Statistical Inference For Data Science

Saul Diaz Infante Velasco

2/23/23

Table of contents

Preface

Who I am. I am Saul Diaz Infante Velasco. I just starting as assistant professor at the Data Science graduate program of Universidad de Sonora at Hermosillo Mexico. My Background is related with numerical analysis and stochastic models. I'm are a enthusiastic of this treading topic called Data-Science, but perhaps at the moment I only have just intuition about what really it is. However, I have been programming almost 20 years an moved from old programming langues as FORTRAN, Pascal, Basic, Cobol, C, C++ to the new well established treading development workflows like R, Python and Julia. This is my firs attempt in R.

More of this book is work in progress. We aim to provide material as well we review and improve our basic skills to face the study of the most popular methods in Machine learning. Thus the book try to cover fundamentals in R programming as data types, flow control structures and put particular importance in the well practice of coding functions. Then we moves to the management of data whit dplyr and other packages. To finish this part we discuss some package to visualize data. Then, we face the problem o estimation and explore techniques based on bootstrap–and another sampling flavors.

This book has been started on January, 2023 as part of a course to the Master on Data Science from Universidad de Sonora.

I'm writing this book to follow a path of self learning, understanding and and joy for this matter called Data Science. I'm not try to become and expert instead I just pursuit the joy of the interaction of math, computational sciences and the generosity of this virtuous learning-teaching process.

https://sauldiazinfante.github.io/statisticalInferenceinR4DS/

Introduction

The focus of this course is int the programming and basic techniques for inference that are usually applied in data science. We start by reviewing and enforcing programming skills. Then we will use the database of entomological data practice and build the required bases for more structured tools like bootstrap or Jackknife cuts.

Figure 1 further explores the impact of temperature on ozone level.

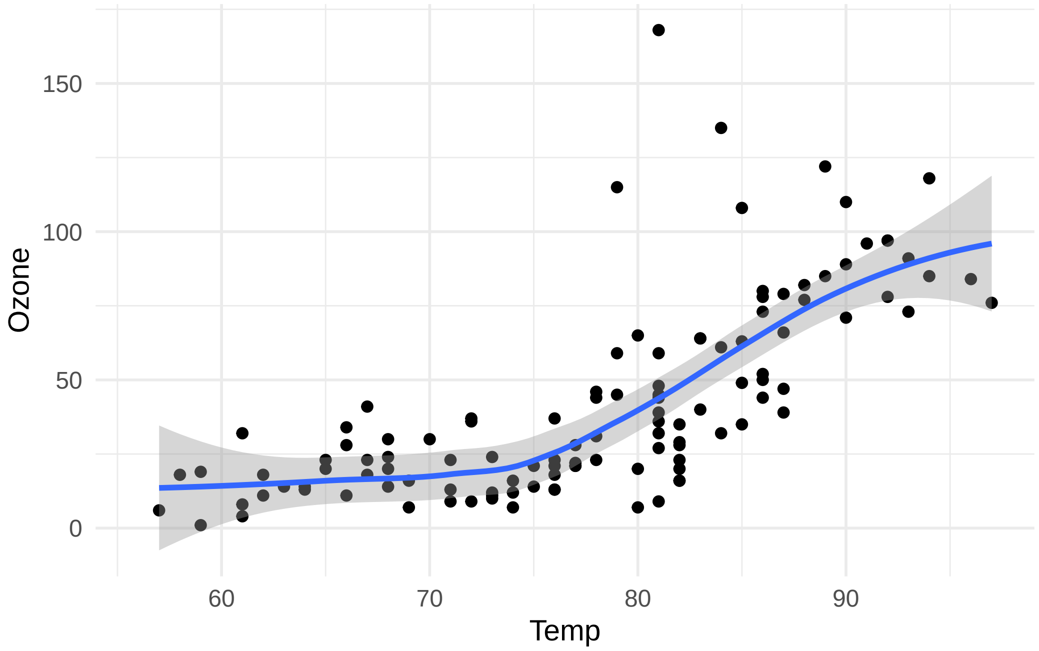

Figure 1: Temperature and ozone level.

Rubrics for a Data Analyst (Associate and professional)

We also want to cover the fundamentals of a common Data Analyst hast to know according to the DataCamp's rubrics at the current date (March-01, 2023).

Associate

Topics	Competnecy
Data Validation	Assess data quality and perform validation tasks
Data Visualization	Create data visualizations to demonstrate the characteristics of data and represent relatio
Communication	Presents data concepts to small, diverse audiences

Professional

Topics	Competency
Data Validation	Assess data quality and perform validation tasks
Data Visualization	Create data visualizations to demonstrate the characteristics of data and represent relatio
Business Focus	Collects relevant information, detects patterns, observes and interprets data
Business Metrics	Benchmarks, monitors, and evaluates business processes
Communication	Employs multiple tactics (written and verbal) to communicate to business leaders

Here we also enclose the material to take the Skill Assessments for Data Analyst and Data Scientist certifications.

Data Analyst

DataCamp recommends that you complete the follwing tracks the associate and professional certification

- Data Analyst with SQL career track for

- Data Analyst with R or
- Data Analyst with Python career tracks for the professional certification being R or Python accordingly

SKILL ASSESMENT	ASSOCIATE	PROFESSIONAL
Data Management in SQL (PostgreSQL)	134	134
Data Analysis in SQL (PostgreSQL)	115	115
Importing & Cleaning Data in R/Python	NA	111
Data Manipulation in R/Python	NA	111
Statistical Fundamentals in R/Python	NA	125

Data Scientist

DataCamp recommends that you complete the following tracks

- SQL Fundamentals skill
- Data Scientist with R (career)
- Data Scientist with Python (career)

you may want to enroll in the Data Communication Concepts course to prepare for the practical exam.

SKILL ASSESMENT	ASSOCIATE	PROFESSIONAL
Data Manipulation in R/Python	131	131
Statistical Fundamentals in R/Python	125	125
Importing & Cleaning in R/Python	130	149
Data Management in SQL (PostgreSQL)	NA	134
Machine Learning Fundamentals in R/Python	119	119
R / Python Programming	120	140

The tidyverse

We need to install a R package. The majority of the packages that we will use are part of the so-called tidyverse package. The packages in the tidyverse share a common philosophy of data and R programming, and are designed to work together naturally.

You can install the complete tidyverse with the line of code:

then we can use it by loading in the preamble section with

```
-- Attaching packages --------------------------------------- tidyverse 1.3.2 --
v tibble  3.1.8      v dplyr   1.0.10
v tidyr   1.2.0      v stringr 1.5.0
v readr   2.1.2      v forcats 0.5.2
v purrr   0.3.4
-- Conflicts ------------------------------------------ tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
```

see https://www.tidyverse.org/ documentation.

Part I

R fundamentals in programming, data managment and visualization

We dedicate this part to overview the basics to program in R. The aim of this part is building the basis for Machine learning, namely **data types as**, **vectors, lists, factors and matrices**

Nuts and bolts: Data types

Entering Input: the assigment operator

The thing that we type on the R console prompt are expressions. The firs expression we discuses here is the assignment operator, please watch the following video https://www.youtube.com/watch?v=vGY5i_J2c-c&t=283s

At the R console, any executable typed text that we put a side of the prompt are called expressions. We start by the `<-` symbol is the assignment operator.

```
[1] 0
```

```
[1] 0
```

```
[1] "what's up"
```

The `[1]` shown in the output indicates that `x` is a `vector` and 0 is the element at position with `index` 1.

Best Coding Practices for R

What we mean when say “better coding practice”

R programmers have a bad reputation writing bad code. Perhaps the main reason is that the people whose write much of the package are not programmers but scientific from other areas. Sometimes we overestimate crucial aspects from a programming standpoint. As R programmers we overcome to write the code for production. Mostly we write scripts and when we deploy it the same when we just wrap it in a function and perhaps a package. It is common to face poorly written code—**columns were referred by numbers, functions were dependent upon global environment variables, 50+ lines functions without arguments and with over-sized lines code 100 characters or more, not indentation, poor naming, conventions** etc,…,.

We strongly encourage to use a style. Yea I know, there is not a unique way to do it, but the philosophy is to follow a consistent style. With respect to this regard made yourself a favor and read this great book for R

https://bookdown.org/content/d1e53ac9-28ce-472f-bc2c-f499f18264a3/

Folder Structure

Code Structure

Sections

Structural Composition

Identation

Styling

Final Comments

Tidyverse Fundamentals with R

Introduction to Tidyverse

Reshaping Data with tidyr

Project

Modeling with Data in the Tidyverse

Communication with Data in the Tidyverse

Categorical Data in the Tydiverse

Data Manipulation

Data Manipulation with dplyr

Joining Data with dplyr

Case Study: Exploratory Data Analysis in R

Data Manipulation with data.table in R

Joining Data with data.table in R

1 Intro to basics

1.1 How it works

In the text editor you should type R code to solve the exercises. When you hit **ctrl + enter**, every line of code is interpreted and executed by R and you get a message whether or not your code was correct.

R makes use of the # sign to add comments, so that you and others can understand what the R code is about. Comments are not run as R code, so they will not influence your result. For example, Calculate 3 + 4 in the editor on the right is a comment.

You can also execute R commands straight in the console. This is a good way to experiment with R code.

Instructions `100 XP`

```
- In the text editor on the right there is already some sample code.
- Can you see which lines are actual R code and which are comments?
- Add a line of code that calculates the sum of 6 and 12,
    and hit the enter button
```

ex_01.R

```r
# Calculate 3 + 4
3 + 4
# Calculate 6 + 12
6 + 12
```

1.2 Arithmetic with R

In its most basic form, R can be used as a simple calculator. Consider the following arithmetic operators:

- Addition: `+`

- Subtraction: `-`
- Multiplication: `*`
- Division: `/`
- Exponentiation: `^`
- Modulo: `%%`

The last two might need some explaining:

- The `^` operator raises the number to its left to the power of the number to its right: for example `3^2` is 9.
- The modulo returns the remainder of the division of the number to the left by the number on its right, for example 5 modulo 3 or `5 %% 3` is 2.

Instructions `100 XP`

```
- Type 2^5 in the editor to calculate 2 to the power 5.
- Type 28 %% 6 to calculate 28 modulo 6.
- Run the answer in the console and have a look at the R output .
- Note how the # symbol is used to add comments on the R code.
```

ex_02.R

```
# An addition
5 + 5

# A subtraction
5 - 5

# A multiplication
3 * 5
  # A division
(5 + 5) / 2
# Exponentiation
2 ^ 5
# Modulo
28 %% 6
```

1.2.1 Variable assignment

A basic concept in (statistical) programming is called a variable.

A variable allows you to store a value (e.g. 4) or an object (e.g. a function description) in R. You can then later use this variable's name to easily access the value or the object that is stored within this variable.

> You can assign a value 4 to a variable my_var with the command
>
> my_var <- 4

1.2.1.1 Instructions 100 XP

Over to you: complete the code in the editor such that it assigns the value 42 to the variable x in the editor. Submit the answer. Notice that when you ask R to print x, the value 42 appears.

ex_03.R

```
# Assign the value 42 to x
x <- 42
# Print out the value of the variable x
print (x)
```

1.2.2 Variable assignment (2)

Suppose you have a fruit basket with five apples. As a data analyst in training, you want to store the number of apples in a variable with the name my_apples.

1.2.2.1 Instructions 100 XP

- Type the following code in the editor: my_apples <- 5. This will assign the value 5 to my_apples.
- Type: my_apples below the second comment. This will print out the value of my_apples.
- Run your answer, and look at the output: you see that the number 5 is printed. So R now links the variable my_apples to the value 5.

ex_04.R

```
# Assign the value 5 to the variable my_apples
my_apples <- 5
# Print out the value of the variable my_apples
```

```
print(my_apples)
```

1.3 Variable assignment (3)

Every tasty fruit basket needs oranges, so you decide to add six oranges. As a data analyst, your reflex is to immediately create the variable my_oranges and assign the value 6 to it. Next, you want to calculate how many pieces of fruit you have in total. Since you have given meaningful names to these values,

> you can now code this in a clear way:
>
> ```
> my_apples + my_oranges
> ```

Instructions `100 XP`

- Assign to my_oranges the value 6.
- Add the variables my_apples and my_oranges and have R simply print the result.
- Assign the result of adding my_apples and my_oranges to a new variable my_fruit.

ex_05.R

```
# Assign a value to the variables my_apples and my_oranges
my_apples <- 5
my_oranges <- 6

# Add these two variables together
my_apples + my_oranges

# Create the variable my_fruit
my_fruit <- my_apples + my_oranges
```

1.4 Apples and oranges

Common knowledge tells you not to add apples and oranges. But hey, that is what you just did, no :-)? The my_apples and my_oranges variables both contained a number in the previous exercise. The + operator works with numeric variables in R. If you really tried to add "apples" and "oranges", and assigned a text value to the variable my_oranges (see the editor), you would be trying to assign the addition of a numeric and a character variable to the variable my_fruit. This is not possible.

Instructions 100 XP

- Run the answer and read the error message. Make sure to understand why this did not work.
- Adjust the code so that R knows you have 6 oranges and thus a fruit basket with 11 pieces of fruit.

ex_06.R

```r
# Assign a value to the variable my_apples
my_apples <- 5
# Fix the assignment of my_oranges
my_oranges <- "six"
# Create the variable my_fruit and print it out
my_fruit <- my_apples + my_oranges
my_fruit
```

Response

ex_06.R

```r
# Assign a value to the variable my_apples
my_apples <- 5
# Fix the assignment of my_oranges
my_oranges <- 6
# Create the variable my_fruit and print it out
my_fruit <- my_apples + my_oranges
my_fruit
```

1.5 Basic data types in R

R works with numerous data types. Some of the most basic types to get started are:

- Decimal values like 4.5 are called numerics.
- Whole numbers like 4 are called integers. Integers are also numerics.
- Boolean values (TRUE or FALSE) are called logical.
- Text (or string) values are called characters.

Note how the quotation marks in the editor indicate that “some text” is a string.

Instructions `100 XP`

Change the value of the:

- `my_numeric` variable to 42.
- `my_character` variable to `"universe"`. Note that the quotation marks indicate that “universe” is a character.
- `my_logical` variable to `FALSE`.

> ℹ Note that R is case sensitive!
>
> Thus despite the varibales called `var`, `Var`, `vAr`, has the same fonetic characters, R understand each of these as different memory addresses.

ex_07.R

```r
# Change my_numeric to be 42
my_numeric <- 42.5

# Change my_character to be "universe"
my_character <- "some text"

# Change my_logical to be FALSE
my_logical <- TRUE
```

Response

ex_07.R

```r
# Change my_numeric to be 42
my_numeric <- 42

# Change my_character to be "universe"
my_character <- "universe"

# Change my_logical to be FALSE
my_logical <- FALSE
```

1.6 What’s that data type?

Do you remember that when you added 5 + “six”, you got an error due to a mismatch in data types? You can avoid such embarrassing situations by checking the data type of a variable beforehand. You can do this with the class() function, as the code in the editor shows.

Instructions 100 XP

Complete the code in the editor and also print out the classes of my_character and my_logical.

ex_08.R

```r
# Declare variables of different types

my_numeric <- 42
my_character <- "universe"
my_logical <- FALSE
# Check class of my_numeric
class(my_numeric)

# Check class of my_character
class(my_character)

# Check class of my_logical
class(my_logical)
```

2 Vectors

2.1 Create a vector

Feeling lucky? You better, because this chapter takes you on a trip to the City of Sins, also known as Statisticians Paradise!

Thanks to R and your new data-analytical skills, you will learn how to uplift your performance at the tables and fire off your career as a professional gambler. This chapter will show how you can easily keep track of your betting progress and how you can do some simple analyses on past actions. Next stop, Vegas Baby… VEGAS!!

Instructions `100 XP`

- Do you still remember what you have learned in the first chapter? Assign the value `"Go!"` to the variable vegas. Remember: R is case sensitive!

ex_08.R

```
# Define the variable vegas
vegas <- "Go!"
```

2.2 Create a vector (2)

Let us focus first!

On your way from rags to riches, you will make extensive use of vectors. Vectors are one-dimension arrays that can hold numeric data, character data, or logical data. In other words, a vector is a simple tool to store data. For example, you can store your daily gains and losses in the casinos.

In R, you create a vector with the combine function `c()`. You place the vector elements separated by a comma between the parentheses.

For example:

```r
numeric_vector <- c(1, 2, 3)
character_vector <- c("a", "b", "c")
```

Once you have created these vectors in R, you can use them to do calculations.

Instructions `100 XP`

Complete the code such that boolean_vector contains the three elements: `TRUE,`FALSE`and`TRUE‘ (in that order).

ex_09.R

```r
numeric_vector <- c(1, 10, 49)
character_vector <- c("a", "b", "c")

# Complete the code for boolean_vector
boolean_vector <-c(TRUE, FALSE, TRUE)
```

2.2.1 Create a vector (3)

After one week in Las Vegas and still zero Ferraris in your garage, you decide that it is time to start using your data analytical superpowers.

Before doing a first analysis, you decide to first collect all the winnings and losses for the last week:

For poker_vector:

- On Monday you won `$140`
- Tuesday you lost `$50`
- Wednesday you won `$20`
- Thursday you lost `$120`
- Friday you won `$240`

For roulette_vector:

- On Monday you lost `$24`
- Tuesday you lost `$50`
- Wednesday you won `$100`
- Thursday you lost `$350`

- Friday you won $10

You only played poker and roulette, since there was a delegation of mediums that occupied the craps tables. To be able to use this data in R, you decide to create the variables poker_vector and roulette_vector.

Instructions 100 XP

Assign the winnings/losses for roulette to the variable roulette_vector. You lost $24, then lost $50 , won $100, lost $350, and won $10.

ex_10.R

```
# Poker winnings from Monday to Friday
poker_vector <- c(140, -50, 20, -120, 240)

# Roulette winnings from Monday to Friday
roulette_vector <- c(-24, -50, 100, -350, 10)
```

2.2.2 Naming a vector

As a data analyst, it is important to have a clear view on the data that you are using. Understanding what each element refers to is therefore essential.

In the previous exercise, we created a vector with your winnings over the week. Each vector element refers to a day of the week but it is hard to tell which element belongs to which day. It would be nice if you could show that in the vector itself.

You can give a name to the elements of a vector with the `names()` function. Have a look at this example:

```
#| code-line-numbers: false
#| code-fold: false
#| code-summary: "Show the code"

some_vector <- c("John Doe", "poker player")
names(some_vector) <- c("Name", "Profession")
```

This code first creates a vector some_vector and then gives the two elements a name. The first element is assigned the name Name, while the second element is labeled Profession. Printing the contents to the console yields following output:

Output

```
      Name     Profession
"John Doe" "poker player"
```

Instructions `100 XP`

The code in the editor names the elements in `poker_vector` with the days of the week. Add code to do the same thing for `roulette_vector`.

ex_11.R

```r
# Poker winnings from Monday to Friday
poker_vector <- c(140, -50, 20, -120, 240)

# Roulette winnings from Monday to Friday
roulette_vector <- c(-24, -50, 100, -350, 10)

# Assign days as names of poker_vector
names(poker_vector) <-
  c("Monday", "Tuesday", "Wednesday", "Thursday", "Friday")

# Assign days as names of roulette_vector

names(roulette_vector) <-
  c("Monday", "Tuesday", "Wednesday", "Thursday", "Friday")
```

2.3 Naming a vector (2)

If you want to become a good statistician, you have to become lazy. (If you are already lazy, chances are high you are one of those exceptional, natural-born statistical talents.)

In the previous exercises you probably experienced that it is boring and frustrating to type and retype information such as the days of the week. However, when you look at it from a higher perspective, there is a more efficient way to do this, namely, to assign the days of the week vector to a **variable**!

Just like you did with your poker and roulette returns, you can also create a variable that contains the days of the week. This way you can use and re-use it.

Instructions `100 XP`

- A variable days_vector that contains the days of the week has already been created for you.
- Use `days_vector` to set the names of `poker_vector` and `roulette_vector`.

ex__12.R

```r
# Poker winnings from Monday to Friday
poker_vector <- c(140, -50, 20, -120, 240)

# Roulette winnings from Monday to Friday
roulette_vector <- c(-24, -50, 100, -350, 10)

# The variable days_vector
days_vector <- c("Monday", "Tuesday", "Wednesday", "Thursday", "Friday")

# Assign the names of the day to roulette_vector and poker_vector
names(poker_vector) <- days_vector
names(roulette_vector) <- days_vector
```

2.4 Calculating total winnings

Now that you have the poker and roulette winnings nicely as named vectors, you can start doing some data analytical magic.

You want to find out the following type of information:

- How much has been your overall profit or loss per day of the week?
- Have you lost money over the week in total?
- Are you winning/losing money on poker or on roulette? To get the answers, you have to do arithmetic calculations on vectors.

It is important to know that if you sum two vectors in R, it takes the element-wise sum. For example, the following three statements are completely equivalent:

You can also do the calculations with variables that represent vectors:

Instructions 100 XP

- Take the sum of the variables `A_vector` and `B_vector` and assign it to total_vector.
- Inspect the result by printing out `total_vector`.

ex_13.R

```
A_vector <- c(1, 2, 3)
B_vector <- c(4, 5, 6)

# Take the sum of A_vector and B_vector
total_vector <- A_vector + B_vector

# Print out total_vector
print(total_vector)
```

2.5 Calculating total winnings (2)

Now you understand how R does arithmetic with vectors, it is time to get those Ferraris in your garage! First, you need to understand what the overall profit or loss per day of the week was. The total daily profit is the sum of the `profit / loss` you realized on poker per day, and the `profit / loss` you realized on roulette per day.

In R, this is just the sum of `roulette_vector` and `poker_vector`.

Instructions 100 XP

Assign to the variable `total_daily` how much you won or lost on each day in total (poker and roulette combined).

ex_14.R

```
# Poker and roulette winnings from Monday to Friday:
poker_vector <- c(140, -50, 20, -120, 240)
roulette_vector <- c(-24, -50, 100, -350, 10)
days_vector <- c("Monday", "Tuesday", "Wednesday", "Thursday", "Friday")
names(poker_vector) <- days_vector
names(roulette_vector) <- days_vector

# Assign to total_daily how much you won/lost on each day
total_daily <- roulette_vector + poker_vector
```

2.6 Calculating total winnings (3)

Based on the previous analysis, it looks like you had a mix of good and bad days. This is not what your ego expected, and you wonder if there may be a very tiny chance you have lost money over the week in total?

A function that helps you to answer this question is `sum()`. It calculates the sum of all elements of a vector. For example, to calculate the total amount of money you have lost/won with poker you do:

```r
total_poker <- sum(poker_vector)
```

Instructions `100 XP`

- Calculate the total amount of money that you have won/lost with roulette and assign to the variable `total_roulette`.
- Now that you have the totals for roulette and poker, you can easily calculate `total_week` (which is the sum of all gains and losses of the week).
- Print out `total_week`.

ex_15.R

```r
# Poker and roulette winnings from Monday to Friday:
poker_vector <- c(140, -50, 20, -120, 240)
roulette_vector <- c(-24, -50, 100, -350, 10)
days_vector <- c("Monday", "Tuesday", "Wednesday", "Thursday", "Friday")
names(poker_vector) <- days_vector
names(roulette_vector) <- days_vector

# Total winnings with poker
total_poker <- sum(poker_vector)

# Total winnings with roulette
total_roulette <-  sum(roulette_vector)

# Total winnings overall
total_week <- total_poker + total_roulette

# Print out total_week
print(total_week)
```

2.7 Comparing total winnings

Oops, it seems like you are losing money. Time to rethink and adapt your strategy! This will require some deeper analysis…

After a short brainstorm in your hotel's jacuzzi, you realize that a possible explanation might be that your skills in roulette are not as well developed as your skills in poker. So maybe your total gains in poker are higher (or `>`) than in roulette.

Instructions `100 XP`

- Calculate `total_poker` and `total_roulette` as in the previous exercise. Use the `sum()` function twice.
- Check if your total gains in poker are higher than for roulette by using a comparison. Simply print out the result of this comparison. What do you conclude, should you focus on roulette or on poker?

ex_16.R

```r
# Poker and roulette winnings from Monday to Friday:
poker_vector <- c(140, -50, 20, -120, 240)
roulette_vector <- c(-24, -50, 100, -350, 10)
days_vector <- c("Monday", "Tuesday", "Wednesday", "Thursday", "Friday")
names(poker_vector) <- days_vector
names(roulette_vector) <- days_vector

# Calculate total gains for poker and roulette
total_poker <- sum(poker_vector)
total_roulette <- sum(roulette_vector)

# Check if you realized higher total gains in poker than in roulette

print(total_poker > total_roulette)
```

2.8 Vector selection: the good times

Your hunch seemed to be right. It appears that the poker game is more your cup of tea than roulette.

Another possible route for investigation is your performance at the beginning of the working week compared to the end of it. You did have a couple of Margarita cocktails at the end of the week…

To answer that question, you only want to focus on a selection of the `total_vector`. In other words, our goal is to select specific elements of the vector. To select elements of a vector (and later matrices, data frames, …), you can use square brackets. Between the square brackets, you indicate what elements to select. For example, to select the first element of the vector, you type `poker_vector[1]`. To select the second element of the vector, you type `poker_vector[2]`, etc. Notice that the first element in a vector has index 1, not 0 as in many other programming languages.

Instructions 100 XP

Assign the poker results of Wednesday to the variable `poker_wednesday`.

ex_17.R

```r
# Poker and roulette winnings from Monday to Friday:
poker_vector <- c(140, -50, 20, -120, 240)
roulette_vector <- c(-24, -50, 100, -350, 10)
days_vector <- c("Monday", "Tuesday", "Wednesday", "Thursday", "Friday")
names(poker_vector) <- days_vector
names(roulette_vector) <- days_vector

# Define a new variable based on a selection
poker_wednesday <- poker_vector[3]
```

2.9 Vector selection: the good times (2)

How about analyzing your midweek results?

To select multiple elements from a vector, you can add square brackets at the end of it. You can indicate between the brackets what elements should be selected. For example: suppose you want to select the first and the fifth day of the week: use the vector `c(1, 5)` between the square brackets. For example, the code below selects the first and fifth element of poker_vector:

```r
poker_vector[c(1, 5)]
```

Instructions 100 XP

Assign the poker results of Tuesday, Wednesday and Thursday to the variable `poker_midweek`.

ex_18.R

```r
# Poker and roulette winnings from Monday to Friday:
poker_vector <- c(140, -50, 20, -120, 240)
roulette_vector <- c(-24, -50, 100, -350, 10)
days_vector <- c("Monday", "Tuesday", "Wednesday", "Thursday", "Friday")
names(poker_vector) <- days_vector
names(roulette_vector) <- days_vector

# Define a new variable based on a selection
poker_midweek <- poker_vector[c(2, 3, 4)]
```

2.10 Vector selection: the good times (3)

Selecting multiple elements of poker_vector with `c(2, 3, 4)` is not very convenient. Many statisticians are lazy people by nature, so they created an easier way to do this: `c(2, 3, 4)` can be abbreviated to `2:4`, which generates a vector with all natural numbers from 2 up to 4.

So, another way to find the mid-week results is `poker_vector[2:4]`. Notice how the vector `2:4` is placed between the square brackets to select element 2 up to 4.

Instructions 100 XP

Assign to `roulette_selection_vector` the roulette results from Tuesday up to Friday; make use of : if it makes things easier for you.

ex_19.R

```r
# Poker and roulette winnings from Monday to Friday:
poker_vector <- c(140, -50, 20, -120, 240)
roulette_vector <- c(-24, -50, 100, -350, 10)
days_vector <- c("Monday", "Tuesday", "Wednesday", "Thursday", "Friday")
names(poker_vector) <- days_vector
names(roulette_vector) <- days_vector

# Define a new variable based on a selection
```

```
roulette_selection_vector <- roulette_vector[2:5]
```

2.11 Vector selection: the good times (4)

Another way to tackle the previous exercise is by using the names of the vector elements (Monday, Tuesday, …) instead of their numeric positions. For example,

```
poker_vector[c("Monday"]
```

will select the first element of poker_vector since "Monday" is the name of that first element.

Just like you did in the previous exercise with numerics, you can also use the element names to select multiple elements, for example:

```
poker_vector[c("Monday","Tuesday")]
```

Instructions `100 XP`

- Select the first three elements in `poker_vector` by using their names: `"Monday"`, `"Tuesday"` and `"Wednesday"`. Assign the result of the selection to `poker_start`.
- Calculate the average of the values in `poker_start` with the `mean()` function. Simply print out the result so you can inspect it.

ex_20.R

```
# Poker and roulette winnings from Monday to Friday:
poker_vector <- c(140, -50, 20, -120, 240)
roulette_vector <- c(-24, -50, 100, -350, 10)
days_vector <- c("Monday", "Tuesday", "Wednesday", "Thursday", "Friday")
names(poker_vector) <- days_vector
names(roulette_vector) <- days_vector

# Select poker results for Monday, Tuesday and Wednesday
poker_start <- poker_vector[c("Monday", "Tuesday", "Wednesday")]

# Calculate the average of the elements in poker_start
mean(poker_start)
```

2.12 Selection by comparison - Step 1

By making use of comparison operators, we can approach the previous question in a more proactive way.

