# Beginner R

**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**

**Instructions**

**100 XP**

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

[**Take Hint (-30 XP)**](javascript:void(0))

# Calculate 3 + 4

3 + 4

# Calculate 6 + 12

6 + 12

> # Calculate 3 + 4

> 3 + 4

[1] 7

>

> # Calculate 6 + 12

> 6+12

[1] 18

> # Calculate 3 + 4

> 3 + 4

[1] 7

>

> # Calculate 6 + 12

> 6 + 12

[1] 18

>

 +100 XP

Awesome! See how the console shows the result of the R code you submitted? Now that you're familiar with the interface, let's get down to R business!

**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. Click 'Submit Answer'. Notice that when you ask R to print x, the value 42 appears.

[**Take Hint (-30 XP)**](javascript:void(0))

# Assign the value 42 to x

x <- 42

# Print out the value of the variable x

X

 +100 XP

Good job! Have you noticed that R does not print the value of a variable to the console when you did the assignment? x <- 42 did not generate any output, because R assumes that you will be needing this variable in the future. Otherwise you wouldn't have stored the value in a variable in the first place, right? Proceed to the next exercise!

**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.
* Click 'Submit Answer', and look at the console: you see that the number 5 is printed. So R now links the variable my\_apples to the value 5.

[**Take Hint (-30 XP)**](javascript:void(0))

# Assign the value 5 to the variable my\_apples

my\_apples <- 5

# Print out the value of the variable my\_apples

my\_apples

**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.

[**Take Hint (-30 XP)**](javascript:void(0))

# Assign a value to the variables my\_apples and my\_oranges

my\_apples <- 5

my\_oranges <- 6

# Add these two variables together

my\_fruit <- my\_apples + my\_oranges

# Create the variable my\_fruit

my\_fruit

> # Assign a value to the variables my\_apples and my\_oranges

> my\_apples <- 5

> my\_oranges <- 6

>

> # Add these two variables together

> my\_fruit <- my\_apples + my\_oranges

>

> # Create the variable my\_fruit

> my\_fruit

[1] 11

>

 +100 XP

Nice one! The great advantage of doing calculations with variables is reusability. If you just change my\_apples to equal 12 instead of 5 and rerun the script, my\_fruit will automatically update as well. Continue to the next exercise.

**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**

**Instructions**

**100 XP**

* Click 'Submit 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.

[**Take Hint (-30 XP)**](javascript:void(0))

**Incorrect Submission**

Your code contains an error that you should fix:

Error: non-numeric argument to binary operator

You can do this by setting the my\_oranges variable to a numeric value, not a string!

# 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

# 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

**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**.
* Natural 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 on the right indicate that "some text" is a character.

**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!

# 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

**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 on the right shows.

**Instructions**

**100 XP**

Complete the code in the editor and also print out the classes of my\_character and my\_logical.

# 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)

**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!

[**Take Hint (-30 XP)**](javascript:void(0))

# Define the variable vegas

vegas <- 'Go!'

**Create a vector (2)**

Let us focus first!

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

In R, you create a vector with the combine function [**c()**](http://www.rdocumentation.org/packages/base/functions/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).

[**Take Hint (-30 XP)**](javascript:void(0))

numeric\_vector <- c(1, 10, 49)

character\_vector <- c("a", "b", "c")

# Complete the code for boolean\_vector

boolean\_vector <- c(TRUE, FALSE, TRUE)

 +100 XP

Perfect! Notice that adding a space behind the commas in the c() function improves the readability of your code. Let's practice some more with vector creation in the next exercise.

**Create a vector (3)**

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

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

For poker\_vector:

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

For roulette\_vector:

* On Monday you lost $24
* Tuesday you lost $50
* Wednesday you won $100
* Thursday you lost $350
* Friday you won $10

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

**Instructions**

**100 XP**

Assign the winnings/losses for roulette to the variable roulette\_vector.

[**Take Hint (-30 XP)**](javascript:void(0))

# 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)

**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 on the right names the elements in poker\_vector with the days of the week. Add code to do the same thing for roulette\_vector.

[**Take Hint (-30 XP)**](javascript:void(0))

# Poker winnings from Monday to Friday

poker\_vector <- c(140, -50, 20, -120, 240)

# Roulette winnings from Monday to Friday

roulette\_vector <- c(-24, -50, 100, -350, 10)

# Assign days as names of poker\_vector

names(poker\_vector) <- c("Monday", "Tuesday", "Wednesday", "Thursday", "Friday")

# Assign days as names of roulette\_vector

names(roulette\_vector) <- c("Monday", "Tuesday", "Wednesday", "Thursday", "Friday")

**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.

[**Take Hint (-30 XP)**](javascript:void(0))

# 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)

**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.

[**Take Hint (-30 XP)**](javascript:void(0))

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

**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).

[**Take Hint (-30 XP)**](javascript:void(0))

# 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 <- poker\_vector+roulette\_vector

total\_daily

**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()**](http://www.rdocumentation.org/packages/base/functions/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.

[**Take Hint (-30 XP)**](javascript:void(0))

**Incorrect Submission**

Your code contains an error that you should fix:

Error: object 'total\_daily' not found

# Poker and roulette winnings from Monday to Friday:

# total\_daily

poker\_vector <- c(140, -50, 20, -120, 240)

roulette\_vector <- c(-24, -50, 100, -350, 10)

days\_vector <- c("Monday", "Tuesday", "Wednesday", "Thursday", "Friday")

names(poker\_vector) <- days\_vector

names(roulette\_vector) <- days\_vector

# Total winnings with poker

total\_poker <- sum(poker\_vector)

# Total winnings with roulette

total\_roulette <- sum(roulette\_vector)

# Total winnings overall

total\_week <- total\_poker+total\_roulette

# Print out total\_week

total\_week

[1] -84

**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?

[**Take Hint (-30 XP)**](javascript:void(0))

# 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

[1] TRUE

**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.

[**Take Hint (-30 XP)**](javascript:void(0))

# Poker and roulette winnings from Monday to Friday:

poker\_vector <- c(140, -50, 20, -120, 240)

roulette\_vector <- c(-24, -50, 100, -350, 10)

days\_vector <- c("Monday", "Tuesday", "Wednesday", "Thursday", "Friday")

names(poker\_vector) <- days\_vector

names(roulette\_vector) <- days\_vector

# Define a new variable based on a selection

poker\_wednesday <- poker\_vector[3]

**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.

[**Take Hint (-30 XP)**](javascript:void(0))

**Incorrect Submission**

It looks like poker\_midweek does not contain the correct values from poker\_vector. You can use the vector c(2, 3, 4) inside square brackets.

Did you find this feedback helpful?

[YesNo](javascript:void(0))

# 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)]

**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.

[**Take Hint (-30 XP)**](javascript:void(0))

# 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]

**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**

**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()**](http://www.rdocumentation.org/packages/base/functions/mean) function. Simply print out the result so you can inspect it.

[**Take Hint (-30 XP)**](javascript:void(0))

# 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 <-

# Calculate the average of the elements in poker\_start

**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**

**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()**](http://www.rdocumentation.org/packages/base/functions/mean) function. Simply print out the result so you can inspect it.

[**Take Hint (-30 XP)**](javascript:void(0))

# 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[1:3]

# Calculate the average of the elements in poker\_start

mean(poker\_start)

[1] 36.66667

**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**

**70 XP**

**Instructions**

**70 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.

[**Show Answer (-70 XP)**](javascript:void(0))

**Hint**

In order to check for which days your poker gains are positive, R should check for each element of poker\_vector whether it is larger than zero. some\_vector > 0 is the way to tell R what you are after.

> # 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

Monday Tuesday Wednesday Thursday Friday

TRUE FALSE TRUE FALSE TRUE

>

**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.

[**Take Hint (-30 XP)**](javascript:void(0))

> # 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]

>

**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.

[**Take Hint (-30 XP)**](javascript:void(0))

> # 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]

>

 +100 XP

Great! This exercise concludes the chapter on vectors. The next chapter will introduce you to the two-dimensional version of vectors: matrices.

**What's a matrix?**

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

You can construct a matrix in R with the [**matrix()**](http://www.rdocumentation.org/packages/base/functions/matrix) function. Consider the following example:

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

In the [**matrix()**](http://www.rdocumentation.org/packages/base/functions/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**

**Instructions**

**100 XP**

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

[**Take Hint (-30 XP)**](javascript:void(0))

> # Construct a matrix with 3 rows that contain the numbers 1 up to 9

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

[,1] [,2] [,3]

[1,] 1 2 3

[2,] 4 5 6

[3,] 7 8 9

>

**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.

[**Take Hint (-30 XP)**](javascript:void(0))

> # 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)

>

> box\_office

[1] 460.998 314.400 290.475 247.900 309.306 165.800

> star\_wars\_matrix

[,1] [,2]

[1,] 460.998 314.4

[2,] 290.475 247.9

[3,] 309.306 165.8

>

 +100 XP

The force is actually with you! Continue to the next exercise.

**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.

[**Take Hint (-30 XP)**](javascript:void(0))

# 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

star\_wars\_matrix

 +100 XP

Great! You're on the way of becoming an R jedi! Continue to the next exercise.

**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()**](http://www.rdocumentation.org/packages/base/functions/colSums) 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.

[**Take Hint (-30 XP)**](javascript:void(0))

> # Construct star\_wars\_matrix

> box\_office <- c(460.998, 314.4, 290.475, 247.900, 309.306, 165.8)

> star\_wars\_matrix <- matrix(box\_office, nrow = 3, byrow = TRUE,

dimnames = list(c("A New Hope", "The Empire Strikes Back", "Return of the Jedi"),

c("US", "non-US")))

>

> # Calculate worldwide box office figures

> worldwide\_vector <- rowSums(star\_wars\_matrix)

>

**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()**](http://www.rdocumentation.org/packages/base/functions/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()**](http://www.rdocumentation.org/packages/base/functions/cbind) function.

