IntermedioR

Table of contents

1 Conditionals and Control Flow

1.1 Equality

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

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

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

1.2 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
```

1.3 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.

1.4 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
```

1.5 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.

1.6 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
```

1.7 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.

1.8 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
```

1.9 & 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.

1.10 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
```

1.11 & 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?

1.12 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
```

1.13 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))

1.14 Answer the question 50XP

Possible Answers

TRUE press 1

FALSE press 2

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

1.15 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.

1.16 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)
```

1.17 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.

1.18 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

```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!")
}
```

1.19 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!

1.20 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

```r
  print("Showing LinkedIn information")
}else {
  print("Unknown medium")
}
```

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

1.21 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!

1.22 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

```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!")
}
```

1.23 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.

1.24 Instructions `50 XP`

Possible Answers

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

- 2 and 3

1.25 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.

1.26 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)
```

1.27 Write a while loop

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

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

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

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

1.28 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.

E12.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)
```

1.29 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?

1.30 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.

E13.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
  }
}
```

1.31 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.

1.32 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.

E14.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
  }
}
```

1.33 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.

1.34 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.

E15.R

```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
```

1.35 Function documentation

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

help(sample) ?sample

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

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

args(sample)

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

1.36 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.

E16.R

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

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

1.37 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.

1.38 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.

E17.R

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

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

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

```
print(avg_fb)
```

1.39 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.

1.40 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?

E18.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)
```

1.41 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.

E19.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)
```

1.42 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!

E20.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)
```

1.43 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)

1.44 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.

1.45 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.

E21.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)
```

1.46 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!

1.47 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.

E22.R

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

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

1.48 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?

1.49 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.

E23.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)
```

1.50 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 }

1.51 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

1.52 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)

1.53 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.

1.54 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.

1.55 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.

E24.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])
```

1.56 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.

1.57 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.

E25.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)
```

1.58 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!

1.59 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.

E26.R

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

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

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

1.60 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:

2 Chunk 1

library(data.table) require(rjson)

3 Chunk 2

library("data.table") require(rjson)

4 Chunk 3

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

5 Chunk 4

library(c("data.table", "rjson")) Select the option that lists all of the chunks that do not generate an error. The console is yours to > experiment in.

Instructions `50 XP`

Possible Answers

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

6 The apply family

6.1 Use lapply with a built-in R function

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!

6.2 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().

E27.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)
```

6.3 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.

6.4 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.

E28.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)
```

6.5 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:

7 Named function

triple <- function(x) { 3 * x }

8 Anonymous function with same implementation

function(x) { 3 * x }

9 Use anonymous function inside lapply()

lapply(list(1,2,3), function(x) { 3 * x }) split_low is defined for you.

9.1 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.

E29.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] })
```

9.2 Use lapply with additional arguments

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

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.

9.3 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.

E30.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)
```

9.4 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]) } })

9.5 Instructions `50 XP`

Possible Answers

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

9.6 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.

9.7 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.

E31.R

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

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

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

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

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

9.8 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.

9.9 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.

E32.R

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

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

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

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

9.10 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!

9.11 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.

E33.R

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

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

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

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

9.12 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.

9.13 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.

E34.R

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

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

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

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

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

9.14 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.

9.15 Instructions 100 XP

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

E35.R

```
# temp is already available in the workspace

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

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

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

9.16 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.

9.17 Instructions 50 XP

Possible Answers

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

9.18 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.

9.19 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.

E36.R

```
# temp is already available in the workspace

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

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

9.20 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.

9.21 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.

E37.R

```
# temp is already available in the workspace

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

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

9.22 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!).

9.23 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).

E38.R

```
# temp is already defined in the workspace

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

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

10 Utilities

10.1 Mathematical utilities

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

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

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

10.2 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.

E40.R

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

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

10.3 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.

10.4 Instructions `100 XP`

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

E41.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()
```

10.5 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.

10.6 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.

E42.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)
```

10.7 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!

10.8 Instructions 100 XP

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

E43.R

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

10.9 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?

10.10 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.

E44.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)
```

10.11 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!

10.12 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".

E45.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]
```

10.13 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.

10.14 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.

E46.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]
```

10.15 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!

10.16 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.

E47.R

```
# The emails vector has already been defined for you
emails <- c("john.doe@ivyleague.edu", "education@world.gov",
"global@peace.org",
            "invalid.edu", "quant@bigdatacollege.edu",
            "cookie.monster@sesame.tv")

# Use sub() to convert the email domains to datacamp.edu
sub("@.*\\.edu$", "@datacamp.edu", emails)
```

10.17 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.

10.18 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

10.19 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.

10.20 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.

E48.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)
```

10.21 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!")

10.22 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.

E49.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")
```

10.23 Create and format times

Similar to working with dates, you can use as.POSIXct() to convert from a character string to a POSIXct object, and format() to convert from a POSIXct object to a character string. Again, you have a wide variety of symbols:

- %H: hours as a decimal number (00-23)
- %I: hours as a decimal number (01-12)
- %M: minutes as a decimal number
- %S: seconds as a decimal number
- %T: shorthand notation for the typical format %H:%M:%S
- %p: AM/PM indicator

For a full list of conversion symbols, consult the strptime documentation in the console:

?strptime

Again,as.POSIXct() uses a default format to match character strings. In this case, it's %Y-%m-%d %H:%M:%S. In this exercise, abstraction is made of different time zones.

10.24 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!

E50.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")
```

10.25 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.

10.26 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.

E51.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))
```

10.27 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.

10.28 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.

E52.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)
```

10.29 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.

10.30 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.

E53.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))
```