The (logical) comparison operators known to R are:

- `<` for less than
- `>` for greater than
- `<=` for less than or equal to
- `>=` for greater than or equal to
- `==` for equal to each other
- `!=` not equal to each other

As seen in the previous chapter, stating 6 > 5 returns TRUE. The nice thing about R is that you can use these comparison operators also on vectors. For example:

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

This command tests for every element of the vector if the condition stated by the comparison operator is `TRUE` or `FALSE`.

Instructions `100 XP`

- Check which elements in `poker_vector` are positive (i.e. `> 0`) and assign this to selection_vector.
- Print out selection_vector so you can inspect it. The printout tells you whether you won `(TRUE)` or lost `(FALSE)` any money for each day.

ex_21.R

```
# Poker and roulette winnings from Monday to Friday:
poker_vector <- c(140, -50, 20, -120, 240)
roulette_vector <- c(-24, -50, 100, -350, 10)
days_vector <- c("Monday", "Tuesday", "Wednesday", "Thursday", "Friday")
names(poker_vector) <- days_vector
names(roulette_vector) <- days_vector

# Which days did you make money on poker?
selection_vector <-
    poker_vector > 0
```

```
# Print out selection_vector
print(selection_vector)
```

2.13 Selection by comparison - Step 2

Working with comparisons will make your data analytical life easier. Instead of selecting a subset of days to investigate yourself (like before), you can simply ask R to return only those days where you realized a positive return for poker.

In the previous exercises you used `selection_vector <- poker_vector > 0` to find the days on which you had a positive poker return. Now, you would like to know not only the days on which you won, but also how much you won on those days.

You can select the desired elements, by putting `selection_vector` between the square brackets that follow poker_vector:

```
poker_vector[selection_vector]
```

R knows what to do when you pass a logical vector in square brackets: it will only select the elements that correspond to `TRUE` in `selection_vector`.

Instructions `100 XP`

Use `selection_vector` in square brackets to assign the amounts that you won on the profitable days to the variable `poker_winning_days`.

ex_22.R

```
# Poker and roulette winnings from Monday to Friday:
poker_vector <- c(140, -50, 20, -120, 240)
roulette_vector <- c(-24, -50, 100, -350, 10)
days_vector <- c("Monday", "Tuesday", "Wednesday", "Thursday", "Friday")
names(poker_vector) <- days_vector
names(roulette_vector) <- days_vector

# Which days did you make money on poker?
selection_vector <- poker_vector > 0

# Select from poker_vector these days
poker_winning_days <- poker_vector[selection_vector]
```

2.14 Advanced selection

Just like you did for poker, you also want to know those days where you realized a positive return for roulette.

2.14.1 Instructions `100 XP`

- Create the variable selection_vector, this time to see if you made profit with roulette for different days.
- Assign the amounts that you made on the days that you ended positively for roulette to the variable roulette_winning_days. This vector thus contains the positive winnings of roulette_vector.

ex_23.R

```r
# Poker and roulette winnings from Monday to Friday:
poker_vector <- c(140, -50, 20, -120, 240)
roulette_vector <- c(-24, -50, 100, -350, 10)
days_vector <- c("Monday", "Tuesday", "Wednesday", "Thursday", "Friday")
names(poker_vector) <- days_vector
names(roulette_vector) <- days_vector

# Which days did you make money on roulette?
selection_vector <- roulette_vector > 0

# Select from roulette_vector these days
roulette_winning_days <- roulette_vector[selection_vector]
```

3 Matrices

In this chapter, you will learn how to work with matrices in R. By the end of the chapter, you will be able to create matrices and understand how to do basic computations with them. You will analyze the box office numbers of the Star Wars movies and learn how to use matrices in R. May the force be with you!

3.1 What's a matrix?

In R, a matrix is a collection of elements of the same data type (numeric, character, or logical) arranged into a fixed number of rows and columns. Since you are only working with rows and columns, a matrix is called two-dimensional.

You can construct a matrix in R with the `matrix()` function. Consider the following example:

```r
matrix(1:9, byrow = TRUE, nrow = 3)
```

In the `matrix()` function:

- The first argument is the collection of elements that R will arrange into the rows and columns of the matrix. Here, we use 1:9 which is a shortcut for `c(1, 2, 3, 4, 5, 6, 7, 8, 9)`.
- The argument byrow indicates that the matrix is filled by the rows. If we want the matrix to be filled by the columns, we just place `byrow = FALSE`.
- The third argument nrow indicates that the matrix should have three rows.

Instructions `100 XP`

Construct a matrix with 3 rows containing the numbers 1 up to 9, filled row-wise.

ex__24.R

```r
# Construct a matrix with 3 rows that contain the numbers 1 up to 9
matrix(1:9, byrow = TRUE, nrow = 3)
```

3.2 Analyze matrices, you shall

It is now time to get your hands dirty. In the following exercises you will analyze the box office numbers of the Star Wars franchise. May the force be with you!

In the editor, three vectors are defined. Each one represents the box office numbers from the first three Star Wars movies. The first element of each vector indicates the US box office revenue, the second element refers to the Non-US box office (source: Wikipedia).

In this exercise, you'll combine all these figures into a single vector. Next, you'll build a matrix from this vector.

Instructions `100 XP`

- Use `c(new_hope, empire_strikes, return_jedi)` to combine the three vectors into one vector. Call this vector `box_office`.
- Construct a matrix with 3 rows, where each row represents a movie. Use the `matrix()` function to do this. The first argument is the vector `box_office`, containing all box office figures. Next, you'll have to specify `nrow = 3` and `byrow = TRUE`. Name the resulting matrix `star_wars_matrix`.

ex__25.R

```r
# Box office Star Wars (in millions!)
new_hope <- c(460.998, 314.4)
empire_strikes <- c(290.475, 247.900)
return_jedi <- c(309.306, 165.8)

# Create box_office
box_office <- c(new_hope, empire_strikes, return_jedi)

# Construct star_wars_matrix
star_wars_matrix <- matrix (box_office, nrow = 3, byrow = TRUE)
```

3.3 Naming a matrix

To help you remember what is stored in star_wars_matrix, you would like to add the names of the movies for the rows. Not only does this help you to read the data, but it is also useful to select certain elements from the matrix.

Similar to vectors, you can add names for the rows and the columns of a matrix

```
rownames(my_matrix) <- row_names_vector
colnames(my_matrix) <- col_names_vector
```

We went ahead and prepared two vectors for you: `region`, and `titles`. You will need these vectors to name the columns and rows of star_wars_matrix, respectively.

Instructions 100 XP

- Use colnames() to name the columns of star_wars_matrix with the region vector.
- Use rownames() to name the rows of star_wars_matrix with the titles vector.
- Print out star_wars_matrix to see the result of your work.

ex_26.R

```
# Box office Star Wars (in millions!)
new_hope <- c(460.998, 314.4)
empire_strikes <- c(290.475, 247.900)
return_jedi <- c(309.306, 165.8)

# Construct matrix
star_wars_matrix <-
  matrix(c(new_hope, empire_strikes, return_jedi), nrow = 3, byrow = TRUE)

# Vectors region and titles, used for naming
region <- c("US", "non-US")
titles <- c("A New Hope", "The Empire Strikes Back", "Return of the Jedi")

# Name the columns with region
colnames(star_wars_matrix) <- region

# Name the rows with titles
rownames(star_wars_matrix) <- titles

# Print out star_wars_matrix
print(star_wars_matrix)
```

3.4 Calculating the worldwide box office

The single most important thing for a movie in order to become an instant legend in Tinseltown is its worldwide box office figures.

To calculate the total box office revenue for the three Star Wars movies, you have to take the sum of the US revenue column and the non-US revenue column.

In R, the function `rowSums()` conveniently calculates the totals for each row of a matrix. This function creates a new vector:

```
rowSums(my_matrix)
```

Instructions `100 XP`

Calculate the worldwide box office figures for the three movies and put these in the vector named `worldwide_vector`.

ex_26.R

```
# Construct star_wars_matrix
box_office <- c(460.998, 314.4, 290.475, 247.900, 309.306, 165.8)
region <- c("US", "non-US")
titles <- c("A New Hope",
                "The Empire Strikes Back",
                "Return of the Jedi")

star_wars_matrix <- matrix(box_office,
                      nrow = 3, byrow = TRUE,
                      dimnames = list(titles, region))

# Calculate worldwide box office figures
worldwide_vector <- rowSums(star_wars_matrix)
```

3.5 Adding a column for the Worldwide box office

In the previous exercise you calculated the vector that contained the worldwide box office receipt for each of the three Star Wars movies. However, this vector is not yet part of star_wars_matrix.

You can add a column or multiple columns to a matrix with the cbind() function, which merges matrices and/or vectors together by column. For example:

```
  big_matrix <- cbind(matrix1, matrix2, vector1 ...)
```

Instructions 100 XP

Add `worldwide_vector` as a new column to the `star_wars_matrix` and assign the result to `all_wars_matrix`. Use the `cbind()` function.

ex_27.R

```r
# Construct star_wars_matrix
box_office <- c(460.998, 314.4, 290.475, 247.900, 309.306, 165.8)
region <- c("US", "non-US")
titles <- c("A New Hope",
            "The Empire Strikes Back",
            "Return of the Jedi")

star_wars_matrix <- matrix(box_office,
                      nrow = 3, byrow = TRUE,
                      dimnames = list(titles, region))

# The worldwide box office figures
worldwide_vector <- rowSums(star_wars_matrix)

# Bind the new variable worldwide_vector as a column to star_wars_matrix
all_wars_matrix <- cbind(star_wars_matrix, worldwide_vector)
```

3.6 Adding a row

Just like every action has a reaction, every `cbind()` has an `rbind()`. (We admit, we are pretty bad with metaphors.)

Your R workspace, where all variables you defined 'live' (check out what a workspace is), has already been initialized and contains two matrices:

- star_wars_matrix that we have used all along, with data on the original trilogy,
- star_wars_matrix2, with similar data for the prequels trilogy.

Explore these matrices in the console if you want to have a closer look. If you want to check out the contents of the workspace, you can type `ls()` in the console.

Instructions 100 XP

Use `rbind()` to paste together `star_wars_matrix` and `star_wars_matrix2`, in this order. Assign the resulting matrix to `all_wars_matrix`.

ex_28.R

```
# star_wars_matrix and star_wars_matrix2 are available in your workspace
star_wars_matrix
star_wars_matrix2

# Combine both Star Wars trilogies in one matrix
all_wars_matrix <- rbind(star_wars_matrix, star_wars_matrix2)
```

3.7 The total box office revenue for the entire saga

Just like `cbind()` has `rbind()`, `colSums()` has `rowSums()`. Your R workspace already contains the `all_wars_matrix` that you constructed in the previous exercise; type `all_wars_matrix` to have another look. Let's now calculate the total box office revenue for the entire saga.

Instructions `100 XP`

- Calculate the total revenue for the US and the non-US region and assign total_revenue_vector. You can use the colSums() function.
- Print out total_revenue_vector to have a look at the results.

ex_29.R

```
# all_wars_matrix is available in your workspace
all_wars_matrix

# Total revenue for US and non-US
total_revenue_vector <- colSums(all_wars_matrix)

# Print out total_revenue_vector
print(total_revenue_vector)
```

3.8 Selection of matrix elements

Similar to vectors, you can use the square brackets `[ ]` to select one or multiple elements from a matrix. Whereas vectors have one dimension, matrices have two dimensions. You should therefore use a comma to separate the rows you want to select from the columns. For example:

- `my_matrix[1,2]` selects the element at the first row and second column.
- `my_matrix[1:3,2:4]` results in a matrix with the data on the rows 1, 2, 3 and columns 2, 3, 4.

If you want to select all elements of a row or a column, no number is needed before or after the comma, respectively:

- `my_matrix[,1]` selects all elements of the first column.
- `my_matrix[1,]` selects all elements of the first row.

Back to Star Wars with this newly acquired knowledge! As in the previous exercise, all_wars_matrix is already available in your workspace.

Instructions `100 XP`

- Select the non-US revenue for all movies (the entire second column of `all_wars_matrix`), store the result as `non_us_all`.
- Use `mean()` on `non_us_all` to calculate the average non-US revenue for all movies. Simply print out the result.
- This time, select the non-US revenue for the first two movies in `all_wars_matrix`. Store the result as `non_us_some`.
- Use `mean()` again to print out the average of the values in `non_us_some`.

ex_30.R

```r
# all_wars_matrix is available in your workspace
all_wars_matrix

# Select the non-US revenue for all movies
non_us_all <- all_wars_matrix[,2]

# Average non-US revenue
print(mean(non_us_all))

# Select the non-US revenue for first two movies
non_us_some <- all_wars_matrix[1:2, 2]

# Average non-US revenue for first two movies
print(mean(non_us_some))
```

3.9 A little arithmetic with matrices

Similar to what you have learned with vectors, the standard operators like `+`, `-`,`/`, `*`, etc. work in an element-wise way on matrices in R.

For example, `2 * my_matrix` multiplies each element of `my_matrix` by two.

As a newly-hired data analyst for Lucasfilm, it is your job to find out how many visitors went to each movie for each geographical area. You already have the total revenue figures in `all_wars_matrix`. Assume that the price of a ticket was 5 dollars. Simply dividing the box office numbers by this ticket price gives you the number of visitors.

3.9.1 Instructions `100 XP`

- Divide `all_wars_matrix` by 5, giving you the number of visitors in millions.
- Assign the resulting matrix to visitors.
- Print out visitors so you can have a look.

ex_31.R

```
# all_wars_matrix is available in your workspace
all_wars_matrix

# Estimate the visitors
visitors <- all_wars_matrix / 5

# Print the estimate to the console
print(visitors)
```

3.10 A little arithmetic with matrices (2)

Just like `2 * my_matrix` multiplied every element of my_matrix by two, `my_matrix1 * my_matrix2` creates a matrix where each element is the product of the corresponding elements in my_matrix1 and my_matrix2.

After looking at the result of the previous exercise, big boss Lucas points out that the ticket prices went up over time. He asks to redo the analysis based on the prices you can find in `ticket_prices_matrix` (source: imagination).

Those who are familiar with matrices should note that this is not the standard matrix multiplication for which you should use %*% in R.

3.10.1 Instructions `100 XP`

- Divide all_wars_matrix by ticket_prices_matrix to get the estimated number of US and non-US visitors for the six movies. Assign the result to visitors.
- From the visitors matrix, select the entire first column, representing the number of visitors in the US. Store this selection as `us_visitors`.
- Calculate the average number of US visitors; print out the result.

ex_32.R

```r
# all_wars_matrix and ticket_prices_matrix are available in your workspace
all_wars_matrix
ticket_prices_matrix

# Estimated number of visitors
visitors <- all_wars_matrix / ticket_prices_matrix

# US visitors
us_visitors <- visitors[, 1]

# Average number of US visitors
print(mean(us_visitors))
```

4 Factors

Data often falls into a limited number of categories. For example, human hair color can be categorized as black, brown, blond, red, grey, or white—and perhaps a few more options for people who color their hair. In R, categorical data is stored in factors. Factors are very important in data analysis, so start learning how to create, subset, and compare them now.

4.1 What's a factor and why would you use it?

In this chapter you dive into the wonderful world of factors.

The term factor refers to a statistical data type used to store categorical variables. The difference between a categorical variable and a continuous variable is that a categorical variable can belong to a limited number of categories. A continuous variable, on the other hand, can correspond to an infinite number of values.

It is important that R knows whether it is dealing with a continuous or a categorical variable, as the statistical models you will develop in the future treat both types differently. (You will see later why this is the case.)

A good example of a categorical variable is sex. In many circumstances you can limit the sex categories to "Male" or "Female". (Sometimes you may need different categories. For example, you may need to consider chromosomal variation, hermaphroditic animals, or different cultural norms, but you will always have a finite number of categories.)

Instructions `100 XP`

Assign to variable theory the value "factors".

ex_33.R

```r
# Assign to the variable theory what this chapter is about!
theory <- "factors"
```

4.2 What's a factor and why would you use it? (2)

To create factors in R, you make use of the function `factor()`. First thing that you have to do is create a vector that contains all the observations that belong to a limited number of categories. For example, `sex_vector` contains the sex of 5 different individuals:

```
sex_vector <- c("Male","Female","Female","Male","Male")
```

It is clear that there are two categories, or in R-terms 'factor levels', at work here: "Male" and "Female".

The function `factor()` will encode the vector as a factor:

```
factor_sex_vector <- factor(sex_vector)
```

Instructions `100 XP`

- Convert the character vector `sex_vector` to a factor with `factor()` and assign the result to `factor_sex_vector`
- Print out factor_sex_vector and assert that R prints out the factor levels below the actual values.

ex_34.R

```
# Sex vector
sex_vector <- c("Male", "Female", "Female", "Male", "Male")

# Convert sex_vector to a factor
factor_sex_vector <- factor(sex_vector)

# Print out factor_sex_vector
print(factor_sex_vector)
```

4.3 What's a factor and why would you use it? (3)

There are two types of categorical variables: a nominal categorical variable and an ordinal categorical variable.

A nominal variable is a categorical variable without an implied order. This means that it is impossible to say that 'one is worth more than the other'. For example, think of the categorical variable animals_vector with the categories "Elephant", "Giraffe", "Donkey" and

"Horse". Here, it is impossible to say that one stands above or below the other. (Note that some of you might disagree ;-)).

In contrast, ordinal variables do have a natural ordering. Consider for example the categorical variable temperature_vector with the categories: "Low", "Medium" and "High". Here it is obvious that "Medium" stands above "Low", and "High" stands above "Medium".

Instructions `100 XP`

Submit the answer to check how R constructs and prints nominal and ordinal variables. Do not worry if you do not understand all the code just yet, we will get to that.

ex_35.R

```
# Animals
animals_vector <- c("Elephant", "Giraffe", "Donkey", "Horse")
factor_animals_vector <- factor(animals_vector)
factor_animals_vector

# Temperature
temperature_vector <- c("High", "Low", "High","Low", "Medium")
factor_temperature_vector <-
    factor(
        temperature_vector,
        order = TRUE,
        levels = c("Low", "Medium", "High")
    )
factor_temperature_vector
```

4.4 Factor levels

When you first get a dataset, you will often notice that it contains factors with specific factor levels. However, sometimes you will want to change the names of these levels for clarity or other reasons. R allows you to do this with the function levels():

```
levels(factor_vector) <- c("name1", "name2",...)
```

A good illustration is the raw data that is provided to you by a survey. A common question for every questionnaire is the sex of the respondent. Here, for simplicity, just two categories were recorded, "M" and "F". (You usually need more categories for survey data; either way, you use a factor to store the categorical data.)

```
survey_vector <- c("M", "F", "F", "M", "M")
```

Recording the sex with the abbreviations "M" and "F" can be convenient if you are collecting data with pen and paper, but it can introduce confusion when analyzing the data. At that point, you will often want to change the factor levels to "Male" and "Female" instead of "M" and "F" for clarity.

Watch out: the order with which you assign the levels is important. If you type `levels(factor_survey_vector)`, you'll see that it outputs `[1] "F" "M"`. If you don't specify the levels of the factor when creating the vector, `R` will automatically assign them alphabetically. To correctly map "F" to "Female" and "M" to "Male", the levels should be set to `c("Female", "Male")`, in this order.

Instructions 100 XP

- Check out the code that builds a factor vector from `survey_vector`. You should use `factor_survey_vector` in the next instruction.
- Change the factor levels of `factor_survey_vector` to `c("Female", "Male")`. Mind the order of the vector elements here.

ex_36.R

```
# Code to build factor_survey_vector
survey_vector <- c("M", "F", "F", "M", "M")
factor_survey_vector <- factor(survey_vector)

# Specify the levels of factor_survey_vector
levels(factor_survey_vector) <- c("F", "M")

levels(factor_survey_vector) <-  c("Female", "Male")
```

4.5 Summarizing a factor

After finishing this course, one of your favorite functions in R will be summary (). This will give you a quick overview of the contents of a variable:

```
summary(my_var)
```

Going back to our survey, you would like to know how many `"Male"` responses you have in your study, and how many `"Female"` responses. The `summary()` function gives you the answer to this question.

Instructions 100 XP

Ask a `summary()` of the survey_vector and factor_survey_vector. Interpret the results of both vectors. Are they both equally useful in this case?

ex_37.R

```r
# Build factor_survey_vector with clean levels
survey_vector <- c("M", "F", "F", "M", "M")
factor_survey_vector <- factor(survey_vector)
levels(factor_survey_vector) <- c("Female", "Male")
factor_survey_vector

# Generate summary for survey_vector
summary(survey_vector)

# Generate summary for factor_survey_vector
summary(factor_survey_vector)
```

4.6 Battle of the sexes

You might wonder what happens when you try to compare elements of a factor. In `factor_survey_vector` you have a factor with two levels: `"Male"` and `"Female"`. But how does R value these relative to each other?

Instructions 100 XP

Read the code in the editor and submit the answer to test if `male` is greater than (`>`) `female`.

ex_38.R

```r
# Build factor_survey_vector with clean levels
survey_vector <- c("M", "F", "F", "M", "M")
factor_survey_vector <- factor(survey_vector)
levels(factor_survey_vector) <- c("Female", "Male")
```

```
# Male
male <- factor_survey_vector[1]

# Female
female <- factor_survey_vector[2]

# Battle of the sexes: Male 'larger' than female?
male > female
```

4.7 Ordered factors

Ordered factors Since "Male" and "Female" are unordered (or nominal) factor levels, R returns a warning message, telling you that the greater than operator is not meaningful. As seen before, R attaches an equal value to the levels for such factors.

But this is not always the case! Sometimes you will also deal with factors that do have a natural ordering between its categories. If this is the case, we have to make sure that we pass this information to R…

Let us say that you are leading a research team of five data analysts and that you want to evaluate their performance. To do this, you track their speed, evaluate each analyst as "slow", "medium" or "fast", and save the results in speed_vector.

Instructions 100 XP

As a first step, assign speed_vector a vector with 5 entries, one for each analyst. Each entry should be either "slow", "medium", or "fast". Use the list below:

- Analyst 1 is medium,
- Analyst 2 is slow,
- Analyst 3 is slow,
- Analyst 4 is medium and
- Analyst 5 is fast.

No need to specify these are factors yet.

ex_39.R

```
speed_vector <- c(
    "medium",
    "slow",
    "slow",
    "medium",
    "fast"
)
```

4.8 Ordered factors (2)

speed_vector should be converted to an ordinal factor since its categories have a natural ordering. By default, the function factor() transforms speed_vector into an unordered factor. To create an ordered factor, you have to add two additional arguments: ordered and levels.

```
factor(some_vector,
       ordered = TRUE,
       levels = c("lev1", "lev2" ...))
```

By setting the argument ordered to TRUE in the function factor(), you indicate that the factor is ordered. With the argument levels you give the values of the factor in the correct order.

Instructions 100 XP

From speed_vector, create an ordered factor vector: factor_speed_vector. Set ordered to TRUE, and set levels to c("slow", "medium", "fast").

ex_40.R

```
# Create speed_vector
speed_vector <- c("medium", "slow", "slow", "medium", "fast")

# Convert speed_vector to ordered factor vector
factor_speed_vector <-
factor(
    speed_vector,
    ordered = TRUE,
    levels = c("slow", "medium", "fast")
)


# Print factor_speed_vector
factor_speed_vector
```

```
summary(factor_speed_vector)
```

4.9 Comparing ordered factors

Having a bad day at work, ‘data analyst number two’ enters your office and starts complaining that ‘data analyst number five’ is slowing down the entire project. Since you know that ‘data analyst number two’ has the reputation of being a smarty-pants, you first decide to check if his statement is true.

The fact that `factor_speed_vector` is now ordered enables us to compare different elements (the data analysts in this case). You can simply do this by using the well-known operators.

Instructions `100 XP`

- Use `[2]` to select from `factor_speed_vector` the factor value for the second data analyst. Store it as `da2`.
- Use `[5]` to select the `factor_speed_vector` factor value for the fifth data analyst. Store it as `da5`.
- Check if `da2` is greater than `da5`; simply print out the result. Remember that you can use the `>` operator to check whether one element is larger than the other.

ex_41.R

```
# Create factor_speed_vector
speed_vector <- c("medium", "slow", "slow", "medium", "fast")
factor_speed_vector <-
  factor(
    speed_vector,
    ordered = TRUE,
    levels = c("slow", "medium", "fast")
  )

# Factor value for second data analyst
da2 <- factor_speed_vector[2]

# Factor value for fifth data analyst
da5 <- factor_speed_vector[5]

# Is data analyst 2 faster than data analyst 5?
 print(da2 > da5)
```

5 Data frames

5.1 What's a data frame?

You may remember from the chapter about matrices that all the elements that you put in a matrix should be of the same type. Back then, your dataset on Star Wars only contained numeric elements.

When doing a market research survey, however, you often have questions such as:

- 'Are you married?' or 'yes/no' questions (logical)
- 'How old are you?' (numeric)
- 'What is your opinion on this product?' or other 'open-ended' questions (character)
- … The output, namely the respondents' answers to the questions formulated above, is a dataset of different data types. You will often find yourself working with datasets that contain different data types instead of only one.

A data frame has the variables of a dataset as columns and the observations as rows. This will be a familiar concept for those coming from different statistical software packages such as SAS or SPSS.