[**Take Hint (-30 XP)**](javascript:void(0))

**Incorrect Submission**

Have you correctly used cbind() to add worldwide\_vector to star\_wars\_matrix? You should pass star\_wars\_matrix and world\_wide\_vector to cbind(), in this order. The resulting matrix, all\_wars\_matrix, should consist of three rows and three columns.

> # Construct star\_wars\_matrix

> box\_office <- c(460.998, 314.4, 290.475, 247.900, 309.306, 165.8)

> star\_wars\_matrix <- matrix(box\_office, nrow = 3, byrow = TRUE,

dimnames = list(c("A New Hope", "The Empire Strikes Back", "Return of the Jedi"),

c("US", "non-US")))

>

> # 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)

>

 +100 XP

Nice job! After adding column to a matrix, the logical next step is adding rows. Learn how in the next exercise.

**Adding a row**

Just like every action has a reaction, every [**cbind()**](http://www.rdocumentation.org/packages/base/functions/cbind) has an [**rbind()**](http://www.rdocumentation.org/packages/base/functions/cbind). (We admit, we are pretty bad with metaphors.)

Your R workspace, where all variables you defined 'live' ([**check out what a workspace is**](http://www.statmethods.net/interface/workspace.html)), 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.

Type the name of these matrices in the console and hit Enter 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.

[**Take Hint (-30 XP)**](javascript:void(0))

> # star\_wars\_matrix and star\_wars\_matrix2 are available in your workspace

> star\_wars\_matrix

US non-US

A New Hope 461.0 314.4

The Empire Strikes Back 290.5 247.9

Return of the Jedi 309.3 165.8

> star\_wars\_matrix2

US non-US

The Phantom Menace 474.5 552.5

Attack of the Clones 310.7 338.7

Revenge of the Sith 380.3 468.5

>

> # Combine both Star Wars trilogies in one matrix

> all\_wars\_matrix <- rbind(star\_wars\_matrix, star\_wars\_matrix2)

>

 +100 XP

Wonderful! Continue with the next exercise and see how you can combine the results of the rbind() function with the colSums() function!

**The total box office revenue for the entire saga**

Just like [**cbind()**](http://www.rdocumentation.org/packages/base/functions/cbind) has [**rbind()**](http://www.rdocumentation.org/packages/base/functions/cbind), [**colSums()**](http://www.rdocumentation.org/packages/base/functions/colSums) has [**rowSums()**](http://www.rdocumentation.org/packages/base/functions/colSums). 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()**](http://www.rdocumentation.org/packages/base/functions/colSums) function.
* Print out total\_revenue\_vector to have a look at the results.

[**Take Hint (-30 XP)**](javascript:void(0))

> # all\_wars\_matrix is available in your workspace

> all\_wars\_matrix

US non-US

A New Hope 461.0 314.4

The Empire Strikes Back 290.5 247.9

Return of the Jedi 309.3 165.8

The Phantom Menace 474.5 552.5

Attack of the Clones 310.7 338.7

Revenge of the Sith 380.3 468.5

>

> # Total revenue for US and non-US

> total\_revenue\_vector <- colSums(all\_wars\_matrix)

>

> # Print out total\_revenue\_vector

> total\_revenue\_vector

US non-US

2226.3 2087.8

>

 +100 XP

Bellissimo! Head over to the next exercise to learn matrix subsetting.

**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**

**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.

[**Take Hint (-30 XP)**](javascript:void(0))

**Incorrect Submission**

Your code contains a syntax error. Check the console output and try to fix the issue.

> # all\_wars\_matrix is available in your workspace

> all\_wars\_matrix

US non-US

A New Hope 461.0 314.4

The Empire Strikes Back 290.5 247.9

Return of the Jedi 309.3 165.8

The Phantom Menace 474.5 552.5

Attack of the Clones 310.7 338.7

Revenge of the Sith 380.3 468.5

>

> # Select the non-US revenue for all movies

> non\_us\_all <- all\_wars\_matrix[, 2]

> non\_us\_all

A New Hope The Empire Strikes Back Return of the Jedi

314.4 247.9 165.8

The Phantom Menace Attack of the Clones Revenge of the Sith

552.5 338.7 468.5

>

> # Average non-US revenue

> mean(non\_us\_all)

[1] 347.9667

>

> # Select the non-US revenue for first two movies

> non\_us\_some <- non\_us\_all[1:2]

> non\_us\_some

A New Hope The Empire Strikes Back

314.4 247.9

>

> # Average non-US revenue for first two movies

> mean(non\_us\_some)

[1] 281.15

>

**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.

[**Take Hint (-30 XP)**](javascript:void(0))

> # all\_wars\_matrix is available in your workspace

> all\_wars\_matrix

US non-US

A New Hope 461.0 314.4

The Empire Strikes Back 290.5 247.9

Return of the Jedi 309.3 165.8

The Phantom Menace 474.5 552.5

Attack of the Clones 310.7 338.7

Revenge of the Sith 380.3 468.5

>

> # Estimate the visitors

> visitors <- all\_wars\_matrix/5

>

> # Print the estimate to the console

> visitors

US non-US

A New Hope 92.20 62.88

The Empire Strikes Back 58.10 49.58

Return of the Jedi 61.86 33.16

The Phantom Menace 94.90 110.50

Attack of the Clones 62.14 67.74

Revenge of the Sith 76.06 93.70

>

 +100 XP

Great! What do these results tell you? A staggering 92 million people went to see A New Hope in US theaters! Continue to the next exercise.

**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.

[**Take Hint (-30 XP)**](javascript:void(0))

> # all\_wars\_matrix and ticket\_prices\_matrix are available in your workspace

> all\_wars\_matrix

US non-US

A New Hope 461.0 314.4

The Empire Strikes Back 290.5 247.9

Return of the Jedi 309.3 165.8

The Phantom Menace 474.5 552.5

Attack of the Clones 310.7 338.7

Revenge of the Sith 380.3 468.5

> ticket\_prices\_matrix

US non-US

A New Hope 5.0 5.0

The Empire Strikes Back 6.0 6.0

Return of the Jedi 7.0 7.0

The Phantom Menace 4.0 4.0

Attack of the Clones 4.5 4.5

Revenge of the Sith 4.9 4.9

>

> # 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)

[1] 75.01401

>

 +100 XP

It's a fact: the R force is with you! This exercise concludes the chapter on matrices. Next stop on your journey through the R language: factors.

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

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

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

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

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

**Instructions**

**100 XP**

Assign to variable theory the value "factors for categorical variables".

[**Take Hint (-30 XP)**](javascript:void(0))

# Assign to the variable theory what this chapter is about!

theory <- "factors for categorical variables"

 +100 XP

Good job! Ready to start? Continue to the next exercise!

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

To create factors in R, you make use of the function [**factor()**](http://www.rdocumentation.org/packages/base/functions/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()**](http://www.rdocumentation.org/packages/base/functions/factor) will encode the vector as a factor:

factor\_sex\_vector <- factor(sex\_vector)

**Instructions**

**100 XP**

**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.

[**Take Hint (-30 XP)**](javascript:void(0))

> # 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

> factor\_sex\_vector

[1] Male Female Female Male Male

Levels: Female Male

>

 +100 XP

Great! If you want to find out more about the factor() function, do not hesitate to type ?factor in the console. This will open up a help page. Continue to the next exercise.

**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**

Click 'Submit 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.

[**Take Hint (-30 XP)**](javascript:void(0))

> # Animals

> animals\_vector <- c("Elephant", "Giraffe", "Donkey", "Horse")

> factor\_animals\_vector <- factor(animals\_vector)

> factor\_animals\_vector

[1] Elephant Giraffe Donkey Horse

Levels: Donkey Elephant Giraffe Horse

>

> # 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

[1] High Low High Low Medium

Levels: Low < Medium < High

>

 +100 XP

Can you already tell what's happening in this exercise? Awesome! Continue to the next exercise and get into the details of factor levels.

**Exercise**

**Factor levels**

When you first get a data set, 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()**](http://www.rdocumentation.org/packages/base/functions/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**

**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.

[**Take Hint (-30 XP)**](javascript:void(0))

> # 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

[1] Male Female Female Male Male

Levels: Female Male

>

**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**

Click 'Submit Answer' to start the first exercise on lists.

[**Take Hint (-30 XP)**](javascript:void(0))

**Creating a list**

Let us create our first list! To construct a list you use the function [**list()**](http://www.rdocumentation.org/packages/base/functions/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.

[**Take Hint (-30 XP)**](javascript:void(0))

# 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)

 +100 XP

Wonderful! Head over to the next exercise.

**Exercise**

**Exercise**

**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.

[**Take Hint (-30 XP)**](javascript:void(0))

**Incorrect Submission**

It looks like my\_list does not contain the correct naming for the components. Make sure you use the names vec, mat and df, respectively. Check out the hint if you're struggling.

> # Vector with numerics from 1 up to 10

> my\_vector <- 1:10

>

> # Matrix with numerics from 1 up to 9

> my\_matrix <- matrix(1:9, ncol = 3)

>

> # First 10 elements of the built-in data frame mtcars

> my\_df <- mtcars[1:10,]

>

> # Adapt list() call to give the components names

> my\_list <- list(my\_vector, my\_matrix, my\_df)

> names(my\_list) <- c('vec', 'mat', 'df')

>

> # Print out my\_list

> my\_list

$vec

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

$mat

[,1] [,2] [,3]

[1,] 1 4 7

[2,] 2 5 8

[3,] 3 6 9

$df

mpg cyl disp hp drat wt qsec vs am gear carb

Mazda RX4 21.0 6 160.0 110 3.90 2.620 16.46 0 1 4 4

Mazda RX4 Wag 21.0 6 160.0 110 3.90 2.875 17.02 0 1 4 4

Datsun 710 22.8 4 108.0 93 3.85 2.320 18.61 1 1 4 1

Hornet 4 Drive 21.4 6 258.0 110 3.08 3.215 19.44 1 0 3 1

Hornet Sportabout 18.7 8 360.0 175 3.15 3.440 17.02 0 0 3 2

Valiant 18.1 6 225.0 105 2.76 3.460 20.22 1 0 3 1

Duster 360 14.3 8 360.0 245 3.21 3.570 15.84 0 0 3 4

Merc 240D 24.4 4 146.7 62 3.69 3.190 20.00 1 0 4 2

Merc 230 22.8 4 140.8 95 3.92 3.150 22.90 1 0 4 2

Merc 280 19.2 6 167.6 123 3.92 3.440 18.30 1 0 4 4

>

 +100 XP

Great! Not only do you know how to construct lists now, you can also name them; a skill that will prove most useful in practice. Continue to the next exercise.

