libro

Javier E

3/3/24

Table of contents

Preface

This is a Quarto book.

To learn more about Quarto books visit https://quarto.org/docs/books.

```
1 + 1
```

```
[1] 2
```

1 Introduction

This is a book created from markdown and executable code.

See Knuth (1984) for additional discussion of literate programming.

```
1 + 1
```

```
[1] 2
```

Part I

Introduction to R

2 Intro to basics

2.1 How it works

In the editor on the right you should type R code to solve the exercises. When you hit the 'Submit Answer' button, every line of code is interpreted and executed by R and you get a message whether or not your code was correct. The output of your R code is shown in the console in the lower right corner.

R makes use of the # sign to add comments, so that you and others can understand what the R code is about. Just like Twitter! 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, as your submission is not checked for correctness.

Instructions `100 XP`

In the editor on the right there is already some sample code. Can you see which ines are actual R code and which are comments? Add a line of code that calculates the sum of 6 and 12, and hit the 'Submit Answer' button.

E1.R

```
# Calculate 3 + 4
3 + 4

# Calculate 6 + 12
6 + 12
```

2.2 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. With this knowledge, follow the instructions to complete the exercise.

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

E2.R

```
# An addition
5 + 5

# A subtraction
5 - 5

# A multiplication
3 * 5

 # A division
(5 + 5) /2

# Exponentiation
2 ^5

# Modulo
28 %% 6
```

2.3 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

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.

E3.R

```
# Assign the value 42 to x
x <- 42

# Print out the value of the variable x
x
```

2.4 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.

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. Submit 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.

E4.R

```
# Assign the value 5 to the variable my_apples
my_apples <- 5

# Print out the value of the variable my_apples
```

```
my_apples
```

2.5 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.

E5.R

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


# Add these two variables together
my_oranges<-6

my_apples
my_oranges

# Create the variable my_fruit
my_fruit = my_apples + my_oranges
my_fruit
```

2.6 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

Submit 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.

E6.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
```

2.7 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.

- Note that R is case sensitive!

E7.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
```

2.8 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.

E8.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)
```

3 Vectors

3.1 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!

E1.R

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

3.2 Create a vector (2)

Let us focus first!

On your way from rags to riches, you will make extensive use of vectors. ectors 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:

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).

E2.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)
```

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 {.unnumbered}

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

E3.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)
```

3.3 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:

> 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:

```
      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

E4.R

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

# Roulette winnings from Monday to Friday
```

```r
roulette_vector <- c(-24, -50, 100, -350, 10)

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

# Assign days as names of roulette_vector
names(poker_vector) <-days_vector
  names(roulette_vector) <-days_vector
```

3.4 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.

E5.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
```

3.5 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:

c(1, 2, 3) + c(4, 5, 6) c(1 + 4, 2 + 5, 3 + 6) c(5, 7, 9)

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

a <- c(1, 2, 3) b <- c(4, 5, 6) c <- a + b

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.

E6.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
total_vector
```

3.6 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).

E7.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
total_daily
```

3.7 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:

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.

E8.R

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

# Total winnings overall
total_week <-(total_roulette + total_poker)

# Print out total_week
  total_poker
  total_roulette
  total_week
```

3.8 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?

E9.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
total_poker > total_roulette
```

3.9 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.

E10.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)
```

```r
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]
```

3.10 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:

poker_vector[c(1, 5)]

Instructions 100 XP

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

E11.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)]
```

3.11 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 to2:4, 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.

E12.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]
```

3.12 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[“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.

E13.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)
```

3.13 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:

c(4, 5, 6) > 5 [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.

E14.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
selection_vector
```

3.14 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.

E15.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]
```

3.15 Advanced selection

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

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.

E16.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]
```

4 Matrices

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:

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.

E1.R

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

4.1 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.

E2.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)
```

4.2 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.

E3.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)
```

4.3 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.

E4.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)
```

4.4 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.

E5.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)
```

4.5 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.

E6.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)
```

4.6 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.

E7.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)
```

4.7 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.

E8.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
mean(all_wars_matrix[,2])

# 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
mean(all_wars_matrix[1:2,2])
```

4.8 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.

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.

E9.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)
```

4.9 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.

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.

E10.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
mean(us_visitors)
```

5 Factors

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".

E1.R

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

5.1 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.

E2.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)
```

5.2 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.

E3.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
```

5.3 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.

E4.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("Female", "Male")

factor_survey_vector
```

5.4 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?

E5.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
```

```r
summary(survey_vector)

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

5.5 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.

E6.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
```

5.6 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.

E7.R

```
# Create speed_vector

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

5.7 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").

E8.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)
```

5.8 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.

E9.R

```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?
da2 >da5
```

6 Lists

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.

E1.R

```
# planets_df is pre-loaded in your workspace
order(planets_df$diameter)
# Use order() to create positions
positions <-  order(planets_df$diameter)

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

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!

E2.R

```
# Just submit the answer
#vector
x <- c(1, 2, 3)
y <-c(7,8,5)
z <- c("dragon", "quinera", "leviatan")
#matrix
matrix <- cbind(x,y)

#data frame
df <- data.frame(x,y,z)

print(df)
```

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.

E3.R

```
# Just submit the answer to start the first exercise on lists.
df <- data.frame(x = c(1, 2, 3), y = c("dragon", "grifo", "wyver"))
lista <- list(df)
```

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.

E4.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.

E5.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(vec = my_vector,
                mat = my_matrix,
                df= my_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).

E6.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.

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.

E7.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.

E68.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)
```

```r
# 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

Intermedio R

7 Conditionals and Control Flow

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?

E1.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
```

7.1 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.

E2.R

```
# Comparison of numerics
-6 * 5 + 2 >= -10 + 1

# Comparison of character strings

"raining" <= "raining dogs"
# Comparison of logicals
TRUE > FALSE
```

7.2 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?

E3.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)

# Popular days

linkedin > 15
# Quiet days

linkedin <= 5
# LinkedIn more popular than Facebook
linkedin > facebook
```

7.3 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.

E4.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
```

7.4 & 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:

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 last 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?

E5.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?
5 < last | last < 10

# Is last between 15 (exclusive) and 20 (inclusive)?
15 <= last & last < 20
```

7.5 & 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?

E6.R

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

7.6 Reverse the result: !

On top of the & and | operators, you also learned about the ! operator, which negates a logical value. To refresh your memory, here are some R expressions that use !. They all evaluate to FALSE:

!TRUE !(5 > 3) !!FALSE

What would the following set of R expressions return?

x <- 5 y <- 7 !(!(x < 4) & !!!(y > 12))

Answer the question 50XP

Possible Answers

TRUE press 1

FALSE press 2

Running this piece of code would throw an error. press 3

7.7 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, that contains 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 TRUEs 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.

E7.R

```
# li_df is pre-loaded in your workspace
li_df
# 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<- c(second < 5 |second > 25)

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

7.8 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.

E8.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!")
}
```

7.9 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:

> 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.

E9.R

```
  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!")
}
```

7.10 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.

E10.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") {print("Showing Facebook information")
  # Add code to print correct string when condition is TRUE
```

```
} 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) {print("Your number of views is
            average")
  # Add code to print correct string when condition is TRUE

} else {
  print("Try to be more visible!")
}
```

7.11 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:

> 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.

Instructions 50 XP

Possible Answers

- 2 and 4
- 1 and 4
- 1 and 3 <- respuesta

- 2 and 3

7.12 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.

E11.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)
```

8 Loops

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.

8.1 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.

E1.R

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

# Code the while loop

while (speed > 30) {
  print(paste("Slow down!"))
  speed <- speed -7
}

# Print out the speed variable
print(speed)
```

8.2 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?

8.3 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.

E2.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
  }
}
```

8.4 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.