Instructions `100 XP`

Submit the answer. The data from the built-in example data frame `mtcars` will be printed to the console.

```
                   mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
Duster 360        14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
Merc 240D         24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
Merc 230            22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
Merc 280            19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
Merc 280C           17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
Merc 450SE          16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
Merc 450SL          17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
Merc 450SLC         15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
Fiat 128            32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
Honda Civic         30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
Toyota Corolla      33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
Toyota Corona       21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
Dodge Challenger    15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
AMC Javelin         15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
Camaro Z28          13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
Pontiac Firebird    19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
Fiat X1-9           27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
Porsche 914-2       26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
Lotus Europa        30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
Ford Pantera L      15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
Ferrari Dino        19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
Maserati Bora       15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
Volvo 142E          21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

5.2 Quick, have a look at your dataset

Wow, that is a lot of cars!

Working with large datasets is not uncommon in data analysis. When you work with (extremely) large datasets and data frames, your first task as a data analyst is to develop a clear understanding of its structure and main elements. Therefore, it is often useful to show only a small part of the entire dataset.

So how to do this in R? Well, the function `head()` enables you to show the first observations of a data frame. Similarly, the function `tail()` prints out the last observations in your dataset.

Both `head()` and `tail()` print a top line called the 'header', which contains the names of the different variables in your dataset.

Instructions `100 XP`

Call head() on the mtcars dataset to have a look at the header and the first observations.

ex_42.R

```r
# Call head() on mtcars
head(mtcars)
```

5.3 Have a look at the structure

Another method that is often used to get a rapid overview of your data is the function `str()`. The function `str()` shows you the structure of your dataset.

For a data frame it tells you:

- The total number of observations (e.g. 32 car types)
- The total number of variables (e.g. 11 car features)
- A full list of the variables names (e.g. mpg, cyl …)
- The data type of each variable (e.g. num)
- The first observations

Applying the `str()` function will often be the first thing that you do when receiving a new dataset or data frame. It is a great way to get more insight in your dataset before diving into the real analysis.

Instructions `100 XP`

Investigate the structure of mtcars. Make sure that you see the same numbers, variables and data types as mentioned above.

ex_43.R

```r
str(mtcars)
```

5.4 Creating a data frame

Since using built-in datasets is not even half the fun of creating your own datasets, the rest of this chapter is based on your personally developed dataset. Put your jet pack on because it is time for some space exploration!

As a first goal, you want to construct a data frame that describes the main characteristics of eight planets in our solar system. According to your good friend Buzz, the main features of a planet are:

- The type of planet (Terrestrial or Gas Giant).
- The planet's diameter relative to the diameter of the Earth.
- The planet's rotation across the sun relative to that of the Earth.
- If the planet has rings or not (TRUE or FALSE).

After doing some high-quality research on Wikipedia, you feel confident enough to create the necessary vectors: name, type, diameter, rotation and rings; these vectors have already been coded up in the editor. The first element in each of these vectors correspond to the first observation.

You construct a data frame with the `data.frame()` function. As arguments, you pass the vectors from before: they will become the different columns of your data frame. Because every column has the same length, the vectors you pass should also have the same length. But don't forget that it is possible (and likely) that they contain different types of data.

Instructions `100 XP`

Use the function `data.frame()` to construct a data frame. Pass the vectors `name`, `type`, `diameter`, `rotation` and `rings` as arguments to `data.frame()`, in this order. Call the resulting data frame `planets_df`.

ex__44.R

```
name <- c("Mercury", "Venus", "Earth",
          "Mars", "Jupiter", "Saturn",
          "Uranus", "Neptune")
type <- c("Terrestrial planet",
          "Terrestrial planet",
          "Terrestrial planet",
          "Terrestrial planet", "Gas giant",
          "Gas giant", "Gas giant", "Gas giant")
diameter <- c(0.382, 0.949, 1, 0.532,
              11.209, 9.449, 4.007, 3.883)
```

```
rotation <- c(58.64, -243.02, 1, 1.03,
              0.41, 0.43, -0.72, 0.67)
rings <- c(FALSE, FALSE, FALSE, FALSE, TRUE, TRUE, TRUE, TRUE)
planets_df <-
    data.frame(name, type, diameter, rotation, rings)
```

5.5 Creating a data frame (2)

The `planets_df` data frame should have 8 observations and 5 variables. It has been made available in the workspace, so you can directly use it.

Instructions 100 XP

Use `str()` to investigate the structure of the new planets_df variable.

ex_45.R

```
str(planets_df)
```

5.6 Selection of data frame elements

Similar to vectors and matrices, you select elements from a data frame with the help of square brackets `[ ]`. By using a comma, you can indicate what to select from the rows and the columns respectively. For example:

- `my_df[1,2]` selects the value at the first row and second column in `my_df`.
- `my_df[1:3,2:4]` selects rows 1, 2, 3 and columns 2, 3, 4 in my_df.

Sometimes you want to select all elements of a row or column. For example, `my_df[1, ]` selects all elements of the first row. Let us now apply this technique on `planets_df`!

Instructions 100 XP

- From `planets_df`, select the diameter of Mercury: this is the value at the first row and the third column. Simply print out the result.
- From `planets_df`, select all data on Mars (the fourth row). Simply print out the result.

ex_46.R

```r
# The planets_df data frame from the previous exercise is pre-loaded

# Print out diameter of Mercury (row 1, column 3)
print(planets_df[1, 3])

# Print out data for Mars (entire fourth row)
print(planets_df[4, ])
```

5.7 Selection of data frame elements (2)

Instead of using numerics to select elements of a data frame, you can also use the variable names to select columns of a data frame.

Suppose you want to select the first three elements of the type column. One way to do this is

```r
planets_df[1:3, 2]
```

A possible disadvantage of this approach is that you have to know (or look up) the column number of type, which gets hard if you have a lot of variables. It is often easier to just make use of the variable name:

```r
planets_df[1:3,"type"]
```

Instructions `100 XP`

Select and print out the first 5 values in the `"diameter"` column of `planets_df`.

ex_47.R

```r
print(planets_df[1:5, "diameter"])
```

5.8 Only planets with rings

You will often want to select an entire column, namely one specific variable from a data frame. If you want to select all elements of the variable diameter, for example, both of these will do the trick:

```
planets_df[,3]
planets_df[,"diameter"]
```

However, there is a short-cut. If your columns have names, you can use the `$` sign:

```
planets_df$diameter
```

Instructions 100 XP

- Use the `$` sign to select the `rings` variable from `planets_df`. Store the vector that results as `rings_vector`.
- Print out `rings_vector` to see if you got it right.

ex_48.R

```
# planets_df is pre-loaded in your workspace

# Select the rings variable from planets_df
rings_vector <- planets_df$rings

# Print out rings_vector
print(rings_vector)
```

5.9 Only planets with rings (2)

You probably remember from high school that some planets in our solar system have rings and others do not. Unfortunately you can not recall their names. Could R help you out?

If you type rings_vector in the console, you get:

> [1] FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE

This means that the first four observations (or planets) do not have a ring (`FALSE`), but the other four do (`TRUE`). However, you do not get a nice overview of the names of these planets, their diameter, etc. Let's try to use `rings_vector` to select the data for the four planets with rings.

Instructions 100 XP

The code in the editor selects the name column of all planets that have rings. Adapt the code so that instead of only the name column, all columns for planets that have rings are selected.

ex_49.R

```
# planets_df and rings_vector are pre-loaded in your workspace

# Adapt the code to select all columns for planets with rings
planets_df[rings_vector, "name"]
planets_df[rings_vector, ]
```

5.10 Only planets with rings but shorter

So what exactly did you learn in the previous exercises? You selected a subset from a data frame (planets_df) based on whether or not a certain condition was true (rings or no rings), and you managed to pull out all relevant data. Pretty awesome! By now, NASA is probably already flirting with your CV ;-).

Now, let us move up one level and use the function subset(). You should see the subset() function as a short-cut to do exactly the same as what you did in the previous exercises.

```
subset(my_df, subset = some_condition)
```

The first argument of subset() specifies the dataset for which you want a subset. By adding the second argument, you give R the necessary information and conditions to select the correct subset.

The code below will give the exact same result as you got in the previous exercise, but this time, you didn't need the rings_vector!

```
subset(planets_df, subset = rings)
```

Instructions 100 XP

Use subset() on planets_df to select planets that have a diameter smaller than Earth. Because the diameter variable is a relative measure of the planet's diameter w.r.t that of planet Earth, your condition is diameter < 1.

ex_50.R

```
# planets_df is pre-loaded in your workspace

# Select planets with diameter < 1
subset(planets_df, diameter < 1)
```

5.11 Sorting

Making and creating rankings is one of mankind's favorite affairs. These rankings can be useful (best universities in the world), entertaining (most influential movie stars) or pointless (best 007 look-a-like).

In data analysis you can sort your data according to a certain variable in the dataset. In R, this is done with the help of the function `order()`.

`order()` is a function that gives you the ranked position of each element when it is applied on a variable, such as a vector for example:

```
a <- c(100, 10, 1000)
order(a)
```

10, which is the second element in a, is the smallest element, so 2 comes first in the output of order(a). 100, which is the first element in a is the second smallest element, so 1 comes second in the output of `order(a)`.

This means we can use the output of order(a) to reshuffle a:

```
a[order(a)]
```

Instructions `100 XP`

Experiment with the `order()` function in the console. Submit the answer when you are ready to continue.

ex_51.R

```
x <- rnorm(10)
order(x)
```

5.12 Sorting your data frame

Alright, now that you understand the `order()` function, let us do something useful with it. You would like to rearrange your data frame such that it starts with the smallest planet and ends with the largest one. A sort on the `diameter` column.

Instructions `100 XP`

- Call `order()` on `planets_df$diameter` (the `diameter` column of `planets_df`). Store the result as `positions`.
- Now reshuffle `planets_df` with the positions vector as row indexes inside square brackets. Keep all columns. Simply print out the result.

ex__52.R

```r
# planets_df is pre-loaded in your workspace

# Use order() to create positions
positions <-  order(planets_df$diameter)

# Use positions to sort planets_df
planets_df[positions, ]
```

6 Lists

As opposed to vectors, lists can hold components of different types, just as your to-do lists can contain different categories of tasks. This chapter will teach you how to create, name, and subset these lists.

6.1 Lists, why would you need them?

Congratulations! At this point in the course you are already familiar with:

- **Vectors (one dimensional array)** can hold numeric, character or logical values. The elements in a vector all have the same data type.
- **Matrices (two dimensional array)** can hold numeric, character or logical values. The elements in a matrix all have the same data type.
- **Data frames (two-dimensional objects)** can hold numeric, character or logical values. Within a column all elements have the same data type, but different columns can be of different data type. Pretty sweet for an R newbie, right? ;-)

Instructions `100 XP`

Submit the answer to start learning everything about lists!

6.2 Lists, why would you need them? (2)

A list in R is similar to your to-do list at work or school: the different items on that list most likely differ in length, characteristic, and type of activity that has to be done.

A list in R allows you to gather a variety of objects under one name (that is, the name of the list) in an ordered way. These objects can be matrices, vectors, data frames, even other lists, etc. It is not even required that these objects are related to each other in any way.

You could say that a list is some kind super data type: you can store practically any piece of information in it!

Instructions 100 XP

Just submit the answer to start the first exercise on lists.

6.3 Creating a list

Let us create our first list! To construct a list you use the function `list()`:

```
my_list <- list(comp1, comp2 ...)
```

The arguments to the list function are the list components. Remember, these components can be matrices, vectors, other lists, …

Instructions 100 XP

Construct a list, named `my_list`, that contains the variables `my_vector`, `my_matrix` and `my_df` as list components.

ex_53.R

```
# Vector with numerics from 1 up to 10
my_vector <- 1:10

# Matrix with numerics from 1 up to 9
my_matrix <- matrix(1:9, ncol = 3)

# First 10 elements of the built-in data frame mtcars
my_df <- mtcars[1:10,]

# Construct list with these different elements:
my_list <- list(my_vector, my_matrix, my_df)
```

6.4 Creating a named list

Well done, you're on a roll!

Just like on your to-do list, you want to avoid not knowing or remembering what the components of your list stand for. That is why you should give names to them:

my_list <- list(name1 = your_comp1, name2 = your_comp2)

This creates a list with components that are named name1, name2, and so on. If you want to name your lists after you've created them, you can use the `names()` function as you did with vectors. The following commands are fully equivalent to the assignment above:

my_list <- list(your_comp1, your_comp2) names(my_list) <- c(“name1”, “name2”)

Instructions `100 XP`

- Change the code of the previous exercise (see editor) by adding names to the components. Use for my_vector the name vec, for my_matrix the name mat and for `my_df` the name df.
- Print out `my_list` so you can inspect the output.

ex_54.R

```
# Vector with numerics from 1 up to 10
my_vector <- 1:10

# Matrix with numerics from 1 up to 9
my_matrix <- matrix(1:9, ncol = 3)

# First 10 elements of the built-in data frame mtcars
my_df <- mtcars[1:10,]

# Adapt list() call to give the components names
my_list <- list(my_vector, my_matrix, my_df)
names(my_list) <- c("vec", "mat", "df")
# Print out my_list
print(my_list)
```

6.5 Creating a named list (2)

Being a huge movie fan (remember your job at LucasFilms), you decide to start storing information on good movies with the help of lists.

Start by creating a list for the movie “The Shining”. We have already created the variables `mov`, `act` and `rev` in your R workspace. Feel free to check them out in the console.

Instructions 100 XP

Complete the code in the editor to create `shining_list`; it contains three elements:

- `moviename`: a character string with the movie title (stored in mov)
- `actors`: a vector with the main actors' names (stored in act)
- `reviews`: a data frame that contains some reviews (stored in rev)

Do not forget to name the list components accordingly (names are moviename, actors and reviews).

ex_55.R

```
# The variables mov, act and rev are available

# Finish the code to build shining_list
shining_list <-
    list(
        moviename = mov,
        actors = act,
        reviews = rev
    )
```

6.6 Selecting elements from a list

Your list will often be built out of numerous elements and components. Therefore, getting a single element, multiple elements, or a component out of it is not always straightforward.

One way to select a component is using the numbered position of that component. For example, to "grab" the first component of `shining_list` you type

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

A quick way to check this out is typing it in the console. Important to remember: to select elements from vectors, you use single square brackets: `[ ]`. Don't mix them up!

You can also refer to the names of the components, with `[[ ]]` or with the `$` sign. Both will select the data frame representing the reviews:

```
shining_list[["reviews"]]
shining_list$reviews
```

Besides selecting components, you often need to select specific elements out of these components. For example, with `shining_list[[2]][1]` you select from the second component, actors (`shining_list[[2]]`), the first element (`[1]`). When you type this in the console, you will see the answer is Jack Nicholson.

6.6.1 Instructions `100 XP`

- Select from shining_list the vector representing the actors. Simply print out this vector.
- Select from shining_list the second element in the vector representing the actors. Do a printout like before.

ex_56.R

```r
# shining_list is already pre-loaded in the workspace

# Print out the vector representing the actors
print(shining_list[["actors"]])

# Print the second element of the vector representing the actors
print(shining_list[["actors"]][2])
```

6.7 Creating a new list for another movie

You found reviews of another, more recent, Jack Nicholson movie: The Departed!

Scores	Comments
4.6	I would watch it again
5	Amazing!
4.8	I liked it
5	One of the best movies
4.2	Fascinating plot

It would be useful to collect together all the pieces of information about the movie, like the title, actors, and reviews into a single variable. Since these pieces of data are different shapes, it is natural to combine them in a list variable.

`movie_title`, containing the title of the movie, and `movie_actors`, containing the names of some of the actors in the movie, are available in your workspace.

Instructions 100 XP

- Create two vectors, called `scores` and `comments`, that contain the information from the reviews shown in the table.
- Find the average of the `scores` vector and save it as `avg_review`.
- Combine the `scores` and `comments` vectors into a data frame called `reviews_df`.
- Create a list, called `departed_list`, that contains the `movie_title`, `movie_actors`, `reviews` data frame as `reviews_df`, and the average review score as `avg_review`, and print it out.

ex__57.R

```r
# Use the table from the exercise to define the comments and scores vectors
scores <- c(4.6, 5, 4.8, 5, 4.2)
comments <-
    c(
        "I would watch it again",
        "Amazing!",
        "I liked it",
        "One of the best movies",
        "Fascinating plot"
    )

# Save the average of the scores vector as avg_review
avg_review <- mean(scores)

# Combine scores and comments into the reviews_df data frame
reviews_df <- data.frame(scores, comments)

# Create and print out a list, called departed_list
departed_list <- list(
    movie_title,
    movie_actors,
    reviews_df,
    avg_review
)
print(departed_list)
```

Part II

R fundamentals in programming, data managment and visualization

We dedicate this part to overview the basics to program in R. The aim of this part is building the basis for Machine learning, namely *control flow*, *loops*, *functional programming*.

Relational and logical operators, and conditional statements

Conditionals and Control Flow

In this chapter, you'll learn about relational operators for comparing R objects, and logical operators like "and" and "or" for combining TRUE and FALSE values. Then, you'll use this knowledge to build conditional statements.

Equality

The most basic form of comparison is equality. Let's briefly recap its syntax. The following statements all evaluate to `TRUE` (feel free to try them out in the console).

```
3 == (2 + 1)
"intermediate" != "r"
TRUE != FALSE
"Rchitect" != "rchitect"
```

Notice from the last expression that R is case sensitive: "R" is not equal to "r". Keep this in mind when solving the exercises in this chapter!

Instructions `100 XP`

- In the editor on the right, write R code to see if `TRUE` equals `FALSE`.
- Likewise, check if `-6 * 14` is not equal to `17 - 101`.
- Next up: comparison of character strings. Ask R whether the strings "useR" and "user" are equal.
- Finally, find out what happens if you compare logicals to numerics: are `TRUE` and 1 equal?

ex_001.R

```
# Comparison of logicals
TRUE == FALSE

# Comparison of numerics
```

```
-6 * 14 != 17 -101
# Comparison of character strings
"useR" == "user"

# Compare a logical with a numeric
TRUE == 1
```

Greater and less than

Apart from equality operators, Filip also introduced the less than and greater than operators: $<$ and $>$. You can also add an equal sign to express less than or equal to or greater than or equal to, respectively. Have a look at the following R expressions, that all evaluate to FALSE:

```
(1 + 2) > 4
"dog" < "Cats"
TRUE <= FALSE
```

Remember that for string comparison, R determines the greater than relationship based on alphabetical order. Also, keep in mind that `TRUE` is treated as 1 for arithmetic, and `FALSE` is treated as 0. Therefore, `FALSE < TRUE` is `TRUE`.

Instructions `100 XP`

- Write R expressions to check whether:
- `-6 * 5 + 2` is greater than or equal to `-10 + 1`.
- “raining” is less than or equal to “raining dogs”.
- `TRUE` is greater than `FALSE`.

ex__002.R

```
# Comparison of numerics

-6 * 5 + 2 >= -10 +1

# Comparison of character strings
"raining" <=  "raining dogs"
```

```
# Comparison of logicals

TRUE > FALSE
```

Compare vectors

You are already aware that R is very good with vectors. Without having to change anything about the syntax, R's relational operators also work on vectors.

Let's go back to the example that was started in the video. You want to figure out whether your activity on social media platforms have paid off and decide to look at your results for LinkedIn and Facebook. The sample code in the editor initializes the vectors linkedin and facebook. Each of the vectors contains the number of profile views your LinkedIn and Facebook profiles had over the last seven days.

Instructions `100 XP`

Using relational operators, find a logical answer, i.e. `TRUE` or `FALSE`, for the following questions:

- On which days did the number of LinkedIn profile views exceed 15? When was your
- LinkedIn profile viewed only 5 times or fewer? When was your LinkedIn profile
- visited more often than your Facebook profile?

ex003.R

```
# The linkedin and facebook vectors have already been created for you
# The linkedin and facebook vectors have already been created for you
linkedin <- c(16, 9, 13, 5, 2, 17, 14)
facebook <- c(17, 7, 5, 16, 8, 13, 14)

# Popular days

linkedin > 15
# Quiet days
linkedin <= 5

# LinkedIn more popular than Facebook
linkedin > facebook
```

Compare matrices

R's ability to deal with different data structures for comparisons does not stop at vectors. Matrices and relational operators also work together seamlessly!

Instead of in vectors (as in the previous exercise), the LinkedIn and Facebook data is now stored in a matrix called `views`. The first row contains the LinkedIn information; the second row the Facebook information. The original vectors `facebook` and `linkedin` are still available as well.

Instructions `100 XP`

Using the relational operators you've learned so far, try to discover the following:

- When were the views exactly equal to 13? Use the views matrix to return a logical matrix.
- For which days were the number of views less than or equal to 14? Again, have R return a logical matrix.

ex_004.R

```r
# The social data has been created for you
linkedin <- c(16, 9, 13, 5, 2, 17, 14)
facebook <- c(17, 7, 5, 16, 8, 13, 14)
views <- matrix(c(linkedin, facebook), nrow = 2, byrow = TRUE)

# When does views equal 13?

views == 13
# When is views less than or equal to 14?

views <= 14
```

& and |

Before you work your way through the next exercises, have a look at the following R expressions. All of them will evaluate to TRUE:

```r
TRUE & TRUE
FALSE | TRUE
```

```
5 <= 5 & 2 < 3
3 < 4 | 7 < 6
```

Watch out: `3 < x < 7` to check if `x` is between 3 and 7 will not work; you'll need `3 < x` & `x < 7` for that.

In this exercise, you'll be working with the `last` variable. This variable equals the value of the `linkedin` vector that you've worked with previously. The `linkedin` vector represents the number of LinkedIn views your profile had in the last seven days, remember? Both the variables `linkedin` and last have been pre-defined for you.

Instructions 100 XP

Write R expressions to solve the following questions concerning the variable last:

- Is `last` under 5 or above 10?
- Is `last` between 15 and 20, excluding 15 but including 20?

ex_005.R

```
# The linkedin and last variable are already defined for you
linkedin <- c(16, 9, 13, 5, 2, 17, 14)
last <- tail(linkedin, 1)

# Is last under 5 or above 10?
last < 5 | last > 10

# Is last between 15 (exclusive) and 20 (inclusive)?
last > 15 | last < 20
```

& and | (2)

Like relational operators, logical operators work perfectly fine with vectors and matrices.

Both the vectors `linkedin` and `facebook` are available again. Also a matrix `-views-` has been defined; its first and second row correspond to the `linkedin` and `facebook` vectors, respectively. Ready for some advanced queries to gain more insights into your social outreach?

Instructions 100 XP

- When did LinkedIn views exceed 10 and did Facebook views fail to reach 10 for a particular day? Use the `linkedin` and `facebook` vectors.
- When were one or both of your LinkedIn and Facebook profiles visited at least 12 times?
- When is the `views` matrix equal to a number between 11 and 14, excluding 11 and including 14?

ex_006.R

```r
# The social data (linkedin, facebook, views) has been created for you

# linkedin exceeds 10 but facebook below 10
linkedin > 10 & facebook < 10


# When were one or both visited at least 12 times?
linkedin >= 12 | facebook >= 12

# When is views between 11 (exclusive) and 14 (inclusive)?

views > 11 & views <= 14
```

Blend it all together

With the things you've learned by now, you're able to solve pretty cool problems.

Instead of recording the number of views for your own LinkedIn profile, suppose you conducted a survey inside the company you're working for. You've asked every employee with a LinkedIn profile how many visits their profile has had over the past seven days. You stored the results in a data frame called `li_df`. This data frame is available in the workspace; type `li_df` in the console to check it out.

Instructions 100 XP

- Select the entire second column, named `day2`, from the `li_df` data frame as a vector and assign it to `second`.

- Use `second` to create a logical vector, thatcontains `TRUE` if the corresponding number of views is strictly greater than 25 or strictly lower than 5 and FALSE otherwise. Store this logical vector as `extremes`.

- Use `sum()` on the `extremes` vector to calculate the number of `TRUE`s in `extremes` (i.e. to calculate the number of employees that are either very popular or very low-profile). Simply print this number to the console.

ex_007.R

```
# li_df is pre-loaded in your workspace

# Select the second column, named day2, from li_df: second
second <- li_df$day2

# Build a logical vector, TRUE if value in second is extreme: extremes
extremes <- second > 25 | second < 5

# Count the number of TRUEs in extremes
print(sum(extremes))
```

The if statement

Before diving into some exercises on the if statement, have another look at its syntax:

```
if (condition) {
  expr
}
```

Remember your vectors with social profile views? Let's look at it from another angle. The `medium` variable gives information about the social website; the `num_views` variable denotes the actual number of views that particular `medium` had on the last day of your recordings. Both variables have been pre- defined for you.

Instructions `100 XP`

- Examine the `if` statement that prints out "Showing LinkedIn information" `if` the `medium` variable equals `"LinkedIn"`. -Code an `if` statement that prints "You are popular!" to the console if the `num_views` variable exceeds 15.

ex_008.R

```r
# Variables related to your last day of recordings
medium <- "LinkedIn"
num_views <- 14

# Examine the if statement for medium
if (medium == "LinkedIn") {
  print("Showing LinkedIn information")
}

# Write the if statement for num_views

if (num_views > 15) {
  print("You are popular!")
}
```

Add an else

You can only use an else statement in combination with an if statement. The else statement does not require a condition; its corresponding code is simply run if all of the preceding conditions in the control structure are FALSE. Here's a recipe for its usage:

```r
if (condition) {
  expr1
} else {
  expr2
}
```

It's important that the else keyword comes on the same line as the closing bracket of the if part!

Both if statements that you coded in the previous exercises are already available to use. It's now up to you to extend them with the appropriate else statements!

Instructions 100 XP

Add an else statement to both control structures, such that

- "Unknown medium" gets printed out to the console when the if-condition on medium does not hold.
  -R prints out "Try to be more visible!" when the if-condition on num_views is not met.

ex_009.R

```
# Variables related to your last day of recordings
medium <- "LinkedIn"
num_views <- 14
# Control structure for medium
if (medium == "LinkedIn") {
  print("Showing LinkedIn information")
} else{
  print("Unknown medium" )
}

# Control structure for num_views
if (num_views > 15) {
  print("You're popular!")
} else{
  print( "Try to be more visible!" )
}
```

Customize further: else if

The `else if` statement allows you to further customize your control structure. You can add as many `else if` statements as you like. Keep in mind that R ignores the remainder of the control structure once a condition has been found that is TRUE and the corresponding expressions have been executed. Here's an overview of the syntax to freshen your memory:

```
if (condition1) {
  expr1
} else if (condition2) {
  expr2
} else if (condition3) {
  expr3
} else {
  expr4
}
```

Again, It's important that the else if keywords comes on the same line as the closing bracket of the previous part of the control construct!

Instructions 100 XP

Add code to both control structures such that:

- R prints out "Showing Facebook information" if `medium` is equal to "Facebook". Remember that R is case sensitive!
- "Your number of views is average" is printed if `num_views` is between 15 (inclusive) and 10 (exclusive). Feel free to change the variables `medium` and `num_views` to see how the control structure respond. In both cases, the existing code should be extended in the `else if` statement. No existing code should be modified.

ex_010.R

```r
# Variables related to your last day of recordings
medium <- "LinkedIn"
num_views <- 14

# Control structure for medium
if (medium == "LinkedIn") {
  print("Showing LinkedIn information")
} else if (medium == "Facebook") {
  # Add code to print correct string when condition is TRUE
  print("Showing Facebook information" )
} else {
  print("Unknown medium")
}

# Control structure for num_views
if (num_views > 15) {
  print("You're popular!")
} else if (num_views <= 15 & num_views > 10) {
  # Add code to print correct string when condition is TRUE
  print("Your number of views is average")
} else {
  print("Try to be more visible!")
}
```

Else if 2.0

You can do anything you want inside if-else constructs. You can even put in another set of conditional statements. Examine the following code chunk:

```r
if (number < 10) {
  if (number < 5) {
    result <- "extra small"
  } else {
    result <- "small"
  }
} else if (number < 100) {
  result <- "medium"
} else {
  result <- "large"
}
print(result)
```

Have a look at the following statements:

(1) If number is set to 6, “small” gets printed to the console.
(2) If number is set to 100, R prints out “medium”.
(3) If number is set to 4, “extra small” gets printed out to the console.
(4) If number is set to 2500, R will generate an error, as result will not be defined.

Select the option that lists all the true statements.

> Run the code or a handwrite test (1, 3).

Take control!

In this exercise, you will combine everything that you’ve learned so far: relational operators, logical operators and control constructs. You’ll need it all!

We’ve pre-defined two values for you: `li` and `fb`, denoting the number of profile views your LinkedIn and Facebook profile had on the last day of recordings. Go through the instructions to create R code that generates a ‘social media score’, sms, based on the values of `li` and `fb`.

Instructions `100 XP`

Finish the control-flow construct with the following behavior:

- If both `li` and `fb` are 15 or higher, set `sms` equal to double the sum of `li` and `fb`.
- If both `li` and `fb` are strictly below 10, set `sms` equal to half the `sum` of `li` and `fb`.
- In all other cases, set sms equal to `li + fb`.
- Finally, print the resulting `sms` variable.

ex_011.R

```r
# Variables related to your last day of recordings
li <- 15
fb <- 9

# Code the control-flow construct
if (li >= 15 & fb >= 15) {
  sms <- 2 * (li + fb)
} else if (li < 10 & fb < 10) {
  sms <- 0.5 * (li + fb)
} else {
  sms <- li + fb
}

# Print the resulting sms to the console
print(sms)
```

7 Loops

Loops can come in handy on numerous occasions. While loops are like repeated if statements, the for loop is designed to iterate over all elements in a sequence. Learn about them in this chapter.

7.1 Write a while loop

Let's get you started with building a while loop from the ground up. Have another look at its recipe:

```
while (condition) {
  expr
}
```

Remember that the condition part of this recipe should become FALSE at some point during the execution. Otherwise, the while loop will go on indefinitely.

If your session expires when you run your code, check the body of your `while` loop carefully.

Have a look at the sample code provided; it initializes the `speed` variables and already provides a `while` loop template to get you started.

Instructions `100 XP`

Code a while loop with the following characteristics:

- The condition of the while loop should check if speed is higher than 30.
- Inside the body of the while loop, print out "Slow down!".
- Inside the body of the while loop, decrease the speed by 7 units and assign this new value to speed again. This step is crucial; otherwise your while loop will never stop and your session will expire.

If your session expires when you run your code, check the body of your while loop carefully: it’s likely that you made a mistake.

ex_012.R

```
# Initialize the speed variable
speed <- 64

# Code the while loop
while (speed >30) {
  print("Slow down!")
  speed <- speed - 7
}

# Print out the speed variable
speed
```

7.2 Throw in more conditionals

In the previous exercise, you simulated the interaction between a driver and a driver’s assistant: When the speed was too high, “Slow down!” got printed out to the console, resulting in a decrease of your speed by 7 units.

There are several ways in which you could make your driver’s assistant more advanced. For example, the assistant could give you different messages based on your speed or provide you with a current speed at a given moment.

A while loop similar to the one you’ve coded in the previous exercise is already available for you to use. It prints out your current speed, but there’s no code that decreases the speed variable yet, which is pretty dangerous. Can you make the appropriate changes?

Instructions 100 XP

If the speed is greater than 48, have R print out “Slow down big time!”, and decrease the speed by 11. Otherwise, have R simply print out “Slow down!”, and decrease the speed by 6. If the session keeps timing out and throwing an error, you are probably stuck in an infinite loop! Check the body of your while loop and make sure you are assigning new values to speed.

ex_013.R

```
# Initialize the speed variable
speed <- 64

# Extend/adapt the while loop
while (speed > 30) {
  print(paste("Your speed is",speed))
  if (speed > 48 ) {
    print("Slow down big time!")
    speed <- speed - 11
  } else {
    print("Slow down!")
    speed <- speed - 6
  }
}
```

7.3 Stop the while loop: break

There are some very rare situations in which severe speeding is necessary: what if a hurricane is approaching and you have to get away as quickly as possible? You don't want the driver's assistant sending you speeding notifications in that scenario, right?

This seems like a great opportunity to include the break statement in the while loop you've been working on. Remember that the break statement is a control statement. When R encounters it, the while loop is abandoned completely.

Instructions 100 XP

Adapt the while loop such that it is abandoned when the speed of the vehicle is greater than 80. This time, the speed variable has been initialized to 88; keep it that way.

ex_014.R

```
# Initialize the speed variable
speed <- 88

while (speed > 30) {
  print(paste("Your speed is", speed))