**Exercise**

**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 on the right 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).

[**Take Hint (-30 XP)**](javascript:void(0))

> # The variables mov, act and rev are available

>

> # Finish the code to build shining\_list

> shining\_list <- list(moviename=mov, actors=act, reviews=rev)

> shining\_list

$moviename

[1] "The Shining"

$actors

[1] "Jack Nicholson" "Shelley Duvall" "Danny Lloyd" "Scatman Crothers"

[5] "Barry Nelson"

$reviews

scores sources comments

1 4.5 IMDb1 Best Horror Film I Have Ever Seen

2 4.0 IMDb2 A truly brilliant and scary film from Stanley Kubrick

3 5.0 IMDb3 A masterpiece of psychological horror

>

 +100 XP

Wonderful! You now know how to construct and name lists. As in the previous chapters, let's look at how to select elements for lists. Head over to the next exercise

**Exercise**

**Exercise**

**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.

[**Take Hint (-30 XP)**](javascript:void(0))

**Incorrect Submission**

Your code contains a syntax error. Check the console output and try to fix the issue.

To select the second actor from the vector representing actors, you should chain your selections: shining\_list$actors represents the actors, so you can add a [2] to select the second element.

Parsing error in script.R:7:16: unexpected numeric constant

6: # Print the second element of the vector representing the actors

7: shining\_list $ 2

^

> # shining\_list is already pre-loaded in the workspace

>

> # Print out the vector representing the actors

> shining\_list $ actors

[1] "Jack Nicholson" "Shelley Duvall" "Danny Lloyd" "Scatman Crothers"

[5] "Barry Nelson"

>

> # Print the second element of the vector representing the actors

> shining\_list[[2]]

[1] "Jack Nicholson" "Shelley Duvall" "Danny Lloyd" "Scatman Crothers"

[5] "Barry Nelson"

> # shining\_list is already pre-loaded in the workspace

>

> # Print out the vector representing the actors

> shining\_list $ actors

[1] "Jack Nicholson" "Shelley Duvall" "Danny Lloyd" "Scatman Crothers"

[5] "Barry Nelson"

>

> # Print the second element of the vector representing the actors

> shining\_list $ actors[2]

[1] "Shelley Duvall"

>

 +100 XP

Great! Selecting elements from lists is rather easy isn't it? Continue to the next exercise.

**Exercise**

**Exercise**

**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.

[**Take Hint (-30 XP)**](javascript:void(0))

**Incorrect Submission**

Have you called data.frame()?

Check your call of list(). Did you specify the argument ..3?

**Hint**

* You'll use the c() function to create the vectors.
* To find the average of a vector, pass the name of the vector to the mean() function.
* Use the data.frame() function to create the reviews\_df data frame.

> # 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)

> avg\_review

[1] 4.72

>

> # Combine scores and comments into the reviews\_df data frame

> reviews\_df <- c(scores, comments)

> reviews\_df

[1] "4.6" "5" "4.8"

[4] "5" "4.2" "I would watch it again"

[7] "Amazing!" "I liked it" "One of the best movies"

[10] "Fascinating plot"

> # Create and print out a list, called departed\_list

> departed\_list <- list(movie\_title, movie\_actors)

> # 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)

> avg\_review

[1] 4.72

>

> # Combine scores and comments into the reviews\_df data frame

> reviews\_df <- data.frame(scores, comments)

> reviews\_df

scores comments

1 4.6 I would watch it again

2 5.0 Amazing!

3 4.8 I liked it

4 5.0 One of the best movies

5 4.2 Fascinating plot

> # Create and print out a list, called departed\_list

> departed\_list <- list(movie\_title, movie\_actors)

> # 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)

> avg\_review

[1] 4.72

>

> # Combine scores and comments into the reviews\_df data frame

> reviews\_df <- data.frame(scores, comments)

> reviews\_df

scores comments

1 4.6 I would watch it again

2 5.0 Amazing!

3 4.8 I liked it

4 5.0 One of the best movies

5 4.2 Fascinating plot

>

> # Create and print out a list, called departed\_list

> departed\_list <- list(movie\_title, movie\_actors, reviews\_df, avg\_review)

> departed\_list

[[1]]

[1] "The Departed"

[[2]]

[1] "Leonardo DiCaprio" "Matt Damon" "Jack Nicholson"

[4] "Mark Wahlberg" "Vera Farmiga" "Martin Sheen"

[[3]]

scores comments

1 4.6 I would watch it again

2 5.0 Amazing!

3 4.8 I liked it

4 5.0 One of the best movies

5 4.2 Fascinating plot

[[4]]

[1] 4.72

>

Solution:

> # 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)

> avg\_review

[1] 4.72

>

> # Combine scores and comments into the reviews\_df data frame

> reviews\_df <- c(scores, comments)

> reviews\_df

[1] "4.6" "5" "4.8"

[4] "5" "4.2" "I would watch it again"

[7] "Amazing!" "I liked it" "One of the best movies"

[10] "Fascinating plot"

> # Create and print out a list, called departed\_list

> departed\_list <- list(movie\_title, movie\_actors)

> # 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)

> avg\_review

[1] 4.72

>

> # Combine scores and comments into the reviews\_df data frame

> reviews\_df <- data.frame(scores, comments)

> reviews\_df

scores comments

1 4.6 I would watch it again

2 5.0 Amazing!

3 4.8 I liked it

4 5.0 One of the best movies

5 4.2 Fascinating plot

> # Create and print out a list, called departed\_list

> departed\_list <- list(movie\_title, movie\_actors)

> # 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)

> avg\_review

[1] 4.72

>

> # Combine scores and comments into the reviews\_df data frame

> reviews\_df <- data.frame(scores, comments)

> reviews\_df

scores comments

1 4.6 I would watch it again

2 5.0 Amazing!

3 4.8 I liked it

4 5.0 One of the best movies

5 4.2 Fascinating plot

>

> # Create and print out a list, called departed\_list

> departed\_list <- list(movie\_title, movie\_actors, reviews\_df, avg\_review)

> departed\_list

[[1]]

[1] "The Departed"

[[2]]

[1] "Leonardo DiCaprio" "Matt Damon" "Jack Nicholson"

[4] "Mark Wahlberg" "Vera Farmiga" "Martin Sheen"

[[3]]

scores comments

1 4.6 I would watch it again

2 5.0 Amazing!

3 4.8 I liked it

4 5.0 One of the best movies

5 4.2 Fascinating plot

[[4]]

[1] 4.72

>

 +100 XP

Good work! You successfully created another list of movie information, and combined different components into a single list. Congratulations on finishing the course!

# Intermediate R

**Exercise**

**Equality**

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

3 == (2 + 1)

"intermediate" != "r"

TRUE != FALSE

"Rchitect" != "rchitect"

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

**Instructions**

**100 XP**

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

[**Take Hint (-30 XP)**](javascript:void(0))

> # Comparison of logicals

> TRUE == FALSE

[1] FALSE

>

> # Comparison of numerics

> -6 \* 14 != 17 - 101

[1] FALSE

>

> # Comparison of character strings

> "useR" == "user"

[1] FALSE

>

> # Compare a logical with a numeric

> TRUE == 1

[1] TRUE

>

 +100 XP

Awesome! Since TRUE coerces to 1 under the hood, TRUE == 1 evaluates to TRUE. Make sure not to mix up == (comparison) and = (assignment), == is what need to check the equality of R objects.

**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.

[**Take Hint (-30 XP)**](javascript:void(0))

> # Comparison of numerics

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

[1] FALSE

>

> # Comparison of character strings

> "raining" <= "raining dogs"

[1] TRUE

>

> # Comparison of logicals

> TRUE > FALSE

[1] TRUE

>

 +100 XP

Great job! Make sure to have a look at the console output to see if R returns the results you expected.

**Exercise**

**Exercise**

**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?

[**Take Hint (-30 XP)**](javascript:void(0))

> # 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

[1] TRUE FALSE FALSE FALSE FALSE TRUE FALSE

>

> # Quiet days

> linkedin <= 5

[1] FALSE FALSE FALSE TRUE TRUE FALSE FALSE

>

> # LinkedIn more popular than Facebook

> linkedin > facebook

[1] FALSE TRUE TRUE FALSE FALSE TRUE FALSE

>

 +100 XP

Wonderful! Have a look at the console output. Your LinkedIn profile was pretty popular on the sixth day, but less so on the fourth and fifth day.

**Exercise**

**Exercise**

**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.

[**Take Hint (-30 XP)**](javascript:void(0))

> # 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

[,1] [,2] [,3] [,4] [,5] [,6] [,7]

[1,] FALSE FALSE TRUE FALSE FALSE FALSE FALSE

[2,] FALSE FALSE FALSE FALSE FALSE TRUE FALSE

>

> # When is views less than or equal to 14?

> views <= 14

[,1] [,2] [,3] [,4] [,5] [,6] [,7]

[1,] FALSE TRUE TRUE TRUE TRUE FALSE TRUE

[2,] FALSE TRUE TRUE FALSE TRUE TRUE TRUE

>

 +100 XP

Nice job! This exercise concludes the part on comparators. Now that you know how to query the relation between R objects, the next step will be to use the results to alter the behavior of your programs. Find out all about that in the next video!