8.5 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.

E3.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
  }
}
```

8.6 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.

8.7 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.

E4.R

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

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

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

9 Functions

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.

9.1 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.

E1.R

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

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

9.2 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.

9.3 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.

E2.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)
print(avg_fb)
```

9.4 Use a function (2)

Check the documentation on the mean() function again:

?mean

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

mean(x, ...)

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

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.

9.5 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?

E3.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
mean(linkedin + facebook)

# Calculate the trimmed mean of the sum
avg_sum_trimmed <- mean(linkedin + facebook, trim = 0.2,na.rm = FALSE,)


# Inspect both new variables
print(avg_sum)
print(avg_sum_trimmed)
```

9.6 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:

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.

E4.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)
```

9.7 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!

E5.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)
```

9.8 Required, or optional?

By now, you will probably have a good understanding of the difference between required and optional arguments. Let's refresh this difference by having one last look at the mean() function:

mean(x, trim = 0, na.rm = FALSE, ...) x is required; if you do not specify it, R will throw an error. trim and na.rm are optional arguments: they have a default value which is used if the arguments are not explicitly specified.

Which of the following statements about the read.table() function are true?

header, sep and quote are all optional arguments. row.names and fileEncoding don't have default values. read.table("myfile.txt", "-", TRUE) will throw an error. read.table("myfile.txt", sep = "-", header = TRUE) will throw an error.

Instructions 50 XP

Possible Answers

(1) and (3) Respuesta

(2) and (4)

(1), (2), and (3)

(1), (2), and (4)

9.9 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.

9.10 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.

E6.R

```
# Create a function pow_two()
pow_two <- function(arg1) {
  arg1*arg1
}

# Use the function
pow_two(12)

# Create a function sum_abs()
sum_abs <- function(arg1, arg2) {
  sum(abs(arg1), abs(arg2))
}
# Use the function
sum_abs(-2, 3)
```

9.11 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.

E7.R

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

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

9.12 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:

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:

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?

9.13 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.

E8.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)
}

pow_two(5, FALSE)
```

9.14 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:

> 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 ourse - x in this case.

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

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

Instructions 50 XP

Possible Answers

- Executing two_dice() causes an error.
- Executing res <- two_dice() makes the contents of dice1 and dice2 available outside the function.
- Whatever the way of calling the two_dice() function, R won't have access to dice1 and dice2 outside the function. Respuesta

9.15 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 function would ripple through to the variable a, outside the function. However, 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

Possible Answers

- a and b equal 7 and 6 respectively after executing this code block.
- After the first call of increment(), where a is defined, a equals 7 and count equals 5.
- In the end, count will equal 10. Respuesta
- In the last expression, the value of count was actually changed because of the explicit assignment.

9.16 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.

9.17 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.

E9.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)
interpret(facebook[2])
```

9.18 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.

9.19 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.

E10.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 == TRUE) {
    return(count)

  } else {
    return(NULL)

  }
}

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

9.20 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!

9.21 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.

E11.R

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

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

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

9.22 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:

10 Chunk 1

library(data.table) require(rjson)

11 Chunk 2

library(“data.table”) require(rjson)

12 Chunk 3

library(data.table) require(rjson, character.only = TRUE)

13 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` {.unnumbered}

Possible Answers

- Only (1)
- (1) and (2) Respuesta
- (1), (2) and (3)
- All of them are valid

14 The apply family

Before you go about solving the exercises below, have a look at the documentation of the lapply() function. The Usage section shows the following expression:

lapply(X, FUN, ...)

To put it generally, lapply takes a vector or list X, and applies the function FUN to each of its members. If FUN requires additional arguments, you pass them after you've specified X and FUN (...). The output of lapply() is a list, the same length as X, where each element is the result of applying FUN on the corresponding element of X.

Now that you are truly brushing up on your data science skills, let's revisit some of the most relevant figures in data science history. We've compiled a vector of famous mathematicians/statisticians and the year they were born. Up to you to extract some information!

14.1 Instructions `100 XP`

- Have a look at the strsplit() calls, that splits the strings in pioneers on the : sign. The result, split_math is a list of 4 character vectors: the first vector element represents the name, the second element the birth year.
- Use lapply() to convert the character vectors in split_math to lowercase letters: apply tolower() on each of the elements in split_math. Assign the result, which is a list, to a new variable split_low.
- Finally, inspect the contents of split_low with str().

E1.R

```
# The vector pioneers has already been created for you
pioneers <- c("GAUSS:1777", "BAYES:1702", "PASCAL:1623", "PEARSON:1857")

# Split names from birth year
split_math <- strsplit(pioneers, split = ":")

# Convert to lowercase strings: split_low
split_low <- lapply(split_math, tolower)
```

```
# Take a look at the structure of split_low
str(split_low)
```

14.2 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.

14.3 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.

E2.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)
```

14.4 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.

14.5 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.

E3.R

```
# split_low has been created for you
split_low <- lapply(split, tolower)
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] })
```

14.6 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:

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.

14.7 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.

E4.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
names <- lapply(split_low, select_el, index = 1)
years <- lapply(split_low, select_el, index = 2)
```

14.8 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:

> 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

> 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!

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

Instructions 50 XP

Possible Answers

- list(NULL, NULL, "1623", "1857")
- list("gauss", "bayes", NULL, NULL)
- list("1777", "1702", NULL, NULL) Respuesta
- list("1777", "1702")

14.9 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 (…):

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.

14.10 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.

E5.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)
```

14.11 sapply with your own function

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

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.

14.12 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.

E6.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)
```

14.13 apply 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!

14.14 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.

E7.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)
```

14.15 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.

14.16 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.

E8.R

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

# Definition of below_zero()
```

```r
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_l)
```

14.17 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.

14.18 Instructions 100 XP

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

E9.R

```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)
```

14.19 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.

Instructions 50 XP

Possible Answers

- (1) and (3)
- (2) and (3) Respuesta
- (1) and (4)
- (2), (3) and (4)

14.20 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.

14.21 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.

E10.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))
```

14.22 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.

14.23 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.

E11.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))
```

14.24 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!).

14.25 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).

E12.R

```
# temp is already defined in the workspace

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

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

15 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.

15.1 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.

E1.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)))
```

15.2 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.

15.3 Instructions 100 XP

Fix the error by including code on the last line. Remember: you want to call mean() only once!

E2.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))
mean(abs(vec2))

# ?mean()
# ?rev()
```

15.4 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.

15.5 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.

E3.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 <- as.vector(linkedin)
fb_vec <- as.vector(facebook)

# Append fb_vec to li_vec: social_vec
social_vec <-unlist(append(li_vec, fb_vec))

# Sort social_vec
sort(social_vec,decreasing = TRUE)
```

15.6 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!

15.7 Instructions 100 XP

Correct the expression. Make sure that your fix still uses the functions rep() and seq().

E4.R

```
# Fix me
rep(seq(1, 7, by = 2), times = 7)
```

15.8 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?

15.9 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.

E5.R

```
# Create first sequence: seq1
seq1<- seq(from = 1, to = 500, by = 3)

# Create second sequence: seq2
seq2<- seq(from = 1200, to = 900, by = -7)

# Calculate total sum of the sequences
sum(seq1, seq2)
```

15.10 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!

15.11 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".

E6.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"
grepl( "edu",  emails)

# Use grep() to match for "edu", save result to hits
hits <- grep("edu", emails)


# Subset emails using hits
emails[hits]
```

15.12 repl & 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.

15.13 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.

E7.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 addresses more robustly
grepl(pattern="@.*\\.edu$",emails)

# Use grep() to match for .edu addresses more robustly, save result to hits
hits <- grep("@.*\\.edu$", emails)

# Subset emails using hits
emails[hits]
```

15.14 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!

15.15 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.

E8.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)
```

15.16 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]+.