  # Break the while loop when speed exceeds 80
  if (speed > 80) {
    break
  }

  if (speed > 48) {
    print("Slow down big time!")
    speed <- speed - 11
  } else {
    print("Slow down!")
    speed <- speed - 6
  }
}
```

7.4 Build a while loop from scratch

The previous exercises guided you through developing a pretty advanced while loop, containing a break statement and different messages and updates as determined by control flow constructs. If you manage to solve this comprehensive exercise using a while loop, you're totally ready for the next topic: the for loop.

Instructions 100 XP

Finish the while loop so that it:

- prints out the triple of i, so 3 * i, at each run.
- is abandoned with a break if the triple of i is divisible by 8, but still prints out this triple before breaking.

ex_015.R

```
# Initialize i as 1
i <- 1

# Code the while loop
while (i <= 10) {
  print(3 * i)
  if (i %% 8 ==0 ) {
    break
  }
  i <- i + 1
}
```

7.5 Loop over a vector

In the previous video, Filip told you about two different strategies for using the for loop. To refresh your memory, consider the following loops that are equivalent in R:

```
primes <- c(2, 3, 5, 7, 11, 13)

# loop version 1
for (p in primes) {
  print(p)
}

# loop version 2
for (i in 1:length(primes)) {
  print(primes[i])
}
```

Remember our `linkedin` vector? It's a vector that contains the number of views your LinkedIn profile had in the last seven days. The linkedin vector has been pre-defined so that you can fully focus on the instructions!

Instructions 100 XP

Write a `for` loop that iterates over all the elements of `linkedin` and prints out every element separately. Do this in two ways: using the *loop version 1* and the *loop version 2* in the example code above.

ex_016.R

```
# The linkedin vector has already been defined for you
linkedin <- c(16, 9, 13, 5, 2, 17, 14)

# Loop version 1
for (element in linkedin) {

    print(element)
}

```

```
# Loop version 2
for (i in 1:length(linkedin)){
    print(linkedin[i])
}
```

7.6 Loop over a list

Looping over a list is just as easy and convenient as looping over a vector. There are again two different approaches here:

```
primes_list <- list(2, 3, 5, 7, 11, 13)

# loop version 1
for (p in primes_list) {
  print(p)
}

# loop version 2
for (i in 1:length(primes_list)) {
  print(primes_list[[i]])
}
```

Notice that you need double square brackets - `[[ ]]` - to select the list elements in loop version 2.

Suppose you have a list of all sorts of information on New York City: its population size, the names of the boroughs, and whether it is the capital of the United States. We've already defined a list nyc containing this information (source: Wikipedia).

Instructions `100 XP`

As in the previous exercise, loop over the nyc list in two different ways to print its elements:

- Loop directly over the nyc list (loop version 1).
- Define a looping index and do subsetting using double brackets (loop version 2).

ex_017.R

```
# The nyc list is already specified
nyc <- list(pop = 8405837,
              boroughs = c(
                "Manhattan",
                "Bronx",
                "Brooklyn",
                "Queens",
                "Staten Island"
              ),
              capital = FALSE)

# Loop version 1
for(i in nyc){
    print(i)
}

# Loop version 2
for(i in 1:length(nyc)){
    print(nyc[[i]])
}
```

7.7 Loop over a matrix

In your workspace, there's a matrix ttt, that represents the status of a tic-tac-toe game. It contains the values "X", "O" and "NA". Print out ttt to get a closer look. On row 1 and column 1, there's "O", while on row 3 and column 2 there's "NA".

To solve this exercise, you'll need a for loop inside a for loop, often called a nested loop. Doing this in R is a breeze! Simply use the following recipe:

```
for (var1 in seq1) {
  for (var2 in seq2) {
    expr
  }
}
```

Instructions 100 XP

Finish the nested for loops to go over the elements in `ttt`:

- The outer loop should loop over the rows, with loop index `i` (use `1:nrow(ttt)`).
- The inner loop should loop over the columns, with loop index `j` (use `1:ncol(ttt)`).

- Inside the inner loop, make use of print() and paste() to print out information in the following format: “On row i and column j the board contains x”, where x is the value on that position.

ex_018.R

```r
# The tic-tac-toe matrix ttt has already been defined for you

# define the double for loop
for (i in 1:nrow(ttt)) {
  for (j in 1:ncol(ttt)) {
    print(
      paste(
        "On row",
        i,
        " and column",
        j,
        "the board contains",
        ttt[i, j]
      )
    )
  }
}
```

7.8 Mix it up with control flow

Let’s return to the LinkedIn profile views data, stored in a vector `linkedin`. In the first exercise on `for` loops you already did a simple printout of each element in this vector. A little more in-depth interpretation of this data wouldn’t hurt, right? Time to throw in some conditionals! As with the `while` loop, you can use the `if` and `else` statements inside the `for` loop.

Instructions `100 XP`

Add code to the `for` loop that loops over the elements of the `linkedin` vector:

- If the vector element’s value exceeds 10, print out “You’re popular!”.
- If the vector element’s value does not exceed 10, print out “Be more visible!”

ex_019.r

```r
# The linkedin vector has already been defined for you
linkedin <- c(16, 9, 13, 5, 2, 17, 14)

# Code the for loop with conditionals
for (li in linkedin) {
  if ( li > 10) {
    print("You're popular!")
  } else {
    print("Be more visible!")
  }
  print(li)
}
```

7.9 Next, you break it

A possible solution to the previous exercise has been provided for you. The code loops over the linkedin vector and prints out different messages depending on the values of li.

In this exercise, you will use the break and next statements:

The break statement abandons the active loop: the remaining code in the loop is skipped and the loop is not iterated over anymore. The next statement skips the remainder of the code in the loop, but continues the iteration.

Instructions `100 XP` Extend the for loop with two new, separate if tests as follows:

- If the vector element's value exceeds 16, print out "This is ridiculous, I'm outta here!" and have R abandon the for loop (break).
- If the value is lower than 5, print out "This is too embarrassing!" and fast-forward to the next iteration (next).

ex_019.R

```r
# The linkedin vector has already been defined for you
linkedin <- c(16, 9, 13, 5, 2, 17, 14)

# Adapt/extend the for loop
for (li in linkedin) {
  if (li > 10) {
    print("You're popular!")
  } else {
    print("Be more visible!")
  }

  # Add if statement with break
  if (li > 16 ){
    print("This is ridiculous, I'm outta here!")
    break
  }

  # Add if statement with next

  if (li < 5){
      print("This is too embarrassing!")
      next
  }

  print(li)
}
```

7.10 Build a for loop from scratch

This exercise will not introduce any new concepts on for loops.

We already went ahead and defined a variable rquote. This variable has been split up into a vector that contains separate letters and has been stored in a vector chars with the strsplit() function.

Can you write code that counts the number of r's that come before the first u in rquote?

Instructions `100 XP`

- Initialize the variable rcount, as 0.
- Finish the for loop:
  - if char equals “r”, increase the value of rcount by 1.
  - if char equals “u”, leave the for loop entirely with a break.
- Finally, print out the variable rcount to the console to see if your code is correct.

ex__020.R

```
# Pre-defined variables
rquote <- "r's internals are irrefutably intriguing"
chars <- strsplit(rquote, split = "")[[1]]

# Initialize rcount
rcount <- 0

# Finish the for loop
for (char in chars) {
  if (char == "r"){
      rcount <- rcount + 1
  }

  if (char == "u"){
      break
  }

}

# Print out rcount
print(rcount)
```

8 Functions

Functions are an extremely important concept in almost every programming language, and R is no different. Learn what functions are and how to use them—then take charge by writing your own functions.

8.1 Function documentation

Before even thinking of using an R function, you should clarify which arguments it expects. All the relevant details such as a description, usage, and arguments can be found in the documentation. To consult the documentation on the sample() function, for example, you can use one of following R commands:

```
help(sample)
?sample
```

If you execute these commands, you'll be redirected to <www.rdocumentation.org.>

A quick hack to see the arguments of the `sample()` function is the `args()` function. Try it out in the console:

```
args(sample)
```

In the next exercises, you'll be learning how to use the `mean()` function with increasing complexity. The first thing you'll have to do is get acquainted with the `mean()` function.

Instructions `100 XP`

- Consult the documentation on the `mean()` function: `?mean` or `help(mean)`.
- Inspect the arguments of the mean() function using the args() function.

ex_021.R

```
# Consult the documentation on the mean() function

```

```
help(mean)
# Inspect the arguments of the mean() function
args(mean)
```

8.2 Use a function

The documentation on the `mean()` function gives us quite some information:

- The `mean()` function computes the arithmetic mean.
- The most general method takes multiple arguments: `x` and `...`.
- The `x` argument should be a vector containing numeric, logical or time-related information.

Remember that R can match arguments both by position and by name. Can you still remember the difference? You'll find out in this exercise!

Once more, you'll be working with the view counts of your social network profiles for the past 7 days. These are stored in the `linkedin` and `facebook` vectors and have already been created for you.

Instructions `100 XP`

- Calculate the average number of views for both `linkedin` and `facebook` and assign the result to avg_li and `avg_fb`, respectively. Experiment with different types of argument matching!

-Print out both `avg_li` and `avg_fb`.

ex_022.R

```
# The linkedin and facebook vectors have already been created for you
linkedin <- c(16, 9, 13, 5, 2, 17, 14)
facebook <- c(17, 7, 5, 16, 8, 13, 14)

# Calculate average number of views
avg_li <- mean(linkedin)
avg_fb <- mean(facebook)


# Inspect avg_li and avg_fb
print(avg_li)
```

```r
print(avg_fb)
```

8.3 Use a function (2)

Check the documentation on the `mean()` function again:

```r
?mean
```

The Usage section of the documentation includes two versions of the `mean()`function. The first usage,

```r
mean(x, ...)
```

is the most general usage of the mean function. The 'Default S3 method', however, is:

```r
mean(x, trim = 0, na.rm = FALSE, ...)
```

The `...` is called the ellipsis. It is a way for R to pass arguments along without the function having to name them explicitly. The ellipsis will be treated in more detail in future courses.

For the remainder of this exercise, just work with the second usage of the mean function. Notice that both trim and na.rm have default values. This makes them *optional arguments*.

Instructions `100 XP`

- Calculate the mean of the element-wise sum of `linkedin` and `facebook` and store the result in a variable avg_sum.
- Calculate the mean once more, but this time set the `trim` argument equal to 0.2 and assign the result to `avg_sum_trimmed`.
- Print out both `avg_sum` and `avg_sum_trimmed`; can you spot the difference?

ex_023.R

```r
# The linkedin and facebook vectors have already been created for you
linkedin <- c(16, 9, 13, 5, 2, 17, 14)
facebook <- c(17, 7, 5, 16, 8, 13, 14)

# Calculate the mean of the sum
avg_sum <- mean(linkedin + facebook)
```

```r
# Calculate the trimmed mean of the sum

avg_sum_trimmed <- mean(linkedin + facebook, trim = 0.2)
# Inspect both new variables
print(avg_sum)
print(avg_sum_trimmed)
```

8.4 Use a function (3)

In the video, Filip guided you through the example of specifying arguments of the `sd()` function. The `sd()` function has an optional argument, `na.rm` that specified whether or not to remove missing values from the input vector before calculating the standard deviation.

If you've had a good look at the documentation, you'll know by now that the `mean()` function also has this argument, `na.rm`, and it does the exact same thing. By default, it is set to `FALSE`, as the Usage of the default S3 method shows:

```r
mean(x, trim = 0, na.rm = FALSE, ...)
```

Let's see what happens if your vectors `linkedin` and `facebook` contain missing values (`NA`).

Instructions 100 XP

- Calculate the average number of LinkedIn profile views, without specifying any optional arguments. Simply print the result to the console.
- Calculate the average number of LinkedIn profile views, but this time tell R to strip missing values from the input vector.

ex_024.R

```r
# The linkedin and facebook vectors have already been created for you
linkedin <- c(16, 9, 13, 5, NA, 17, 14)
facebook <- c(17, NA, 5, 16, 8, 13, 14)

# Basic average of linkedin

mean(linkedin)
# Advanced average of linkedin
mean(linkedin, na.rm = TRUE)
```

8.5 Functions inside functions

You already know that R functions return objects that you can then use somewhere else. This makes it easy to use functions inside functions, as you've seen before:

```
speed <- 31
print(paste("Your speed is", speed))
```

Notice that both the `print()` and `paste()` functions use the ellipsis - `...` - as an argument. Can you figure out how they're used?

Instructions `100 XP`

Use `abs()` on `linkedin - facebook` to get the absolute differences between the daily LinkedIn and Facebook profile views. Place the call to `abs()` inside `mean()` to calculate the Mean Absolute Deviation. In the `mean()` call, make sure to specify `na.rm` to treat missing values correctly!

ex_025.R

```
# The linkedin and facebook vectors have already been created for you
linkedin <- c(16, 9, 13, 5, NA, 17, 14)
facebook <- c(17, NA, 5, 16, 8, 13, 14)

# Calculate the mean absolute deviation
mean(abs(linkedin - facebook), na.rm = TRUE)
```

8.6 Write your own function

Wow, things are getting serious… you're about to write your own function! Before you have a go at it, have a look at the following function template:

```
my_fun <- function(arg1, arg2) {
body
}
```

Notice that this recipe uses the assignment operator (`<-`) just as if you were assigning a vector to a variable for example. This is not a coincidence. Creating a function in R basically is the assignment of a function object to a variable! In the recipe above, you're creating a new R variable my_fun, that becomes available in the workspace as soon as you execute the definition. From then on, you can use the my_fun as a function.

Instructions 100 XP

- Create a function `pow_two()`: it takes one argument and returns that number squared (that number times itself).
- Call this newly defined function with 12 as input.
- Next, create a function `sum_abs()`, that takes two arguments and returns the sum of the absolute values of both arguments.
- Finally, call the function `sum_abs()` with arguments `-2` and `3` afterwards.

ex_026.R

```
# Create a function pow_two()

pow_two <- function(x) {
  x ^ 2
}

# Use the function
pow_two(12)

# Create a function sum_abs()

sum_abs <- function(x1, x2){
  abs(x1 + x2)
}

# Use the function
sum_abs(-2, 3)
```

8.7 Write your own function (2)

There are situations in which your function does not require an input. Let's say you want to write a function that gives us the random outcome of throwing a fair die:

```
throw_die <- function() {
number <- sample(1:6, size = 1)
number
}

throw_die()
```

Up to you to code a function that doesn't take any arguments!

Instructions 100 XP

- Define a function, `hello()`. It prints out "Hi there!" and returns `TRUE`. It has no arguments.
- Call the function `hello()`, without specifying arguments of course.

ex_027.R

```r
# Define the function hello()
hello <- function(){
  print("Hi there!")
  return TRUE
}

# Call the function hello()
hello()
```

8.8 Write your own function (3)

Do you still remember the difference between an argument with and without default values? The usage section in the sd() documentation shows the following information:

```r
sd(x, na.rm = FALSE)
```

This tells us that `x` has to be defined for the `sd()` function to be called correctly, however, `na.rm` already has a default value. Not specifying this argument won't cause an error.

You can define default argument values in your own R functions as well. You can use the following recipe to do so:

```r
my_fun <- function(arg1, arg2 = val2) {
body
}
```

The editor on the right already includes an extended version of the `pow_two()` function from before. Can you finish it?

Instructions 100 XP

- Add an optional argument, named `print_info`, that is `TRUE` by default.
- Wrap an if construct around the `print()` function: this function should only be executed if `print_info` is `TRUE`.
- Feel free to experiment with the `pow_two()` function you've just coded.

ex_028.R

```r
# Finish the pow_two() function
pow_two <- function(x, print_info = TRUE) {
y <- x ^ 2
if(print_info){
  print(paste(x, "to the power two equals", y))
}
return(y)
}
```

8.9 Function scoping

An issue that Filip did not discuss in the video is function scoping. It implies that variables that are defined inside a function are not accessible outside that function. Try running the following code and see if you understand the results:

```r
pow_two <- function(x) {
y <- x ^ 2
return(y)
}
pow_two(4)
y
x
```

`y` was defined inside the `pow_two()` function and therefore it is not accessible outside of that function. This is also true for the function's arguments of course - `x` in this case.

Which statement is correct about the following chunk of code? The function `two_dice()` is already available in the workspace.

```r
two_dice <- function() {
possibilities <- 1:6
dice1 <- sample(possibilities, size = 1)
dice2 <- sample(possibilities, size = 1)
dice1 + dice2
}
```

Instructions 50 XP

- *Whatever the way of calling the two_dice() function, R won't have access to dice1 and dice2 outside the function.*

8.10 R passes arguments by value

The title gives it away already: R passes arguments by value. What does this mean? Simply put, it means that an R function cannot change the variable that you input to that function. Let's look at a simple example (try it in the console):

```
triple <- function(x) {
x <- 3*x
x
}
a <- 5
triple(a)
a
```

Inside the `triple()` function, the argument `x` gets overwritten with its value times three. Afterwards this new `x` is returned. If you call this function with a variable `a` set equal to 5, you obtain 15. But did the value of `a` change? If R were to pass `a` to `triple()` by reference, the override of the `x` inside the R passes by value, so the R objects you pass to a function can never change unless you do an explicit assignment. a remains equal to 5, even after calling `triple(a)`.

Can you tell which one of the following statements is false about the following piece of code?

```
increment <- function(x, inc = 1) {
x <- x + inc
x
}
count <- 5
a <- increment(count, 2)
b <- increment(count)
count <- increment(count, 2)
```

Instructions 50 XP

- *In the end, count will equal 10.*

8.11 R you functional?

Now that you've acquired some skills in defining functions with different types of arguments and return values, you should try to create more advanced functions. As you've noticed in the previous exercises, it's perfectly possible to add control-flow constructs, loops and even other functions to your function body.

Remember our social media example? The vectors `linkedin` and `facebook` are already defined in the workspace so you can get your hands dirty straight away. As a first step, you will be writing a function that can interpret a single value of this vector. In the next exercise, you will write another function that can handle an entire vector at once.

Instructions 100 XP

- Finish the function definition for `interpret()`, that interprets the number of profile views on a single day:
- The function takes one argument, num_views.
- If `num_views` is greater than 15, the function prints out "You're popular!" to the console and returns `num_views`.
- Else, the function prints out "Try to be more visible!" and returns 0. Finally, call the interpret() function twice: on the first value of the linkedin vector and on the second element of the facebook vector.

ex_029.R

```r
# The linkedin and facebook vectors have already been created for you

# Define the interpret function
interpret <- function(num_views) {
if (num_views > 15) {
  print("You're popular!")
  return(num_views)
} else {
  print("Try to be more visible!")
  return(0)
}
}
```

```
# Call the interpret function twice
interpret(linkedin[1])
interpret(facebook[1])
```

8.12 R you functional? (2)

A possible implementation of the `interpret()` function has been provided for you. In this exercise you'll be writing another function that will use the `interpret()` function to interpret all the data from your daily profile views inside a vector. Furthermore, your function will return the sum of views on popular days, if asked for. A `for` loop is ideal for iterating over all the vector elements. The ability to return the sum of views on popular days is something you can code through a function argument with a default value.

8.12.1 Instructions `100 XP`

Finish the template for the `interpret_all()` function:

- Make return_sum an optional argument, that is `TRUE` by default.
- Inside the for loop, iterate over all views: on every iteration, add the result of `interpret(v)` to count. Remember that `interpret(v)` returns v for popular days, and 0 otherwise. At the same time, `interpret(v)` will also do some printouts.
- Finish the if construct:
- If return_sum is TRUE, return count.
- Else, return NULL.

Call this newly defined function on both linkedin and facebook.

ex_029.R

```
# The linkedin and facebook vectors have already been created for you
linkedin <- c(16, 9, 13, 5, 2, 17, 14)
facebook <- c(17, 7, 5, 16, 8, 13, 14)

# The interpret() can be used inside interpret_all()
interpret <- function(num_views) {
if (num_views > 15) {
  print("You're popular!")
  return(num_views)
} else {
  print("Try to be more visible!")
  return(0)
}
}

# Define the interpret_all() function
# views: vector with data to interpret
# return_sum: return total number of views on popular days?
interpret_all <- function(views, return_sum = TRUE) {
count <- 0

for (v in views) {
  count <- count + interpret(v)
}

if (return_sum) {
  return (count)
} else {
  return (NULL)
}
}

# Call the interpret_all() function on both linkedin and facebook

interpret_all(linkedin)
interpret_all(facebook)
```

8.13 Load an R Package

There are basically two extremely important functions when it comes down to R packages:

- `install.packages()`, which as you can expect, installs a given package.
- library() which loads packages, i.e. attaches them to the search list on your R workspace.

To install packages, you need administrator privileges. This means that `install.packages()` will thus not work in the DataCamp interface. However, almost all CRAN packages are installed on our servers. You can load them with `library()`.

In this exercise, you'll be learning how to load the `ggplot2` package, a powerful package for data visualization. You'll use it to create a plot of two variables of the `mtcars` data frame. The data has already been prepared for you in the workspace.

Before starting, execute the following commands in the console:

- search(), to look at the currently attached packages and
- qplot(mtcars$wt, mtcars$hp), to build a plot of two variables of the mtcars data frame. An error should occur, because you haven't loaded the `ggplot2` package yet!

Instructions 100 XP

- To fix the error you saw in the console, load the `ggplot2` package. Make sure you are loading (and not installing) the package!
- Now, retry calling the qplot() function with the same arguments.
- Finally, check out the currently attached packages again.

ex_030.R

```r
# Load the ggplot2 package
library(ggplot2)

# Retry the qplot() function
qplot(mtcars$wt, mtcars$hp)

# Check out the currently attached packages again
search()
```

8.14 Different ways to load a package

The `library()` and `require()` functions are not very picky when it comes down to argument types: both `library(rjson)` and `library("rjson")` work perfectly fine for loading a package.

Have a look at some more code chunks that (attempt to) load one or more packages:

```r
# Chunk 1
library(data.table)
require(rjson)

# Chunk 2
library("data.table")
require(rjson)

# Chunk 3
library(data.table)
```

```r
require(rjson, character.only = TRUE)

# Chunk 4
library(c("data.table", "rjson"))
```

Select the option that lists all of the chunks that do not generate an error. The console is yours to experiment in.

Instructions `50 XP`

Possible Answers: (1) and (2)

9 The apply family

Whenever you're using a for loop, you may want to revise your code to see whether you can use the lapply function instead. Learn all about this intuitive way of applying a function over a list or a vector, and how to use its variants, sapply and vapply.

9.1 Use lapply with your own function

As Filip explained in the instructional video, you can use lapply() on your own functions as well. You just need to code a new function and make sure it is available in the workspace. After that, you can use the function inside lapply() just as you did with base R functions.

In the previous exercise you already used lapply() once to convert the information about your favorite pioneering statisticians to a list of vectors composed of two character strings. Let's write some code to select the names and the birth years separately.

The sample code already includes code that defined select_first(), that takes a vector as input and returns the first element of this vector.

Instructions 100 XP

- Apply select_first() over the elements of split_low with lapply() and assign the result to a new variable names.
- Next, write a function select_second() that does the exact same thing for the second element of an inputted vector.
- Finally, apply the select_second() function over split_low and assign the output to the variable years.

ex_0.32.R

```
# Code from previous exercise:
pioneers <-
 c("GAUSS:1777", "BAYES:1702", "PASCAL:1623", "PEARSON:1857")
split <- strsplit(pioneers, split = ":")
split_low <- lapply(split, tolower)
# Write function select_first()
```

```
select_first <- function(x) {
 x[1]
}
# Apply select_first() over split_low: names
names <- lapply(split_low, select_first)
# Write function select_second()
select_second <- function(x) {
 x[2]
}
# Apply select_second() over split_low: years
years <- lapply(split_low, select_second)
```

## lapply and anonymous functions Writing your own functions and then using them inside `lapply()` is quite an accomplishment! But defining functions to use them only once is kind of overkill, isn't it? That's why you can use so-called anonymous functions in R.

Previously, you learned that functions in R are objects in their own right. This means that they aren't automatically bound to a name. When you create a function, you can use the assignment operator to give the function a name. It's perfectly possible, however, to not give the function a name. This is called an anonymous function:

```
# Named function
triple <- function(x) { 3 * x }

# Anonymous function with same implementation
function(x) { 3 * x }

# Use anonymous function inside lapply()
lapply(list(1,2,3), function(x) { 3 * x })
```

`split_low` is defined for you.

Instructions 100 XP

- Transform the first call of `lapply()` such that it uses an anonymous function that does the same thing.
- In a similar fashion, convert the second call of `lapply` to use an anonymous version of the `select_second()` function.
- Remove both the definitions of `select_first()` and `select_second()`, as they are no longer useful.

ex__033.R

```r
# split_low has been created for you
split_low
# Transform: use anonymous function inside lapply
names <- lapply(split_low, function(x){ x[1] })
# Transform: use anonymous function inside lapply
years <- lapply(split_low, function(x){ x[2] })
```

## Use lapply with additional arguments

In the video, the triple() function was transformed to the multiply() function to allow for a more generic approach. lapply() provides a way to handle functions that require more than one argument, such as the multiply() function:

```r
multiply <- function(x, factor) {
  x * factor
}
lapply(list(1,2,3), multiply, factor = 3)
```

On the right we've included a generic version of the select functions that you've coded earlier: select_el(). It takes a vector as its first argument, and an index as its second argument. It returns the vector's element at the specified index.

Instructions 100 XP

Use lapply() twice to call select_el() over all elements in split_low: once with the index equal to 1 and a second time with the index equal to 2. Assign the result to names and years, respectively.

ex_034.R

```r
# Definition of split_low
pioneers <-
c("GAUSS:1777", "BAYES:1702", "PASCAL:1623", "PEARSON:1857")
split <- strsplit(pioneers, split = ":")
split_low <- lapply(split, tolower)

# Generic select function
select_el <- function(x, index) {
  x[index]
}

# Use lapply() twice on split_low: names and years
```

```r
names <- lapply(split_low, select_el, 1)
years <- lapply(split_low, select_el, 2)
```

9.2 Apply functions that return NULL

In all of the previous exercises, it was assumed that the functions that were applied over vectors and lists actually returned a meaningful result. For example, the `tolower()` function simply returns the strings with the characters in lowercase. This won't always be the case. Suppose you want to display the structure of every element of a list. You could use the `str()` function for this, which returns `NULL`:

```r
lapply(list(1, "a", TRUE), str)
```

This call actually returns a list, the same size as the input list, containing all `NULL` values. On the other hand calling

```r
  str(TRUE)
```

on its own prints only the structure of the logical to the console, not `NULL`. That's because `str()` uses `invisible()` behind the scenes, which returns an invisible copy of the return value, `NULL` in this case. This prevents it from being printed when the result of `str()` is not assigned.

What will the following code chunk return (`split_low` is already available in the workspace)? Try to reason about the result before simply executing it in the console!