**Exercise**

**& 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 already been defined in the editor.

**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?

[**Take Hint (-30 XP)**](javascript:void(0))

Parsing error in script.R:9:11: unexpected '<='

8: # Is last between 15 (exclusive) and 20 (inclusive)?

9: 15 < last <=

^

> # The linkedin and last variable are already defined for you

> linkedin <- c(16, 9, 13, 5, 2, 17, 14)

> last <- tail(linkedin, 1)

>

> # Is last under 5 or above 10?

> last < 5 | last > 10

[1] TRUE

>

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

> 15 < last & last <= 20

[1] FALSE

>

 +100 XP

Great! Have one last look at the console before proceeding; do the results of the different expressions make sense?

**& 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?

[**Take Hint (-30 XP)**](javascript:void(0))

**Incorrect Submission**

Your code contains a syntax error. Check the console output and try to fix the issue.

In your code for the second instruction, you should use linkedin >= 12.

Have another look at your code for the second instruction. Make sure to use the correct logical operator: |.

 +100 XP

Bravo! You'll have noticed how easy it is to use logical operators to vectors and matrices. What do these results tell us? The third day of the recordings was the only day where your LinkedIn profile was visited more than 10 times, while your Facebook profile wasn't. Can you draw similar conclusions for the other results?

> views

[,1] [,2] [,3] [,4] [,5] [,6] [,7]

[1,] 16 9 13 5 2 17 14

[2,] 17 7 5 16 8 13 14

Parsing error in script.R:10:12: unexpected '<='

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

10: 11 < views <=

^

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

>

> # linkedin exceeds 10 but facebook below 10

> linkedin > 10 & facebook < 10

[1] FALSE FALSE TRUE FALSE FALSE FALSE FALSE

>

> # When were one or both visited at least 12 times?

> views >= 12

[,1] [,2] [,3] [,4] [,5] [,6] [,7]

[1,] TRUE FALSE TRUE FALSE FALSE TRUE TRUE

[2,] TRUE FALSE FALSE TRUE FALSE TRUE TRUE

>

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

> 11 < views & views <= 14

[,1] [,2] [,3] [,4] [,5] [,6] [,7]

[1,] FALSE FALSE TRUE FALSE FALSE FALSE TRUE

[2,] FALSE FALSE FALSE FALSE FALSE TRUE TRUE

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

>

> # linkedin exceeds 10 but facebook below 10

> linkedin > 10 & facebook < 10

[1] FALSE FALSE TRUE FALSE FALSE FALSE FALSE

>

> # When were one or both visited at least 12 times?

> linkedin >= 12 & facebook >= 12

[1] TRUE FALSE FALSE FALSE FALSE TRUE TRUE

>

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

> 11 < views & views <= 14

[,1] [,2] [,3] [,4] [,5] [,6] [,7]

[1,] FALSE FALSE TRUE FALSE FALSE FALSE TRUE

[2,] FALSE FALSE FALSE FALSE FALSE TRUE TRUE

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

>

> # linkedin exceeds 10 but facebook below 10

> linkedin > 10 & facebook < 10

[1] FALSE FALSE TRUE FALSE FALSE FALSE FALSE

>

> # When were one or both visited at least 12 times?

> linkedin >= 12 | facebook >= 12

[1] TRUE FALSE TRUE TRUE FALSE TRUE TRUE

>

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

> 11 < views & views <= 14

[,1] [,2] [,3] [,4] [,5] [,6] [,7]

[1,] FALSE FALSE TRUE FALSE FALSE FALSE TRUE

[2,] FALSE FALSE FALSE FALSE FALSE TRUE TRUE

>

**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))

**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.

[**Take Hint (-30 XP)**](javascript:void(0))

> # li\_df is pre-loaded in your workspace

> li\_df

day1 day2 day3 day4 day5 day6 day7

employee\_1 2 3 3 6 4 2 0

employee\_2 19 23 18 22 23 29 25

employee\_3 24 18 15 19 18 22 17

employee\_4 22 18 27 26 19 21 25

employee\_5 25 25 26 31 24 36 37

employee\_6 22 20 29 26 23 22 29

employee\_7 0 4 2 2 3 4 2

employee\_8 12 3 15 7 1 15 11

employee\_9 19 22 22 19 25 24 23

employee\_10 23 12 19 25 18 22 22

employee\_11 29 27 23 25 29 30 17

employee\_12 13 13 20 17 12 22 20

employee\_13 7 17 9 5 11 9 9

employee\_14 26 27 28 36 29 31 30

employee\_15 7 6 4 11 5 5 15

employee\_16 32 35 31 35 24 25 36

employee\_17 7 17 9 12 13 6 12

employee\_18 9 6 3 12 3 8 6

employee\_19 0 1 11 6 0 4 11

employee\_20 9 12 6 13 12 13 11

employee\_21 6 15 15 10 9 7 18

employee\_22 17 17 12 4 14 17 7

employee\_23 1 12 8 2 4 4 11

employee\_24 5 8 0 1 6 3 1

employee\_25 2 7 5 3 1 5 5

employee\_26 29 25 32 28 28 27 27

employee\_27 17 15 17 23 23 17 22

employee\_28 26 32 33 30 33 28 26

employee\_29 27 29 24 29 26 31 28

employee\_30 4 1 1 2 1 7 4

employee\_31 22 22 17 20 14 19 13

employee\_32 9 11 7 10 8 15 5

employee\_33 6 5 12 5 17 17 4

employee\_34 18 17 12 22 22 13 12

employee\_35 2 12 13 7 10 6 2

employee\_36 32 26 20 23 24 25 21

employee\_37 5 13 12 11 6 5 10

employee\_38 6 10 11 6 6 2 5

employee\_39 30 37 32 35 37 41 42

employee\_40 34 33 32 35 33 27 35

employee\_41 15 19 21 18 22 26 22

employee\_42 28 29 30 19 21 19 26

employee\_43 6 8 6 7 17 11 14

employee\_44 17 22 27 24 18 28 24

employee\_45 6 10 17 18 13 10 7

employee\_46 18 19 22 17 21 15 23

employee\_47 21 27 28 28 26 17 25

employee\_48 10 18 20 18 12 19 17

employee\_49 6 15 15 15 10 14 2

employee\_50 30 28 29 31 24 20 25

>

> # Select the second column, named day2, from li\_df: second

> second <- li\_df $ day2

>

> # Build a logical vector, TRUE if value in second is extreme: extremes

> extremes <- second < 5 | second > 25

> extremes

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

[13] FALSE TRUE FALSE TRUE FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE

[25] FALSE FALSE FALSE TRUE TRUE TRUE FALSE FALSE FALSE FALSE FALSE TRUE

[37] FALSE FALSE TRUE TRUE FALSE TRUE FALSE FALSE FALSE FALSE TRUE FALSE

[49] FALSE TRUE

>

> # Count the number of TRUEs in extremes

> sum(extremes)

[1] 16

>

 +100 XP

Great! Head over to the next video and learn how relational and logical operators can be used to alter the flow of your R scripts.

**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 these variables have already been defined in the editor.

**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.

[**Take Hint (-30 XP)**](javascript:void(0))

**Incorrect Submission**

Inside the body of your if statement, use the function print() to print the message "You are popular!" when the number of views exceeds 15. Remember that R is case sensitive.

> # Variables related to your last day of recordings

> medium <- "LinkedIn"

> num\_views <- 14

>

> "Test print w/o function."

[1] "Test print w/o function."

>

> # Examine the if statement for medium

> if (medium == "LinkedIn") {

print("Showing LinkedIn information")

}

[1] "Showing LinkedIn information"

>

> # Write the if statement for num\_views

> if (num\_views > 15) {print("You are popular!")}

>

 +100 XP

Great! Try to see what happens if you change the medium and num\_views variables and run your code again. Let's further customize these if statements in the next exercise.

**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 in the editor. 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.

[**Take Hint (-30 XP)**](javascript:void(0))

Parsing error in script.R:11:1: unexpected 'else'

10:

11: else

^

>

> # Variables related to your last day of recordings

> medium <- "LinkedIn"

> num\_views <- 14

>

> # Control structure for medium

> if (medium == "LinkedIn") {

print("Showing LinkedIn information")

} else {

print("Unknown medium")

}

[1] "Showing LinkedIn information"

>

> # Control structure for num\_views

> if (num\_views > 15) {

print("You're popular!")

} else {

print("Try to be more visible!")

}

[1] "Try to be more visible!"

>

+100 XP

Great job! You also had Facebook information available, remember? Time to add some more statements to our control structures using else if!

**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**

**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.

[**Take Hint (-30 XP)**](javascript:void(0))

> # Variables related to your last day of recordings

> medium <- "LinkedIn"

> num\_views <- 14

>

> # Control structure for medium

> if (medium == "LinkedIn") {

print("Showing LinkedIn information")

} else if (medium == "Facebook") {

# Add code to print correct string when condition is TRUE

print("Showing Facebook information")

} else {

print("Unknown medium")

}

[1] "Showing LinkedIn information"

>

> # Control structure for num\_views

> if (num\_views > 15) {

print("You're popular!")

} else if (num\_views <= 15 & num\_views > 10) {

# Add code to print correct string when condition is TRUE

print("Your number of views is average")

} else {

print("Try to be more visible!")

}

[1] "Your number of views is average"

>

 +100 XP

Awesome! Have another look at the second control structure. Because R abandons the control flow as soon as it finds a condition that is met, you can simplify the condition for the else if part in the second construct to num\_views > 10.

