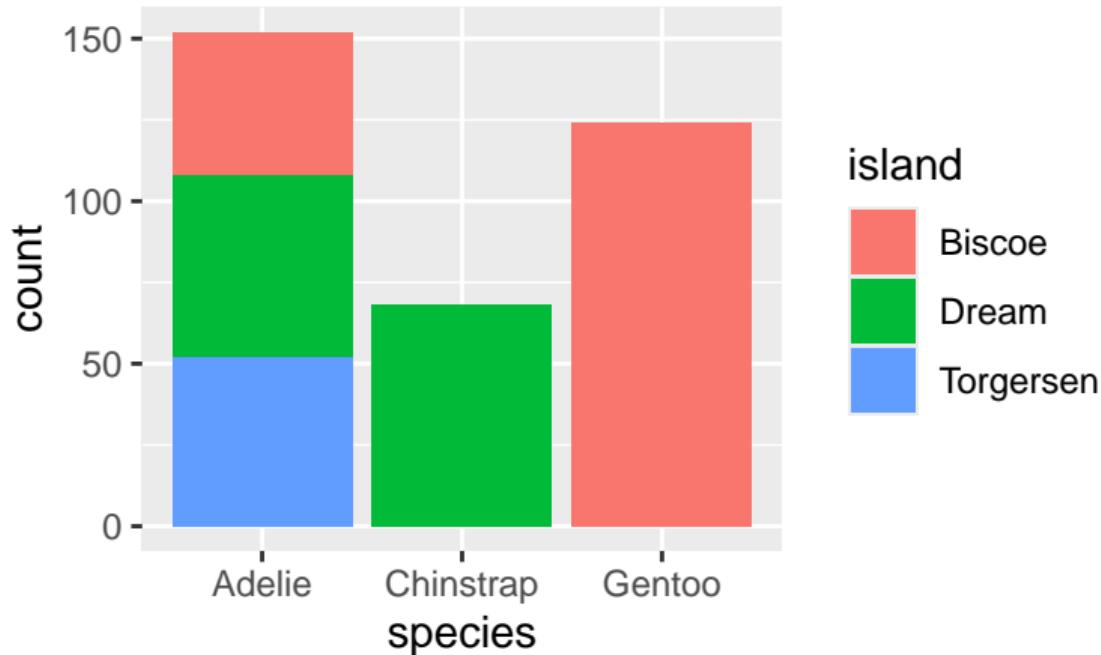
```
table(penguins$species, penguins$island)
```

|           | Biscoe | Dream | Torgersen |
|-----------|--------|-------|-----------|
| Adelie    | 44     | 56    | 52        |
| Chinstrap | 0      | 68    | 0         |
| Gentoo    | 124    | 0     | 0         |

- Adelie penguins appear on all 3 islands.
- Chinstrap penguins appear only on Dream island.
- Gentoo penguins appear only on Biscoe island.