  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". Respuesta

15.17 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.

15.18 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.

E9.R

```
# Get the current date: today
today <- Sys.Date()
today

# See what today looks like under the hood
unclass(today)
```

```
# Get the current time: now
now <- Sys.Date()
now

# See what now looks like under the hood
unclass(now)
```

15.19 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!")

15.20 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.

E10.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 %b, '%d")
date3 <- as.Date(str3, format = "%d/%B/%Y")

# Convert dates to formatted strings
format(date1, "%A")
format(date2, "%d")
format(date3, "%b %Y")
```

15.21 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.

15.22 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!

E11.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)

# Convert times to formatted strings
format(time1, "%M")
format(time2, "%I:%M %p")
```

15.23 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.

15.24 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.

E12.R

```
# day1, day2, day3, day4 and day5 are already available in the workspace

# Difference between last and first pizza day
day5 - day1

# Create vector pizza
pizza <- c(day1, day2, day3, day4, day5)

# Create differences between consecutive pizza days: day_diff
day_diff <- diff(pizza, lag = 1, differences = 1)
day_diff

# Average period between two consecutive pizza days
print(mean(day_diff))
```

15.25 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:

> now <- Sys.time() now + 3600 # add an hour now - 3600 * 24 # subtract a day

Adding or subtracting time objects is also straightforward:

> 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.

15.26 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.

E13.R

```
# login and logout are already defined in the workspace
# Calculate the difference between login and logout: time_online
time_online <- logout- login

# Inspect the variable time_online
time_online

# Calculate the total time online
sum(time_online)

# Calculate the average time online
mean(time_online)
```

15.27 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.

15.28 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.

E14.R

```
# Convert astro to vector of Date objects: astro_dates
astro_dates <-as.Date(astro, format = "%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))
```

Part III

Introduction to Writing Functions in R

16 How to Write a Function

One way to make your code more readable is to be careful about the order you pass arguments when you call functions, and whether you pass the arguments by position or by name.

gold_medals, a numeric vector of the number of gold medals won by each country in the 2016 Summer Olympics, is provided.

For convenience, the arguments of median() and rank() are displayed using args(). Setting rank()'s na.last argument to "keep" means "keep the rank of NA values as NA".

Best practice for calling functions is to include them in the order shown by args(), and to only name rare arguments.

Instructions `100 XP`

- The final line calculates the median number of gold medals each country won.
- Rewrite the call to median(), following best practices.

E1.R

```
# Look at the gold medals data
gold_medals

# Note the arguments to median()
args(median)

# Rewrite this function call, following best practices
median(gold_medals, na.rm = TRUE)
```

16.1 The benefits of writing functions

There are lots of great reasons that you should write your own functions.

Which of these is not one of them?

Answer the question 50XP

Possible Answers

- You can type less code, saving effort and making your analyses more readable. Respuesta
- You make less "copy and paste"-related errors.
- You can reuse your code from project to project.
- You can make your code harder to read, potentially improving your job security because only you can maintain it.

16.2 Your first function: tossing a coin

Time to write your first function! It's a really good idea when writing functions to start simple. You can always make a function more complicated later if it's really necessary, so let's not worry about arguments for now.

Instructions 100 XP

- Simulate a single coin toss by using sample() to sample from coin_sides once.
- Write a template for your function, naming it toss_coin. The function should take no arguments. Don't include the body of the function yet.
- Copy your script, and paste it into the function body.
- Call your function.

E2.R

```
coin_sides <- c("head", "tail")

# Sample from coin_sides once
sample(coin_sides,1)

# Write a template for your function, toss_coin()
toss_coin <- function() {


  # (Leave the contents of the body for later)
# Add punctuation to finish the body
}
```

```r
# Your script, from a previous step
coin_sides <- c("head", "tail")

# Paste your script into the function body
toss_coin <- function() {
  sample(coin_sides, 1)

}

# Your functions, from previous steps
toss_coin <- function() {
  coin_sides <- c("head", "tail")
  sample(coin_sides, 1)
}

# Call your function
toss_coin()
```

16.3 Inputs to functions

Most functions require some sort of input to determine what to compute. The inputs to functions are called arguments. You specify them inside the parentheses after the word "function."

As mentioned in the video, the following exercises assume that you are using sample() to do random sampling.

Instructions 100 XP

- Sample from coin_sides n_flips times with replacement.
- Update the definition of toss_coin() to accept a single argument, n_flips. The function should sample coin_sides n_flips times with replacement. Remember to change the signature and the body.
- Generate 10 coin flips.

E3.R

```r
coin_sides <- c("head", "tail")
n_flips <- 10

# Sample from coin_sides n_flips times with replacement
sample(coin_sides,n_flips,replace = TRUE)

# Update the function to return n coin tosses
toss_coin <- function(n_flips) {
  coin_sides <- c("head", "tail")
  sample(coin_sides, n_flips,replace = TRUE)
}

# Generate 10 coin tosses
toss_coin(10)
```

16.4 Multiple inputs to functions

If a function should have more than one argument, list them in the function signature, separated by commas.

To solve this exercise, you need to know how to specify sampling weights to sample(). Set the prob argument to a numeric vector with the same length as x. Each value of prob is the probability of sampling the corresponding element of x, so their values add up to one. In the following example, each sample has a 20% chance of "bat", a 30% chance of "cat" and a 50% chance of "rat".

sample(c("bat", "cat", "rat"), 10, replace = TRUE, prob = c(0.2, 0.3, 0.5))

##Instructions `100 XP` {.unnumbered}

- Bias the coin by weighting the sampling. Specify the prob argument so that heads are sampled with probability p_head (and tails are sampled with probability 1 - p_head).
- Update the definition of toss_coin() so it accepts an argument, p_head, and weights the samples using the code you wrote in the previous step.
- Generate 10 coin tosses with an 80% chance of each head.

E4.R

```r
coin_sides <- c("head", "tail")
n_flips <- 10
p_head <- 0.8
```

```r
# Define a vector of weights
weights <- c(p_head, 1 - p_head)

# Update so that heads are sampled with prob p_head
sample(coin_sides, n_flips, replace = TRUE, prob = weights)

# Update the function so heads have probability p_head
toss_coin <- function(n_flips,p_head) {
  coin_sides <- c("head", "tail")
  # Define a vector of weights
  weights <- c(p_head,1-p_head)
  # Modify the sampling to be weighted
  sample(coin_sides, n_flips, replace = TRUE,prob=weights)
}

# Generate 10 coin tosses
toss_coin(10,0.8)
```

16.5 Renaming GLM

R's generalized linear regression function, glm(), suffers the same usability problems as lm(): its name is an acronym, and its formula and data arguments are in the wrong order.

To solve this exercise, you need to know two things about generalized linear regression:

glm() formulas are specified like lm() formulas: response is on the left, and explanatory variables are added on the right. To model count data, set glm()'s family argument to poisson, making it a Poisson regression. Here you'll use data on the number of yearly visits to Snake River at Jackson Hole, Wyoming, snake_river_visits.

Instructions 100 XP

- Run a generalized linear regression by calling glm(). Model n_visits vs. gender, income, and travel on the snake_river_visits dataset, setting the family to poisson.
- Define a function, run_poisson_regression(), to run a Poisson regression. This should take two arguments: data and formula, and call glm(), passing those arguments and setting family to poisson.
- Recreate the Poisson regression model from the first step, this time by calling your run_poisson_regression() function.