```r
lapply(split_low, function(x) {
  if (nchar(x[1]) > 5) {
    return(NULL)
  } else {
    return(x[2])
  }
})
```

9.3 How to use sapply

You can use `sapply()` similar to how you used `lapply()`. The first argument of `sapply()` is the list or vector `X` over which you want to apply a function, FUN. Potential additional arguments to this function are specified afterwards `(...)`:

```r
sapply(X, FUN, ...)
```

In the next couple of exercises, you'll be working with the variable `temp`, that contains temperature measurements for 7 days. `temp` is a list of length 7, where each element is a vector of length 5, representing 5 measurements on a given day. This variable has already been defined in the workspace: type str(temp) to see its structure.

9.3.1 Instructions `100 XP`

- Use `lapply()` to calculate the minimum (built-in function `min()`) of the temperature measurements for every day.
- Do the same thing but this time with `sapply()`. See how the output differs.
- Use `lapply()` to compute the the maximum (`max()`) temperature for each day. Again, use `sapply()` to solve the same question and see how `lapply()` and `sapply()` differ.

ex_035.R

```r
# temp has already been defined in the workspace

# Use lapply() to find each day's minimum temperature

lapply(temp, min)

# Use sapply() to find each day's minimum temperature
sapply(temp, min)

# Use lapply() to find each day's maximum temperature
lapply(temp, max)

# Use sapply() to find each day's maximum temperature
sapply(temp, max)
```

9.4 sapply with your own function

Like `lapply()`, `sapply()` allows you to use self-defined functions and apply them over a vector or a list:

```r
sapply(X, FUN, ...)
```

Here, FUN can be one of R's built-in functions, but it can also be a function you wrote. This self-written function can be defined before hand, or can be inserted directly as an anonymous function.

Instructions 100 XP

- Finish the definition of extremes_avg(): it takes a vector of temperatures and calculates the average of the minimum and maximum temperatures of the vector.
- Next, use this function inside sapply() to apply it over the vectors inside temp.
- Use the same function over temp with lapply() and see how the outputs differ.

ex_036.R

```
# temp is already defined in the workspace

# Finish function definition of extremes_avg
extremes_avg <- function(x) {
  ( min(x) + max(x) ) / 2
}

# Apply extremes_avg() over temp using sapply()
sapply(temp, extremes_avg)

# Apply extremes_avg() over temp using lapply()
lapply(temp, extremes_avg)
```

9.5 sapply with function returning vector

In the previous exercises, you've seen how sapply() simplifies the list that lapply() would return by turning it into a vector. But what if the function you're applying over a list or a vector returns a vector of length greater than 1? If you don't remember from the video, don't waste more time in the valley of ignorance and head over to the instructions!

Instructions 100 XP

- Finish the definition of the extremes() function. It takes a vector of numerical values and returns a vector containing the minimum and maximum values of a given vector, with the names "min" and "max", respectively.
- Apply this function over the vector temp using sapply().
- Finally, apply this function over the vector temp using lapply() as well.

ex_037.R

```r
# temp is already available in the workspace

# Create a function that returns min and max of a vector: extremes
extremes <- function(x) {
  c(min = min(x), max = max(x))
}

# Apply extremes() over temp with sapply()
sapply(temp, extremes)

# Apply extremes() over temp with lapply()
lapply(temp, extremes)
```

9.6 sapply can't simplify, now what?

It seems like we've hit the jackpot with `sapply()`. On all of the examples so far, `sapply()` was able to nicely simplify the rather bulky output of `lapply()`. But, as with life, there are things you can't simplify. How does `sapply()` react?

We already created a function, below_zero(), that takes a vector of numerical values and returns a vector that only contains the values that are strictly below zero.

Instructions `100 XP`

- Apply `below_zero()` over `temp` using `sapply()` and store the result in `freezing_s`.
- Apply `below_zero()` over temp using `lapply()`. Save the resulting list in a variable `freezing_l`.
- Compare `freezing_s` to `freezing_l` using the `identical()` function.

ex_038.R

```r
# temp is already prepared for you in the workspace

# Definition of below_zero()
below_zero <- function(x) {
  return(x[x < 0])
}
```

```
# Apply below_zero over temp using sapply(): freezing_s
freezing_s <- sapply(temp, below_zero)

# Apply below_zero over temp using lapply(): freezing_l
freezing_l <- lapply(temp, below_zero)

# Are freezing_s and freezing_l identical?
identical(freezing_s, freezing_s)
```

9.7 sapply with functions that return NULL

You already have some apply tricks under your sleeve, but you're surely hungry for some more, aren't you? In this exercise, you'll see how `sapply()` reacts when it is used to apply a function that returns `NULL` over a vector or a list.

A function `print_info()`, that takes a vector and prints the average of this vector, has already been created for you. It uses the `cat()` function.

Instructions `100 XP`

- Apply `print_info()` over the contents of temp with `sapply()`.
- Repeat this process with `lapply()`. Do you notice the difference?

ex__039.R

```
# temp is already available in the workspace

# Definition of print_info()
print_info <- function(x) {
  cat("The average temperature is", mean(x), "\n")
}

# Apply print_info() over temp using sapply()
sapply(temp, print_info)

# Apply print_info() over temp using lapply()
lapply(temp, print_info)
```

9.8 Reverse engineering sapply

```
sapply(list(runif (10), runif (10)),
       function(x) c(min = min(x), mean = mean(x), max = max(x)))
```

Without going straight to the console to run the code, try to reason through which of the following statements are correct and why.

(1) `sapply()` can't simplify the result that `lapply()` would return, and thus returns a list of vectors.
(2) This code generates a matrix with 3 rows and 2 columns.
(3) The function that is used inside sapply() is anonymous.
(4) The resulting data structure does not contain any names.

Select the option that lists all correct statements.

answer : (2) and (3)

9.9 Use vapply

Before you get your hands dirty with the third and last apply function that you'll learn about in this intermediate R course, let's take a look at its syntax. The function is called `vapply()`, and it has the following syntax:

```
vapply(X, FUN, FUN.VALUE, ..., USE.NAMES = TRUE)
```

Over the elements inside `X`, the function `FUN` is applied. The `FUN.VALUE` argument expects a template for the return argument of this function `FUN`. `USE.NAMES` is `TRUE` by default; in this case `vapply()` tries to generate a named array, if possible.

For the next set of exercises, you'll be working on the temp list again, that contains 7 numerical vectors of length 5. We also coded a function `basics()` that takes a vector, and returns a named vector of length 3, containing the `minimum`, `mean` and `maximum` value of the vector respectively.

Instructions 100 XP

Apply the function `basics()` over the list of temperatures, `temp`, using `vapply()`. This time, you can use `numeric(3)` to specify the `FUN.VALUE` argument.

ex_040.R

```
# temp is already available in the workspace

# Definition of basics()
basics <- function(x) {
  c(min = min(x), mean = mean(x), max = max(x))
}

# Apply basics() over temp using vapply()
vapply(temp, basics, numeric(3))
```

9.10 Use vapply (2)

So far you've seen that `vapply()` mimics the behavior of `sapply()` if everything goes according to plan. But what if it doesn't?

In the video, Filip showed you that there are cases where the structure of the output of the function you want to apply, `FUN`, does not correspond to the template you specify in `FUN.VALUE`. In that case, `vapply()` will throw an error that informs you about the misalignment between expected and actual output.

Instructions `100 XP`

- Inspect the pre-loaded code and try to run it. If you haven't changed anything, an error should pop up. That's because `vapply()` still expects `basics()` to return a vector of length 3. The error message gives you an indication of what's wrong.
- Try to fix the error by editing the `vapply()` command.

ex_041.R

```
# temp is already available in the workspace

# Definition of the basics() function
basics <- function(x) {
  c(min = min(x), mean = mean(x), median = median(x), max = max(x))
}

# Fix the error:
vapply(temp, basics, numeric(4))
```

9.11 From sapply to vapply

As highlighted before, `vapply()` can be considered a more robust version of `sapply()`, because you explicitly restrict the output of the function you want to `apply`. Converting your `sapply()` expressions in your own R scripts to `vapply()` expressions is therefore a good practice (and also a breeze!).

Instructions `100 XP`

Convert all the `sapply()` expressions on the right to their `vapply()` counterparts. Their results should be exactly the same; you're only adding robustness. You'll need the templates `numeric(1)` and `logical(1)`.

ex_042.R

```
# temp is already defined in the workspace

# Convert to vapply() expression
vapply(temp, max, numeric(1))

# Convert to vapply() expression
vapply(temp, function(x, y) { mean(x) > y }, y = 5, logical(1))
```

10 Utilities

Mastering R programming is not only about understanding its programming concepts. Having a solid understanding of a wide range of R functions is also important. This chapter introduces you to many useful functions for data structure manipulation, regular expressions, and working with times and dates.

10.1 Mathematical utilities

Have another look at some useful math functions that R features:

- `abs()`: Calculate the absolute value.
- `sum()`: Calculate the sum of all the values in a data structure.
- `mean()`: Calculate the arithmetic mean.
- `round()`: Round the values to 0 decimal places by default. Try out ?round in the console for variations of `round()` and ways to change the number of digits to round to.

As a data scientist in training, you've estimated a regression model on the sales data for the past six months. After evaluating your model, you see that the training error of your model is quite regular, showing both positive and negative values. A vector `errors` containing the error values has been pre-defined for you.

Instructions `100 XP`

Calculate the sum of the absolute rounded values of the training errors. You can work in parts, or with a single one-liner. There's no need to store the result in a variable, just have R print it.

ex_043.R

```r
# The errors vector has already been defined for you
errors <- c(1.9, -2.6, 4.0, -9.5, -3.4, 7.3)

# Sum of absolute rounded values of errors
sum(abs(round(errors)))
```

10.2 Find the error

We went ahead and pre-loaded some code for you, but there's still an error. Can you trace it and fix it?

In times of despair, help with functions such as sum() and rev() are a single command away; simply execute the code ?sum and ?rev.

Instructions `100 XP`

Fix the error by including code on the last line. Remember: you want to call `mean()` only once!

ex_044.R

```r
# Don't edit these two lines
vec1 <- c(1.5, 2.5, 8.4, 3.7, 6.3)
vec2 <- rev(vec1)

# Fix the error
mean(abs(vec1))
```

10.3 Data Utilities

R features a bunch of functions to juggle around with data structures::

- seq(): Generate sequences, by specifying the from, to, and by arguments.
- rep(): Replicate elements of vectors and lists.
- sort(): Sort a vector in ascending order. Works on numerics, but also on character strings and logicals.
- rev(): Reverse the elements in a data structures for which reversal is defined.
- str(): Display the structure of any R object.
- append(): Merge vectors or lists.
- is.*(): Check for the class of an R object.
- as.*(): Convert an R object from one class to another.
- unlist(): Flatten (possibly embedded) lists to produce a vector.

Remember the social media profile views data? Your LinkedIn and Facebook view counts for the last seven days have been pre-defined as lists.

Instructions 100 XP

- Convert both linkedin and facebook lists to a vector, and store them as `li_vec` and fb_vec respectively.
- Next, append fb_vec to the li_vec (Facebook data comes last). Save the result as social_vec.
- Finally, sort social_vec from high to low. Print the resulting vector.

ex_045.R

```r
# The linkedin and facebook lists have already been created for you
linkedin <- list(16, 9, 13, 5, 2, 17, 14)
facebook <- list(17, 7, 5, 16, 8, 13, 14)

# Convert linkedin and facebook to a vector: li_vec and fb_vec
li_vec <- unlist(linkedin)
fb_vec <- unlist(facebook)

# Append fb_vec to li_vec: social_vec
social_vec <- append(li_vec, fb_vec)

# Sort social_vec
print(sort(social_vec, decreasing=TRUE))
```

10.4 Find the error (2)

Just as before, let's switch roles. It's up to you to see what unforgivable mistakes we've made. Go fix them!

Instructions 100 XP

Correct the expression. Make sure that your fix still uses the functions `rep()` and `seq()`.

ex_046.R

```r
# Fix me
rep(seq(1, 7, by = 2), times = 7)
```

10.5 Beat Gauss using R

There is a popular story about young Gauss. As a pupil, he had a lazy teacher who wanted to keep the classroom busy by having them add up the numbers 1 to 100. Gauss came up with an answer almost instantaneously, 5050. On the spot, he had developed a formula for calculating the sum of an arithmetic series. There are more general formulas for calculating the sum of an arithmetic series with different starting values and increments. Instead of deriving such a formula, why not use R to calculate the sum of a sequence?

Instructions `100 XP`

- Using the function `seq()`, create a sequence that ranges from 1 to 500 in increments of 3. Assign the resulting vector to a variable `seq1`.
- Again with the function `seq()`, create a sequence that ranges from 1200 to 900 in increments of -7. Assign it to a variable `seq2`.
- Calculate the total sum of the sequences, either by using the `sum()` function twice and adding the two results, or by first concatenating the sequences and then using the `sum()` function once. Print the result to the console.

ex_047.R

```r
# Create first sequence: seq1
seq1 <- seq(1, 500, by = 3)

# Create second sequence: seq2
seq2 <- seq(1200, 900, by = -7)

# Calculate total sum of the sequences
print(sum(seq1) + sum(seq2))
```

10.6 grepl & grep

In their most basic form, regular expressions can be used to see whether a pattern exists inside a character string or a vector of character strings. For this purpose, you can use:

`grepl()`, which returns `TRUE` when a pattern is found in the corresponding character string. `grep()`, which returns a vector of indices of the character strings that contains the pattern. Both functions need a pattern and an `x` argument, where pattern is the regular expression you want to match for, and the `x` argument is the character vector from which matches should be sought.

In this and the following exercises, you'll be querying and manipulating a character vector of email addresses! The vector emails has been pre-defined so you can begin with the instructions straight away!

Instructions `100 XP`

- Use `grepl()` to generate a vector of logicals that indicates whether these email addresses contain `"edu"`. Print the result to the output.
- Do the same thing with `grep()`, but this time save the resulting indexes in a variable hits.
- Use the variable hits to select from the emails vector only the emails that contain `"edu"`.

ex_48.R

```
# The emails vector has already been defined for you
emails <-
    c(
        "john.doe@ivyleague.edu",
        "education@world.gov",
        "dalai.lama@peace.org",
        "invalid.edu",
        "quant@bigdatacollege.edu",
        "cookie.monster@sesame.tv"
    )

# Use grepl() to match for "edu"
print(grepl('edu', emails))

# Use grep() to match for "edu", save result to hits
hits <- grep('edu', emails)

# Subset emails using hits
emails[hits]
```

10.7 grepl & grep (2)

You can use the caret, ^, and the dollar sign, $ to match the content located in the start and end of a string, respectively. This could take us one step closer to a correct pattern for matching only the ".edu" email addresses from our list of emails. But there's more that can be added to make the pattern more robust:

- @, because a valid email must contain an at-sign.
- .* , which matches any character (.) zero or more times (*). Both the dot and the asterisk are metacharacters. You can use them to match any character between the at-sign and the ".edu" portion of an email address.
- `\\.edu$`, to match the ".edu" part of the email at the end of the string. The `\\` part escapes the dot: it tells R that you want to use the `.` as an actual character.

Instructions `100 XP`

- Use `grepl()` with the more advanced regular expression to return a logical vector. Simply print the result.
- Do a similar thing with `grep()` to create a vector of indices. Store the result in the variable `hits`.
- Use `emails[hits]` again to subset the `emails` vector.

ex_049.R

```r
# The emails vector has already been defined for you
emails <- c(
    "john.doe@ivyleague.edu",
    "education@world.gov",
    "dalai.lama@peace.org",
    "invalid.edu",
    "quant@bigdatacollege.edu",
    "cookie.monster@sesame.tv"
)
grepl("@.*\\.edu$", emails)
hits <- grep("@.*\\.edu$", emails)
emails[hits]
```

10.8 sub & gsub

While grep() and grepl() were used to simply check whether a regular expression could be matched with a character vector, `sub()` and `gsub()` take it one step further: you can specify a replacement argument. If inside the character vector `x`, the regular expression pattern is found, the matching element(s) will be replaced with replacement. `sub()` only replaces the first match, whereas `gsub()` replaces all matches.

Suppose that emails vector you've been working with is an excerpt of DataCamp's email database. Why not offer the owners of the `.edu` email addresses a new email address on the `datacamp.edu` domain? This could be quite a powerful marketing stunt: Online education is taking over traditional learning institutions! Convert your email and be a part of the new generation!

Instructions 100 XP

With the advanced regular expression `"@.*\\.edu$"`, use `sub()` to replace the match with `"@datacamp.edu"`. Since there will only be one match per character string, `gsub()` is not necessary here. Inspect the resulting output.

ex_050.R

```
# The emails vector has already been defined for you
emails <- c(
    "john.doe@ivyleague.edu",
    "education@world.gov",
    "global@peace.org",
    "invalid.edu",
    "quant@bigdatacollege.edu",
    "cookie.monster@sesame.tv"
)

# Use sub() to convert the email domains to datacamp.edu
sub("@.*\\.edu$", "@datacamp.edu", emails)
```

10.9 sub & gsub (2)

Regular expressions are a typical concept that you'll learn by doing and by seeing other examples. Before you rack your brains over the regular expression in this exercise, have a look at the new things that will be used:

- .*: A usual suspect! It can be read as "any character that is matched zero or more times".
- \s: Match a space. The "s" is normally a character, escaping it (\) makes it a metacharacter.
- [0-9]+: Match the numbers 0 to 9, at least once (+).
- ([0-9]+): The parentheses are used to make parts of the matching string available to define the replacement. The \1 in the replacement argument of `sub()` gets set to the string that is captured by the regular expression [0-9]+. `{. r code-line-numbers="false"} awards <- c(     "Won 1 Oscar.",     "Won 1 Oscar. Another 9 wins & 24 nominations.",     "1 win and 2 nominations.",     "2 wins & 3 nominations.",     "Nominated for 2 Golden Globes. 1 more win & 2 nominations.",     "4`

`wins & 1 nomination." ) sub(".*\\s([0-9]+)\\snomination.*$", "\\1", awards)` What does this code chunk return? awards is already defined in the workspace so you can start playing in the console straight away.

Instructions 50 XP

Possible Answers

- A vector of integers containing: `1, 24, 2, 3, 2, 1`.
- The vector awards gets returned as there isn't a single element in awards that matches the regular expression.
- A vector of character strings containing `"1", "24", "2", "3", "2", "1"`.
- A vector of character strings containing `"Won 1 Oscar.", "24", "2", "3", "2", "1"`.

10.10 Right here, right now

In R, dates are represented by Date objects, while times are represented by `POSIXct` objects. Under the hood, however, these dates and times are simple numerical values. Date objects store the number of days since the 1st of January in 1970. `POSIXct` objects on the other hand, store the number of seconds since the 1st of January in 1970.

The 1st of January in 1970 is the common origin for representing times and dates in a wide range of programming languages. There is no particular reason for this; it is a simple convention. Of course, it's also possible to create dates and times before 1970; the corresponding numerical values are simply negative in this case.

Instructions 100 XP

- Ask R for the current date, and store the result in a variable today.
- To see what today looks like under the hood, call `unclass()` on it.
- Ask R for the current time, and store the result in a variable, now.
- To see the numerical value that corresponds to now, call `unclass()` on it.

ex_50.R

```r
# Get the current date: today
today <- Sys.date()
# See what today looks like under the hood
unclass(today)
# Get the current time: now
```

```
now <-  Sys.time()
# See what now looks like under the hood
unclass(now)
```

10.11 Create and format dates

To create a Date object from a simple character string in R, you can use the `as.Date()` function. The character string has to obey a format that can be defined using a set of symbols (the examples correspond to 13 January, 1982):

- %Y: 4-digit year (1982)
- %y: 2-digit year (82)
- %m: 2-digit month (01)
- %d: 2-digit day of the month (13)
- %A: weekday (Wednesday)
- %a: abbreviated weekday (Wed)
- %B: month (January)
- %b: abbreviated month (Jan)

The following R commands will all create the same Date object for the 13th day in January of 1982:

```
as.Date("1982-01-13")
as.Date("Jan-13-82", format = "%b-%d-%y")
as.Date("13 January, 1982", format = "%d %B, %Y")
```

Notice that the first line here did not need a format argument, because by default R matches your character string to the formats `"%Y-%m-%d"` or `"%Y/%m/%d"`.

In addition to creating dates, you can also convert dates to character strings that use a different date notation. For this, you use the `format()` function. Try the following lines of code:

today <- Sys.Date() format(Sys.Date(), format = "%d %B, %Y") format(Sys.Date(), format = "Today is a %A!")

Instructions 100 XP

- Three character strings representing dates have been created for you. Convert them to dates using as.Date(), and assign them to date1, date2, and date3 respectively. The code for date1 is already included.

- Extract useful information from the dates as character strings using `format()`. From the first date, select the weekday. From the second date, select the day of the month. From the third date, you should select the abbreviated month and the 4-digit year, separated by a space.

ex_51.R

```
# Definition of character strings representing dates
str1 <- "May 23, '96"
str2 <- "2012-03-15"
str3 <- "30/January/2006"

# Convert the strings to dates: date1, date2, date3
date1 <- as.Date(str1, format = "%b %d, '%y")
date2 <- as.Date(str2, format = "%Y-%m-%d")
date3 <- as.Date(str3, format = "%d/%B/%Y")

# Convert dates to formatted strings
format(date1, "%A")
format(date2, "%d")
format(date3, "%b %Y")
```

10.12 Create and format times

Similar to working with dates, you can use `as.POSIXct()` to convert from a character string to a `POSIXct` object, and `format()` to convert from a `POSIXct` object to a character string. Again, you have a wide variety of symbols:

- `%H`: hours as a decimal number (00-23)
- `%I`: hours as a decimal number (01-12)
- `%M`: minutes as a decimal number
- `%S`: seconds as a decimal number
- `%T`: shorthand notation for the typical format %H:%M:%S
- `%p`: AM/PM indicator

For a full list of conversion symbols, consult the strptime documentation in the console:

```
?strptime
```

Again, `as.POSIXct()` uses a default format to match character strings. In this case, it's `%Y-%m-%d %H:%M:%S`. In this exercise, abstraction is made of different time zones.

10.12.1 Instructions 100 XP

- Convert two strings that represent timestamps, str1 and str2, to POSIXct objects called time1 and time2.
- Using format(), create a string from time1 containing only the minutes.
- From time2, extract the hours and minutes as "hours:minutes AM/PM". Refer to the assignment text above to find the correct conversion symbols!

ex_52.R

```
    # Definition of character strings representing times
str1 <- "May 23, '96 hours:23 minutes:01 seconds:45"
str2 <- "2012-3-12 14:23:08"
# Convert the strings to POSIXct objects: time1, time2
time1 <- as.POSIXct(str1, format = "%B %d, '%y hours:%H minutes:%M seconds:%S")

time2 <-
    as.POSIXct(
        str2, format = "%Y-%m-%d %H:%M:%S")

# Convert times to formatted strings
format(time1, '%M')
format(time2, '%H:%M%p')
```

10.13 Calculations with Dates

Both Date and POSIXct R objects are represented by simple numerical values under the hood. This makes calculation with time and date objects very straightforward: R performs the calculations using the underlying numerical values, and then converts the result back to human-readable time information again.

You can increment and decrement Date objects, or do actual calculations with them:

```
today <- Sys.Date()
today + 1
today - 1
as.Date("2015-03-12") - as.Date("2015-02-27")
```

To control your eating habits, you decided to write down the dates of the last five days that you ate pizza. In the workspace, these dates are defined as five Date objects, day1 to day5. A vector pizza containing these 5 Date objects has been pre-defined for you.

Instructions `100 XP`

- Calculate the number of days that passed between the last and the first day you ate pizza. Print the result.
- Use the function `diff()` on pizza to calculate the differences between consecutive pizza days. Store the result in a new variable `day_diff`.
- Calculate the average period between two consecutive pizza days. Print the result.

ex_053.R

```r
# day1, day2, day3, day4 and day5 are already available in the
# workspace Difference between last and first pizza day
print(day5 - day1)
# Create vector pizza
pizza <- c(day1, day2, day3, day4, day5)
# Create differences between consecutive pizza days: day_diff
day_diff <- diff(pizza)
# Average period between two consecutive pizza days
print(mean(day_diff))
```

10.14 Calculations with Times

Calculations using POSIXct objects are completely analogous to those using Date objects. Try to experiment with this code to increase or decrease POSIXct objects:

```r
now <- Sys.time()
now + 3600          # add an hour
now - 3600 * 24     # subtract a day
```

Adding or subtracting time objects is also straightforward:

```r
birth <- as.POSIXct("1879-03-14 14:37:23")
death <- as.POSIXct("1955-04-18 03:47:12")
einstein <- death - birth
einstein
```

You're developing a website that requires users to log in and out. You want to know what is the total and average amount of time a particular user spends on your website. This user has logged in 5 times and logged out 5 times as well. These times are gathered in the vectors login and logout, which are already defined in the workspace.

Instructions 100 XP

- Calculate the difference between the two vectors logout and login, i.e. the time the user was online in each independent session. Store the result in a variable time_online.
- Inspect the variable time_online by printing it.
- Calculate the total time that the user was online. Print the result.
- Calculate the average time the user was online. Print the result.

ex_054.R

```r
# login and logout are already defined in the workspace
# Calculate the difference between login and logout: time_online

time_on_line <- logout - login

# Inspect the variable time_online
time_on_line

# Calculate the total time online
sum(time_on_line)

# Calculate the average time online
mean(time_on_line)
```

10.15 Time is of the essence

The dates when a season begins and ends can vary depending on who you ask. People in Australia will tell you that spring starts on September 1st. The Irish people in the Northern hemisphere will swear that spring starts on February 1st, with the celebration of St. Brigid's Day. Then there's also the difference between astronomical and meteorological seasons: while astronomers are used to equinoxes and solstices, meteorologists divide the year into 4 fixed seasons that are each three months long. (source: www.timeanddate.com)

A vector `astro`, which contains character strings representing the dates on which the 4 astronomical seasons start, has been defined on your workspace. Similarly, a vector `meteo` has already been created for you, with the meteorological beginnings of a season.

Instructions 100 XP

- Use `as.Date()` to convert the astro vector to a vector containing `Date` objects. You will need the `%d`, `%b` and `%Y` symbols to specify the format. Store the resulting vector as

astro_dates.
- Use `as.Date()` to convert the meteo vector to a vector with Date objects. This time, you will need the `%B`, `%d` and `%y` symbols for the format argument. Store the resulting vector as`meteo_dates`.
- With a combination of `max()`, `abs()` and `-`, calculate the maximum absolute difference between the astronomical and the meteorological beginnings of a season, i.e. `astro_dates` and `meteo_dates`. Simply print this maximum difference to the console output.

ex_055.R

```r
# Convert astro to vector of Date objects: astro_dates

astro_dates <- as.Date(astro, "%d-%b-%Y")
# Convert meteo to vector of Date objects: meteo_dates
meteo_dates <- as.Date(meteo, format = "%B %d, %y")

# Calculate the maximum absolute difference between astro_dates and meteo_dates
max(abs(astro_dates - meteo_dates))
```

11 Data visualization with ggplot2 and friends

Part III

Introduction to Writing Functions in R

Being able to write your own functions makes your analyses more readable, with fewer errors, and more reusable from project to project. Function writing will increase your productivity more than any other skill! In this course you'll learn the basics of function writing, focusing on the arguments going into the function and the return values. You'll be writing useful data science functions, and using real-world data on Wyoming tourism, stock price/earnings ratios, and grain yields.

12

13 Data visualization with ggplot2 and friends

Part IV

importing and cleaning data

14 Importing data from flat files with utils

A lot of data comes in the form of flat files: simple tabular text files. Learn how to import the common formats of flat file data with base R functions.

14.1 read.csv

The `utils` package, which is automatically loaded in your R session on startup, can import `CSV` files with the `read.csv()` function.

In this exercise, you'll be working with `swimming_pools.csv` (view); it contains data on swimming pools in Brisbane, Australia (Source: data.gov.au). The file contains the column names in the first row. It uses a comma to separate values within rows.

Type `dir()` in the console to list the files in your working directory. You'll see that it contains swimming_pools.csv, so you can start straight away.

14.1.1 Instructions `100 XP`

- Use read.csv() to import "swimming_pools.csv" as a data frame with the name pools.
- Print the structure of pools using `str()`.

ex_001.R

```r
# Import swimming_pools.csv: pools
pools <- read.csv("swimming_pools.csv")
# Print the structure of pools
str(pools)
```

References

- http://s3.amazonaws.com/assets.datacamp.com/production/course_1477/datasets/swimming_pools.csv
- https://data.gov.au/dataset/swimming-pools-brisbane-city-council

14.2 stringsAsFactors

With `stringsAsFactors`, you can tell R whether it should convert strings in the flat file to factors.

For all importing functions in the utils package, this argument is TRUE, which means that you import strings as factors. This only makes sense if the strings you import represent categorical variables in R. If you set `stringsAsFactors` to `FALSE`, the data frame columns corresponding to strings in your text file will be character.

You'll again be working with the `swimming_pools.csv` (view in data folder) file. It contains two columns (`Name` and `Address`), which shouldn't be factors.

Instructions `100 XP`

- Use `read.csv()` to import the data in `"swimming_pools.csv"` as a data frame called pools; make sure that strings are imported as characters, not as factors.
- Using `str()`, display the structure of the dataset and check that you indeed get character vectors instead of factors.

ex_002.R

```
# Import swimming_pools.csv correctly: pools
pools <- read.csv("swimming_pools.csv", stringsAsFactors = FALSE)

# Check the structure of pools
str(pools)
```

14.3 Any changes?

Consider the code below that loads data from `swimming_pools.csv` in two distinct ways:

```
# Option A
pools <- read.csv("swimming_pools.csv", stringsAsFactors = TRUE)

# Option B
pools <- read.csv("swimming_pools.csv", stringsAsFactors = FALSE)
```

ex_003.R

```
library(projmgr)

# the following could be run in RMarkdown
todo_path <- system.file(
    "extdata",
    "todo-ex.yml",
     package = "projmgr",
     mustWork = TRUE
)

my_todo <- read_todo(todo_path)
report_todo(my_todo)
```

14.4 read.delim

Aside from `.csv` files, there are also the `.txt` files which are basically text files. You can import these functions with `read.delim()`. By default, it sets the sep argument to `"\t"` (fields in a record are delimited by tabs) and the header argument to `TRUE` (the first row contains the field names).

In this exercise, you will import hotdogs.txt (view), containing information on sodium and calorie levels in different hotdogs (Source: UCLA). The dataset has 3 variables, but the variable names are not available in the first line of the file. The file uses tabs as field separators.

Instructions `100 XP`

- Import the data in `"hotdogs.txt"` with `read.delim()`. Call the resulting data frame `hotdogs`. The variable names are not on the first line, so make sure to set the header argument appropriately.
- Call `summary()` on `hotdogs`. This will print out some summary statistics about all variables in the data frame.

ex_004.R

```
# Import hotdogs.txt: hotdogs
hotdogs <- read.delim(
    "hotdogs.txt",
    sep = '\t',
    header = FALSE
)
```

```r
# Summarize hotdogs
summary(hotdogs)
```

14.5 read.table

If you're dealing with more exotic flat file formats, you'll want to use read.table(). It's the most basic importing function; you can specify tons of different arguments in this function. Unlike read.csv() and read.delim(), the header argument defaults to FALSE and the sep argument is " " by default.

Up to you again! The data is still hotdogs.txt (view). It has no column names in the first row, and the field separators are tabs. This time, though, the file is in the data folder inside your current working directory. A variable path with the location of this file is already coded for you.

Instructions 100 XP

- Finish the read.table() call that's been prepared for you. Use the path variable, and make sure to set sep correctly.
- Call head() on hotdogs; this will print the first 6 observations in the data frame.

ex_005.R