# Basic Bar Plot with Cross-Tabulation

```
ggplot(penguins, aes(species, fill = island)) +  
  geom_bar()
```



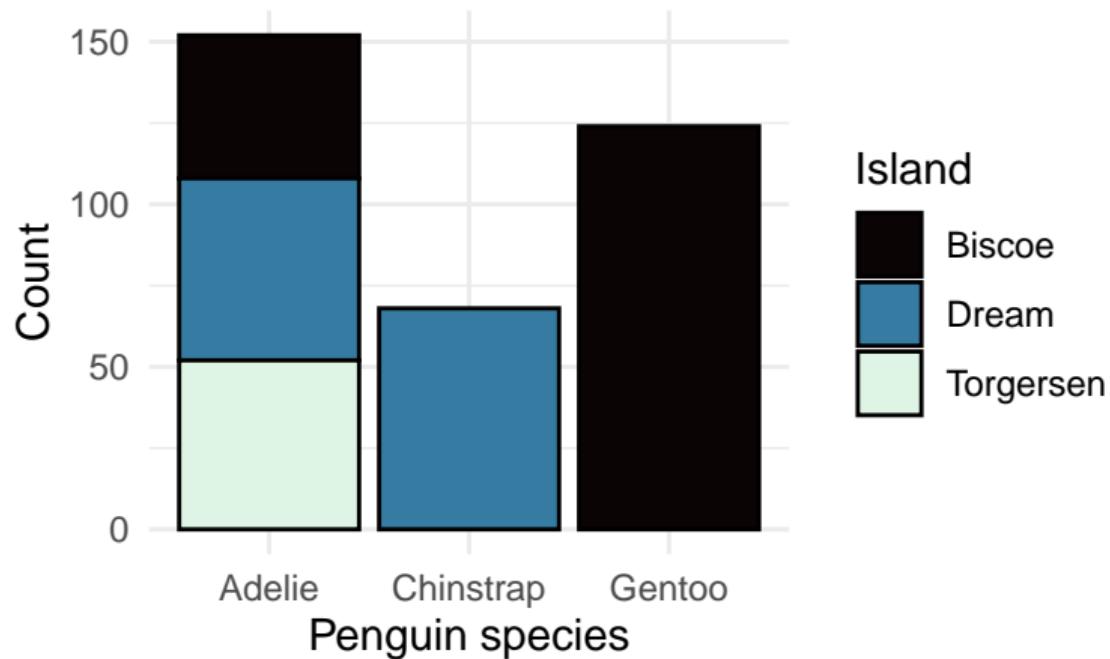
# Customizing Bar Plots

- We can change the name of the legend and the “fill” colors using the `scale_fill_discrete()` option.
- We can also specify colors with their *hexadecimal format* instead of color names.
  - ▶ We can find the hexadecimal format of a color using any color picker tool.

```
ggplot(penguins, aes(species, fill = island)) +  
  geom_bar(color = "black") +  
  xlab("Penguin species") +  
  ylab("Count") +  
  scale_fill_discrete(name = "Island",  
                      type = c("#0B0405", "#357BA2", "#DEF5E5")) +  
  theme_minimal()
```

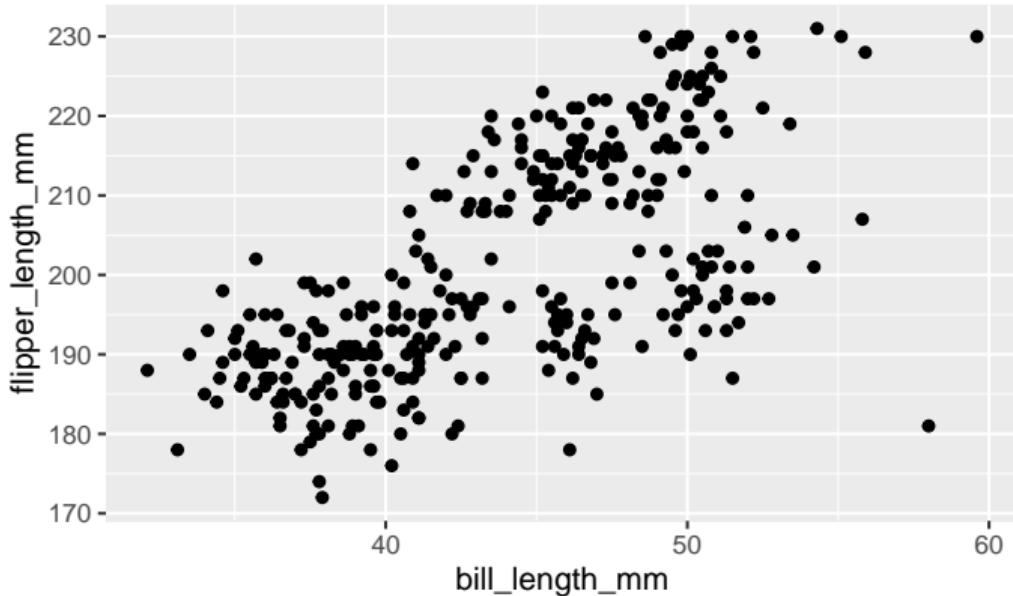
(Output on the next slide)

# Customizing Bar Plots



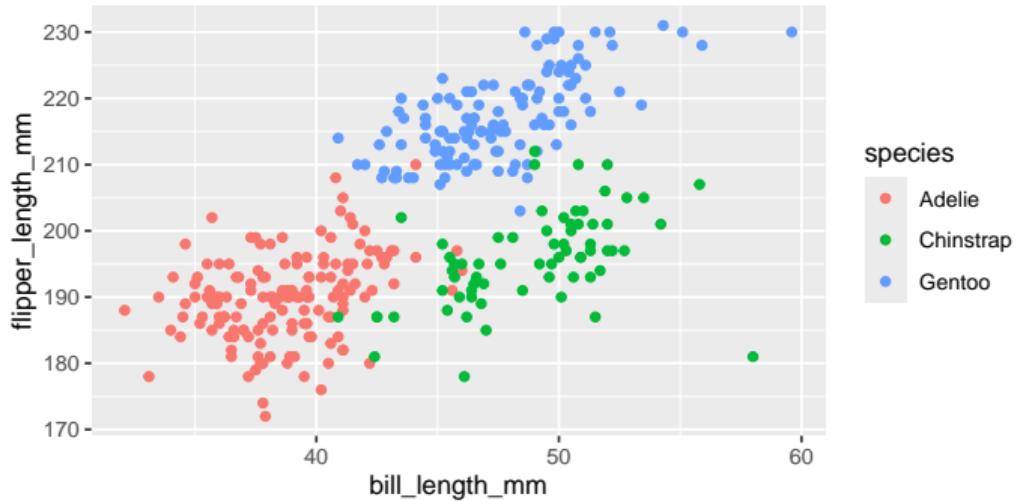
## Basic Scatter Plots

```
ggplot(penguins, aes(bill_length_mm, flipper_length_mm)) +  
  geom_point()
```



# Different Colors for Different Categories

```
ggplot(penguins, aes(bill_length_mm, flipper_length_mm, color = species))  
  geom_point()
```