**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

2 and 3

[**Take Hint (-15 XP)**](javascript:void(0))

> if (number < 10) {

+ if (number < 5) {

+ result <- "extra small"

+ } else {

+ result <- "small"

+ }

+ } else if (number < 100) {

+ result <- "medium"

+ } else {

+ result <- "large"

+ }

+ print(result)

Error: object 'number' not found

Error: object 'result' not found

>

> number <- 6

> if (number < 10) {

+ if (number < 5) {

+ result <- "extra small"

+ } else {

+ result <- "small"

+ }

+ } else if (number < 100) {

+ result <- "medium"

+ } else {

+ result <- "large"

+ }

+ print(result)

[1] "small"

>

> number <- 100

> if (number < 10) {

+ if (number < 5) {

+ result <- "extra small"

+ } else {

+ result <- "small"

+ }

+ } else if (number < 100) {

+ result <- "medium"

+ } else {

+ result <- "large"

+ }

+ print(result)

[1] "large"

>

> number <- 4

> if (number < 10) {

+ if (number < 5) {

+ result <- "extra small"

+ } else {

+ result <- "small"

+ }

+ } else if (number < 100) {

+ result <- "medium"

+ } else {

+ result <- "large"

+ }

+ print(result)

[1] "extra small"

>

> number <- 2500

> if (number < 10) {

+ if (number < 5) {

+ result <- "extra small"

+ } else {

+ result <- "small"

+ }

+ } else if (number < 100) {

+ result <- "medium"

+ } else {

+ result <- "large"

+ }

+ print(result)

[1] "large"

>

 +50 XP

Wonderful! If you got this one right, the next exercise will be a piece of cake.

**Exercise**

**Exercise**

**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!

In the editor, we've coded two values beforehand: 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 to the console.

[**Take Hint (-30 XP)**](javascript:void(0))

> # 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

> sms

[1] 24

>

 +100 XP

Bellissimo! Feel free to play around some more with your solution by changing the values of li and fb.

**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 code on the right; it initializes the speed variables and already provides a while loop template to get you started.

**Instructions**

**100 XP**

Code a while loop with the following characteristics:

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

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

[**Take Hint (-30 XP)**](javascript:void(0))

> # Initialize the speed variable

> speed <- 64

>

> # Code the while loop

> while (speed > 30) {

print("Slow down!")

speed <- speed - 7

}

[1] "Slow down!"

[1] "Slow down!"

[1] "Slow down!"

[1] "Slow down!"

[1] "Slow down!"

>

> # Print out the speed variable

> 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 in the editor. It prints out your current speed, but there's no code that decreases the speed variable yet, which is pretty dangerous. Can you make the appropriate changes?

**Instructions**

**100 XP**

* If the speed is greater than 48, have R print out "Slow down big time!", and decrease the speed by 11.
* Otherwise, have R simply print out "Slow down!", and decrease the speed by 6.

If the session keeps timing out and throwing an error, you are probably stuck in an infinite loop! Check the body of your while loop and make sure you are assigning new values to speed.

[**Take Hint (-30 XP)**](javascript:void(0))

> # 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] "Your speed is 64"

[1] "Slow down big time!"

[1] "Your speed is 53"

[1] "Slow down big time!"

[1] "Your speed is 42"

[1] "Slow down!"

[1] "Your speed is 36"

[1] "Slow down!"

>

 +100 XP

Wonderful! To further improve our driver assistant model, head over to the next exercise!

**Stop the while loop: break**

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

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

**Instructions**

**100 XP**

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

[**Take Hint (-30 XP)**](javascript:void(0))

> # 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] "Your speed is 88"

>

 +100 XP

Wonderful! Now that you've correctly solved this exercise, feel free to play around with different values of speed to see how the while loop handles the different cases.

**Build a while loop from scratch**

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

**Instructions**

**100 XP**

Finish the while loop so that it:

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

[**Take Hint (-30 XP)**](javascript:void(0))

**Incorrect Submission**

Your code contains a syntax error. Check the console output and try to fix the issue.

Have you correctly specified the if condition? Use if ((3 \* i) %% 8 == 0)

Parsing error in script.R:8:8: unexpected input

7:

8: if (i%8 == 0) {

^

> # Initialize i as 1

> i <- 1

>

> # Code the while loop

> while (i <= 10) {

print(3 \* i)

if (i%%8 == 0) {

break

}

i <- i + 1

}

[1] 3

[1] 6

[1] 9

[1] 12

[1] 15

[1] 18

[1] 21

[1] 24

> # Initialize i as 1

> i <- 1

>

> # Code the while loop

> while (i <= 10) {

i3 = 3 \* i

print(i3)

if (i3%%8 == 0) {

break

}

i <- i + 1

}

[1] 3

[1] 6

[1] 9

[1] 12

[1] 15

[1] 18

[1] 21

[1] 24

> # Initialize i as 1

> i <- 1

>

> # Code the while loop

> while (i <= 10) {

i3 = 3 \* i

print(i3)

if ((3 \* i) %% 8 == 0) {

break

}

i <- i + 1

}

[1] 3

[1] 6

[1] 9

[1] 12

[1] 15

[1] 18

[1] 21

[1] 24

>

**Exercise**

**Exercise**

**Loop over a vector**

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

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

# loop version 1

for (p in primes) {

print(p)

}

# loop version 2

for (i in 1:length(primes)) {

print(primes[i])

}

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

**Instructions**

**100 XP**

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

[**Take Hint (-30 XP)**](javascript:void(0))

> # The linkedin vector has already been defined for you

> linkedin <- c(16, 9, 13, 5, 2, 17, 14)

>

> # loop version 1

> for (li in linkedin) {

print(li)

}

[1] 16

[1] 9

[1] 13

[1] 5

[1] 2

[1] 17

[1] 14

>

> # loop version 2

> for (i in 1:length(linkedin)) {

print(linkedin[i])

}

[1] 16

[1] 9

[1] 13

[1] 5

[1] 2

[1] 17

[1] 14

>

**Loop over a list**

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

primes\_list <- list(2, 3, 5, 7, 11, 13)

# loop version 1

for (p in primes\_list) {

print(p)

}

# loop version 2

for (i in 1:length(primes\_list)) {

print(primes\_list[[i]])

}

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

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

**Instructions**

**100 XP**

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

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

[**Take Hint (-30 XP)**](javascript:void(0))

> # The nyc list is already specified

> nyc <- list(pop = 8405837,

boroughs = c("Manhattan", "Bronx", "Brooklyn", "Queens", "Staten Island"),

capital = FALSE)

>

> # loop version 1

> for (n in nyc) {

print(n)

}

[1] 8405837

[1] "Manhattan" "Bronx" "Brooklyn" "Queens"

[5] "Staten Island"

[1] FALSE

>

> # loop version 2

> for (i in 1:length(nyc)) {

print(nyc[[i]])

}

[1] 8405837

[1] "Manhattan" "Bronx" "Brooklyn" "Queens"

[5] "Staten Island"

[1] FALSE

>

 +100 XP

Good job! Filip mentioned that for loops can also be used for matrices. Let's put that to a test in the next exercise.

**Loop over a matrix**

In your workspace, there's a matrix ttt, that represents the status of a [**tic-tac-toe**](http://en.wikipedia.org/wiki/Tic-tac-toe) game. It contains the values "X", "O" and "NA". Print out ttt in the console so you can have a closer look. On row 1 and column 1, there's "O", while on row 3 and column 2 there's "NA".

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

for (var1 in seq1) {

for (var2 in seq2) {

expr

}

}

**Instructions**

**100 XP**

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

* The outer loop should loop over the rows, with loop index i (use 1:nrow(ttt)).
* The inner loop should loop over the columns, with loop index j (use 1:ncol(ttt)).
* Inside the inner loop, make use of [**print()**](http://www.rdocumentation.org/packages/base/functions/print) and [**paste()**](http://www.rdocumentation.org/packages/base/functions/paste) to print out information in the following format: "On row i and column j the board contains x", where x is the value on that position.

[**Take Hint (-30 XP)**](javascript:void(0))

**Incorrect Submission**

Check the first for statement. Check the body. Check the first for statement. Check the body. Have you called paste()?

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

# define the double for loop

for (i in 1:nrow(ttt)) {

for (j in 1:ncol(ttt)) {

if (!is.na(ttt[i,j])) {

if (ttt[i, j] == 'X')

{print(paste("On row", i, "and column", j, "the board contains x"))}

#{print(sprintf("On row %s and column %s the board contains x", i, j))}

}

}

}

**Hint**

* For the outer loop, use for (i in 1:nrow(ttt)).
* For the inner loop, use for (j in 1:ncol(ttt)).
* Inside the inner loop, use this expression to print the board info:

print(paste("On row", i, "and column", j, "the board contains", ttt[i,j]))

**Incorrect Submission**

The system couldn't find all the printouts that should have printed out. If you properly iterated over i and j, you can use print(paste("On row", i, "and column", j, "the board contains", ttt[i,j])) to make the appropriate printouts.

Answer:

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

# define the double for loop

for (i in 1:nrow(ttt)) {

for (j in 1:ncol(ttt)) {

print(paste("On row", i, "and column", j, "the board contains", ttt[i,j]))

}

}

[1] "On row 1 and column 1 the board contains O"

[1] "On row 1 and column 2 the board contains NA"

[1] "On row 1 and column 3 the board contains X"

[1] "On row 2 and column 1 the board contains NA"

[1] "On row 2 and column 2 the board contains O"

[1] "On row 2 and column 3 the board contains O"

[1] "On row 3 and column 1 the board contains X"

[1] "On row 3 and column 2 the board contains NA"

[1] "On row 3 and column 3 the board contains X"

>

 +0 XP

Awesome! You're sufficiently comfortable with basic for looping, so it's time to step it up a notch!

**Mix it up with control flow**

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

**Instructions**

**100 XP**

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

* If the vector element's value exceeds 10, print out "You're popular!".
* If the vector element's value does not exceed 10, print out "Be more visible!"

[**Take Hint (-30 XP)**](javascript:void(0))

> # The linkedin vector has already been defined for you

> linkedin <- c(16, 9, 13, 5, 2, 17, 14)

>

> # Code the for loop with conditionals

> for (li in linkedin) {

if (li <= 10) {

print("Be more visible!")

} else {

print("You're popular!")

}

print(li)

}

[1] "You're popular!"

[1] 16

[1] "Be more visible!"

[1] 9

[1] "You're popular!"

[1] 13

[1] "Be more visible!"

[1] 5

[1] "Be more visible!"