E5.R

```
# Run a generalized linear regression
glm(
  # Model no. of visits vs. gender, income, travel
  n_visits ~ gender + income + travel,
  # Use the snake_river_visits dataset
  data = snake_river_visits,
  # Make it a Poisson regression
  family = poisson
)

# Write a function to run a Poisson regression
run_poisson_regression <- function(data, formula) {
    glm(formula, data, family = poisson)
}

# From previous step
run_poisson_regression <- function(data, formula) {
  glm(formula, data, family = poisson)
}

# Re-run the Poisson regression, using your function
model <- snake_river_visits %>%
  run_poisson_regression(n_visits ~ gender + income + travel)

# Run this to see the predictions
snake_river_explanatory %>%
  mutate(predicted_n_visits = predict(model, ., type = "response"))%>%
  arrange(desc(predicted_n_visits))
```

16.6 Numeric defaults

cut_by_quantile() converts a numeric vector into a categorical variable where quantiles define the cut points. This is a useful function, but at the moment you have to specify five arguments to make it work. This is too much thinking and typing.

By specifying default arguments, you can make it easier to use. Let's start with n, which specifies how many categories to cut x into.

A numeric vector of the number of visits to Snake River is provided as n_visits.

Instructions 100 XP

Update the definition of cut_by_quantile() so that the n argument defaults to 5. Remove the n argument from the call to cut_by_quantile().

E6.R

```
# Set the default for n to 5
cut_by_quantile <- function(x, n=5, na.rm, labels, interval_type) {
  probs <- seq(0, 1, length.out = n + 1)
  qtiles <- quantile(x, probs, na.rm = na.rm, names = FALSE)
  right <- switch(interval_type, "(lo, hi]" = TRUE, "[lo, hi)" = FALSE)
  cut(x, qtiles, labels = labels, right = right, include.lowest = TRUE)
}

# Remove the n argument from the call
cut_by_quantile(
  n_visits,
  na.rm = FALSE,
  labels = c("very low", "low", "medium", "high", "very high"),
  interval_type = "(lo, hi]"
)
formals(cut_by_quantile)
```

16.7 Logical defaults

cut_by_quantile() is now slightly easier to use, but you still always have to specify the na.rm argument. This removes missing values—it behaves the same as the na.rm argument to mean() or sd().

Where functions have an argument for removing missing values, the best practice is to not remove them by default (in case you hadn't spotted that you had missing values). That means that the default for na.rm should be FALSE.

Instructions 100 XP

Update the definition of cut_by_quantile() so that the na.rm argument defaults to FALSE. Remove the na.rm argument from the call to cut_by_quantile().

E7.R

```r
# Set the default for na.rm to FALSE
cut_by_quantile <- function(x, n = 5, na.rm = FALSE, labels, interval_type) {
  probs <- seq(0, 1, length.out = n + 1)
  qtiles <- quantile(x, probs, na.rm = na.rm, names = FALSE)
  right <- switch(interval_type, "(lo, hi]" = TRUE, "[lo, hi)" = FALSE)
  cut(x, qtiles, labels = labels, right = right, include.lowest = TRUE)
}

# Remove the na.rm argument from the call
cut_by_quantile(
  n_visits,
  labels = c("very low", "low", "medium", "high", "very high"),
  interval_type = "(lo, hi]"
)
```

17 Return Values and Scope

Sometimes, you don't need to run through the whole body of a function to get the answer. In that case you can return early from that function using return().

To check if x is divisible by n, you can use is_divisible_by(x, n) from assertive.

Alternatively, use the modulo operator, %%. x %% n gives the remainder when dividing x by n, so x %% n == 0 determines whether x is divisible by n. Try 1:10 %% 3 == 0 in the console.

To solve this exercise, you need to know that a leap year is every 400th year (like the year 2000) or every 4th year that isn't a century (like 1904 but not 1900 or 1905).

assertive is loaded.

17.1 Instructions `100 XP`

Complete the definition of is_leap_year(), checking for the cases of year being divisible by 400, then 100, then 4, returning early from the function in each case.

E1.R

```
is_leap_year <- function(year) {
  # If year is div. by 400 return TRUE
  if(is_divisible_by(year,400)) {
    return(TRUE)
  }
  # If year is div. by 100 return FALSE
  if(is_divisible_by(year,100)) {
    return(FALSE)
  }
  # If year is div. by 4 return TRUE
  if(is_divisible_by(year,4)) {
    return(TRUE)
  }
```

```
  # Otherwise return FALSE
  FALSE
}
```

17.2 Returning invisibly

When the main purpose of a function is to generate output, like drawing a plot or printing something in the console, you may not want a return value to be rinted as well. In that case, the value should be invisibly returned.

The base R plot function returns NULL, since its main purpose is to draw a plot. This isn't helpful if you want to use it in piped code: instead it should invisibly return the plot data to be piped on to the next step.

Recall that plot() has a formula interface: instead of giving it vectors for x and y, you can specify a formula describing which columns of a data frame go on the x and y axes, and a data argument for the data frame. Note that just like lm(), the arguments are the wrong way round because the detail argument, formula, comes before the data argument.

> plot(y ~ x, data = data)

Instructions `100 XP`

- Use the cars dataset and the formula interface to plot(), draw a scatter plot of dist versus speed.
- Give pipeable_plot() data and formula arguments (in that order) and make it draw the plot, then invisibly return data.
- Draw the scatter plot of dist vs. speed again by calling pipeable_plot()

E2.R

```
# Using cars, draw a scatter plot of dist vs. speed
plt_dist_vs_speed <- plot(dist ~ speed, data = cars)

# Oh no! The plot object is NULL
plt_dist_vs_speed

# Define a pipeable plot fn with data and formula args
pipeable_plot <- function(data, formula) {
```

```
  # Call plot() with the formula interface
  plot(formula, data)
  # Invisibly return the input dataset
  invisible(data)
}

# Draw the scatter plot of dist vs. speed again
plt_dist_vs_speed <- cars %>%
  pipeable_plot(dist ~ speed)
```

17.3 Returning many things

Functions can only return one value. If you want to return multiple things, then you can store them all in a list.

If users want to have the list items as separate variables, they can assign each list element to its own variable using zeallot's multi-assignment operator, %<-%.

glance(), tidy(), and augment() each take the model object as their only rgument.

The Poisson regression model of Snake River visits is available as model. broom and zeallot are loaded.

Instructions 100 XP

- Examine the structure of model.
- Use broom functions on model to create a list containing the model-, coefficient-, and observation-level parts of model.
- Wrap the code into a function, groom_model(), that accepts model as its only argument.
- Call groom_model() on model, multi-assigning the result to three variables at once: mdl, cff, and obs.

E3.R

```
# Look at the structure of model (it's a mess!)
str(model)

# Use broom tools to get a list of 3 data frames
list(
```

```
  # Get model-level values
  model = glance(model),
  # Get coefficient-level values
  coefficients = tidy(model),
  # Get observation-level values
  observations = augment(model)
)

# Wrap this code into a function, groom_model
groom_model <- function(model){
  list(
    model = glance(model),
    coefficients = tidy(model),
    observations = augment(model)
  )
}

groom_model(model)

# From previous step
groom_model <- function(model) {
  list(
    model = glance(model),
    coefficients = tidy(model),
    observations = augment(model)
  )
}

# Call groom_model on model, assigning to 3 variables
c(mdl, cff, obs) %<-% groom_model(model)
#c(var1, var2, var3) %<-% fn(args)

# See these individual variables
mdl; cff; obs
```

17.4 Returning metadata

Sometimes you want to return multiple things from a function, but you want the result to have a particular class (for example, a data frame or a numeric vector), so returning a list isn't appropriate. This is common when you have a result plus metadata about the result.

(Metadata is "data about the data". For example, it could be the file a dataset was loaded from, or the username of the person who created the variable, or the number of iterations for an algorithm to converge.)