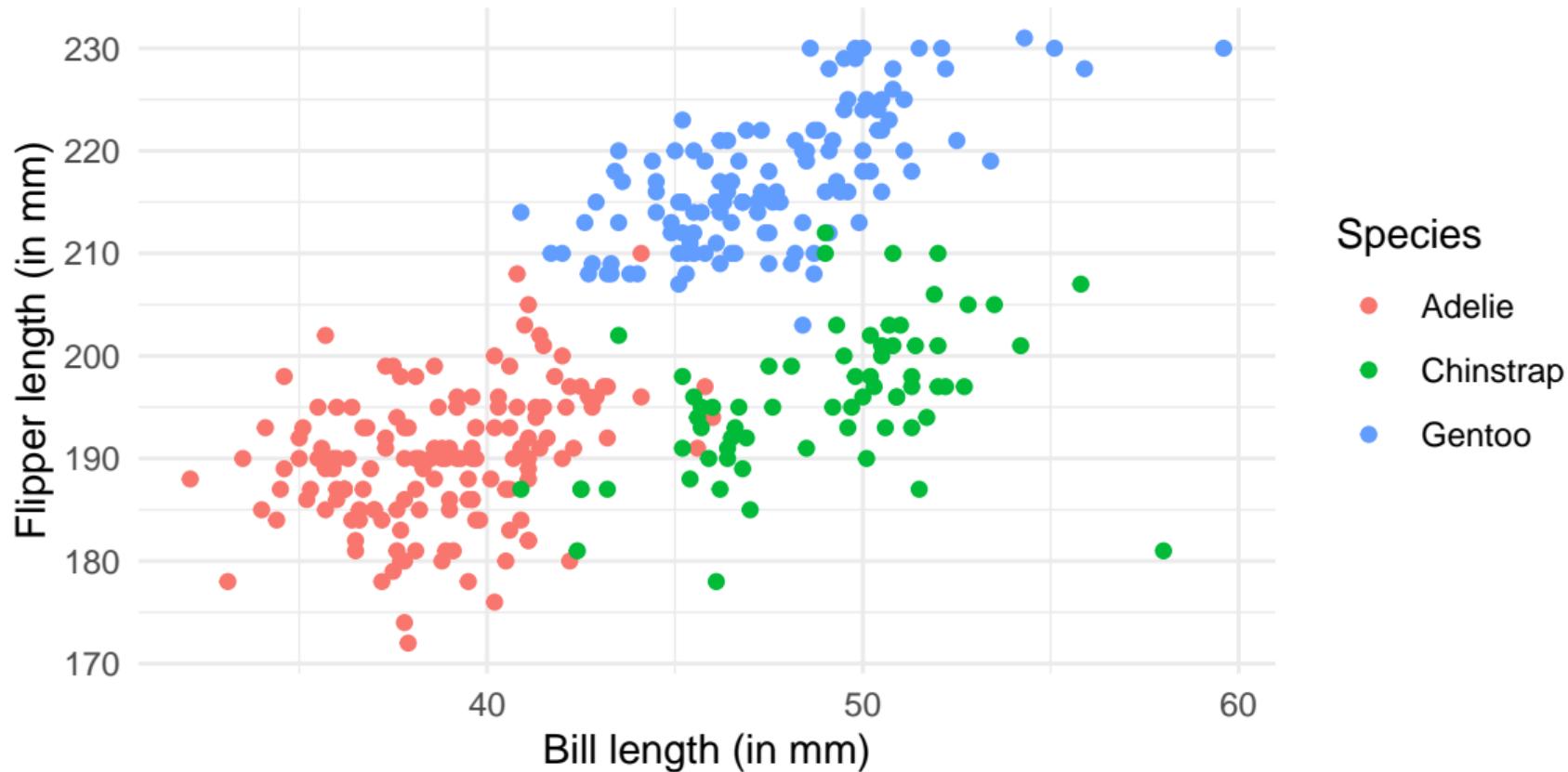


# Customizing Scatter Plots

```
ggplot(penguins, aes(bill_length_mm, flipper_length_mm, color = species))  
  geom_point() +  
  scale_color_discrete(name = "Species") +  
  xlab("Bill length (in mm)") +  
  ylab("Flipper length (in mm)") +  
  theme_minimal()
```

(Output on the next slide)

# Customizing Scatter Plots



# Making your own R Functions

- It's very easy to create your own R functions.
- Consider the quadratic function:

$$f(x) = -8 - 2x + x^2$$

- We can create an R function, which we call `f()`, to calculate the output of this function as follows:

```
f <- function(x) {  
  y <- -8 - 2 * x + x^2  
  return(y)  
}
```

## Making your own R Functions

- We can then use this custom function like we would any other R function.
- The function evaluated at  $x = 2$  should equal:

$$f(2) = -8 - 2 \times (2) + (2)^2 = -8 - 4 + 4 = -8$$

- We can check that our custom function gets the same answer:

```
f(2)
```

```
[1] -8
```

- We can also pass vectors into our custom function:

```
f(c(2, 3, 4))
```

```
[1] -8 -5  0
```