[1] 2

[1] "You're popular!"

[1] 17

[1] "You're popular!"

[1] 14

>

+100 XP

Outstanding! In the next exercise, you'll customize this for loop even further with break and next statements.

**Next, you break it**

In the editor on the right you'll find a possible solution to the previous exercise. The code loops over the linkedin vector and prints out different messages depending on the values of li.

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

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

**Instructions**

**100 XP**

Extend the for loop with two new, separate if tests in the editor as follows:

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

**Take Hint (-30 XP)**

> # The linkedin vector has already been defined for you

> linkedin <- c(16, 9, 13, 5, 2, 17, 14)

>

> # Adapt/extend the for loop

> for (li in linkedin) {

if (li > 10) {

print("You're popular!")

} else {

print("Be more visible!")

}

# Add if statement with break

if (li > 16) {

print("This is ridiculous, I'm outta here!")

break

}

# Add if statement with next

if (li < 5) {

print("This is too embarrassing!")

next

}

print(li)

}

[1] "You're popular!"

[1] 16

[1] "Be more visible!"

[1] 9

[1] "You're popular!"

[1] 13

[1] "Be more visible!"

[1] 5

[1] "Be more visible!"

[1] "This is too embarrassing!"

[1] "You're popular!"

[1] "This is ridiculous, I'm outta here!"

>

 +100 XP

Great. for, break, next? We name it, you can do it!

**Build a for loop from scratch**

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

In the editor on the right, we already went ahead and defined a variable rquote. This variable has been split up into a vector that contains separate letters and has been stored in a vector chars with the [**strsplit()**](http://www.rdocumentation.org/packages/base/functions/strsplit) function.

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

**Instructions**

**100 XP**

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

[**Take Hint (-30 XP)**](javascript:void(0))

**Incorrect Submission**

Your code contains a syntax error. Check the console output and try to fix the issue.

Do not forget to finish by printing out rcount.

Parsing error in script.R:10:19: unexpected ','

9: for (char in chars) {

10: if (char == "r"),

^

Parsing error in script.R:10:29: unexpected '='

9: for (char in chars) {

10: if (char == "r") {rcount +=

^

> # Pre-defined variables

> rquote <- "r's internals are irrefutably intriguing"

> chars <- strsplit(rquote, split = "")[[1]]

>

> # Initialize rcount

> rcount <- 0

>

> # Finish the for loop

> for (char in chars) {

if (char == "r") {rcount <- rcount+1}

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

}

>

> # Print out rcount

> # Pre-defined variables

> rquote <- "r's internals are irrefutably intriguing"

> chars <- strsplit(rquote, split = "")[[1]]

>

> # Initialize rcount

> rcount <- 0

>

> # Finish the for loop

> for (char in chars) {

if (char == "r") {rcount <- rcount+1}

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

}

>

> # Print out rcount

> rcount

[1] 5

>

 +100 XP

For-midable! This exercise concludes the chapter on while and for loops.

**Exercise**

**Exercise**

**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()**](http://www.rdocumentation.org/packages/base/functions/sample) function, for example, you can use one of following R commands:

help(sample)

?sample

If you execute these commands in the console of the DataCamp interface, you'll be redirected to www.rdocumentation.org.

A quick hack to see the arguments of the [**sample()**](http://www.rdocumentation.org/packages/base/functions/sample) function is the [**args()**](http://www.rdocumentation.org/packages/base/functions/args) function. Try it out in the console:

args(sample)

In the next exercises, you'll be learning how to use the [**mean()**](http://www.rdocumentation.org/packages/base/functions/mean) function with increasing complexity. The first thing you'll have to do is get acquainted with the [**mean()**](http://www.rdocumentation.org/packages/base/functions/mean) function.

**Instructions**

**100 XP**

* Consult the documentation on the [**mean()**](http://www.rdocumentation.org/packages/base/functions/mean) function: ?mean or help(mean).
* Inspect the arguments of the [**mean()**](http://www.rdocumentation.org/packages/base/functions/mean) function using the [**args()**](http://www.rdocumentation.org/packages/base/functions/args) function.

[**Take Hint (-30 XP)**](javascript:void(0))

# Consult the documentation on the mean() function

?mean

help(mean)

# Inspect the arguments of the mean() function

args(mean)

> ?sample

> ?gila

No documentation found in loaded packages and libraries.

> help

function (topic, package = NULL, lib.loc = NULL, verbose = getOption("verbose"),

try.all.packages = getOption("help.try.all.packages"), help\_type = getOption("help\_type"))

{

types <- c("text", "html", "pdf")

help\_type <- if (!length(help\_type))

"text"

else match.arg(tolower(help\_type), types)

if (!missing(package))

if (is.name(y <- substitute(package)))

package <- as.character(y)

if (missing(topic)) {

if (!is.null(package)) {

if (interactive() && help\_type == "html") {

port <- tools::startDynamicHelp(NA)

if (port <= 0L)

return(library(help = package, lib.loc = lib.loc,

character.only = TRUE))

browser <- if (.Platform$GUI == "AQUA") {

get("aqua.browser", envir = as.environment("tools:RGUI"))

}

else getOption("browser")

browseURL(paste0("http://127.0.0.1:", port, "/library/",

package, "/html/00Index.html"), browser)

return(invisible())

}

else return(library(help = package, lib.loc = lib.loc,

character.only = TRUE))

}

if (!is.null(lib.loc))

return(library(lib.loc = lib.loc))

topic <- "help"

package <- "utils"

lib.loc <- .Library

}

ischar <- tryCatch(is.character(topic) && length(topic) ==

1L, error = identity)

if (inherits(ischar, "error"))

ischar <- FALSE

if (!ischar) {

reserved <- c("TRUE", "FALSE", "NULL", "Inf", "NaN",

"NA", "NA\_integer\_", "NA\_real\_", "NA\_complex\_", "NA\_character\_")

stopic <- deparse(substitute(topic))

if (!is.name(substitute(topic)) && !stopic %in% reserved)

stop("'topic' should be a name, length-one character vector or reserved word")

topic <- stopic

}

paths <- index.search(topic, find.package(if (is.null(package))

loadedNamespaces()

else package, lib.loc, verbose = verbose))

try.all.packages <- !length(paths) && is.logical(try.all.packages) &&

!is.na(try.all.packages) && try.all.packages && is.null(package) &&

is.null(lib.loc)

if (try.all.packages) {

for (lib in .libPaths()) {

packages <- .packages(TRUE, lib)

packages <- packages[is.na(match(packages, .packages()))]

paths <- c(paths, index.search(topic, file.path(lib,

packages)))

}

paths <- paths[nzchar(paths)]

}

structure(unique(paths), call = match.call(), topic = topic,

tried\_all\_packages = try.all.packages, type = help\_type,

class = "help\_files\_with\_topic")

}

<bytecode: 0x49b99c8>

<environment: namespace:utils>

> help(help)

> # Consult the documentation on the mean() function

> ?mean

> help(mean)

>

> # Inspect the arguments of the mean() function

> args(mean)

function (x, ...)

NULL

>

 +100 XP

Great! That wasn't too hard, was it? Take a look at the documentation and head over to the next exercise.

**Use a function**

The documentation on the [**mean()**](http://www.rdocumentation.org/packages/base/functions/mean) function gives us quite some information:

* The [**mean()**](http://www.rdocumentation.org/packages/base/functions/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 defined in the editor on the right.

**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.

[**Take Hint (-30 XP)**](javascript:void(0))

Parsing error in script.R:6:7: unexpected ','

5: # Calculate average number of views

6: avg\_li,

^

> # 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

> avg\_li

[1] 10.85714

> avg\_fb

[1] 11.42857

>

 +100 XP

Nice! I'm sure you've already called more advanced R functions in your history as a programmer. Now you also know what actually happens under the hood ;-)

**Use a function (2)**

Check the documentation on the [**mean()**](http://www.rdocumentation.org/packages/base/functions/mean) function again:

?mean

The Usage section of the documentation includes two versions of the [**mean()**](http://www.rdocumentation.org/packages/base/functions/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**.

**Instructions**

**100 XP**

**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?

[**Take Hint (-30 XP)**](javascript:void(0))

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

> linkedin <- c(16, 9, 13, 5, 2, 17, 14)

> facebook <- c(17, 7, 5, 16, 8, 13, 14)

>

> # Calculate the mean of the sum

> avg\_sum = mean(linkedin+facebook)

>

> # Calculate the trimmed mean of the sum

> avg\_sum\_trimmed = mean(linkedin+facebook, trim=0.2)

>

> # Inspect both new variables

> avg\_sum

[1] 22.28571

> avg\_sum\_trimmed

[1] 22.6

>

>

 +100 XP

Nice! When the trim argument is not zero, it chops off a fraction (equal to trim) of the vector you pass as argument x.

**Use a function (3)**

In the video, Filip guided you through the example of specifying arguments of the [**sd()**](http://www.rdocumentation.org/packages/stats/functions/sd) function. The [**sd()**](http://www.rdocumentation.org/packages/stats/functions/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()**](http://www.rdocumentation.org/packages/base/functions/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.

[**Take Hint (-30 XP)**](javascript:void(0))

**Incorrect Submission**

For the second instruction, you should call mean() on linkedin, and in addition specify the na.rm argument; use na.rm = TRUE to do this.

> # 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)

[1] NA

>

> # Advanced average of linkedin

> mean(linkedin, na.rm=TRUE)

[1] 12.33333

>

**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()**](http://www.rdocumentation.org/packages/base/functions/print) and [**paste()**](http://www.rdocumentation.org/packages/base/functions/paste) functions use the ellipsis - ... - as an argument. Can you figure out how they're used?

**Instructions**

**100 XP**

**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!

[**Take Hint (-30 XP)**](javascript:void(0))

**Incorrect Submission**

The first argument you pass to mean() should be abs(linkedin - facebook). Make sure to also set na.rm = TRUE.