```r
# Path to the hotdogs.txt file: path
path <- file.path("data", "hotdogs.txt")

# Import the hotdogs.txt file: hotdogs
hotdogs <-
    read.table(
        path,
        sep = '\t',
        col.names = c("type", "calories", "sodium")
    )

# Call head() on hotdogs
head(hotdogs)
```

14.6 Arguments

Lily and Tom are having an argument because they want to share a hot dog but they can't seem to agree on which one to choose. After some time, they simply decide that they will have one each. Lily wants to have the one with the fewest calories while Tom wants to have the one with the most sodium.

Next to calories and sodium, the hotdogs have one more variable: type. This can be one of three things: Beef, Meat, or Poultry, so a categorical variable: a factor is fine.

Instructions `100 XP`

- Finish the read.delim() call to import the data in "hotdogs.txt". It's a tab- delimited file without names in the first row.
- The code that selects the observation with the lowest calorie count and stores it in the variable lily is already available. It uses the function which.min(), that returns the index the smallest value in a vector.
- Do a similar thing for Tom: select the observation with the most sodium and store it in tom. Use which.max() this time.
- Finally, print both the observations lily and tom.

ex_006.R

```r
# Finish the read.delim() call
hotdogs <-
    read.delim(
        "hotdogs.txt",
        header = FALSE,
        col.names = c("type", "calories", "sodium")
    )

# Select the hot dog with the least calories: lily
lily <- hotdogs[which.min(hotdogs$calories), ]

# Select the observation with the most sodium: tom

tom <- hotdogs[which.max(hotdogs$sodium), ]
# Print lily and tom
lily
tom
```

14.7 Column classes

Next to column names, you can also specify the column types or column classes of the resulting data frame. You can do this by setting the colClasses argument to a vector of strings representing classes:

```
read.delim("my_file.txt",
           colClasses = c("character",
                          "numeric",
                          "logical"))
```

This approach can be useful if you have some columns that should be factors and others that should be characters. You don't have to bother with `stringsAsFactors` anymore; just state for each column what the class should be.

If a column is set to "NULL" in the `colClasses` vector, this column will be skipped and will not be loaded into the data frame.

Instructions 100 XP

- The `read.delim()` call from before is already included and creates the `hotdogs` data frame. Go ahead and display the structure of `hotdogs`.
- Edit the second `read.delim()` call. Assign the correct vector to the `colClasses` argument. `NA` should be replaced with a character vector: `c("factor", "NULL", "numeric")`.
- Display the structure of `hotdogs2` and look for the difference.

ex_007.R

```
# Previous call to import hotdogs.txt
hotdogs <-
    read.delim(
        "hotdogs.txt",
         header = FALSE,
         col.names = c("type", "calories", "sodium")
    )

# Display structure of hotdogs
str(hotdogs)

# Edit the colClasses argument to import the data correctly: hotdogs2
hotdogs2 <- read.delim(
```

```r
    "hotdogs.txt",
    header = FALSE,
    col.names = c("type", "calories", "sodium"),
    colClasses =  c("factor", "NULL", "numeric")
)


# Display structure of hotdogs2
str(hotdogs2)
```

15 readr & data.table

In addition to base R, there are dedicated packages to easily and efficiently import flat file data. We'll talk about two such packages: `readr` and `data.table`.

15.1 read_csv

CSV files can be imported with `read_csv()`. It's a wrapper function around `read_delim()` that handles all the details for you. For example, it will assume that the first row contains the column names.

The dataset you'll be working with here is `potatoes.csv` (view in dataset folder). It gives information on the impact of storage period and cooking on potatoes' flavor. It uses commas to delimit fields in a record, and contains column names in the first row. The file is available in your workspace.

Instructions `100 XP`

Load the `readr` package with `library()`.

Import `"potatoes.csv"` using `read_csv()`. Assign the resulting data frame to the variable potatoes.

ex_008.R

```r
# Load the readr package
library(readr)

# Import potatoes.csv with read_csv(): potatoes
potatoes <- read_csv("potatoes.csv")
```

15.2 read_tsv

Where you use `read_csv()` to easily read in CSV files, you use `read_tsv()` to easily read in TSV files. TSV is short for tab- separated values.

This time, the potatoes data comes in the form of a tab-separated values file; `potatoes.txt`. In contrast to `potatoes.csv`, this file does not contain columns names in the first row, though.

There's a vector properties that you can use to specify these column names manually.

Instructions 100 XP

- Use `read_tsv()` to import the potatoes data from potatoes.txt and store it in the data frame potatoes. In addition to the path to the file, you'll also have to specify the `col_names` argument; you can use the properties vector for this.
- Call `head()` on potatoes to show the first observations of your dataset.

ex_009.R

```r
# readr is already loaded

# Column names
properties <- c(
    "area",
    "temp",
    "size",
    "storage",
    "method",
    "texture",
    "flavor",
    "moistness"
)

# Import potatoes.txt: potatoes
potatoes <- read_tsv(
    "potatoes.txt",
    col_names = properties
)

# Call head() on potatoes
head(potatoes)
```

15.3 read_delim

Just as `read.table()` was the main utils function, `read_delim()` is the main readr function.

`read_delim()` takes two mandatory arguments:

- `file`: the file that contains the data
- `delim`: the character that separates the values in the data file

You'll again be working with `potatoes.txt` (view); the file uses tabs (`"\t"`) to delimit values and does not contain column names in its first line. It's available in your working directory so you can start right away. As before, the vector `properties` is available to set the `col_names`.

Instructions 100 XP

- Import all the data in "potatoes.txt" using read_delim(); store the resulting data frame in potatoes.
- Print out potatoes.

ex_010.R

```
# readr is already loaded

# Column names
properties <- c(
    "area",
    "temp",
    "size",
    "storage",
    "method",
    "texture",
    "flavor",
    "moistness"
)

# Import potatoes.txt using read_delim(): potatoes
potatoes <- read_delim(
    "potatoes.txt",
    delim = "\t",
    col_names = properties
)
# Print out potatoes
```

```
potatoes
```

15.4 skip and n_max

Through `skip` and `n_max` you can control which part of your flat file you're actually importing into R.

- skip specifies the number of lines you're ignoring in the flat file before actually starting to import data.
- n_max specifies the number of lines you're actually importing. Say for example you have a CSV file with 20 lines, and set `skip = 2` and `n_max = 3`, you're only reading in lines 3, 4 and 5 of the file.

Watch out: Once you skip some lines, you also skip the first line that can contain column names!

`potatoes.txt`, a flat file with tab-delimited records and without column names, is available in your workspace.

Instructions 100 XP

- Finish the first read_tsv() call to import observations 7, 8, 9, 10 and 11 from `potatoes.txt`.

ex_011.R

```
# readr is already loaded

# Column names
properties <- c(
    "area",
    "temp",
    "size",
    "storage",
    "method",
    "texture",
    "flavor",
    "moistness"
)

# Import 5 observations from potatoes.txt: potatoes_fragment
```

```
potatoes_fragment <- read_tsv(
    "potatoes.txt",
    skip = 6,
    n_max = 5,
    col_names = properties
)
```

15.5 col_types

You can also specify which types the columns in your imported data frame should have. You can do this with `col_types`. If set to `NULL`, the default, functions from the readr package will try to find the correct types themselves. You can manually set the types with a `string`, where each character denotes the class of the column: `character`, `double`, `integer` and `logical`. `_` skips the column as a whole.

`potatoes.txt`, a flat file with tab-delimited records and without column names, is again available in your workspace.

Instructions 100 XP

- In the second `read_tsv()` call, edit the `col_types` argument to import all columns as characters (`c`). Store the resulting data frame in `potatoes_char`.
- Print out the structure of `potatoes_char` and verify whether all column types are `chr`, short for `character`.

ex_012.R

```
# Column names
properties <- c(
    "area",
    "temp",
    "size",
    "storage",
    "method",
    "texture",
    "flavor",
    "moistness"
)

# Import all data, but force all columns to be character: potatoes_char
```

```
potatoes_char <- read_tsv(
    "potatoes.txt",
    col_types = "cccccccc",
    col_names = properties
)
# Print out structure of potatoes_char
str(potatoes_char)
```

15.6 col_types with collectors

Another way of setting the types of the imported columns is using collectors. Collector functions can be passed in a `list()` to the `col_types` argument of `read_` functions to tell them how to interpret values in a column.

For a complete list of collector functions, you can take a look at the `collector` documentation. For this exercise you will need two collector functions:

- `col_integer()`: the column should be interpreted as an integer.
- `col_factor(levels, ordered = FALSE)`: the column should be interpreted as a factor with levels.

In this exercise, you will work with 'hotdogs.txt, which is a tab-delimited file without column names in the first row.

Instructions `100 XP`

- `hotdogs` is created for you without setting the column types. Inspect its summary using the `summary()` function.
- Two collector functions are defined for you: `fac` and `int`. Have a look at them, do you understand what they're collecting?
- In the second `read_tsv()` call, edit the `col_types` argument: Pass a `list()` with the elements `fac`, `int` and `int`, so the first column is imported as a factor, and the second and third column as integers.
- Create a `summary()` of `hotdogs_factor`. Compare this to the summary of hotdogs.

ex_013.R

```
# check if readr is already loaded

# Import without col_types
hotdogs <- read_tsv(
    "hotdogs.txt",
    col_names = c("type", "calories", "sodium")
)

# Display the summary of hotdogs
summary(hotdogs)

# The collectors you will need to import the data
fac <- col_factor(levels = c("Beef", "Meat", "Poultry"))
int <- col_integer()

# Edit the col_types argument to import the data correctly: hotdogs_factor
hotdogs_factor <- read_tsv(
    "hotdogs.txt",
    col_names = c("type", "calories", "sodium"),
    col_types = list(fac, int, int)
)

# Display the summary of hotdogs_factor
summary(hotdogs_factor)
```

15.7 fread

You still remember how to use `read.table()`, right? Well, `fread()` is a function that does the same job with very similar arguments. It is extremely easy to use and blazingly fast! Often, simply specifying the path to the file is enough to successfully import your data.

Don't take our word for it, try it yourself! You'll be working with the `potatoes.csv` file, that's available the dataset folder. Fields are delimited by commas, and the first line contains the column names.

Instructions `100 XP`

-Use `library()` to load (NOT install) the `data.table` package. Check If you
need to install this package.

-Import `"potatoes.csv"` with `fread()`. Simply pass it the file path and see if it worked. Store the result in a variable potatoes. Print out potatoes.

ex_014.R

```r
# load the data.table package using library()
library(data.table)

# Import potatoes.csv with fread(): potatoes
potatoes <- fread("potatoes.csv")

# Print out potatoes
potatoes
```

15.8 fread: more advance use

Now that you know the basics about `fread()`, you should know about two arguments of the function: `drop` and `select`, to drop or select variables of interest.

Suppose you have a dataset that contains 5 variables and you want to keep the first and fifth variable, named “a” and “e”. The following options will all do the trick:

```r
fread("path/to/file.txt", drop = 2:4)
fread("path/to/file.txt", select = c(1, 5))
fread("path/to/file.txt", drop = c("b", "c", "d"))
fread("path/to/file.txt", select = c("a", "e"))
```

Let’s stick with potatoes since we’re particularly fond of them here at DataCamp. The data is again available in the file potatoes.csv (view), containing comma- separated records.

Instructions `100 XP`

Using `fread()` and `select` or `drop` as arguments, only import the `texture` and `moistness` columns of the flat file. They correspond to the columns 6 and 8 in `"potatoes.csv"`. Store the result in a variable potatoes. - plot() 2 columns of the potatoes data frame: texture on the x-axis, moistness on the y-axis. Use the dollar sign notation twice. Feel free to name your axes and plot.

ex_014.R

```r
# fread is already loaded
library("data.table")
path <- "introduction_to_importing_data_in_R/potatoes.csv"
# Import columns 6 and 8 of potatoes.csv: potatoes
potatoes <- fread(path, select = c(6, 8))
```

```
# Plot texture (x) and moistness (y) of potatoes
plot(potatoes$texture, potatoes$moistness)
```

16 List the sheets of an Excel file

List the sheets of an Excel file Before you can start importing from Excel, you should find out which sheets are available in the workbook. You can use the `excel_sheets()` function for this.

You will find the Excel file `urbanpop.xlsx` (view) in your working directory (type dir() to see it). This dataset contains urban population metrics for practically all countries in the world throughout time (Source: Gapminder). It contains three sheets for three different time periods. In each sheet, the first row contains the column names.

Instructions `100 XP`

Load the `readxl` package using`library()`. It's already installed on DataCamp's servers. Use excel_sheets() to print out the names of the sheets in `urbanpop.xlsx`.

ex_15.R

```
# Load the readxl package
library(readxl)

# Print the names of all worksheets
excel_sheets('urbanpop.xlsx')
```

16.1 Import an Excel sheet

Now that you know the names of the sheets in the Excel file you want to import, it is time to import those sheets into R. You can do this with the `read_excel()` function. Have a look at this recipe:

```
data <- read_excel("data.xlsx", sheet = "my_sheet")
```

This call simply imports the sheet with the name `"my_sheet"` from the `"data.xlsx"` file. You can also pass a number to the sheet argument; this will cause `read_excel()` to import the sheet with the given sheet number. `sheet = 1` will import the first sheet, sheet = 2 will import the second sheet, and so on.

In this exercise, you'll continue working with the `urbanpop.xlsx` (view) file.

Instructions 100 XP

- The code to import the first and second sheets is already included. Can you add a command to also import the third sheet, and store the resulting data frame in `pop_3`?
- Store the data frames `pop_1`, `pop_2` and `pop_3` in a list that you call `pop_list`.
- Display the structure of `pop_list`.

ex__16.R

```
# The readxl package is already loaded

# Read the sheets, one by one
pop_1 <- read_excel("urbanpop.xlsx", sheet = 1)
pop_2 <- read_excel("urbanpop.xlsx", sheet = 2)
pop_3 <- read_excel("urbanpop.xlsx", sheet = 3)

# Put pop_1, pop_2 and pop_3 in a list: pop_list
pop_list <- list(pop_1, pop_2, pop_3)

# Display the structure of pop_list
str(pop_list)
```

16.2 Reading a workbook00

In the previous exercise you generated a list of three Excel sheets that you imported. However, loading in every sheet manually and then merging them in a list can be quite tedious. Luckily, you can automate this with `lapply()`. If you have no experience with `lapply()`, feel free to take Chapter 4 of the Intermediate R course.

Have a look at the example code below:

```
my_workbook <- lapply(excel_sheets("data.xlsx"),
                      read_excel,
                      path = "data.xlsx")
```

The `read_excel()` function is called multiple times on the `"data.xlsx"` file and each sheet is loaded in one after the other. The result is a list of data frames, each data frame representing one of the sheets in `data.xlsx`.

You're still working with the `urbanpop.xlsx` (view) file.

Instructions `100 XP`

- Use `lapply()` in combination with `excel_sheets()` and `read_excel()` to read all the Excel sheets in `"urbanpop.xlsx"`. Name the resulting list `pop_list`.
- Print the structure of pop_list.

ex_17.R

```
# The readxl package is already loaded

# Read all Excel sheets with lapply(): pop_list
pop_list <- lapply(
    excel_sheets("urbanpop.xlsx"),
    read_excel,
    path = "urbanpop.xlsx"
)

# Display the structure of pop_list
str(pop_list)
```

16.3 The col_names argument

Apart from path and sheet, there are several other arguments you can specify in `read_excel()`. One of these arguments is called `col_names`.

By default it is `TRUE`, denoting whether the first row in the Excel sheets contains the column names. If this is not the case, you can set `col_names` to `FALSE`. In this case, R will choose column names for you. You can also choose to set `col_names` to a character vector with names for each column. It works exactly the same as in the `readr` package.

You'll be working with the `urbanpop_nonames.xlsx` (view) file. It contains the same data as `urbanpop.xlsx` (view) but has no column names in the first row of the excel sheets.

Instructions `100 XP`

- Import the first Excel sheet of `"urbanpop_nonames.xlsx"` and store the result in `pop_a`. Have R set the column names of the resulting data frame itself.
- Import the first Excel sheet of `urbanpop_nonames.xlsx`; this time, use the `cols` vector that has already been prepared for you to specify the column names. Store the resulting data frame in `pop_b`.
- Print out the summary of `pop_a`.
- Print out the summary of `pop_b`. Can you spot the difference with the other summary?

ex_18.R

```
# The readxl package is already loaded

# Import the first Excel sheet of urbanpop_nonames.xlsx (R gives names): pop_a
pop_a <- read_excel("urbanpop_nonames.xlsx", col_names = FALSE)

# Import the first Excel sheet of urbanpop_nonames.xlsx (specify col_names): pop_b
cols <- c("country", paste0("year_", 1960:1966))
pop_b <- read_excel("urbanpop_nonames.xlsx", col_names = cols)

# Print the summary of pop_a
summary(pop_a)

# Print the summary of pop_b
summary(pop_b)
```

16.4 The skip argument

Another argument that can be very useful when reading in Excel files that are less tidy, is skip. With skip, you can tell R to ignore a specified number of rows inside the Excel sheets you're trying to pull data from. Have a look at this example:

```
read_excel("data.xlsx", skip = 15)
```

In this case, the first 15 rows in the first sheet of `"data.xlsx"` are ignored.

If the first row of this sheet contained the column names, this information will also be ignored by `readxl`. Make sure to set `col_names` to `FALSE` or manually specify column names in this case!

The file `urbanpop.xlsx` (view) is available in your directory; it has column names in the first rows.

Instructions 100 XP

Import the second sheet of `"urbanpop.xlsx"`, but skip the first 21 rows. Make sure to set `col_names = FALSE`. Store the resulting data frame in a variable `urbanpop_sel`. Select the first observation from `urbanpop_sel` and print it out.

ex__18.R

```r
# The readxl package is already loaded

# Import the second sheet of urbanpop.xlsx, skipping the first 21 rows: urbanpop_sel

urbanpop_sel <-
    read_excel(
        sheet=2,
        "urbanpop.xlsx",
         col_names = FALSE,
        skip=21
    )
# Print out the first observation from urbanpop_sel

head(urbanpop_sel, 1)
```

16.5 Import a local file

In this part of the chapter you'll learn how to import `.xls` files using the `gdata` package. Similar to the `readxl` package, you can import single Excel sheets from Excel sheets to start your analysis in R.

You'll be working with the `urbanpop.xls` (view) dataset, the `.xls` version of the Excel file you've been working with before. It's available in your current working directory.

Instructions 100 XP

- Load the gdata package with `library()`. `gdata` and `Perl` are already installed on DataCamp's Servers.

- Import the second sheet, named `"1967-1974"`, of `"urbanpop.xls"` with `read.xls()`. Store the resulting data frame as `urban_pop`.
- Print the first 11 observations of `urban_pop` with `head()`.

ex_019.R

```r
# Load the gdata package
library(gdata)

# Import the second sheet of urbanpop.xls: urban_pop
urban_pop <- read.xls(
    "urbanpop.xls",
    sheet = "1967-1974"
)

# Print the first 11 observations using head()
head(urban_pop, 11)
```

16.6 read.xls() wraps around read.table()

Remember how `read.xls()` actually works? It basically comes down to two steps: converting the Excel file to a .csv file using a Perl script, and then reading that .csv file with the read.csv() function that is loaded by default in R, through the utils package.

This means that all the options that you can specify in read.csv(), can also be specified in read.xls().

The urbanpop.xls (view) dataset is already available in your workspace. It's still comprised of three sheets, and has column names in the first row of each sheet.

Instructions `100 XP`

- Finish the `read.xls()` call that reads data from the second sheet of `urbanpop.xls`: skip the first `50 rows` of the sheet. Make sure to set header appropriately and that the country names are not imported as factors.
- Print the first 10 observations of `urban_pop` with `head()`.

ex_20.R

```
# The gdata package is alreaded loaded

# Column names for urban_pop
columns <- c("country", paste0("year_", 1967:1974))

# Finish the read.xls call
urban_pop <- read.xls(
    "urbanpop.xls",
    sheet = 2,
    skip = 50,
    header = FALSE,
    stringsAsFactors = FALSE,
    col.names = columns
)

# Print first 10 observation of urban_pop
head(urban_pop, 10)
```

16.7 Work that Excel data!

Now that you can read in Excel data, let's try to clean and merge it. You already used the `cbind()` function some exercises ago. Let's take it one step further now.

The `urbanpop.xls` (view) dataset is available in your working directory. The file still contains three sheets, and has column names in the first row of each sheet.

Instructions `100 XP`

- Add code to read the data from the third sheet in `"urbanpop.xls"`. You want to end up with three data frames: `urban_sheet1`, `urban_sheet2` and `urban_sheet3`.
- Extend the `cbind()` call so that it also includes `urban_sheet3`. Make sure the first column of `urban_sheet2` and `urban_sheet3` are removed, so you don't have duplicate columns. Store the result in `urban`.
- Use `na.omit()` on the urban data frame to remove all rows that contain `NA` values. Store the cleaned data frame as `urban_clean`.
- Print a summary of `urban_clean` and assert that there are no more `NA` values.

ex__21.R

```r
# Add code to import data from all three sheets in urbanpop.xls
path <- "urbanpop.xls"
urban_sheet1 <- read.xls(path, sheet = 1, stringsAsFactors = FALSE)
urban_sheet2 <- read.xls(path, sheet = 2, stringsAsFactors = FALSE)
urban_sheet3 <- read.xls(path, sheet = 3, stringsAsFactors = FALSE)


# Extend the cbind() call to include urban_sheet3: urban
urban <- cbind(
    urban_sheet1,
    urban_sheet2[-1],
    urban_sheet3[-1]
    )

# Remove all rows with NAs from urban: urban_clean

urban_clean <- na.omit(urban)
# Print out a summary of urban_clean
summary(urban_clean)
```

17 Reproducible Excel work with XLConnect

Beyond importing data from Excel, you can take things one step further with XLConnect. Learn all about it and bridge the gap between R and Excel.

17.1 Connect to a workbook

When working with `XLConnect`, the first step will be to load a workbook in your R session with `loadWorkbook()`; this function will build a "bridge" between your Excel file and your R session.

In this and the following exercises, you will continue to work with `urbanpop.xlsx` (view data folder), containing urban population data throughout time.

Instructions `100 XP`

- Load the `XLConnect` package using `library()`; it is already installed on DataCamp's servers.
- Use `loadWorkbook()` to build a connection to the `"urbanpop.xlsx"` file in R. Call the workbook `my_book`.
- Print out the class of `my_book`. What does this tell you?

ex_023.R

```r
# urbanpop.xlsx is available in your working directory
# Load the XLConnect package
library(XLConnect)

# Build connection to urbanpop.xlsx: my_book
my_book <- loadWorkbook("urbanpop.xlsx")

# Print out the class of my_book
class(my_book)
```

17.2 List and read Excel sheets

Just as readxl and gdata, you can use XLConnect to import data from Excel file into R.

To list the sheets in an Excel file, use getSheets(). To actually import data from a sheet, you can use readWorksheet(). Both functions require an XLConnect workbook object as the first argument.

You'll again be working with urbanpop.xlsx. The my_book object that links to this Excel file has already been created.

Instructions 100 XP

- Print out the sheets of the Excel file that my_book links to.
- Import the second sheet in my_book as a data frame. Print it out.

ex_024.R

```
# XLConnect is already available

# Build connection to urbanpop.xlsx
my_book <- loadWorkbook("urbanpop.xlsx")

# List the sheets in my_book
getSheets(my_book)

# Import the second sheet in my_book
readWorksheet(my_book, sheet = "1967-1974")
```

17.3 Customize readWorksheet

To get a clear overview about urbanpop.xlsx (view) without having to open up the Excel file, you can execute the following code:

```
my_book <- loadWorkbook("urbanpop.xlsx")
sheets <- getSheets(my_book)
all <- lapply(sheets, readWorksheet, object = my_book)
str(all)
```

Suppose we're only interested in urban population data of the years 1968, 1969 and 1970. The data for these years is in the columns 3, 4, and 5 of the second sheet. Only selecting these columns will leave us in the dark about the actual countries the figures belong to.

Instructions 100 XP

- Extend the readWorksheet() command with the startCol and endCol arguments to only import the columns 3, 4, and 5 of the second sheet. -urbanpop_sel no longer contains information about the countries now. Can you write another readWorksheet() command that imports only the first column from the second sheet? Store the resulting data frame as countries. -Use cbind() to paste together countries and urbanpop_sel, in this order. Store the result as selection.

ex_025.R

```r
# XLConnect is already available

# Build connection to urbanpop.xlsx
my_book <- loadWorkbook("urbanpop.xlsx")

# Import columns 3, 4, and 5 from second sheet in my_book: urbanpop_sel
urbanpop_sel <-
    readWorksheet(
        my_book,
        sheet = 2,
        startCol = 3,
        endCol = 5
    )

# Import first column from second sheet in my_book:
# countries
countries <-
    readWorksheet(
        my_book,
        sheet = 2,
        startCol = 1,
        endCol = 1
    )
# cbind() urbanpop_sel and countries together: selection
selection <- cbind(countries, urbanpop_sel)
```

17.4 Add worksheet

Where `readxl` and gdata were only able to import Excel data, `XLConnect's` approach of providing an actual interface to an Excel file makes it able to edit your Excel files from inside R. In this exercise, you'll create a new sheet. In the next exercise, you'll populate the sheet with data, and save the results in a new Excel file.

You'll continue to work with `urbanpop.xlsx`. The `my_book` object that links to this Excel file is already available.

Instructions 100 XP

- Use `createSheet()`, to create a new sheet in `my_book`, named `"data_summary"`.
- Use `getSheets()` to verify that `my_book` now represents an Excel file with four sheets.

ex_026.R

```r
# XLConnect is already available

# Build connection to urbanpop.xlsx
my_book <- loadWorkbook("urbanpop.xlsx")

# Add a worksheet to my_book, named "data_summary"
createSheet(my_book, "data_summary")

# Use getSheets() on my_book
getSheets(my_book)
```

17.5 Populate worksheet

The first step of creating a sheet is done; let's populate it with some data now! summ, a data frame with some summary statistics on the two Excel sheets is already coded so you can take it from there.

Instructions 100 XP

- Use `writeWorksheet()` to populate the `"data_summary"` sheet with the summ data frame.
- Call `saveWorkbook()` to store the adapted Excel workbook as a new file, `"summary.xlsx"`

ex_026.R

```
# XLConnect is already available

# Build connection to urbanpop.xlsx
my_book <- loadWorkbook("urbanpop.xlsx")

# Add a worksheet to my_book, named "data_summary"
createSheet(my_book, "data_summary")

# Create data frame: summ
sheets <- getSheets(my_book)[1:3]
dims <- sapply(
    sheets,
    function(x) dim(readWorksheet(my_book, sheet = x)),
    USE.NAMES = FALSE
)
summ <- data.frame(sheets = sheets,
                   nrows = dims[1, ],
                   ncols = dims[2, ])

# Add data in summ to "data_summary" sheet
writeWorksheet(my_book, summ, sheet = "data_summary")

# Save workbook as summary.xlsx
saveWorkbook(my_book, "summary.xlsx")
```

17.6 Renaming sheets

Come to think of it, “data_summary” is not an ideal name. As the summary of these excel sheets is always data-related, you simply want to name the sheet “summary”.

The code to build a connection to `"urbanpop.xlsx"` and create my_book is already provided for you. It refers to an Excel file with 4 sheets: the three data sheets, and the “data_summary” sheet.

Instructions `100 XP`

- Use `renameSheet()` to rename the fourth sheet to `"summary"`.
- Next, call `getSheets()` on `my_book` to print out the sheet names.

- Finally, make sure to actually save the `my_book` object to a new Excel file, `"renamed.xlsx"`.

ex_027.R

```r
# Build connection to urbanpop.xlsx: my_book
my_book <- loadWorkbook("urbanpop.xlsx")
# Rename "data_summary" sheet to "summary"
renameSheet(my_book, "data_summary", "summary")
# Print out sheets of my_book
getSheets(my_book)
# Save workbook to "renamed.xlsx"
saveWorkbook(my_book, "renamed.xlsx")
```

17.7 Removing sheets

After presenting the new Excel sheet to your peers, it appears not everybody is a big fan. Why summarize sheets and store the info in Excel if all the information is implicitly available? To hell with it, just remove the entire fourth sheet!

Instructions `100 XP`

- Load the XLConnect package.
- Build a connection to "renamed.xlsx", the Excel file that you've built in the previous exercise; it's available in your working directory. Store this connection as my_book.
- Use `removeSheet()` to remove the fourth sheet from `my_book`. The sheet name is `"summary"`. Recall that `removeSheet()` accepts either the index or the name of the sheet as the second argument.
- Save the resulting workbook, `my_book`, to a file `"clean.xlsx"`.

ex_027.R

```r
# Load the XLConnect package

library(XLConnect)
# Build connection to renamed.xlsx: my_book
my_book <- loadWorkbook("renamed.xlsx")