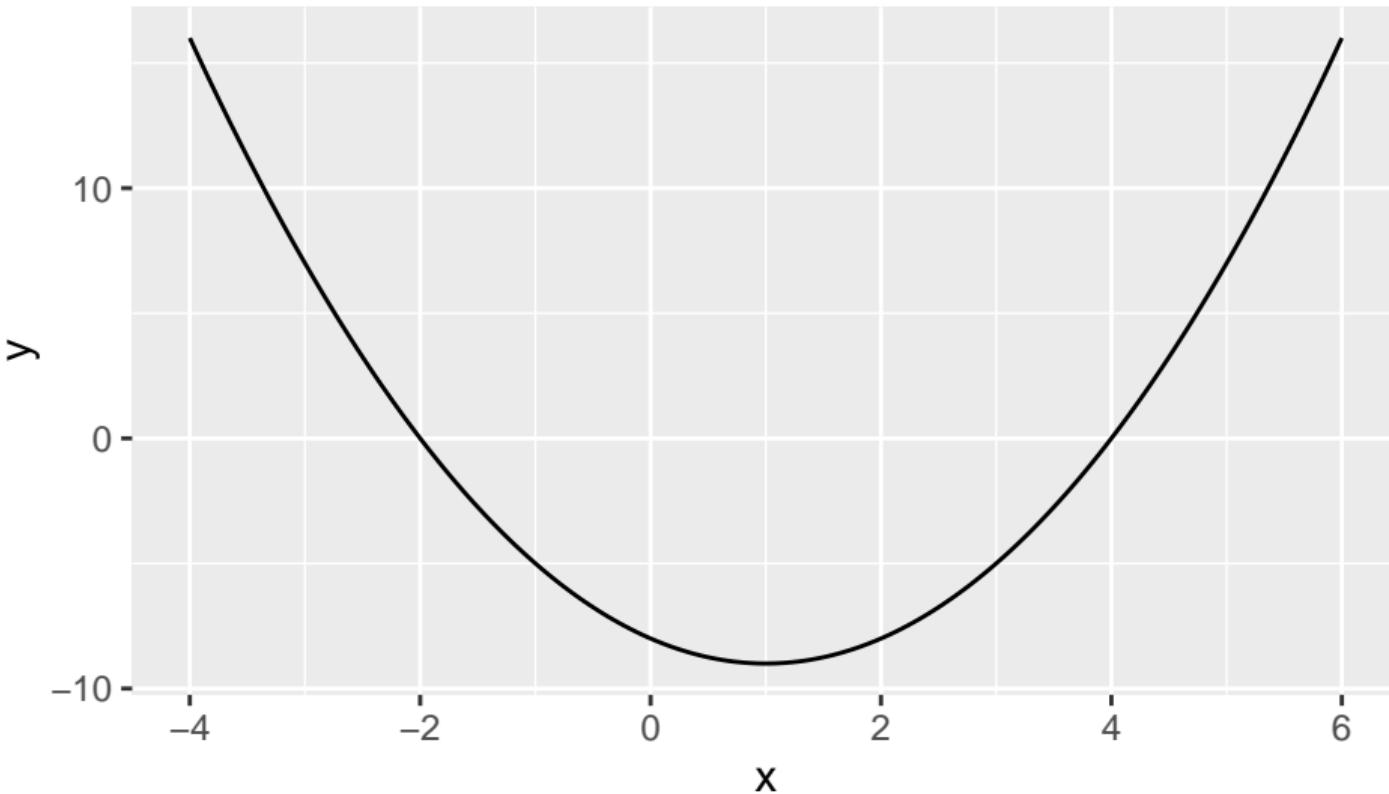
# Plotting Functions with ggplot.

- We can also plot custom functions like this with ggplot.
- We first choose a range of values of  $x$  for which we want to plot the function.
- We then create a sequence of values of  $x$  in this range.
- We then evaluate the function at each of these  $x$  values to get  $y$ .
- We then put these  $x$  and  $y$  values in a data.frame and plot it with ggplot.

```
library(ggplot2)
x <- seq(from = -4, to = 6, length.out = 200)
y <- f(x)
df <- data.frame(x, y)
ggplot(df, aes(x, y)) + geom_line()
```

- Sometimes we don't know what range of  $x$  to choose. In this case it's good to pick some values, make the plot, and then adjust the values and make the plot again.

# Plotting Functions with ggplot.



# Univariate Unconstrained Optimization

- When we plotted the function  $f(x) = -8 - 2x + x^2$ , we saw that it achieved a minimum at  $x = 1$ .
- We could solve for the minimum analytically by setting the first derivative of the function to zero:

$$\frac{df(x)}{dx} = -2 + 2x = 0 \quad \Rightarrow \quad x = 1$$

- We can also use R to find the minimum of the function using the `optimize()` function.

## The optimize( ) Function

- We need to specify an interval (lower bound and upper bound) to search for the extreme point.
- We also need to specify if we want a maximum or a minimum using the maximum option.

```
optimize(f, interval = c(-100, 100), maximum = FALSE)
```

```
$minimum  
[1] 1
```

```
$objective  
[1] -9
```

- The optimize( ) function finds a minimum at 1, and the function takes a value of  $-9$  at the minimum, i.e.  $f(1) = -9$
- If we instead wanted to find the maximum of a function, we would specify maximum = TRUE.