> # 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=1)

[1] 4.8

> # 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] 4.8

>

**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()**](http://www.rdocumentation.org/packages/base/functions/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()**](http://www.rdocumentation.org/packages/utils/functions/read.table) function are true?

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

**Instructions**

**50 XP**

**Instructions**

**50 XP**

**Possible Answers**

(1) and (3)

(2) and (4)

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

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

 +50 XP

Great! Using functions that are already available in R is pretty straightforward, but how about writing your own functions to supercharge your R programs? The next video will tell you how.

**Write your own function**

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

my\_fun <- function(arg1, arg2) {

body

}

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

**Instructions**

**100 XP**

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

[**Take Hint (-30 XP)**](javascript:void(0))

**Incorrect Submission**

Your code contains a syntax error. Check the console output and try to fix the issue.

Your code contains an error that you should fix:

Error: invalid (NULL) left side of assignment

> # Use the function

> sum\_abs(-2, 3)

Error: could not find function "sum\_abs"

> sumtest()

Error: could not find function "sumtest"

>

> # Create a function pow\_two()

> pow\_two <- function(num) {

return(num^2)

}

>

>

> # Use the function

> pow\_two(12)

[1] 144

>

> # Create a function sum\_abs()

> sum\_abs <- function(num1, num2) {

return (abs(num1)+abs(num2))

}

>

> # Use the function

> sum\_abs(-2, 3)

[1] 5

>

**Exercise**

**Exercise**

**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.

[**Take Hint (-30 XP)**](javascript:void(0))

**Incorrect Submission**

Inside the function body of hello(), use print("Hi there!").

> # Define the function hello()

> hello <- function() {"Hi there!"; return(TRUE)}

>

>

> # Call the function hello()

> hello()

[1] TRUE

> # Define the function hello()

> hello <- function() {print("Hi there!"); return(TRUE)}

>

>

> # Call the function hello()

> hello()

[1] "Hi there!"

[1] TRUE

>

 +100 XP

Truly impressive! Head over to the next exercise.

> # Finish the pow\_two() function

> pow\_two <- function(x, print\_info=1) {

y <- x ^ 2

if (print\_info) {print(paste(x, "to the power two equals", y))}

return(y)

}

>

**Exercise**

**Exercise**

**Write your own function (3)**

Do you still remember the difference between an argument with and without default values? Have another look at the [**sd()**](http://www.rdocumentation.org/packages/stats/functions/sd) function by typing ?sd in the console. The usage section shows the following information:

sd(x, na.rm = FALSE)

This tells us that x has to be defined for the [**sd()**](http://www.rdocumentation.org/packages/stats/functions/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?

**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.

[**Take Hint (-30 XP)**](javascript:void(0))

> # Finish the pow\_two() function

> pow\_two <- function(x, print\_info=1) {

y <- x ^ 2

if (print\_info) {print(paste(x, "to the power two equals", y))}

return(y)

}

>

> pow\_two(5)

[1] "5 to the power two equals 25"

[1] 25

> pow\_two(5, TRUE)

[1] "5 to the power two equals 25"

[1] 25

> pow\_two(5, FALSE)

[1] 25

>

 +100 XP

Wonderful! Have you tried calling this pow\_two() function? Try pow\_two(5), pow\_two(5, TRUE) and pow\_two(5, FALSE). Which ones give different results?

**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 course - x in this case.

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

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.

[**Take Hint (-15 XP)**](javascript:void(0))

 +50 XP

Great! If you're familiar with other programming languages, you might wonder whether R passes arguments *by value* or *by reference*. Find out in the next exercise!

**Exercise**

**Exercise**

**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.

In the last expression, the value of count was actually changed because of the explicit assignment.

[**Take Hint (-15 XP)**](javascript:void(0))

 +50 XP

Well done! Given that R passes arguments *by value* and not *by reference*, the value of count is not changed after the first two calls of increment(). Only in the final expression, where count is re-assigned explicitly, does the value of count change.

> increment <- function(x, inc = 1) {

+ x <- x + inc

+ x

+ }

+ count <- 5

+ a <- increment(count, 2)

+ b <- increment(count)

+ count <- increment(count, 2)

>

> count, a, b

unexpected ','

1: count,

^

> c(count, a, b)

[1] 7 7 6

> count

[1] 7

>

>

**R you functional?**

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

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

**Instructions**

**100 XP**

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

[**Take Hint (-30 XP)**](javascript:void(0))

**Incorrect Submission**

Make sure to use return(num\_views) inside the body of the if part of the control struct.

Make sure to use return(0) inside the body of the else part of the control struct.

Make sure to call interpret() on the second value of the facebook vector (facebook[2]).

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

>

> # Define the interpret function

> interpret <- function(num\_views) {

if (num\_views > 15) {

print("You're popular!")

return(num\_views)

} else {

print("Try to be more visible!")

return(0)

}

}

>

> # Call the interpret function twice

> interpret(linkedin[1])

[1] "You're popular!"

[1] 16

> interpret(facebook[2])

[1] "Try to be more visible!"

[1] 0

>

 +100 XP

Funkadelic! The annoying thing here is that interpret() only takes one argument. Proceed to the next exercise to implement something more useful.

**Exercise**

**Exercise**

**R you functional? (2)**

A possible implementation of the interpret() function is already available in the editor. 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.

**Instructions**

**100 XP**

**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.

[**Take Hint (-30 XP)**](javascript:void(0))

**Incorrect Submission**

Your code contains an error that you should fix:

Error: argument "return\_sum" is missing, with no default

> # 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=1) {

count <- 0

for (v in views) {

count <- count + interpret(v)

}

if (return\_sum) {

return (count)

} else {

return (NULL)

}

}

>

> # Call the interpret\_all() function on both linkedin and facebook

> interpret\_all(linkedin)

[1] "You're popular!"

[1] "Try to be more visible!"

[1] "Try to be more visible!"

[1] "Try to be more visible!"

[1] "Try to be more visible!"

[1] "You're popular!"

[1] "Try to be more visible!"

[1] 33

> interpret\_all(facebook)

[1] "You're popular!"

[1] "Try to be more visible!"

[1] "Try to be more visible!"

[1] "You're popular!"

[1] "Try to be more visible!"

[1] "Try to be more visible!"

[1] "Try to be more visible!"

[1] 33

>

+100 XP

Perfect! Have a look at the results; it appears that the sum of views on popular days are the same for Facebook and LinkedIn, what a coincidence! Your different social profiles must be fairly balanced ;-) Head over to the next video!

**Load an R Package**

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

* [**install.packages()**](http://www.rdocumentation.org/packages/utils/functions/install.packages), which as you can expect, installs a given package.
* [**library()**](http://www.rdocumentation.org/packages/base/functions/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()**](http://www.rdocumentation.org/packages/utils/functions/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()**](http://www.rdocumentation.org/packages/base/functions/library).

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

Before starting, execute the following commands in the console:

* search(), to look at the currently attached packages and
* qplot(mtcars$wt, mtcars$hp), to build a plot of two variables of the mtcars data frame.

An error should occur, because you haven't loaded the ggplot2 package yet!

**Instructions**

**100 XP**

* To fix the error you saw in the console, **load** the [**ggplot2**](http://www.rdocumentation.org/packages/ggplot2) package. Make sure you are *loading* (and not *installing*) the package!
* Now, retry calling the [**qplot()**](http://www.rdocumentation.org/packages/ggplot2/functions/qplot) function with the same arguments.
* Finally, check out the currently attached packages again.

[**Take Hint (-30 XP)**](javascript:void(0))

**Exercise**

**Exercise**

**Load an R Package**

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

* [**install.packages()**](http://www.rdocumentation.org/packages/utils/functions/install.packages), which as you can expect, installs a given package.
* [**library()**](http://www.rdocumentation.org/packages/base/functions/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()**](http://www.rdocumentation.org/packages/utils/functions/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()**](http://www.rdocumentation.org/packages/base/functions/library).

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

Before starting, execute the following commands in the console:

* search(), to look at the currently attached packages and
* qplot(mtcars$wt, mtcars$hp), to build a plot of two variables of the mtcars data frame.

An error should occur, because you haven't loaded the ggplot2 package yet!

**Instructions**

**100 XP**

* To fix the error you saw in the console, **load** the [**ggplot2**](http://www.rdocumentation.org/packages/ggplot2) package. Make sure you are *loading* (and not *installing*) the package!
* Now, retry calling the [**qplot()**](http://www.rdocumentation.org/packages/ggplot2/functions/qplot) function with the same arguments.
* Finally, check out the currently attached packages again.

[**Take Hint (-30 XP)**](javascript:void(0))

**Incorrect Submission**

Make sure to call the library() function to load the ggplot2 package.

Don't forget to finish off with the search() function to list the packages in memory.

There is something wrong in the following function call **search(ggplot2)**: *unused argument (ggplot2)*

> # Load the ggplot2 package

> install.packages(ggplot2)

Error: object 'ggplot2' not found

>

> # Retry the qplot() function

> qplot(ggplot2)

Error: could not find function "qplot"

>

> # Check out the currently attached packages again

> # Load the ggplot2 package

> install.packages(ggplot2)

Error: object 'ggplot2' not found

>

> # Retry the qplot() function

> qplot(mtcars$wt, mtcars$hp)

Error: could not find function "qplot"

>

> # Check out the currently attached packages again

> # Load the ggplot2 package

> install.packages(ggplot2)

Error: object 'ggplot2' not found

> library(ggplot2)

>

> # Retry the qplot() function

> qplot(mtcars$wt, mtcars$hp)

>

> # Check out the currently attached packages again

>

> # Load the ggplot2 package

> library(ggplot2)

>

> # Retry the qplot() function

> qplot(mtcars$wt, mtcars$hp)

>

> # Check out the currently attached packages again

> search()

[1] ".GlobalEnv" "package:ggplot2" "package:RBackend"

[4] "package:stats" "package:graphics" "package:grDevices"

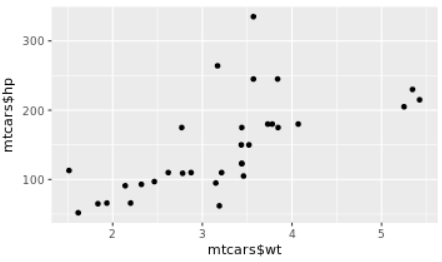
[7] "package:utils" "package:datasets" "package:methods"

[10] "Autoloads" "package:base"

>

 +100 XP

Awesome! Notice how search() and library() are closely interconnected functions. Head over to the next exercise.