In that case, you can store the metadata in attributes. Recall the syntax for assigning attributes is as follows.

> attr(object, "attribute_name") <- attribute_value

Instructions 100 XP

- Update pipeable_plot() so the result has an attribute named "formula" with the value of formula.
- plt_dist_vs_speed, that you previously created, is shown. Examine its updated structure.

E4.R

```
pipeable_plot <- function(data, formula) {
  plot(formula, data)
  # Add a "formula" attribute to data
  attr(data, "formula") <- formula
  invisible(data)
}

# From previous exercise
plt_dist_vs_speed <- cars %>%
  pipeable_plot(dist ~ speed)

# Examine the structure of the result
str(plt_dist_vs_speed)
```

17.5 Creating and exploring environments

Environments are used to store other variables. Mostly, you can think of them as lists, but there's an important extra property that is relevant to writing functions. Every environment has a parent environment (except the empty environment, at the root of the environment tree). This determines which variables R know about at different places in your code.

Facts about the Republic of South Africa are contained in capitals, national_parks, and population.

Instructions 100 XP

- Create rsa_lst, a named list from capitals, national_parks, and population. Use those values as the names.
- List the structure of each element of rsa_lst using ls.str().
- Convert the list to an environment, rsa_env, using list2env().
- List the structure of each element of rsa_env
- Find the parent environment of rsa_env and print its name.

E5.R

```
# Add capitals, national_parks, & population to a named list
rsa_lst <- list(
  capitals = capitals,
  national_parks = national_parks,
  population = population
)

# List the structure of each element of rsa_lst
ls.str(rsa_lst)

# From previous step
rsa_lst <- list(
  capitals = capitals,
  national_parks = national_parks,
  population = population
)

# Convert the list to an environment
rsa_env <- list2env(rsa_lst)

# List the structure of each variable
ls.str(rsa_env)

# From previous steps
rsa_lst <- list(
  capitals = capitals,
  national_parks = national_parks,
  population = population
)
```

```
rsa_env <- list2env(rsa_lst)

# Find the parent environment of rsa_env
parent <- parent.env(rsa_env)
environmentName(parent)

# Print its name
print(environmentName)
```

17.6 Do variables exist?

If R cannot find a variable in the current environment, it will look in the parent environment, then the grandparent environment, and so on until it finds it.

rsa_env has been modified so it includes capitals and national_parks, but not population.

Instructions 100 XP

- Check if population exists in rsa_env, using default inheritance rules.
- Check if population exists in rsa_env, ignoring inheritance.

E6.R

```
# Compare the contents of the global environment and rsa_env
ls.str(globalenv())
ls.str(rsa_env)

# Does population exist in rsa_env?
exists("population", envir = rsa_env)

# Does population exist in rsa_env, ignoring inheritance?
exists("population", envir = rsa_env,inherits = FALSE)
```

17.7 Variable precedence 1

Consider this code, run in a fresh R session.

x_plus_y <- function(x) { y <- 3 x + y } y <- 7

If you now call x_plus_y(5), what is the result?

17.8 Answer the question `50XP`

Possible Answers

8. Respuesta
9.
10.

An error is thrown.

17.9 Variable precedence 2

Consider this (slightly different) code, run in a fresh R session.

x_plus_y <- function(x) { x <- 6 y <- 3 x + y } y <- 7 If you now call x_plus_y(5), what is the result?

17.10 Answer the question `50XP`

Possible Answers

8.
9. respuesta
10.
11. An error is thrown.

18 Return Values and Scope

Sometimes, you don't need to run through the whole body of a function to get the answer. In that case you can return early from that function using return().

To check if x is divisible by n, you can use is_divisible_by(x, n) from assertive.

Alternatively, use the modulo operator, %%. x %% n gives the remainder when dividing x by n, so x %% n == 0 determines whether x is divisible by n. Try 1:10 %% 3 == 0 in the console.

To solve this exercise, you need to know that a leap year is every 400th year (like the year 2000) or every 4th year that isn't a century (like 1904 but not 1900 or 1905).

assertive is loaded.

18.1 Instructions `100 XP`

Complete the definition of is_leap_year(), checking for the cases of year being divisible by 400, then 100, then 4, returning early from the function in each case.

E1.R

```
is_leap_year <- function(year) {
  # If year is div. by 400 return TRUE
  if(is_divisible_by(year,400)) {
    return(TRUE)
  }
  # If year is div. by 100 return FALSE
  if(is_divisible_by(year,100)) {
    return(FALSE)
  }
  # If year is div. by 4 return TRUE
  if(is_divisible_by(year,4)) {
    return(TRUE)
  }
```

```
  # Otherwise return FALSE
  FALSE
}
```

18.2 Returning invisibly

When the main purpose of a function is to generate output, like drawing a plot or printing something in the console, you may not want a return value to be rinted as well. In that case, the value should be invisibly returned.

The base R plot function returns NULL, since its main purpose is to draw a plot. This isn't helpful if you want to use it in piped code: instead it should invisibly return the plot data to be piped on to the next step.

Recall that plot() has a formula interface: instead of giving it vectors for x and y, you can specify a formula describing which columns of a data frame go on the x and y axes, and a data argument for the data frame. Note that just like lm(), the arguments are the wrong way round because the detail argument, formula, comes before the data argument.

> plot(y ~ x, data = data)

Instructions 100 XP

- Use the cars dataset and the formula interface to plot(), draw a scatter plot of dist versus speed.
- Give pipeable_plot() data and formula arguments (in that order) and make it draw the plot, then invisibly return data.
- Draw the scatter plot of dist vs. speed again by calling pipeable_plot()

E2.R

```
# Using cars, draw a scatter plot of dist vs. speed
plt_dist_vs_speed <- plot(dist ~ speed, data = cars)

# Oh no! The plot object is NULL
plt_dist_vs_speed

# Define a pipeable plot fn with data and formula args
pipeable_plot <- function(data, formula) {
```

```
  # Call plot() with the formula interface
  plot(formula, data)
  # Invisibly return the input dataset
  invisible(data)
}

# Draw the scatter plot of dist vs. speed again
plt_dist_vs_speed <- cars %>%
  pipeable_plot(dist ~ speed)
```

18.3 Returning many things

Functions can only return one value. If you want to return multiple things, then you can store them all in a list.

If users want to have the list items as separate variables, they can assign each list element to its own variable using zeallot's multi-assignment operator, %<-%.

glance(), tidy(), and augment() each take the model object as their only rgument.

The Poisson regression model of Snake River visits is available as model. broom and zeallot are loaded.

Instructions 100 XP

- Examine the structure of model.
- Use broom functions on model to create a list containing the model-, coefficient-, and observation-level parts of model.
- Wrap the code into a function, groom_model(), that accepts model as its only argument.
- Call groom_model() on model, multi-assigning the result to three variables at once: mdl, cff, and obs.

E3.R

```
# Look at the structure of model (it's a mess!)
str(model)

# Use broom tools to get a list of 3 data frames
list(
```

```
  # Get model-level values
  model = glance(model),
  # Get coefficient-level values
  coefficients = tidy(model),
  # Get observation-level values
  observations = augment(model)
)

# Wrap this code into a function, groom_model
groom_model <- function(model){
  list(
    model = glance(model),
    coefficients = tidy(model),
    observations = augment(model)
  )
}

groom_model(model)

# From previous step
groom_model <- function(model) {
  list(
    model = glance(model),
    coefficients = tidy(model),
    observations = augment(model)
  )
}

# Call groom_model on model, assigning to 3 variables
c(mdl, cff, obs) %<-% groom_model(model)
#c(var1, var2, var3) %<-% fn(args)

# See these individual variables
mdl; cff; obs
```

18.4 Returning metadata

Sometimes you want to return multiple things from a function, but you want the result to have a particular class (for example, a data frame or a numeric vector), so returning a list isn't appropriate. This is common when you have a result plus metadata about the result.