# The optimize( ) Function

- The output of the `optimize()` function is a list.
- If we assign the output to an object called `f_min`, we can extract the minimum with `f_min$minimum`.
  - ▶ The `$` extraction operator works for lists just like with dataframes.
- We can similarly get the value of the function at the minimum with `f_min$objective`.

```
f_min <- optimize(f, interval = c(-100, 100), maximum = FALSE)  
f_min$minimum
```

```
[1] 1
```

```
f_min$objective
```

```
[1] -9
```

## Conditional Statements (“If-else”)

- Very often we have to perform different actions depending on whether something is true or not.
- For this we use *if-else statements*, which are also called *conditional statements*.
- The absolute value function is a simple example of this:

$$|x| = \begin{cases} -x & x < 0 \\ x & \text{otherwise} \end{cases}$$

- We ask, “is  $x < 0$ ?” If yes, then return  $-x$ . If not, then return  $x$ .
- Of course, we can always use the `abs()` function in R to calculate the absolute value. But let’s create our own function doing exactly this.

## Custom absolute value function

```
my_abs <- function(x) {  
  if (x < 0) {  
    return(-x)  
  } else {  
    return(x)  
  }  
}  
my_abs(-2)
```

```
[1] 2
```

```
my_abs(3)
```

```
[1] 3
```

# If-Else Statements for Vectors

- The previous function we wrote only works with scalars (vectors of length 1).
- If we would try to do `my_abs(c(-2, 3))`, we would get an error.
- To do if-else statements with vectors, we can use the `ifelse()` function.
- The `ifelse()` function takes 3 arguments:
  - ① A logical vector (such as testing the condition  $x < 0$ ).
  - ② What to return if TRUE (i.e. if  $x < 0$ ).
  - ③ What to return if FALSE (i.e. if  $x \geq 0$ ).

```
x <- -5:5
```

```
x
```

```
[1] -5 -4 -3 -2 -1  0  1  2  3  4  5
```

```
ifelse(x < 0, -x, x)
```

```
[1] 5 4 3 2 1 0 1 2 3 4 5
```

## “If-Else If-Else” Statements

- Sometimes there can be more than 2 cases to test.
- For example, consider the following function which gives the “sign” in front of a number:

$$sgn(x) = \begin{cases} -1 & \text{if } x < 0 \\ 0 & \text{if } x = 0 \\ +1 & \text{otherwise} \end{cases}$$

- For example,  $sgn(-2) = -1$ ,  $sgn(0) = 0$ , and  $sgn(3) = 1$ .
- We can code a function in R to do this by nesting if-else statements.

## “If-Else If-Else” Statements

```
sgn <- function(x) {  
  if (x < 0) {  
    return(-1)  
  } else if (x == 0) {  
    return(0)  
  } else {  
    return(+1)  
  }  
}  
sgn(-2)
```

```
[1] -1
```

```
sgn(3)
```

```
[1] 1
```

## “If-Else If-Else” Statements with Vectors

We can also nest the `ifelse()` function inside of itself to get the sign of a vector of numbers:

```
x <- -3:3  
x
```

```
[1] -3 -2 -1  0  1  2  3
```

```
ifelse(x < 0, -1, ifelse(x == 0, 0, 1))
```

```
[1] -1 -1 -1  0  1  1  1
```

# Merging

- Merging (or joining) is the R equivalent of the VLOOKUP function in Excel.
- When two datasets have a common ID variable linking them together, we can merge them.
- For example:
  - ▶ One dataset with total sales on each day, and another dataset with the temperature on each day. We can link the datasets using the date.
  - ▶ One dataset with the total sales in each municipality over a year, and another with the demographic characteristics of each municipality. We can link the datasets using the municipality name.

# Merging Example

- We will show a merging example using 2 datasets:
  - 1 Average daily petrol prices from 2014-2022.
  - 2 Brent crude oil spot prices from 1987-2022 (excludes weekends and holidays).
- We will merge the datasets by date.

## Data Cleaning: Average Daily Petrol Price Data

```
df1 <- read.csv("avg_daily_petrol_prices.csv")
head(df1$date)
```

```
[1] "2014-06-08" "2014-06-09" "2014-06-10" "2014-06-11" "2014-06-12"
[6] "2014-06-13"
```

```
df1$date <- as.Date(df1$date, format = "%Y-%m-%d")
summary(df1)
```

|         | date        | e5            | e10           | diesel         |
|---------|-------------|---------------|---------------|----------------|
| Min.    | :2014-06-08 | Min. :1.159   | Min. :1.130   | Min. :0.9558   |
| 1st Qu. | :2016-07-03 | 1st Qu.:1.340 | 1st Qu.:1.318 | 1st Qu.:1.1322 |
| Median  | :2018-07-29 | Median :1.402 | Median :1.379 | Median :1.2353 |
| Mean    | :2018-07-29 | Mean :1.456   | Mean :1.423   | Mean :1.2811   |
| 3rd Qu. | :2020-08-23 | 3rd Qu.:1.522 | 3rd Qu.:1.479 | 3rd Qu.:1.3217 |
| Max.    | :2022-09-18 | Max. :2.261   | Max. :2.203   | Max. :2.3343   |