# Remove the fourth sheet
removeSheet(my_book, "summary")
```

```
# Save workbook to "clean.xlsx"
saveWorkbook(my_book, "clean.xlsx")
```

18 Importing data from databases (Part 1)

Many companies store their information in relational databases. The R community has also developed R packages to get data from these architectures. You'll learn how to connect to a database and how to retrieve data from it.

18.1 Establish a connection

The first step to import data from a SQL database is creating a connection to it. As Filip explained, you need different packages depending on the database you want to connect to. All of these packages do this in a uniform way, as specified in the `DBI` package.

`dbConnect()` creates a connection between your R session and a SQL database. The first argument has to be a `DBIdriver` object, that specifies how connections are made and how data is mapped between R and the database. Specifically for MySQL databases, you can build such a driver with `RMySQL::MySQL()`.

If the MySQL database is a remote database hosted on a server, you'll also have to specify the following arguments in `dbConnect()`: `dbname`, `host`, `port`, `user` and `password`. Most of these details have already been provided.

Instructions `100 XP`

- Load the `DBI` library, which is already installed on DataCamp's servers.
- Edit the `dbConnect()` call to connect to the `MySQL` database. Change the port argument (`3306`) and user argument (`"student"`).

ex_001.R

```
# Load the DBI package
library(DBI)

# Edit dbConnect() call
con <- dbConnect(RMySQL::MySQL(),
                 dbname = "tweater",
                 host = "courses.csrrinzqubik.us-east-1.rds.amazonaws.com",
                 port = 3306,
                 user = "student",
                 password = "datacamp")
```

18.2 List the database tables

After you've successfully connected to a remote `MySQL` database, the next step is to see what tables the database contains. You can do this with the `dbListTables()` function. As you might remember from the video, this function requires the connection object as an input, and outputs a character vector with the table names.

Instructions `100 XP`

- Add code to create a vector tables, that contains the tables in the tweater database. You can connect to this database through the con object.
- Display the structure of tables; what's the class of this vector?

ex_002.R

```
# Load the DBI package
library(DBI)
# Connect to the MySQL database: con
con <- dbConnect(RMySQL::MySQL(),
                 dbname = "tweater",
                 host = "courses.csrrinzqubik.us-east-1.rds.amazonaws.com",
                 port = 3306,
                 user = "student",
                 password = "datacamp")
# Build a vector of table names: tables
tables <- dbListTables(con)
# Display structure of tables
str(tables)
class(tables)
```

18.3 Import users

As you might have guessed by now, the database contains data on a more tasty version of Twitter, namely Tweater. Users can post tweats with short recipes for delicious snacks. People can comment on these tweats. There are three tables: *users, tweats*, and *comments* that have relations among them. Which ones, you ask? You'll discover in a moment!

Let's start by importing the data on the users into your R session. You do this with the 'dbReadTable()' function. Simply pass it the connection object ('con'), followed by the name of the table you want to import. The resulting object is a standard R data frame.

Instructions `100 XP`

- Add code that imports the "users" table from the tweater database and store the resulting data frame as users.
- Print the users data frame.

ex_003.R

```
# Load the DBI package
library(DBI)

# Connect to the MySQL database: con
con <- dbConnect(RMySQL::MySQL(),
                 dbname = "tweater",
                 host = "courses.csrrinzqubik.us-east-1.rds.amazonaws.com",
                 port = 3306,
                 user = "student",
                 password = "datacamp")

# Import the users table from tweater: users
users <- dbReadTable(con, "users")

# Print users
users
```

18.4 Import all tables

Next to the users, we're also interested in the `tweats` and `comments` tables. However, separate `dbReadTable()` calls for each and every one of the tables in your database would mean a lot of code duplication. Remember about the `lapply()` function? You can use it again here! A connection is already coded for you, as well as a vector `table_names`, containing the names of all the tables in the database. ### Instructions `100 XP` {.unnumbered}

- Finish the `lapply()` function to import the `users`, `tweats` and `comments` tables in a single call. The result, a list of data frames, will be stored in the variable `tables`.
- Print tables to check if you got it right.

ex_004.R

```r
# Load the DBI package
library(DBI)

# Connect to the MySQL database: con
con <- dbConnect(RMySQL::MySQL(),
                 dbname = "tweater",
                 host = "courses.csrrinzqubik.us-east-1.rds.amazonaws.com",
                 port = 3306,
                 user = "student",
                 password = "datacamp")

# Get table names
table_names <- dbListTables(con)

# Import all tables
tables <- lapply(table_names, dbReadTable, conn = con)

# Print out tables
tables
```

19 Importing data from databases (Part 2)

19.1 Query tweater (1)

In your life as a data scientist, you'll often be working with huge databases that contain tables with millions of rows. If you want to do some analyses on this data, it's possible that you only need a fraction of this data. In this case, it's a good idea to send SQL queries to your database, and only import the data you actually need into R.

`dbGetQuery()` is what you need. As usual, you first pass the connection object to it. The second argument is an SQL query in the form of a character string. This example selects the age variable from the people dataset where gender equals "male":

```
dbGetQuery(con, "SELECT age FROM people WHERE gender = 'male'")
```

A connection to the tweater database has already been coded for you.

Instructions 100 XP

- Use `dbGetQuery()` to create a data frame, elisabeth, that **selects** the `tweat_id` column **from** the comments table **where** elisabeth is the commenter, her`user_id` is 1
- Print out elisabeth so you can see if you queried the database correctly.

ex_005.R

```
# Connect to the database
library(DBI)
con <- dbConnect(RMySQL::MySQL(),
    dbname = "tweater",
    host = "courses.csrrinzqubik.us-east-1.rds.amazonaws.com",
    port = 3306,
    user = "student",
    password = "datacamp"
)

# Import tweat_id column of comments where user_id is 1: elisabeth
```

```
qry <- "SELECT tweat_id FROM comments WHERE user_id = 1"
elisabeth <- dbGetQuery(con, qry)

# Print elisabeth
elisabeth
```

19.2 Query tweater (2)

Apart from checking equality, you can also check for less than and greater than relationships, with < and >, just like in R.

con, a connection to the tweater database, is again available.

Instructions 100 XP

- Create a data frame, `latest`, that **selects** the post column from the `tweats` table observations where the `date` is higher than `'2015-09-21'`.
- Print out latest.

ex_006.R

```
# Connect to the database
library(DBI)
con <- dbConnect(RMySQL::MySQL(),
                 dbname = "tweater",
                 host = "courses.csrrinzqubik.us-east-1.rds.amazonaws.com",
                 port = 3306,
                 user = "student",
                 password = "datacamp")

# Import post column of tweats where date is higher than '2015-09-21': latest

qry <-
    "SELECT post FROM tweats WHERE date > '2015-09-21'"

latest <- dbGetQuery(con, qry)
# Print latest
latest
```

19.3 Query tweater (3)

Suppose that you have a `people` table, with a bunch of information. This time, you want to find out the `age` and `country` of `married` males. Provided that there is a `married` column that's 1 when the person in question is married, the following query would work.

```
SELECT age, country
  FROM people
    WHERE gender = "male" AND married = 1
```

Can you use a similar approach for a more specialized query on the `tweater` database?

Instructions `100 XP`

Create an R data frame, specific, that selects the message column from the comments table where the tweat_id is 77 and the user_id is greater than 4. Print specific.

ex_007.R

```
# Connect to the database
library(DBI)
con <- dbConnect(RMySQL::MySQL(),
                 dbname = "tweater",
                 host = "courses.csrrinzqubik.us-east-1.rds.amazonaws.com",
                 port = 3306,
                 user = "student",
                 password = "datacamp")

# Create data frame specific
qry <-
    "SELECT message
        FROM comments
            WHERE tweat_id = 77 AND user_id > 4"
specific <- dbGetQuery(con, qry)

# Print specific
specific
```

19.4 Query tweater (4)

There are also dedicated SQL functions that you can use in the WHERE clause of an SQL query. For example, CHAR_LENGTH() returns the number of characters in a string.

Instructions 100 XP

- Create a data frame, short, that **selects** the id and name columns **from** the users table **where** the number of characters in the name is strictly less than 5.
- Print short.

ex_008.R

```
# Connect to the database
library(DBI)
con <- dbConnect(RMySQL::MySQL(),
                 dbname = "tweater",
                 host = "courses.csrrinzqubik.us-east-1.rds.amazonaws.com",
                 port = 3306,
                 user = "student",
                 password = "datacamp")

# Create data frame short
qry <-
    "SELECT id, name
        FROM users
            WHERE CHAR_LENGTH(name) < 5
    "
short <- dbGetQuery(con, qry)

# Print short
short
```

19.5 Send - Fetch - Clear

You've used dbGetQuery() multiple times now. This is a virtual function from the DBI package, but is actually implemented by the RMySQL package. Behind the scenes, the following steps are performed:

- Sending the specified query with dbSendQuery();
- Fetching the result of executing the query on the database with dbFetch();

- Clearing the result with `dbClearResult()`.

Let's not use `dbGetQuery()` this time and implement the steps above. This is tedious to write, but it gives you the ability to fetch the query's result in chunks rather than all at once. You can do this by specifying the n argument inside `dbFetch()`.

Instructions `100 XP`

- Inspect the `dbSendQuery()` call that has already been coded for you. It selects the comments for the users with an id above 4.
- Use `dbFetch()` twice. In the first call, import only two records of the query result by setting the `n` argument to `2`. In the second call, import all remaining queries (don't specify `n`). In both calls, simply print the resulting data frames.
- Clear res with dbClearResult().

ex_009.R

```r
# Connect to the database
library(DBI)
con <- dbConnect(RMySQL::MySQL(),
                 dbname = "tweater",
                 host = "courses.csrrinzqubik.us-east-1.rds.amazonaws.com",
                 port = 3306,
                 user = "student",
                 password = "datacamp")

# Send query to the database
res <- dbSendQuery(con, "SELECT * FROM comments WHERE user_id > 4")

# Use dbFetch() twice

dbFetch(res, n = 2)
dbFetch(res)

# Clear res
dbClearResult(res)
```

19.6 Be polite and ...

Every time you connect to a database using `dbConnect()`, you're creating a new connection to the database you're referencing. `RMySQL` automatically specifies a maximum of open connections and closes some of the connections for you, but still: it's always polite to manually disconnect from the database afterwards. You do this with the `dbDisconnect()` function.

The code that connects you to the database is already available, can you finish the script?

Instructions 100 XP

- Using the technique you prefer, build a data frame `long_tweats`. It **selects** the `post` and `date` columns from the observations in `tweats` where the character length of the post variable exceeds `40`.
- Print `long_tweats`.
- Disconnect from the database by using `dbDisconnect()`.

ex_010.R

```
# Connect to the database
library(DBI)
con <- dbConnect(RMySQL::MySQL(),
                 dbname = "tweater",
                 host = "courses.csrrinzqubik.us-east-1.rds.amazonaws.com",
                 port = 3306,
                 user = "student",
                 password = "datacamp")

# Create the data frame  long_tweats
qry <-
    "SELECT post, date
        FROM tweats
            WHERE CHAR_LENGTH(post) > 40
    "
long_tweats <- dbGetQuery(con, qry)
# Print long_tweats
print(long_tweats)

# Disconnect from the database
dbDisconnect(con)
```

20 Importing data from the web (Part 1)

21 Importing data from the web (Part 2)

21.1 From JSON to R

In the simplest setting, fromJSON() can convert character strings that represent JSON data into a nicely structured R list. Give it a try!

Instructions 100 XP

- Load the jsonlite package. It's already installed on DataCamp's servers.
- wine_json represents a JSON. Use fromJSON() to convert it to a list, named wine.
- Display the structure of wine

ex_017.R

```r
# Load the jsonlite package
library(jsonlite)

# wine_json is a JSON
wine_json <- '
    {
        "name":"Chateau Migraine",
        "year":1997,
        "alcohol_pct":12.4,
        "color":"red",
        "awarded":false
    }
'

# Convert wine_json into a list: wine
wine <- fromJSON(wine_json)

# Print structure of wine
print(str(wine))
```

21.2 Quandl API

As Filip showed in the video, `fromJSON()` also works if you pass a URL as a character string or the path to a local file that contains JSON data. Let's try this out on the Quandl API, where you can fetch all sorts of financial and economical data.

Instructions `100 XP`

- `quandl_url` represents a URL. Use `fromJSON()` directly on this URL and store the result in `quandl_data`.
- Display the structure of `quandl_data` .

ex_018.R

```
# jsonlite is preloaded

# Definition of quandl_url
quandl_url <- "https://www.quandl.com/api/v3/datasets/WIKI/FB/data.json?auth_token=i83asDs

# Import Quandl data: quandl_data
quandl_data <- fromJSON(quandl_url)

# Print structure of quandl_data
str(quandl_data)
```

21.3 OMDb API

In the video, you saw how easy it is to interact with an API once you know how to formulate requests. You also saw how to fetch all information on Rain Man from OMDb. Simply perform a `GET()` call, and next ask for the contents with the `content()` function. This `content()` function, which is part of the `httr` package, uses jsonlite behind the scenes to import the JSON data into R.

However, by now you also know that jsonlite can handle URLs itself. Simply passing the request URL to fromJSON() will get your data into R. In this exercise, you will be using this technique to compare the release year of two movies in the Open Movie Database.

Instructions 100 XP

- Two URLs are included in the sample code, as well as a `fromJSON()` call to build `sw4`. Add a similar call to build `sw3`.
- Print out the element named `Title` of both `sw4` and `sw3`. You can use the `$` operator. What movies are we dealing with here? -Write an expression that evaluates to `TRUE if sw4` was released later than `sw3`. This information is stored in the `Year` element of the named lists.

ex_019.R

```
# The package jsonlite is already loaded

# Definition of the URLs
url_sw4 <- "http://www.omdbapi.com/?apikey=72bc447a&i=tt0076759&r=json"
url_sw3 <- "http://www.omdbapi.com/?apikey=72bc447a&i=tt0121766&r=json"

# Import two URLs with fromJSON(): sw4 and sw3
sw3 <- fromJSON(url_sw3)
sw4 <- fromJSON(url_sw4)


# Print out the Title element of both lists
print(sw3$Title)
print(sw4$Title)

# Is the release year of sw4 later than sw3?
sw4$Year > sw3$Year
```

21.4 JSON practice (1)

JSON is built on two structures: objects and arrays. To help you experiment with these, two JSON strings are included in the sample code. It's up to you to change them appropriately and then call jsonlite's fromJSON() function on them each time.

Instrucions 100 XP

- Change the assignment of json1 such that the R vector after conversion contains the numbers 1 up to 6, in ascending order. Next, call `fromJSON()` on `json1`.

- Adapt the code for json2 such that it's converted to a named list with two elements: `a`, containing the numbers 1, 2 and 3 and `b`, containing the numbers 4, 5 and 6. Next, call `fromJSON()` on `json2`.

ex_020.R

```
# jsonlite is already loaded
# Challenge 1
json1 <- '[1, 2, 3, 4, 5, 6]'
fromJSON(json1)
# Challenge 2
json2 <- '{
    "a": [1, 2, 3],
    "b": [4, 5, 6]
}'
fromJSON(json2)
```

21.5 JSON practice (2)

We prepared two more `JSON` strings in the sample code. Can you change them and call `jsonlite`'s `fromJSON()` function on them, similar to the previous exercise?

Instrucions `100 XP`

- Remove characters from `json1` to build a 2 by 2 matrix containing only 1, 2, 3 and 4. Call `fromJSON()` on `json1`.
- Add characters to `json2` such that the data frame in which the json is converted contains an additional observation in the last row. For this observations, `a` equals 5 and `b` equals 6. Call `fromJSON()` one last time, on `json2`.

21.6 JSON practice (2)

Instrucions `100 XP`

- Remove characters from json1 to build a 2 by 2 matrix containing only 1, 2, 3 and 4. Call `fromJSON()` on json1.

- Add characters to json2 such that the data frame in which the json is converted contains an additional observation in the last row. For this observations, a equals 5 and b equals 6. Call fromJSON() one last time, on json2.

ex_021.R

```
# jsonlite is already loaded
# Challenge 1
json1 <- '[[1, 2], [3, 4]]'
fromJSON(json1)

# Challenge 2
json2 <-
   '[{"a": 1, "b": 2}, {"a": 3, "b": 4}, {"a": 5, "b": 6}
]'
fromJSON(json2)
```

21.7 toJSON()

Apart from converting JSON to R with fromJSON(), you can also use toJSON() to convert R data to a JSON format. In its most basic use, you simply pass this function an R object to convert to a JSON. The result is an R object of the class json, which is basically a character string representing that JSON.

For this exercise, you will be working with a .csv file containing information on the amount of desalinated water that is produced around the world. As you'll see, it contains a lot of missing values. This data can be found on the URL that is specified in the sample code.

Instrucions 100 XP

- Use a function of the utils package to import the .csv file directly from the URL specified in url_csv. Save the resulting data frame as water. Make sure that strings are not imported as factors.
- Convert the data frame water to a JSON. Call the resulting object water_json.
- Print out water_json.

ex_021.R

```
# jsonlite is already loaded

# URL pointing to the .csv file
```

```
url_csv <- "http://s3.amazonaws.com/assets.datacamp.com/production/course_1478/datasets/wa

# Import the .csv file located at url_csv
water <- read.csv(url_csv)

# Convert the data file according to the requirements
water_json <- toJSON(water)

# Print out water_json
print(water_json)
```

21.8 Minify and prettify

JSONs can come in different formats. Take these two JSONs, that are in fact exactly the same: the first one is in a minified format, the second one is in a pretty format with indentation, whitespace and new lines:

```
# Mini
{"a":1,"b":2,"c":{"x":5,"y":6}}

# Pretty
{
  "a": 1,
  "b": 2,
  "c": {
    "x": 5,
    "y": 6
  }
}
```

Unless you're a computer, you surely prefer the second version. However, the standard form that `toJSON()` returns, is the minified version, as it is more concise. You can adapt this behavior by setting the `pretty` argument inside `toJSON()` to `TRUE`. If you already have a JSON string, you can use `prettify()` or `minify()` to make the JSON pretty or as concise as possible.

Instrucions `100 XP`

- Convert the `mtcars` dataset, which is available in R by default, to a `pretty JSON`. Call the resulting JSON `pretty_json`.

- Print out `pretty_json`. Can you understand the output easily?
- Convert `pretty_json` to a minimal version using `minify()`. Store this version under a new variable, `mini_json`.
- Print out `mini_json`. Which version do you prefer, the pretty one or the minified one?

ex__022.R

```r
# jsonlite is already loaded
# Convert mtcars to a pretty JSON: pretty_json
pretty_json <- toJSON(mtcars, pretty = TRUE)
# Print pretty_json
print(pretty_json)

# Minify pretty_json: mini_json

mini_json <- minify(pretty_json)
# Print mini_json
print(mini_json)
```

22 Importing data from statistical software packages

Next to R, there are also other commonly used statistical software packages: SAS, STATA and SPSS. Each of them has their own file format. Learn how to use the haven and foreign packages to get them into R with remarkable ease!

22.1 Import SAS data with haven

`haven` is an extremely easy-to-use package to import data from three software packages: SAS, STATA and SPSS. Depending on the software, you use different functions:

- SAS: read_sas()
- STATA: read_dta() (or read_stata(), which are identical)
- SPSS: read_sav() or read_por(), depending on the file type.

All these functions take one key argument: the path to your local file. In fact, you can even pass a URL; `haven` will then automatically download the file for you before importing it.

You'll be working with data on the age, gender, income, and purchase level (0 = low, 1 = high) of 36 individuals (Source: SAS). The information is stored in a SAS file, `sales.sas7bdat`, which is available in the dataset directory. You can also download the data here http://s3.amazonaws.com/assets.datacamp.com/production/course_1478/datasets/sales.sas7bdat.

Instructions `100 XP`

- Load the `haven` package.
- Import the data file `"sales.sas7bdat"`. Call the imported data frame sales.
- Display the structure of `sales` with `str()`. Some columns represent categorical variables, so they should be factors.

ex_023.R

```
# Load the haven package
library(haven)
```

```
# Import sales.sas7bdat: sales
sales <- read_sas("sales.sas7bdat")

# Display the structure of sales
str(sales)
```

22.2 Import STATA data with haven

Next up are STATA data files; you can use `read_dta()` for these.

When inspecting the result of the `read_dta()` call, you will notice that one column will be imported as a `labelled` vector, an R equivalent for the common data structure in other statistical environments. In order to effectively continue working on the data in R, it's best to change this data into a standard R class. To convert a variable of the class labelled to a factor, you'll need haven's `as_factor()` function.

In this exercise, you will work with data on yearly import and export numbers of sugar, both in USD and in weight. The data can be found at: http://assets.datacamp.com/production/course_1478/datasets/trade.dta

Instructions `100 XP`

- Import the data file directly from the URL using read_dta(), and store it as sugar.
- Print out the structure of sugar. The Date column has class labelled.
- Convert the values in the Date column of sugar to dates, using `as.Date(as_factor(___))`.
- Print out the structure of sugar once more. Looks better now?

ex_024.R

```
# haven is already loaded
library(haven)
# Import the data from the URL: sugar
url <- "http://assets.datacamp.com/production/course_1478/datasets/trade.dta"

sugar <- read_dta(url)

# Structure of sugar
str(sugar)

# Convert values in Date column to dates
```

```
sugar$Date <- as.Date(as_factor(sugar$Date))

# Structure of sugar again
str(sugar)
```

22.3 Import SPSS data with haven

The haven package can also import data files from SPSS. Again, importing the data is pretty straightforward. Depending on the SPSS data file you're working with, you'll need either `read_sav()` - for `.sav` files - or `read_por()` - for `.por` files.

In this exercise, you will work with data on four of the Big Five personality traits for 434 persons (Source: University of Bath). The Big Five is a psychological concept including, originally, five dimensions of personality to classify human personality. The SPSS dataset is called person.sav and is available in your working directory.

Instructions `100 XP`

- Use `read_sav()` to import the `SPSS` data in `"person.sav"`. Name the imported data frame `traits`.
- `traits` contains several missing values, or `NAs`. Run `summary()` on it to find out how many`NAs` are contained in each variable.
- Print out a subset of those individuals that scored high on Extroversion and on Agreeableness, i.e. scoring higher than 40 on each of these two categories. You can use `subset()` for this.

ex_025.R

```
# haven is already loaded

# Import person.sav: traits
traits <- read_sav("person.sav")

# Summarize traits
summary(traits)

# Print out a subset

subset(traits, Extroversion > 40 & Agreeableness > 40)
```

22.4 Factorize, round two

In the last exercise you learned how to import a data file using the command `read_sav()`. With SPSS data files, it can also happen that some of the variables you import have the `labelled` class. This is done to keep all the labelling information that was originally present in the `.sav` and `.por` files. It's advised to coerce (or change) these variables to factors or other standard R classes.

The data for this exercise involves information on employees and their demographic and economic attributes (Source: QRiE). The data can be found on the following URL: http://s3.amazonaws.com/assets.datacamp.com/production/course_1478/datasets/employee.sav

Instructions 100 XP

- Import the SPSS data straight from the URL and store the resulting data frame as work.
- Display the summary of the GENDER column of work. This information doesn't give you a lot of useful information, right?
- Convert the GENDER column in work to a factor, the class to denote categorical variables in R. Use as_factor().
- Once again display the summary of the GENDER column. This time, the printout makes much more sense.

ex_027.R

```
# haven is already loaded

# Import SPSS data from the URL: work
url <- "http://s3.amazonaws.com/assets.datacamp.com/production/course_1478/datasets/employ
work <- read_sav(url)

# Display summary of work$GENDER
summary(work$GENDER)
# Convert work$GENDER to a factor
work$GENDER <- as_factor(work$GENDER)
# Display summary of work$GENDER again
summary(work$GENDER)
```

22.5 Import STATA data with foreign (1)

The `foreign` package offers a simple function to import and read STATA data: `read.dta()`.

In this exercise, you will import data on the US presidential elections in the year 2000. The data in `florida.dta` contains the total numbers of votes for each of the four candidates as well as the total number of votes per election area in the state of Florida (Source: Florida Department of State). The file is available in the dataset directory, you can download it here http://s3.amazonaws.com/assets.datacamp.com/production/course_1478/datasets/florida.dta

if you want to experiment some more.

Instructions 100 XP

- Load the `foreign` package; it's already installed on DataCamp's servers.
- Import the data on the elections in Florida, `"florida.dta"`, and name the resulting data frame florida. Use `read.dta()` without specifying extra arguments.
- Check out the last 6 observations of florida with `tail()`.

ex_028.R

```
# Load the foreign package
library(foreign)

# Import florida.dta and name the resulting data frame florida

florida <- read.dta("florida.dta")
# Check tail() of florida
tail(florida)
```

22.6 Import STATA data with foreign (2)

Data can be very diverse, going from character vectors to categorical variables, dates and more. It's in these cases that the additional arguments of `read.dta()` will come in handy.

The arguments you will use most often are convert.dates, convert.factors, missing.type and convert.underscore. Their meaning is pretty straightforward, as Filip explained in the video. It's all about correctly converting STATA data to standard R data structures. Type ?read.dta to find out about about the default values.

The dataset for this exercise contains socio-economic measures and access to education for different individuals (Source: World Bank). This data is available as `edequality.dta`, which is located in the worldbank folder in your working directory.

Instructions 100 XP

ex_029.R

```r
# foreign is already loaded

# Specify the file path using file.path(): path
path <- file.path("worldbank", "edequality.dta")

# Create and print structure of edu_equal_1
edu_equal_1 <- read.dta(path)
str(edu_equal_1)


# Create and print structure of edu_equal_2
edu_equal_2 <-
    read.dta(
        path,
        convert.factors = FALSE
    )
str(edu_equal_2)


# Create and print structure of edu_equal_3

edu_equal_3 <-
    read.dta(
        path,
        convert.underscore = TRUE
    )
str(edu_equal_3)


nrow(
    subset(
        edu_equal_1,
        edu_equal_1$ethnicity_head == "Bulgarian",
        income > 1000
    )
)
```

22.7 Import SPSS data with foreign (1)

All great things come in pairs. Where foreign provided `read.dta()` to read Stata data, there's also `read.spss()` to read SPSS data files. To get a data frame, make sure to set `to.data.frame = TRUE` inside `read.spss()`.

In this exercise, you'll be working with socio-economic variables from different countries (Source: Quantative Data Analysis in Education). The SPSS data is in a file called international.sav, which is in your working directory. You can also download it here if you want to play around with it some more.

Instructions `100 XP`

ex_030.R

```
# foreign is already loaded

# Import international.sav as a data frame: demo

demo <- read.spss(
    "international.sav", to.data.frame=TRUE)
# Create boxplot of gdp variable of demo
boxplot(demo$gdp)
```

22.8 Import SPSS data with foreign (2)

In the previous exercise, you used the to.data.frame argument inside `read.spss()`. There are many other ways in which to customize the way your SPSS data is imported.

In this exercise you will experiment with another argument, `use.value.labels`. It specifies whether variables with value labels should be converted into R factors with levels that are named accordingly. The argument is TRUE by default which means that so called labelled variables inside SPSS are converted to factors inside R.

You'll again be working with the `international.sav` data, which is available in your current working directory.

Instructions 100 XP

- Import the data file "international.sav" as a data frame, demo_1.
- Print the first few rows of demo_1 using the head() function.
- Import the data file "international.sav" as a data frame, demo_2, but this time in a way such that variables with value labels are not converted to R factors.
- Again, print the first few rows of demo_2. Can you tell the difference between the two data frames?

ex_031.R

```
# foreign is already loaded
# Import international.sav as demo_1
demo_1 <- read.spss(
    "international.sav",
    to.data.frame = TRUE,
    )
# Print out the head of demo_1
head(demo_1)

# Import international.sav as demo_2
demo_2 <- read.spss(
    "international.sav",
    to.data.frame = TRUE,
    use.value.labels = FALSE
    )

# Print out the head of demo_2
head(demo_2)
```

23 Common Data Problems

In this chapter, you'll learn how to overcome some of the most common dirty data problems. You'll convert data types, apply range constraints to remove future data points, and remove duplicated data points to avoid double-counting.

23.1 Converting data types

Throughout this chapter, you'll be working with San Francisco bike share ride data called bike_share_rides. It contains information on start and end stations of each trip, the trip duration, and some user information.

Before beginning to analyze any dataset, it's important to take a look at the different types of columns you'll be working with, which you can do using `glimpse()`.

In this exercise, you'll take a look at the data types contained in `bike_share_rides` and see how an incorrect data type can flaw your analysis.

`dplyr` and `assertive` are loaded and `bike_share_rides` is available.

Instructions `100 XP`

- Examine the data types of the columns of bike_share_rides.
- Get a summary of the user_birth_year column of bike_share_rides.

ex_001.R

```r
# Glimpse at bike_share_rides
glimpse(bike_share_rides)

# Summary of user_birth_year
summary(bike_share_rides$user_birth_year)

# Convert user_birth_year to factor: user_birth_year_fct
bike_share_rides <- bike_share_rides %>%
  mutate(user_birth_year_fct = as.factor(user_birth_year))
```

```r
# Assert user_birth_year_fct is a factor
assert_is_factor(bike_share_rides$user_birth_year_fct)