(Metadata is "data about the data". For example, it could be the file a dataset was loaded from, or the username of the person who created the variable, or the number of iterations for an algorithm to converge.)

In that case, you can store the metadata in attributes. Recall the syntax for assigning attributes is as follows.

> attr(object, "attribute_name") <- attribute_value

Instructions 100 XP

- Update pipeable_plot() so the result has an attribute named "formula" with the value of formula.
- plt_dist_vs_speed, that you previously created, is shown. Examine its updated structure.

E4.R

```
pipeable_plot <- function(data, formula) {
  plot(formula, data)
  # Add a "formula" attribute to data
  attr(data, "formula") <- formula
  invisible(data)
}

# From previous exercise
plt_dist_vs_speed <- cars %>%
  pipeable_plot(dist ~ speed)

# Examine the structure of the result
str(plt_dist_vs_speed)
```

18.5 Creating and exploring environments

Environments are used to store other variables. Mostly, you can think of them as lists, but there's an important extra property that is relevant to writing functions. Every environment has a parent environment (except the empty environment, at the root of the environment tree). This determines which variables R know about at different places in your code.

Facts about the Republic of South Africa are contained in capitals, national_parks, and population.

Instructions 100 XP

- Create rsa_lst, a named list from capitals, national_parks, and population. Use those values as the names.
- List the structure of each element of rsa_lst using ls.str().
- Convert the list to an environment, rsa_env, using list2env().
- List the structure of each element of rsa_env
- Find the parent environment of rsa_env and print its name.

E5.R

```
# Add capitals, national_parks, & population to a named list
rsa_lst <- list(
  capitals = capitals,
  national_parks = national_parks,
  population = population
)

# List the structure of each element of rsa_lst
ls.str(rsa_lst)

# From previous step
rsa_lst <- list(
  capitals = capitals,
  national_parks = national_parks,
  population = population
)

# Convert the list to an environment
rsa_env <- list2env(rsa_lst)

# List the structure of each variable
ls.str(rsa_env)

# From previous steps
rsa_lst <- list(
  capitals = capitals,
  national_parks = national_parks,
  population = population
)
```

```
rsa_env <- list2env(rsa_lst)

# Find the parent environment of rsa_env
parent <- parent.env(rsa_env)
environmentName(parent)

# Print its name
print(environmentName)
```

18.6 Do variables exist?

If R cannot find a variable in the current environment, it will look in the parent environment, then the grandparent environment, and so on until it finds it.

rsa_env has been modified so it includes capitals and national_parks, but not population.

Instructions 100 XP

- Check if population exists in rsa_env, using default inheritance rules.
- Check if population exists in rsa_env, ignoring inheritance.

E6.R

```
# Compare the contents of the global environment and rsa_env
ls.str(globalenv())
ls.str(rsa_env)

# Does population exist in rsa_env?
exists("population", envir = rsa_env)

# Does population exist in rsa_env, ignoring inheritance?
exists("population", envir = rsa_env,inherits = FALSE)
```

18.7 Variable precedence 1

Consider this code, run in a fresh R session.

x_plus_y <- function(x) { y <- 3 x + y } y <- 7

If you now call x_plus_y(5), what is the result?

18.8 Answer the question `50XP`

Possible Answers

8. Respuesta
9.
10.

An error is thrown.

18.9 Variable precedence 2

Consider this (slightly different) code, run in a fresh R session.

x_plus_y <- function(x) { x <- 6 y <- 3 x + y } y <- 7 If you now call x_plus_y(5), what is the result?

18.10 Answer the question `50XP`

Possible Answers

8.
9. respuesta
10.
11. An error is thrown.

19 Case Study on Grain Yields

In this chapter, you'll be working with grain yield data from the United States Department of Agriculture, National Agricultural Statistics Service. Unfortunately, they report all areas in acres. So, the first thing you need to do is write some utility functions to convert areas in acres to areas in hectares.

To solve this exercise, you need to know the following:

There are 4840 square yards in an acre. There are 36 inches in a yard and one inch is 0.0254 meters. There are 10000 square meters in a hectare.

19.1 Instructions `100 XP`

- Write a function, acres_to_sq_yards(), to convert areas in acres to areas in square yards. This should take a single argument, acres.
- Write a function, yards_to_meters(), to convert distances in yards to distances in meters. This should take a single argument, yards.
- Write a function, sq_meters_to_hectares(), to convert areas in square meters to areas in hectares. This should take a single argument, sq_meters.

E1.R

```
# Write a function to convert acres to sq. yards
acres_to_sq_yards <- function(x) {
  x * 4840
}


# Write a function to convert yards to meters
yards_to_meters <- function(x) {
  x * 36*0.0254
}

# Write a function to convert sq. meters to hectares
```

```
sq_meters_to_hectares <- function(sq_meters) {
  (sq_meters)/10000
}
```

19.2 Converting areas to metric 2

You're almost there with creating a function to convert acres to hectares. You need another utility function to deal with getting from square yards to square meters. Then, you can bring everything together to write the overall acres-to-hectares conversion function. Finally, in the next exercise you'll be calculating area conversions in the denominator of a ratio, so you'll need a harmonic acre-to-hectare conversion function.

Free hints: magrittr's raise_to_power() will be useful here. The last step is similar to Chapter 2's Harmonic Mean.

The three utility functions from the last exercise (acres_to_sq_yards(), yards_to_meters(), and sq_meters_to_hectares()) are available, as is your get_reciprocal() from Chapter 2. magrittr is loaded.

Instructions 100 XP

- Write a function to convert distance in square yards to square meters. It should take the square root of the input, then convert yards to meters, then square the result.
- Write a function to convert areas in acres to hectares. The function should convert the input from acres to square yards, then to square meters, then to hectares.
- Write a function to harmonically convert areas in acres to hectares. The function should get the reciprocal of the input, then convert from acres to hectares, then get the reciprocal again.

E2.R

```
# Write a function to convert sq. yards to sq. meters
sq_yards_to_sq_meters <- function(sq_yards) {
  sq_yards %>%
    # Take the square root
    sqrt() %>%
    # Convert yards to meters
    yards_to_meters() %>%
    # Square it
```

```
    raise_to_power(2)
     }

# Load the function from the previous step
load_step2()

# Write a function to convert acres to hectares
acres_to_hectares <- function(acres) {
  acres %>%
    # Convert acres to sq yards
    acres_to_sq_yards() %>%
    # Convert sq yards to sq meters
    sq_yards_to_sq_meters() %>%
    # Convert sq meters to hectares
    sq_meters_to_hectares()
}

# Load the functions from the previous steps
load_step3()

# Define a harmonic acres to hectares function
harmonic_acres_to_hectares <- function(acres) {
  acres %>%
    # Get the reciprocal
    get_reciprocal() %>%
    # Convert acres to hectares
    acres_to_hectares() %>%
    # Get the reciprocal again
    get_reciprocal()
}
```

19.3 Converting yields to metric

The yields in the NASS corn data are also given in US units, namely bushels per acre. You'll need to write some more utility functions to convert this unit to the metric unit of kg per hectare.

Bushels historically meant a volume of 8 gallons, but in the context of grain, they are now defined as masses. This mass differs for each grain! To solve this exercise, you need to know these facts.

One pound (lb) is 0.45359237 kilograms (kg). One bushel is 48 lbs of barley, 56 lbs of corn, or 60 lbs of wheat. magrittr is loaded.