## Data Cleaning: Brent Crude Oil Spot Price Data

```
df2 <- read.csv("Europe_Brent_Spot_Price_FOB.csv", skip = 4)
head(df2$Day)
```

```
[1] "09/19/2022" "09/16/2022" "09/15/2022" "09/14/2022" "09/13/2022"
[6] "09/12/2022"
```

```
df2$Day <- as.Date(df2$Day, format = "%m/%d/%Y")
names(df2) <- c("date", "crude_oil")
summary(df2)
```

|         | date       | crude_oil      |
|---------|------------|----------------|
| Min.    | 1987-05-20 | Min. : 9.10    |
| 1st Qu. | 1996-03-06 | 1st Qu.: 19.03 |
| Median  | 2005-01-04 | Median : 38.08 |
| Mean    | 2005-01-12 | Mean : 48.22   |
| 3rd Qu. | 2013-11-24 | 3rd Qu.: 69.67 |
| Max.    | 2022-09-19 | Max. :143.95   |

## The merge( ) Command:

```
df <- merge(df1, df2, by = "date")
```

```
summary(df)
```

|         | date        | e5            | e10           | diesel         |
|---------|-------------|---------------|---------------|----------------|
| Min.    | :2014-06-09 | Min. :1.159   | Min. :1.130   | Min. :0.9558   |
| 1st Qu. | :2016-07-04 | 1st Qu.:1.339 | 1st Qu.:1.318 | 1st Qu.:1.1302 |
| Median  | :2018-07-26 | Median :1.402 | Median :1.379 | Median :1.2345 |
| Mean    | :2018-07-27 | Mean :1.455   | Mean :1.422   | Mean :1.2797   |
| 3rd Qu. | :2020-08-19 | 3rd Qu.:1.521 | 3rd Qu.:1.478 | 3rd Qu.:1.3216 |
| Max.    | :2022-09-16 | Max. :2.261   | Max. :2.203   | Max. :2.3343   |
|         | crude_oil   |               |               |                |
| Min.    | : 9.12      |               |               |                |
| 1st Qu. | : 48.54     |               |               |                |
| Median  | : 61.18     |               |               |                |
| Mean    | : 63.47     |               |               |                |
| 3rd Qu. | : 72.97     |               |               |                |
| Max.    | :133.18     |               |               |                |

## Merging: Dropped Observations

- The merged dataset only includes observations where there is a match.
- Observations where there is no corresponding match are dropped:

```
nrow(df1)
```

```
[1] 3025
```

```
nrow(df2)
```

```
[1] 8970
```

```
nrow(df)
```

```
[1] 2107
```

- To avoid dropping rows, we can use the `all.x = TRUE` and/or `all.y = TRUE` options.

## Merging with all.x = TRUE

- `all.x = TRUE` : Keeps all observations in the 1st dataset, but only merges data from the 2nd dataset when there is a match. When there is no match, variables in the 2nd dataset get assigned NA values.
- `all.y = TRUE` : Keeps all observations in the 2nd dataset, but only merges data from the 1st dataset when there is a match. When there is no match, variables in the 1st dataset get assigned NA values.
- `all = TRUE` : This keeps all observations from both datasets, and variables get assigned NA values when there is no match. This is equivalent to setting both `all.x = TRUE` and `all.y = TRUE`.

## Merging with all.x = TRUE

```
df <- merge(df1, df2, by = "date", all.x = TRUE)  
summary(df)
```

|         | date        | e5            | e10           | diesel         |
|---------|-------------|---------------|---------------|----------------|
| Min.    | :2014-06-08 | Min. :1.159   | Min. :1.130   | Min. :0.9558   |
| 1st Qu. | :2016-07-03 | 1st Qu.:1.340 | 1st Qu.:1.318 | 1st Qu.:1.1322 |
| Median  | :2018-07-29 | Median :1.402 | Median :1.379 | Median :1.2353 |
| Mean    | :2018-07-29 | Mean :1.456   | Mean :1.423   | Mean :1.2811   |
| 3rd Qu. | :2020-08-23 | 3rd Qu.:1.522 | 3rd Qu.:1.479 | 3rd Qu.:1.3217 |
| Max.    | :2022-09-18 | Max. :2.261   | Max. :2.203   | Max. :2.3343   |

|         | crude_oil |
|---------|-----------|
| Min.    | : 9.12    |
| 1st Qu. | : 48.54   |
| Median  | : 61.18   |
| Mean    | : 63.47   |
| 3rd Qu. | : 72.97   |
| Max.    | :133.18   |
| NA's    | :918      |

## Further Remarks on Merging

- If merging on multiple variables, use `by = c("var1", "var2")`.
- If merging variables differ in `df1` and `df2`, use `by.x` and `by.y` instead of `by`. For example:

```
df <- merge(df1, df2, by.x = c("market_area", "date"),
             by.y = c("market", "date"))
```

- To avoid sorting the data, use option `sort = FALSE`.