# Summary of user_birth_year_fct
summary(bike_share_rides$user_birth_year_fct)
```

23.2 Trimming strings

In the previous exercise, you were able to identify the correct data type and convert `user_birth_year` to the correct type, allowing you to extract counts that gave you a bit more insight into the dataset.

Another common dirty data problem is having extra bits like percent signs or periods in numbers, causing them to be read in as `characters`. In order to be able to crunch these numbers, the extra bits need to be removed and the numbers need to be converted from `character` to `numeric`. In this exercise, you'll need to convert the duration column from character to `numeric`, but before this can happen, the word "minutes" needs to be removed from each value.

`dplyr`, `assertive`, and `stringr` are loaded and bike_share_rides is available.

Instructions `100 XP`

- Use `str_remove()` to remove `"minutes"` from the duration column of `bike_share_rides`. Add this as a new column called `duration_trimmed`.
- Convert the `duration_trimmed` column to a numeric type and add this as a new column called `duration_mins`.
- Glimpse at `bike_share_rides` and assert that the `duration_mins` column is `numeric`.
- Calculate the mean of `duration_mins`.

ex_002.R

```r
bike_share_rides <- bike_share_rides %>%
  # Remove 'minutes' from duration: duration_trimmed
  mutate(duration_trimmed = str_remove(duration, "minutes"),
  # Convert duration_trimmed to numeric: duration_mins
         duration_mins = as.numeric(duration_trimmed))

# Glimpse at bike_share_rides
glimpse(bike_share_rides)
```

```r
# Assert duration_mins is numeric
assert_is_numeric(bike_share_rides$duration_mins)

# Calculate mean duration
mean(bike_share_rides$duration_mins)
```

23.3 Ride duration constraints

Values that are out of range can throw off an analysis, so it's important to catch them early on. In this exercise, you'll be examining the `duration_min` column more closely. Bikes are not allowed to be kept out for more than 24 hours, or 1440 minutes at a time, but issues with some of the bikes caused inaccurate recording of the time they were returned.

In this exercise, you'll replace erroneous data with the range limit (1440 minutes), however, you could just as easily replace these values with NAs.

load `dplyr`, `assertive`, and `ggplot2` and`bike_share_rides`.

Instructions `100 XP`

Create a three-bin histogram of the `duration_min` column of `bike_share_rides` using `ggplot2` to identify if there is out-of-range data.

ex_003.R

```r
# Create breaks
breaks <-
  c(
    min(bike_share_rides$duration_min),
    0,
    1440,
    max(bike_share_rides$duration_min)
  )

# Create a histogram of duration_min
ggplot(bike_share_rides, aes(duration_min)) +
  geom_histogram(breaks = breaks)

# duration_min_const: replace vals of duration_min > 1440 with 1440
bike_share_rides <- bike_share_rides %>%
  mutate(
    duration_min_const = replace(
      duration_min,
      duration_min > 1440 ,
      1440
    )
  )

# Make sure all values of
# duration_min_const are between 0 and 1440
assert_all_are_in_closed_range(
  bike_share_rides$duration_min_const,
  lower = 0, upper = 1440
)
```

23.4 Back to the future

Something has gone wrong and it looks like you have data with dates from the future, which is way outside of the date range you expected to be working with. To fix this, you'll need to remove any rides from the dataset that have a `date` in the future. Before you can do this, the date column needs to be converted from a `character` to a `Date`. Having these as `Date` objects will make it much easier to figure out which rides are from the future, since R makes it easy to check if one Date object is before (<) or after (>) another.

load `dplyr`, `assertive` and `bike_share_rides`.

Instructions `100 XP`

- Convert the date column of `bike_share_rides` from `character` to the `Date` data type.
- Assert that all values in the `date` column happened sometime in the past and not in the future.
- Filter `bike_share_rides` to get only the rides from the past or today, and save this as `bike_share_rides_past`.
- Assert that the dates in `bike_share_rides_past` occurred only in the past.

ex_004.R

```r
library(lubridate)
# Convert date to Date type
bike_share_rides <- bike_share_rides %>%
  mutate(date = as.Date(date))

# Make sure all dates are in the past
assert_all_are_in_past(bike_share_rides$date)


# Filter for rides that occurred before or on today's date
bike_share_rides_past <- bike_share_rides %>%
  filter(date <= today())

# Make sure all dates from bike_share_rides_past are in the past
assert_all_are_in_past(bike_share_rides_past$date)
```

23.5 Full duplicates

You've been notified that an update has been made to the bike sharing data pipeline to make it more efficient, but that duplicates are more likely to be generated as a result. To make sure that you can continue using the same scripts to run your weekly analyses about ride statistics, you'll need to ensure that any duplicates in the dataset are removed first.

When multiple rows of a data frame share the same values for all columns, they're full duplicates of each other. Removing duplicates like this is important, since having the same value repeated multiple times can alter summary statistics like the mean and median. Each ride, including its `ride_id` should be unique.

be sure that`dplyr` is loaded and `bike_share_rides` is available.

Instructions `100 XP`

- Get the total number of full duplicates in `bike_share_rides`.
- Remove all full duplicates from `bike_share_rides` and save the new data frame as `bike_share_rides_unique`.
- Get the total number of full duplicates in the new `bike_share_rides_unique` data frame.

ex__005.R

```
# Count the number of full duplicate

sum(duplicated(bike_share_rides))
```

```r
# Remove duplicates
bike_share_rides_unique <- distinct(bike_share_rides)

# Count the full duplicates in bike_share_rides_unique
sum(duplicated(bike_share_rides_unique))
```

23.6 Removing partial duplicates

Now that you've identified and removed the full duplicates, it's time to check for partial duplicates. Partial duplicates are a bit tricker to deal with than full duplicates. In this exercise, you'll first identify any partial duplicates and then practice the most common technique to deal with them, which involves dropping all partial duplicates, keeping only the first.

`dplyr` is loaded and `bike_share_rides` is available.

Instructions `100 XP`

- Count the number of occurrences of each `ride_id`.
- Filter for `ride_id`s that occur multiple times.
- Remove full and partial duplicates from `bike_share_rides` based on `ride_id` only, keeping all columns.
- Store this as `bike_share_rides_unique`.

ex_006.R

```r
# Find duplicated ride_ids
bike_share_rides %>%
  count(ride_id) %>%
  filter(n > 1)

# Remove full and partial duplicates
bike_share_rides_unique <- bike_share_rides %>%
  # Only based on ride_id instead of all cols
  distinct(ride_id, .keep_all = TRUE)

# Find duplicated ride_ids in bike_share_rides_unique
bike_share_rides_unique %>%
  # Count the number of occurrences of each ride_id
  count(ride_id) %>%
  # Filter for rows with a count > 1
  filter(n>1)
```

23.7 Aggregating partial duplicates

Another way of handling partial duplicates is to compute a summary statistic of the values that differ between partial duplicates, such as mean, median, maximum, or minimum. This can come in handy when you're not sure how your data was collected and want an average, or if based on domain knowledge, you'd rather have too high of an estimate than too low of an estimate (or vice versa).

`dplyr` is loaded and `bike_share_rides` is available.

23.7.1 Instructions `100 XP`

- Group `bike_share_rides` by ride_id and date.
- Add a column called duration_min_avg that contains the mean ride duration for the row's ride_id and date.
- Remove duplicates based on ride_id and date, keeping all columns of the data frame.
- Remove the duration_min column.

ex_007.R

```
bike_share_rides %>%
  # Group by ride_id and date
  group_by(ride_id, date) %>%
  # Add duration_min_avg column
  mutate(duration_min_avg = mean(duration_min)) %>%
  # Remove duplicates based on ride_id and date, keep all cols
  distinct(ride_id, date, .keep_all = TRUE) %>%
  # Remove duration_min column
  select(-duration_min)
```

24 Categorical and Text Data

Categorical and text data can often be some of the messiest parts of a dataset due to their unstructured nature. In this chapter, you'll learn how to fix whitespace and capitalization inconsistencies in category labels, collapse multiple categories into one, and reformat strings for consistency.

24.1 Not a member

Now that you've practiced identifying membership constraint problems, it's time to fix these problems in a new dataset. Throughout this chapter, you'll be working with a dataset called `sfo_survey`, containing survey responses from passengers taking flights from San Francisco International Airport (SFO). Participants were asked questions about the airport's cleanliness, wait times, safety, and their overall satisfaction.

There were a few issues during data collection that resulted in some inconsistencies in the dataset. In this exercise, you'll be working with the `dest_size` column, which categorizes the size of the destination airport that the passengers were flying to. A data frame called `dest_sizes` is available that contains all the possible destination sizes. Your mission is to find rows with invalid `dest_sizes` and remove them from the data frame.

Load `dplyr`, `sfo_survey` and `dest_sizes`.

Instructions `100 XP`

- Count the number of occurrences of each `dest_size` in `sfo_survey`.

ex_008.R

```
# Count the number of occurrences of dest_size
sfo_survey %>%
  count(dest_size)

# Find bad dest_size rows
sfo_survey %>%
  # Join with dest_sizes data frame to get bad dest_size rows
  anti_join(dest_sizes, by = "dest_size") %>%
  # Select id, airline, destination, and dest_size cols
  select(id, airline, destination, dest_size)

# Remove bad dest_size rows
sfo_survey %>%
  # Join with dest_sizes
  semi_join(dest_sizes, by = "dest_size") %>%
  # Count the number of each dest_size
  count(dest_size)
```

24.2 Identifiying inconsistency

In the video exercise, you learned about different kinds of inconsistencies that can occur within categories, making it look like a variable has more categories than it should.

In this exercise, you'll continue working with the sfo_survey dataset. You'll examine the `dest_size` column again as well as the cleanliness column and determine what kind of issues, if any, these two categorical variables face.

Load `dplyr` and `sfo_survey` .

Instructions `100 XP`

- Count the number of occurrences of each category of the `dest_size` variable of `sfo_survey`.

ex__009.R

```
# Count dest_size
# Count dest_size
sfo_survey %>%
  count(dest_size)

# Count cleanliness
sfo_survey %>%
  count(cleanliness)
```

24.3 Correcting inconsistency

Now that you've identified that dest_size has whitespace inconsistencies and cleanliness has capitalization inconsistencies, you'll use the new tools at your disposal to fix the inconsistent values in `sfo_survey` instead of removing the data points entirely, which could add bias to your dataset if more than 5% of the data points need to be dropped.

Load `dplyr`, `stringr` and `sfo_survey`.

Instructions 100 XP

- Add a column to sfo_survey called dest_size_trimmed that contains the values in the dest_size column with all leading and trailing whitespace removed.
- Add another column called cleanliness_lower that contains the values in the cleanliness column converted to all lowercase.
- Count the number of occurrences of each category in dest_size_trimmed.
- Count the number of occurrences of each category in cleanliness_lower.

ex_010.R

```r
# Add new columns to sfo_survey
sfo_survey <- sfo_survey %>%
  # dest_size_trimmed: dest_size without whitespace
  mutate(dest_size_trimmed =  str_trim(dest_size),
         # cleanliness_lower: cleanliness converted to lowercase
         cleanliness_lower = str_to_lower(cleanliness))

# Count values of dest_size_trimmed
sfo_survey %>%
  count(dest_size_trimmed)

# Count values of cleanliness_lower
sfo_survey %>%
  count(cleanliness_lower)
```

24.4 Collapsing categories

One of the tablets that participants filled out the `sfo_survey` on was not properly configured, allowing the response for `dest_region` to be free text instead of a dropdown menu. This resulted in some inconsistencies in the `dest_region` variable that you'll need to correct in this exercise to ensure that the numbers you report to your boss are as accurate as possible.

Make dplyr, forcats and sfo_survey available.

Instructions 100 XP

- Count the categories of dest_region.

ex_011.R

```
# Count categories of dest_region
sfo_survey %>%
  count(dest_region)

# Categories to map to Europe
europe_categories <- c("EU","eur", "Europ")

# Add a new col dest_region_collapsed
sfo_survey %>%
  # Map all categories in europe_categories to Europe
  mutate(
    dest_region_collapsed = fct_collapse(dest_region,
                                     Europe = europe_categories)) %>%
  # Count categories of dest_region_collapsed
  count(dest_region_collapsed)
```

24.5 Detecting inconsistency text data

You've recently received some news that the customer support team wants to ask the SFO survey participants some follow-up questions. However, the auto-dialer that the call center uses isn't able to parse all of the phone numbers since they're all in different formats. After some investigation, you found that some phone numbers are written with hyphens (-) and some are written with parentheses ((,)). In this exercise, you'll figure out which phone numbers have these issues so that you know which ones need fixing.

Load dplyr, stringr and sfo_survey.

Instructions 100 XP

- Filter for rows with phone numbers that contain "-"s.
- Filter for rows with phone numbers that contain "(", or ")". Remember to use fixed() when searching for parentheses.

ex_012.R

```
# Filter for rows with "-" in the phone column
sfo_survey %>%
  filter(str_detect(phone, "-"))

# Filter for rows with "(" or ")" in the phone column
sfo_survey %>%
  filter(str_detect(phone, fixed("(")) | str_detect(phone, fixed(")")))
```

24.6 Replacing and removing

In the last exercise, you saw that the `phone` column of `sfo_survey` is plagued with unnecessary parentheses and hyphens. The customer support team has requested that all phone numbers be in the format `"123 456 7890"`. In this exercise, you'll use your new stringr skills to fulfill this request.

Make `dplyr`, `stringr` and `sfo_survey` available.

Instructions `100 XP`

- Remove opening and closing parentheses from the phone column. Store this as a variable called phone_no_parens. Remember to use fixed()!
- Add a new column to sfo_survey called phone_no_parens that contains the contents of phone_no_parens.

ex_013.R

```
# Remove parentheses from phone column
phone_no_parens <- sfo_survey$phone %>%
  # Remove "("s
  str_remove(fixed("(")) %>%
  # Remove ")"s
  str_remove(fixed(")"))

  # Remove parentheses from phone column
phone_no_parens <- sfo_survey$phone %>%
  # Remove "("s
  str_remove_all(fixed("(")) %>%
  # Remove ")"s
  str_remove_all(fixed(")"))

# Add phone_no_parens as column
sfo_survey %>%
  mutate(phone_no_parens=phone_no_parens)

# Remove parentheses from phone column
phone_no_parens <- sfo_survey$phone %>%
  # Remove "("s
  str_remove_all(fixed("(")) %>%
  # Remove ")"s
  str_remove_all(fixed(")"))

# Add phone_no_parens as column
sfo_survey %>%
  mutate(
    phone_no_parens = phone_no_parens,
  # Replace all hyphens in phone_no_parens with spaces
    phone_clean = str_replace_all(
      phone_no_parens, "-", " "))
```

24.7 Invalid phone numbers

The customer support team is grateful for your work so far, but during their first day of calling participants, they ran into some phone numbers that were invalid. In this exercise, you'll remove any rows with invalid phone numbers so that these faulty numbers don't keep slowing the team down.

Make `dplyr`, `stringr` and `sfo_survey` available.

Instructions `100 XP`

- Examine the invalid phone numbers by filtering for numbers whose length is not equal to 12.
- Remove the rows with invalid numbers by filtering for numbers with a length of exactly 12

ex__014.R

```
# Check out the invalid numbers
sfo_survey %>%
  filter(str_length(phone) != 12)

# Remove rows with invalid numbers
sfo_survey %>%
   filter(str_length(phone) == 12)
```

25 Advanced Data Problems

In this chapter, you'll dive into more advanced data cleaning problems, such as ensuring that weights are all written in kilograms instead of pounds. You'll also gain invaluable skills that will help you verify that values have been added correctly and that missing values don't negatively impact your analyses.

25.1 Date uniformity

In this chapter, you work at an asset management company and you'll be working with the `accounts` dataset, which contains information about each customer, the amount in their account, and the date their account was opened. Your boss has asked you to calculate some summary statistics about the average value of each account and whether the age of the account is associated with a higher or lower account value. Before you can do this, you need to make sure that the accounts dataset you've been given doesn't contain any uniformity problems. In this exercise, you'll investigate the `date_opened` column and clean it up so that all the dates are in the same format.

Make `dplyr` and `lubridate` and `accounts` available. ### Instructions `100 XP` {.unnumbered} Take a look at the head of accounts to get a sense of the data you're working with.

ex__015.R

```r
head(accounts)

# Define the date formats
formats <- c("%Y-%m-%d", "%B %d, %Y")

# Convert dates to the same format
accounts %>%
  mutate(date_opened_clean = parse_date_time(date_opened, formats))
```

25.2 Currency uniformity

Now that your dates are in order, you'll need to correct any unit differences. When you first plot the data, you'll notice that there's a group of very high values, and a group of relatively lower values. The bank has two different offices - one in New York, and one in Tokyo, so you suspect that the accounts managed by the Tokyo office are in Japanese yen instead of U.S. dollars. Luckily, you have a data frame called `account_offices` that indicates which office manages each customer's account, so you can use this information to figure out which `totals` need to be converted from `yen` to dollars.

The formula to convert yen to dollars is USD = JPY / 104.

load `dplyr`, `ggplot2` , the `accounts` and `account_offices` data frames.

Instructions `100 XP`

- Create a scatter plot with date_opened on the x-axis and total on the y-axis.
- Left join accounts and account_offices by their id columns.

ex_016.R

```r
# Scatter plot of opening date and total amount
accounts %>%
  ggplot(aes(x = date_opened, y = total)) +
  geom_point()

# Left join accounts to account_offices by id
accounts %>%
  left_join(account_offices, by = "id") %>%
  # Convert totals from the Tokyo office to USD
  mutate(total_usd = ifelse(office == "Tokyo", total / 104, total)) %>%
  # Scatter plot of opening date vs total_usd
  ggplot(aes(x = date_opened, y = total_usd)) +
    geom_point()
```

25.3 Validating totals

In this lesson, you'll continue to work with the accounts data frame, but this time, you have a bit more information about each account. There are three different funds that account holders can store their money in. In this exercise, you'll validate whether the total amount in each account is equal to the sum of the amount in `fund_A`, `fund_B`, and `fund_C`. If there are any accounts that don't match up, you can look into them further to see what went wrong in the bookkeeping that led to inconsistencies.

Load `dplyr` and `accounts`.

Instructions 100 XP

- Create a new column called `theoretical_total` that contains the sum of the amounts in each fund.
- Find the accounts where the total doesn't match the `theoretical_total`.

ex__017.R

```
# Find invalid totals
accounts %>%
  # theoretical_total: sum of the three funds
  mutate(
    theoretical_total = fund_A + fund_B + fund_C
  )%>%
  # Find accounts where total doesn't match theoretical_total
  filter(theoretical_total != total)
```

25.4 Validating age

Now that you found some inconsistencies in the `total` amounts, you're suspicious that there may also be inconsistencies in the `acct_age` column, and you want tosee if these inconsistencies are related. Using the skills you learned from the video exercise, you'll need to validate the age of each account and see if rows with inconsistent `acct_ages` are the same ones that had inconsistent `totals`

Make `dplyr` and `lubridate` and `accounts` available.

Instructions 100 XP

- Create a new column called `theoretical_age` that contains the age of each account based on the `date_opened`.
- Find the accounts where the `acct_age` doesn't match the `theoretical_age`.

ex__018.R

```
# Find invalid acct_age
accounts %>%
  # theoretical_age: age of acct based on date_opened
  mutate(theoretical_age =
    floor(
      as.numeric(
        as.Date(date_opened) %--% today(),
        "years"
      )
    )
  ) %>%
  # Filter for rows where acct_age is different from theoretical_age
  filter(theoretical_age != acct_age)
```

25.5 Visualizing missing data

Dealing with missing data is one of the most common tasks in data science. There are a variety of types of missingness, as well as a variety of types of solutions to missing data.

You just received a new version of the accounts data frame containing data on the amount held and amount invested for new and existing customers. However, there are rows with missing `inv_amount` values.

You know for a fact that most customers below 25 do not have investment accounts yet, and suspect it could be driving the missingness.

Load `dplyr` and `visdat` packages and `accounts` dataframe.

Instructions `100 XP`

- Visualize the missing values in `accounts` by column using a function from the visdat package.
- Add a logical column to accounts called `missing_inv` that indicates whether each row is missing the inv_amount or not.
- Group by `missing_inv`.
- Calculate the mean age for each group of missing_inv.

ex_019.R

```
vis_miss(accounts)
# Visualize the missing values by column
vis_miss(accounts)

accounts %>%
  # missing_inv: Is inv_amount missing?
  mutate(missing_inv = is.na(inv_amount)) %>%
  # Group by missing_inv
  group_by(missing_inv) %>%
  # Calculate mean age for each missing_inv group
  summarize(avg_age = mean(age))

# Visualize the missing values by column
vis_miss(accounts)

accounts %>%
  # missing_inv: Is inv_amount missing?
  mutate(missing_inv = is.na(inv_amount)) %>%
  # Group by missing_inv
  group_by(missing_inv) %>%
  # Calculate mean age for each missing_inv group
  summarize(avg_age = mean(age))

# Sort by age and visualize missing vals
accounts %>%
  arrange(age) %>%
  vis_miss()
```

25.6 Treating missing data

In this exercise, you're working with another version of the accounts data that contains missing values for both the `cust_id` and `acct_amount` columns.

You want to figure out how many unique customers the bank has, as well as the average amount held by customers. You know that rows with missing `cust_id` don't really help you, and that on average, the `acct_amount` is usually 5 times the amount of `inv_amount`.

In this exercise, you will drop rows of `accounts` with missing `cust_ids`, and impute missing values of `inv_amount` with some domain knowledge.

Load `dplyr`, `assertive` and `accounts`.

Instructions 100 XP

- Filter accounts to remove rows with missing cust_ids and save as `accounts_clean`.
- Create a new column called `acct_amount_filled`, which contains the values of `acct_amount`, except all `NA` values should be replaced with 5 times the amount in `inv_amount`.
- Assert that there are no missing values in the `cust_id` column of `accounts_clean`.
- Assert that there are no missing values in the `acct_amount_filled` column of accounts_clean.

ex_020.R

```
accounts_clean <- accounts %>%
  # Filter to remove rows with missing cust_id
  filter(!is.na(cust_id))


accounts_clean
# Create accounts_clean
accounts_clean <- accounts %>%
  # Filter to remove rows with missing cust_id
  filter(!is.na(cust_id)) %>%
  # Add new col acct_amount_filled with replaced NAs
  mutate(
    acct_amount_filled = ifelse(
      is.na(acct_amount), 5 * inv_amount, acct_amount))

accounts_clean

# Create accounts_clean
# Create accounts_clean
accounts_clean <- accounts %>%
  # Filter to remove rows with missing cust_id
  filter(!is.na(cust_id)) %>%
  # Add new col acct_amount_filled with replaced NAs
    mutate(
        acct_amount_filled = ifelse(
            is.na(acct_amount),
            inv_amount * 5,
            acct_amount
        )
    )
```

```
# Assert that cust_id has no missing vals
assert_all_are_not_na(accounts_clean$cust_id)
assert_all_are_not_na(accounts_clean$acc)
```

26 Record Linkage

Record linkage is a powerful technique used to merge multiple datasets together, used when values have typos or different spellings. In this chapter, you'll learn how to link records by calculating the similarity between strings—you'll then use your new skills to join two restaurant review datasets into one clean master dataset.

26.1 Small distance, small difference

In the video exercise, you learned that there are multiple ways to calculate how similar or different two strings are. Now you'll practice using the `stringdist` package to compute string distances using various methods. It's important to be familiar with different methods, as some methods work better on certain datasets, while others work better on other datasets.

The stringdist package has been loaded for you.

Instructions 100 XP

- Calculate the Damerau-Levenshtein distance between "las angelos" and "los angeles".

ex_021.R

```r
# Calculate Damerau-Levenshtein distance
stringdist(
    "las angelos",
    "los angeles", method = "dl"
)
# Calculate LCS distance
stringdist(
    "las angelos",
    "los angeles", method = "lcs"
)

# Calculate Jaccard distance
stringdist(
    "las angelos",
    "los angeles",
    method = "jaccard"
)
```

26.2 Fixing typos with string distance

In this chapter, one of the datasets you'll be working with, `zagat`, is a set of restaurants in New York, Los Angeles, Atlanta, San Francisco, and Las Vegas. The data is from `Zagat`, a company that collects restaurant reviews, and includes the restaurant names, addresses, phone numbers, as well as other restaurant information.

The `city` column contains the name of the `city` that the restaurant is located in. However, there are a number of typos throughout the column. Your task is to map each `city` to one of the five correctly-spelled cities contained in the `cities` data frame.

Load `dplyr`, `fuzzyjoin`, `zagat` and `cities`.

Instructions `100 XP`

ex_022.R

```
# Count the number of each city variation
# Count the number of each city variation
zagat %>%
  count(city)

# Join zagat and cities and look at results
zagat %>%
  # Left join based on stringdist using city and city_actual cols
  stringdist_left_join(
    cities,
    by = c("city" = "city_actual"), method = "dl"
  )%>%
  # Select the name, city, and city_actual cols
  select(name, city, city_actual)
```

26.3 Pair blocking

Zagat and Fodor's are both companies that gather restaurant reviews. The `zagat` and `fodors` datasets both contain information about various restaurants, including addresses, phone numbers, and cuisine types. Some restaurants appear in both datasets, but don't necessarily have the same exact name or phone number written down. In this chapter, you'll work towards figuring out which restaurants appear in both datasets.

The first step towards this goal is to generate pairs of records so that you can compare them. In this exercise, you'll first generate all possible pairs, and then use your newly-cleaned `city` column as a blocking variable.

Make `zagat` and `fodors` available.

Instructions `100 XP`

- Load the reclin package.
- Generate all possible pairs of records between the `zagat` and `fodors` datasets.

ex_023.R

```r
# Load reclin
library(reclin)
# Generate all possible pairs
pair_blocking(zagat, fodors)

# Generate pairs with same city
pair_blocking(zagat, fodors, blocking_var ="city")
```

26.4 Comparing pairs

Now that you've generated the pairs of restaurants, it's time to compare them. You can easily customize how you perform your comparisons using the by and `default_comparator` arguments. There's no right answer as to what each should be set to, so in this exercise, you'll try a couple options out.

Load `dplyr`, `reclin`, `zagat` and `fodors`.

Instructions 100 XP

- Compare pairs by name using lcs() distance.

ex_024.R

```
# Generate pairs
pair_blocking(zagat, fodors, blocking_var = "city") %>%
  # Compare pairs by name using lcs()
  compare_pairs(
    by = "name",
    default_comparator = lcs()
  )
# Generate pairs
pair_blocking(zagat, fodors, blocking_var = "city") %>%
  # Compare pairs by name, phone, addr
  compare_pairs(
    by = c("name", "phone", "addr" ),
    default_comparator = jaro_winkler()
  )
```

26.5 Putting it together

During this chapter, you've cleaned up the `city` column of `zagat` using string similarity, as well as generated and compared pairs of restaurants from `zagat` and `fodors`. The end is near - all that's left to do is score and select pairs and link the data together, and you'll be able to begin your analysis in no time!

Load `reclin`, `dplyr`, `zagat`, and `fodors`.

Instructions 100 XP

- Score the pairs of records probabilistically.
- Select the pairs that are considered matches.
- Link the two data frames together.

ex_025.R

```
```

Instructions 100 XP

ex_026.R

```
# Create pairs
pair_blocking(zagat, fodors, blocking_var = "city") %>%
  # Compare pairs
  compare_pairs(by = c("name", "addr"), default_comparator = jaro_winkler()) %>%
  # Score pairs
  score_problink() %>%
  # Select pairs
  select_n_to_m() %>%
  # Link data
  link()
```

Part V

The whole game of statistical Inference

Our goal in this part of the book is to give you a rapid overview of the main tools of data science: **importing**, **tidying**, **transforming**, and **visualizing data**, as shown in **?@fig-ds-whole-game**. We want to show you the "whole game" of data science giving you just enough of all the major pieces so that you can tackle real, if simple, data sets. The later parts of the book, will hit each of these topics in more depth, increasing the range of data science challenges that you can tackle.

27 Statistical Inference with resampling: Bootstrap and Jacknife.

27.1 Likelihood inference.

27.2 Variance analisys.

27.3 ROC Curves

28 Lienar Regression

28.1 Linear Regression

28.2 Multiple linear_regression and generalized linear regresion

29 Summary

In summary, this book has no content whatsoever.

```
[1] 2
```

References

[1] T. Hastie, R. Tibshirani, J. Friedman, The elements of statistical learning, Second, Springer, New York, 2009.

[2] W.J. Krzanowski, D.J. Hand, ROC curves for continuous data, CRC Press, Boca Raton, FL, 2009.

[3] R. Martin, A statistical inference course based on p-values, The American Statistician. 71 (2017) 128–136.

[4] P. McCullagh, J.A. Nelder, Generalized linear models, Chapman & Hall, London, 1989.

[5] B. Ratner, Statistical and machine-learning data mining:: Techniques for better predictive modeling and analysis of big data, third edition, CRC Press, 2017.

[6] D.A. Sprott, Statistical inference in science, Springer-Verlag, New York, 2000.