Instructions 100 XP

- Write a function to convert masses in lb to kg. This should take a single argument, lbs.
- Write a function to convert masses in bushels to lbs. This should take two arguments, bushels and crop. It should define a lookup vector of scale factors for each crop (barley, corn, wheat), extract the scale factor for the crop, then multiply this by the number of bushels.
- Write a function to convert masses in bushels to kgs. This should take two arguments, bushels and crop. It should convert the mass in bushels to lbs then to kgs.
- Write a function to convert yields in bushels/acre to kg/ha. The arguments should be bushels_per_acre and crop. Three choices of crop should be allowed: "barley", "corn", and "wheat". It should match the crop argument, then convert bushels to kgs, then convert harmonic acres to hectares.

E3.R

```
# Write a function to convert lb to kg
lbs_to_kgs <- function(lbs) {lbs * 0.45359237}

# Write a function to convert bushels to lbs
bushels_to_lbs <- function(bushels, crop) {
  # Define a lookup table of scale factors
  c(barley = 48, corn = 56, wheat = 60) %>%
    # Extract the value for the crop
    extract(crop) %>%
    # Multiply by the no. of bushels
    multiply_by(bushels)
}

# Load fns defined in previous steps
load_step3()

# Write a function to convert bushels to kg
bushels_to_kgs <- function(bushels, crop) {
  bushels %>%
    # Convert bushels to lbs for this crop
```

```
    bushels_to_lbs(crop) %>%
    # Convert lbs to kgs
    lbs_to_kgs()
}

# Load fns defined in previous steps
load_step4()

# Write a function to convert bushels/acre to kg/ha
bushels_per_acre_to_kgs_per_hectare <- function(bushels_per_acre,
crop = c("barley", "corn", "wheat")) {
  # Match the crop argument
  crop <- match.arg(crop)
  bushels_per_acre %>%
    # Convert bushels to kgs for this crop
    bushels_to_kgs(crop) %>%
    # Convert harmonic acres to ha
    harmonic_acres_to_hectares()
}
```

19.4 Applying the unit conversion

Now that you've written some functions, it's time to apply them! The NASS corn dataset is available, and you can fortify it (jargon for "adding new columns") with metrics areas and yields.

This fortification process can also be turned into a function, so you'll define a function for this, and test it on the NASS wheat dataset.

Instructions `100 XP`

- Look at the corn dataset. Add two columns: farmed_area_ha should be farmed_area_acres converted to hectares; yield_kg_per_ha should be yield_bushels_per_acre converted to kilograms per hectare.
- Wrap the mutation code into a function called fortify_with_metric_units that takes two arguments, data and crop with no defaults. (In the function body, swap corn for the data argument and pass the function's local crop variable to the crop argument.)
- Use fortify_with_metric_units() on the wheat dataset.

E4.R

```
# View the corn dataset
glimpse(corn)

corn %>%
  # Add some columns
  mutate(
    # Convert farmed area from acres to ha
    farmed_area_ha = acres_to_hectares(farmed_area_acres),
    # Convert yield from bushels/acre to kg/ha
    yield_kg_per_ha = bushels_per_acre_to_kgs_per_hectare(
      yield_bushels_per_acre,
      crop = "corn"
    )
  )

# Wrap this code into a function
fortify_with_metric_units <- function(data, crop) {
  data %>%
    mutate(
      farmed_area_ha = acres_to_hectares(farmed_area_acres),
      yield_kg_per_ha = bushels_per_acre_to_kgs_per_hectare(
        yield_bushels_per_acre,
        crop = crop
      )
    )
}

# Try it on the wheat dataset
fortify_with_metric_units(wheat, crop = "wheat")
```

19.5 Plotting yields over time

Now that the units have been dealt with, it's time to explore the datasets. An obvious question to ask about each crop is, "how do the yields change over time in each US state?" Let's draw a line plot to find out!

ggplot2 is loaded, and corn and wheat datasets are available with metric units.

Instructions 100 XP

- Using the corn dataset, plot yield_kg_per_ha versus year. Add a line layer grouped by state and a smooth trend layer.
- Turn the plotting code into a function, plot_yield_vs_year(). This should accept a single argument, data.

E5.R

```
# Using corn, plot yield (kg/ha) vs. year
ggplot(corn, aes(year, yield_kg_per_ha)) +
  # Add a line layer, grouped by state
  geom_line(aes(group = state)) +
  # Add a smooth trend layer
  geom_smooth()

# Wrap this plotting code into a function
plot_yield_vs_year <- function(data) {
  ggplot(data, aes(year, yield_kg_per_ha)) +
    geom_line(aes(group = state)) +
    geom_smooth()
}

# Test it on the wheat dataset
plot_yield_vs_year(wheat)
```

19.6 A nation divided

The USA has a varied climate, so we might expect yields to differ between states. Rather than trying to reason about 50 states separately, we can use the USA Census Regions to get 9 groups.

The "Corn Belt", where most US corn is grown is in the "West North Central" and "East North Central" regions. The "Wheat Belt" is in the "West South Central" region.

dplyr is loaded, the corn and wheat datasets are available, as is usa_census_regions.

Instructions 100 XP

- Inner join corn to usa_census_regions by "state".
- Turn the code into a function, fortify_with_census_region(). This should accept a single argument, data.

E6.R

```
# Inner join the corn dataset to usa_census_regions by state
corn %>%
  inner_join(usa_census_regions, by = "state")

# Wrap this code into a function
fortify_with_census_region <- function(data){
  data %>%
    inner_join(usa_census_regions, by = "state")
}

# Try it on the wheat dataset
fortify_with_census_region(wheat)
```

19.7 Plotting yields over time by region

So far, you have used a function to plot yields over time for each crop, and you've added a census_region column to the crop datasets. Now you are ready to look at how the yields change over time in each region of the USA.

ggplot2 is loaded. corn and wheat have been fortified with census regions. plot_yield_vs_year() is available.

Instructions 100 XP

- Use the function you wrote to plot yield versus year for the corn dataset, then facet the plot, wrapped by census_region.
- Turn the code into a function, plot_yield_vs_year_by_region(), that should take a single argument, data.

E7.R

```
# Plot yield vs. year for the corn dataset
plot_yield_vs_year(corn) +
  # Facet, wrapped by census region
  facet_wrap(vars(census_region))

# Wrap this code into a function
plot_yield_vs_year_by_region <- function(data) {

  plot_yield_vs_year(data) +
    facet_wrap(vars(census_region))
}

# Try it on the wheat dataset
plot_yield_vs_year_by_region(wheat)
```

19.8 Running a model

The smooth trend line you saw in the plots of yield over time use a generalized additive model (GAM) to determine where the line should lie. This sort of model is ideal for fitting nonlinear curves. So we can make predictions about future yields, let's explicitly run the model. The syntax for running this GAM takes the following form.

gam(response ~ s(explanatory_var1) + explanatory_var2, data = dataset)

Here, s() means "make the variable smooth", where smooth very roughly means nonlinear.

mgcv and dplyr are loaded; the corn and wheat datasets are available.

Instructions 100 XP

- Run a GAM of yield_kg_per_ha versus smoothed year and census region, using the corn dataset.
- Wrap the modeling code into a function, run_gam_yield_vs_year_by_region. This should take a single argument, data, with no default.

E8.R

```
# Run a generalized additive model of yield vs. smoothed year and census region

gam(yield_kg_per_ha ~ s(year) + census_region, data = corn)
```

```r
# Wrap the model code into a function
run_gam_yield_vs_year_by_region <- function(data){
  gam(yield_kg_per_ha ~ s(year) + census_region, data = corn)
}

# Try it on the wheat dataset
run_gam_yield_vs_year_by_region(wheat)
```

19.9 Making yield predictions

The fun part of modeling is using the models to make predictions. You can do this using a call to predict(), in the following form.

predict(model, cases_to_predict, type = "response") mgcv and dplyr are loaded; GAMs of the corn and wheat datasets are available as corn_model and wheat_model. A character vector of census regions is stored as census_regions.