# Reshaping

Suppose you have a dataset structured like this (long format):

|   | <b>id</b> | <b>variable</b> | <b>value</b> |
|---|-----------|-----------------|--------------|
| 1 | 1         | x               | 3            |
| 2 | 1         | y               | 5            |
| 3 | 2         | x               | 4            |
| 4 | 2         | y               | 8            |
| 5 | 3         | x               | 3            |
| 6 | 3         | y               | 1            |

And you wanted to *reshape* it to look like this (wide format):

|   | <b>id</b> | <b>x</b> | <b>y</b> |
|---|-----------|----------|----------|
| 1 | 1         | 3        | 5        |
| 2 | 2         | 4        | 8        |
| 3 | 3         | 3        | 1        |

# Reshaping

- Base R has a function that can do this called `reshape()`, but it's not very easy to use.
- The package `reshape2` contains functions to make it easy to go from long to wide format and vice-versa:
  - ▶ `dcast(df, id ~ variable)`: long to wide format.
  - ▶ `melt(df, idvars = "id")`: wide to long format.

## Reshaping: Long to Wide

```
long <- data.frame(  
  id      = rep(1:3, each = 2),  
  variable = rep(c("x", "y"), times = 3),  
  value    = c(3, 5, 4, 8, 3, 1)  
)  
library(reshape2)  
wide <- dcast(long, id ~ variable)  
wide
```

|   | id | x | y |
|---|----|---|---|
| 1 | 1  | 3 | 5 |
| 2 | 2  | 4 | 8 |
| 3 | 3  | 3 | 1 |

## Reshaping: Wide to Long

```
melt(wide, id.vars = "id")
```

|   | <b>id</b> | <b>variable</b> | <b>value</b> |
|---|-----------|-----------------|--------------|
| 1 | 1         | x               | 3            |
| 2 | 2         | x               | 4            |
| 3 | 3         | x               | 3            |
| 4 | 1         | y               | 5            |
| 5 | 2         | y               | 8            |
| 6 | 3         | y               | 1            |

## Overlay Line Plots with ggplot

To overlay line plots with ggplot we need our data in long format.

```
df <- read.csv("avg_daily_petrol_prices.csv")
df$date <- as.Date(df$date, format = "%Y-%m-%d")
head(df, n = 2)
```

|   | date       | e5       | e10      | diesel   |
|---|------------|----------|----------|----------|
| 1 | 2014-06-08 | 1.551987 | 1.477774 | 1.353583 |
| 2 | 2014-06-09 | 1.576623 | 1.483362 | 1.385182 |

```
df2 <- melt(df, id.vars = "date")
head(df2, n = 2)
```

|   | date       | variable | value    |
|---|------------|----------|----------|
| 1 | 2014-06-08 | e5       | 1.551987 |
| 2 | 2014-06-09 | e5       | 1.576623 |