19.10 Instructions 100 XP

- In predict_this, set the prediction year to 2050.
- Predict the yield using corn_model and the cases specified in predict_this.
- Mutate predict_this to add the prediction vector as a new column named pred_yield_kg_per_ha.
- Wrap the script into a function, predict_yields. It should take two arguments, model and year, with no defaults. Remember to update 2050 to the year argument. Try predict_yields() on wheat_model with year set to 2050.

E9.R

```r
# Make predictions in 2050
predict_this <- data.frame(
  year = 2050,
  census_region = census_regions
)

# Predict the yield
pred_yield_kg_per_ha <- predict(corn_model, predict_this, type = "response")

predict_this %>%
```

```
  # Add the prediction as a column of predict_this
    mutate(pred_yield_kg_per_ha = pred_yield_kg_per_ha)

# Wrap this prediction code into a function
predict_yields <- function(model, year) {
  predict_this <- data.frame(
    year = 2050,
    census_region = census_regions
  )
  pred_yield_kg_per_ha <- predict(model, predict_this, type = "response")
  predict_this %>%
    mutate(pred_yield_kg_per_ha = pred_yield_kg_per_ha)
}

# Try it on the wheat dataset
predict_yields(wheat_model,2050)
```

19.11 Do it all over again

Hopefully, by now, you've realized that the real benefit to writing functions is that you can reuse your code easily. Now you are going to rerun the whole analysis from this chapter on a new crop, barley. Since all the infrastructure is in place, that's less effort than it sounds!

Barley prefers a cooler climate compared to corn and wheat and is commonly grown in the US mountain states of Idaho and Montana.

dplyr and ggplot2, and mgcv are loaded; fortify_with_metric_units(), fortify_with_census_region(), plot_yield_vs_year_by_region(), run_gam_yield_vs_year_by_region(), and predict_yields() are available.

Instructions 100 XP

- Fortify the barley data with metric units, then with census regions.
- Using the fortified barley data, plot the yield versus year by census region.
- Using the fortified barley data, run a GAM of yield versus year by census region, then predict the yields in 2050.

E10.R

```
fortified_barley <- barley %>%
  # Fortify with metric units
    fortify_with_metric_units() %>%
  # Fortify with census regions
  fortify_with_census_region()

# See the result
glimpse(fortified_barley)

# From previous step
fortified_barley <- barley %>%
  fortify_with_metric_units() %>%
  fortify_with_census_region()

# Plot yield vs. year by region
plot_yield_vs_year_by_region(fortified_barley)

# From previous step
fortified_barley <- barley %>%
  fortify_with_metric_units() %>%
  fortify_with_census_region()

fortified_barley %>%
  # Run a GAM of yield vs. year by region
  run_gam_yield_vs_year_by_region()  %>%
  # Make predictions of yields in 2050
  predict_yields(2050)
```

Part IV

Introduction to Importing Data in R

20 Importing data from flat files with utils

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.

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().

E1.R

```
# Import swimming_pools.csv: pools
pools <- read.csv("swimming_pools.csv")

# Print the structure of pools
str(pools)
```

20.1 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) 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.

E2.R

```
# Import swimming_pools.csv correctly: pools
pools <- read.csv("swimming_pools.csv")

# Check the structure of pools
str(pools)
```

20.2 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)

How many variables in the resulting pools data frame have different types if you specify the stringsAsFactors argument differently?

The swimming_pools.csv (view) file is available in your current working directory so you can experiment in the console.

Instructions 50 XP

Possible Answers

Just one: Name.

Two variables: Name and Address. Respuesta

Three columns: all but Longitude.

All four of them!

20.3 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 "‾" (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.

E3.R

```
# Import hotdogs.txt: hotdogs
hotdogs <- read.delim("hotdogs.txt", header = FALSE)

# Summarize hotdogs
summary(hotdogs)
```

20.4 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.

E4.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)
```

20.5 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.

E5.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
print(lily)

print(tom)
```

20.6 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.

E6.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("hotdogs.txt", header = FALSE,
                       col.names = c("type", "calories", "sodium"),
                       colClasses = c("factor", "NULL", "numeric"))


# Display structure of hotdogs2
str(hotdogs2)
```

21 readr & data.table

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). 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. Remember that you can inspect your workspace with dir().

Instructions 100 XP

Load the readr package with library(). You do not need to install the package, it is already installed on DataCamp's servers. Import "potatoes.csv" using read_csv(). Assign the resulting data frame to the variable potatoes.

E1.R

```
# Load the readr package
library(readr)

# Import potatoes.csv with read_csv(): potatoes
potatoes <- read_csv("potatoes.csv")
```

21.1 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 (view) is available in your workspace. 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.

E2.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)
```

21.2 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 (“”) 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.

E3.R

```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
print(potatoes)
```

21.3 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 (view), 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.

E4.R

```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)
```

21.4 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 (view), 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.

E5.R

```
# readr is already loaded

# 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)
```

21.5 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 (view), 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.

E6.R

```
# 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)
```

21.6 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 (view) file, that's available in your workspace. 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. You do not need to install the package, it is already installed on DataCamp's servers.
- 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.

E7.R

```
# load the data.table package using library()
library(data.table)

# Import potatoes.csv with fread(): potatoes
potatoes <- fread("potatoes.csv")

# Print out potatoes
print(potatoes)
```

21.7 fread: more advanced 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:

> 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.

E8.R

```
# fread is already loaded

# Import columns 6 and 8 of potatoes.csv: potatoes
potatoes<- fread("potatoes.csv", select = c(6, 8))

# Plot texture (x) and moistness (y) of potatoes
plot(potatoes$texture, potatoes$moistness)
```

22 Importing Excel data

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.

E1.R

```
# Load the readxl package
library(readxl)

# Print the names of all worksheets
excel_sheets("urbanpop.xlsx")
```

22.1 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.

E2.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)
```

22.2 Reading a workbook

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.

E3.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)
```

22.3 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?

E4.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",sheet =1 ,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" ,sheet =1 , col_names = cols)

# Print the summary of pop_a
summary(pop_a)

# Print the summary of pop_b
summary(pop_b)
```

22.4 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.

E5.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("urbanpop.xlsx", sheet = 2, col_names = FALSE,
skip = 21)

# Print out the first observation from urbanpop_sel
head(urbanpop_sel,n=1 )
```

22.5 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().

E6.R

```
# Load the gdata package
library(gdata)

# Import the second sheet of urbanpop.xls: urban_pop
urban_pop <-read.xls("urbanpop.xls", sheet=2, )

# Print the first 11 observations using head()
head(urban_pop, n=11)
```

22.6 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().

E7.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, n=10)
```

22.7 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.

E8.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)
```

23 Reproducible Excel work with XLConnect

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), containing urban population data throughout time. The Excel file is available in your current working directory.

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?

E1.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)
```

23.1 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 (view). 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.

E2.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 = 2)
```

23.2 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.

E3.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)
```

23.3 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 (view). 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.

E4.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, name = "data_summary")

# Use getSheets() on my_book
getSheets(my_book)
```

23.4 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".

E5.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 = FAL
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, file="summary.xlsx")
```

23.5 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".

E6.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, file = "renamed.xlsx")
```

23.6 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".

E7.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, sheet = 4)

# Save workbook to "clean.xlsx"
saveWorkbook(my_book , file="clean.xlsx")
```

24 Summary

In summary, this book has no content whatsoever.

```
1 + 1
```

```
[1] 2
```

References

Knuth, Donald E. 1984. “Literate Programming.” *Comput. J.* 27 (2): 97–111. https://doi.org/10.1093/comjnl/27.2.97.