# Overlay Line Plots with ggplot

```
ggplot(df2, aes(date, value, color = variable)) + geom_line()
```

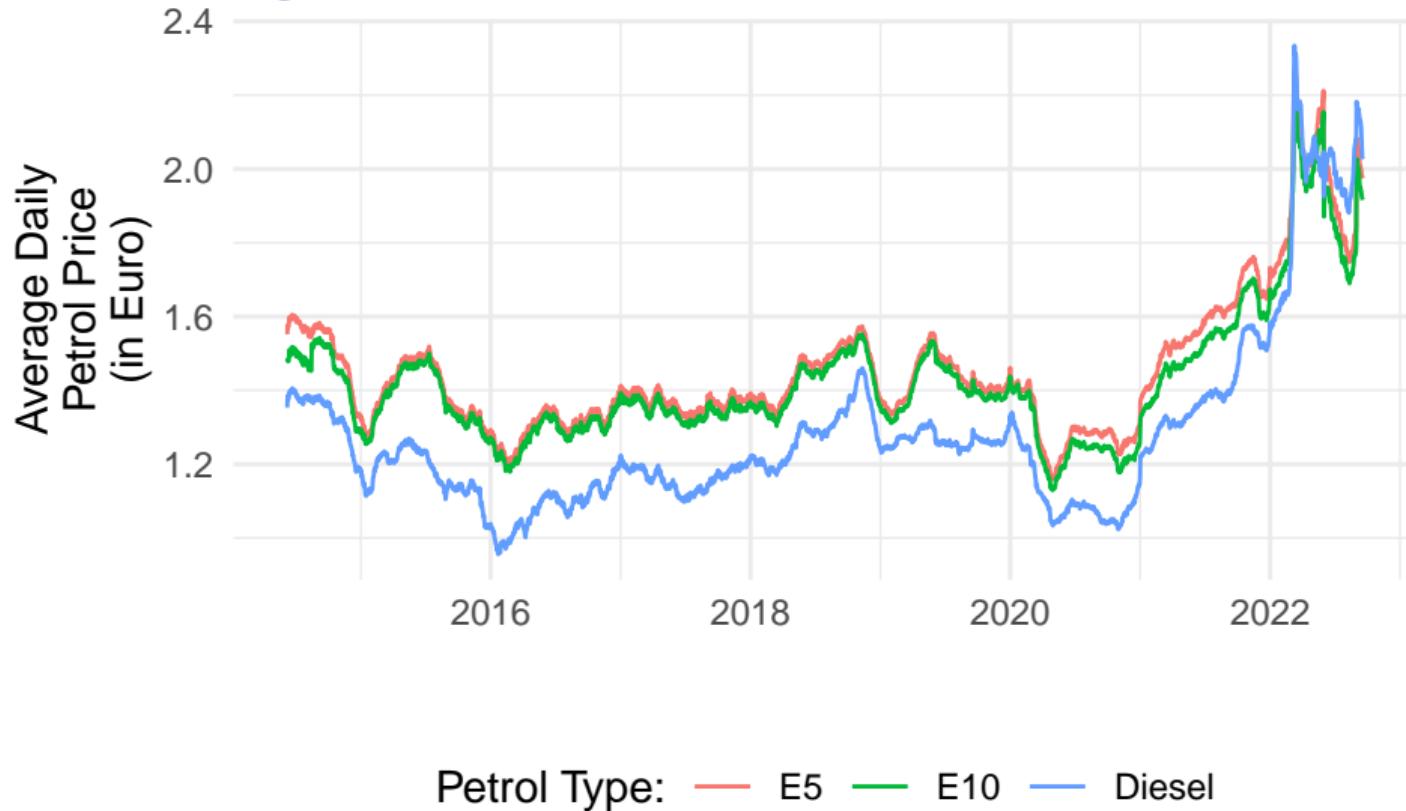


## Customizing the Plot

```
levels(df2$variable) <- c("E5", "E10", "Diesel")
library(ggplot2)
ggplot(df2, aes(date, value, color = variable)) +
  geom_line() +
  xlab("") +
  ylab("Average Daily\nPetrol Price\n(in Euro)") +
  scale_color_discrete(name = "Petrol Type:") +
  theme_minimal() +
  theme(legend.direction = "horizontal",
        legend.position = "bottom")
```

(Output on next slide)

## Customizing the Plot



## Aggregating by Group

- If we want to get the sum or average by group we can use the `aggregate()` function.
- The `aggregate()` function is a bit like the R version of pivot tables in Excel.
- If we want to get the average of `x` by group `g` in the dataframe `df`, we use:

```
aggregate(x ~ g, FUN = mean, data = df)
```

- We'll show this using the petrol price data as an example.

## Average Price of E5 Petrol by Year

```
df <- read.csv("avg_daily_petrol_prices.csv")
df$date <- as.Date(df$date, format = "%Y-%m-%d")
library(lubridate)
df$year <- year(df$date)
aggregate(e5 ~ year, FUN = mean, data = df)
```

|   | year | e5       |
|---|------|----------|
| 1 | 2014 | 1.519195 |
| 2 | 2015 | 1.393170 |
| 3 | 2016 | 1.302767 |
| 4 | 2017 | 1.368414 |
| 5 | 2018 | 1.454098 |
| 6 | 2019 | 1.430592 |
| 7 | 2020 | 1.288080 |
| 8 | 2021 | 1.579992 |
| 9 | 2022 | 1.933512 |

## Maximum Price of E5 by Year

```
aggregate(e5 ~ year, FUN = max, data = df)
```

|   | year | e5       |
|---|------|----------|
| 1 | 2014 | 1.605252 |
| 2 | 2015 | 1.519229 |
| 3 | 2016 | 1.395620 |
| 4 | 2017 | 1.414228 |
| 5 | 2018 | 1.573760 |
| 6 | 2019 | 1.555623 |
| 7 | 2020 | 1.461060 |
| 8 | 2021 | 1.761985 |
| 9 | 2022 | 2.260581 |

## Average Price of E5 and E10 by Year

```
aggregate(cbind(e5, e10) ~ year, FUN = mean, data = df)
```

|   | year | e5       | e10      |
|---|------|----------|----------|
| 1 | 2014 | 1.519195 | 1.461632 |
| 2 | 2015 | 1.393170 | 1.373425 |
| 3 | 2016 | 1.302767 | 1.282375 |
| 4 | 2017 | 1.368414 | 1.345430 |
| 5 | 2018 | 1.454098 | 1.430955 |
| 6 | 2019 | 1.430592 | 1.408177 |
| 7 | 2020 | 1.288080 | 1.253603 |
| 8 | 2021 | 1.579992 | 1.522833 |
| 9 | 2022 | 1.933512 | 1.875897 |

## Average Price of All Variables by Year

```
aggregate(. ~ year, FUN = mean, data = df)
```

|   | year | date    | e5       | e10      | diesel   |
|---|------|---------|----------|----------|----------|
| 1 | 2014 | 16332.0 | 1.519195 | 1.461632 | 1.334612 |
| 2 | 2015 | 16618.0 | 1.393170 | 1.373425 | 1.173013 |
| 3 | 2016 | 16983.5 | 1.302767 | 1.282375 | 1.081282 |
| 4 | 2017 | 17349.0 | 1.368414 | 1.345430 | 1.161306 |
| 5 | 2018 | 17714.0 | 1.454098 | 1.430955 | 1.287264 |
| 6 | 2019 | 18079.0 | 1.430592 | 1.408177 | 1.265341 |
| 7 | 2020 | 18444.5 | 1.288080 | 1.253603 | 1.111068 |
| 8 | 2021 | 18810.0 | 1.579992 | 1.522833 | 1.387114 |
| 9 | 2022 | 19123.0 | 1.933512 | 1.875897 | 1.941386 |