**Different ways to load a package**

The [**library()**](http://www.rdocumentation.org/packages/base/functions/library) and [**require()**](http://www.rdocumentation.org/packages/base/functions/library) 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:

# Chunk 1

library(data.table)

require(rjson)

# Chunk 2

library("data.table")

require(rjson)

# Chunk 3

library(data.table)

require(rjson, character.only = TRUE)

# Chunk 4

library(c("data.table", "rjson"))

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

**Instructions**

**50 XP**

**Possible Answers**

Only (1)

(1) and (2)

(1), (2) and (3)

All of them are valid

Submit Answer

[**Take Hint (-15 XP)**](javascript:void(0))

 +50 XP

Great! Indeed, only chunk 1 and chunk 2 are correct. Can you figure out why the last two aren't valid? This exercise concludes the chapter on functions. Well done!

**Exercise**

**Exercise**

**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()**](http://www.rdocumentation.org/packages/base/functions/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!

**Instructions**

**100 XP**

* Have a look at the [**strsplit()**](http://www.rdocumentation.org/packages/base/functions/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()**](http://www.rdocumentation.org/packages/base/functions/lapply) to convert the character vectors in split\_math to lowercase letters: apply [**tolower()**](http://www.rdocumentation.org/packages/base/functions/chartr) 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()**](http://www.rdocumentation.org/packages/utils/functions/str).

[**Take Hint (-30 XP)**](javascript:void(0))

**Incorrect Submission**

Your code contains an error that you should fix:

Error: object 'split\_low' not found

Don't forget to display the structure of split\_low with str().

> pioneers

Error: object 'pioneers' not found

> pioneers <- c("GAUSS:1777", "BAYES:1702", "PASCAL:1623", "PEARSON:1857")

> strsplit(pioneers, split = ":")

[[1]]

[1] "GAUSS" "1777"

[[2]]

[1] "BAYES" "1702"

[[3]]

[1] "PASCAL" "1623"

[[4]]

[1] "PEARSON" "1857"

>

> split\_math <- strsplit(pioneers, split = ":")

> split\_math# <- strsplit(pioneers, split = ":")

[[1]]

[1] "GAUSS" "1777"

[[2]]

[1] "BAYES" "1702"

[[3]]

[1] "PASCAL" "1623"

[[4]]

[1] "PEARSON" "1857"

> # 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\_low, tolower)

Error: object 'split\_low' not found

>

> # Take a look at the structure of split\_low

> str(split\_low)

Error: object 'split\_low' not found

> # 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\_math)

List of 4

$ : chr [1:2] "GAUSS" "1777"

$ : chr [1:2] "BAYES" "1702"

$ : chr [1:2] "PASCAL" "1623"

$ : chr [1:2] "PEARSON" "1857"

> # 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)

List of 4

$ : chr [1:2] "gauss" "1777"

$ : chr [1:2] "bayes" "1702"

$ : chr [1:2] "pascal" "1623"

$ : chr [1:2] "pearson" "1857"

>

split\_low

[[1]]

[1] "gauss" "1777"

[[2]]

[1] "bayes" "1702"

[[3]]

[1] "pascal" "1623"

[[4]]

[1] "pearson" "1857"

> str(split\_low)

List of 4

$ : chr [1:2] "gauss" "1777"

$ : chr [1:2] "bayes" "1702"

$ : chr [1:2] "pascal" "1623"

$ : chr [1:2] "pearson" "1857"

>

**Use lapply with your own function**

As Filip explained in the instructional video, you can use [**lapply()**](http://www.rdocumentation.org/packages/base/functions/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()**](http://www.rdocumentation.org/packages/base/functions/lapply) just as you did with base R functions.

In the previous exercise you already used [**lapply()**](http://www.rdocumentation.org/packages/base/functions/lapply) once to convert the information about your favorite pioneering statisticians to a list of vectors composed of two character strings. Let's write some code to select the names and the birth years separately.

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

**Instructions**

**100 XP**

* Apply select\_first() over the elements of split\_low with [**lapply()**](http://www.rdocumentation.org/packages/base/functions/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.

[**Take Hint (-30 XP)**](javascript:void(0))

**Exercise**

**Exercise**

**Use lapply with your own function**

As Filip explained in the instructional video, you can use [**lapply()**](http://www.rdocumentation.org/packages/base/functions/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()**](http://www.rdocumentation.org/packages/base/functions/lapply) just as you did with base R functions.

In the previous exercise you already used [**lapply()**](http://www.rdocumentation.org/packages/base/functions/lapply) once to convert the information about your favorite pioneering statisticians to a list of vectors composed of two character strings. Let's write some code to select the names and the birth years separately.

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

**Instructions**

**100 XP**

* Apply select\_first() over the elements of split\_low with [**lapply()**](http://www.rdocumentation.org/packages/base/functions/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.

[**Take Hint (-30 XP)**](javascript:void(0))

**Incorrect Submission**

Your code contains an error that you should fix:

Error: object 'split\_low' of mode 'function' was not found

> # 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)

>

**Exercise**

**Exercise**

**lapply and anonymous functions**

Writing your own functions and then using them inside [**lapply()**](http://www.rdocumentation.org/packages/base/functions/lapply) is quite an accomplishment! But defining functions to use them only once is kind of overkill, isn't it? That's why you can use so-called **anonymous functions** in R.

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

# Named function

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

# Anonymous function with same implementation

function(x) { 3 \* x }

# Use anonymous function inside lapply()

lapply(list(1,2,3), function(x) { 3 \* x })

split\_low is defined for you.

**Instructions**

**100 XP**

* Transform the first call of [**lapply()**](http://www.rdocumentation.org/packages/base/functions/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.

[**Take Hint (-30 XP)**](javascript:void(0))

> # split\_low has been created for you

> split\_low

[[1]]

[1] "gauss" "1777"

[[2]]

[1] "bayes" "1702"

[[3]]

[1] "pascal" "1623"

[[4]]

[1] "pearson" "1857"

>

> # 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]

})

>

 +100 XP

Great! Now, there's another way to solve the issue of using the select\_\*() functions only once: you can make a more generic function that can be used in more places. Find out more about this in the next exercise.

**Exercise**

**Exercise**

**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()**](http://www.rdocumentation.org/packages/base/functions/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.

**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.

[**Take Hint (-30 XP)**](javascript:void(0))

**Incorrect Submission**

Your code contains an error that you should fix:

Error: object 'years' not found

Did you define the variable names without errors?

> # 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, 1)

> years <- lapply(split\_low, select\_el, 2)

>

 +100 XP

Awesome! Your lapply skills are growing by the minute!

**Exercise**

**Exercise**

**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()**](http://www.rdocumentation.org/packages/base/functions/chartr) 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()**](http://www.rdocumentation.org/packages/utils/functions/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()**](http://www.rdocumentation.org/packages/utils/functions/str) uses [**invisible()**](http://www.rdocumentation.org/packages/base/functions/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()**](http://www.rdocumentation.org/packages/utils/functions/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)

* 

list("1777", "1702")

Submit Answer

[**Take Hint (-15 XP)**](javascript:void(0))

 +50 XP

Wonderful! Feel free to experiment some more with your code in the console. Did you notice that lapply() *always* returns a list, no matter the input? This can be kind of annoying. In the next video tutorial you'll learn about sapply() to solve this.

> split\_low

[[1]]

[1] "gauss" "1777"

[[2]]

[1] "bayes" "1702"

[[3]]

[1] "pascal" "1623"

[[4]]

[1] "pearson" "1857"

> split\_low[1]

[[1]]

[1] "gauss" "1777"

> split\_low[1][1]

[[1]]

[1] "gauss" "1777"

> split\_low[1][2]

[[1]]

NULL

> split\_low[1][1] > 5

Error: (list) object cannot be coerced to type 'double'

> nchar(split\_low[1][1]) > 5

[1] TRUE

> nchar(split\_low[1][1])

[1] 18

> split\_low

[[1]]

[1] "gauss" "1777"

[[2]]

[1] "bayes" "1702"

[[3]]

[1] "pascal" "1623"

[[4]]

[1] "pearson" "1857"

> (split\_low[1][1])

[[1]]

[1] "gauss" "1777"

>

> lapply(split\_low, function(x) {

+ if (nchar(x[1]) > 5) {

+ return(NULL)

+ } else {

+ return(x[2])

+ }

+ })

[[1]]

[1] "1777"

[[2]]

[1] "1702"

[[3]]

NULL

[[4]]

NULL

>

> split\_low[1]

[[1]]

[1] "gauss" "1777"

> nchar(split\_low[1])

[1] 18

> x

Error: object 'x' not found

>

**Exercise**

**Exercise**

**How to use sapply**

You can use [**sapply()**](http://www.rdocumentation.org/packages/base/functions/lapply) similar to how you used [**lapply()**](http://www.rdocumentation.org/packages/base/functions/lapply). The first argument of [**sapply()**](http://www.rdocumentation.org/packages/base/functions/lapply) 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.

**Instructions**

**100 XP**

**Instructions**

**100 XP**

* Use [**lapply()**](http://www.rdocumentation.org/packages/base/functions/lapply) to calculate the minimum (built-in function [**min()**](http://www.rdocumentation.org/packages/base/functions/Extremes)) of the temperature measurements for every day.
* Do the same thing but this time with [**sapply()**](http://www.rdocumentation.org/packages/base/functions/lapply). See how the output differs.
* Use [**lapply()**](http://www.rdocumentation.org/packages/base/functions/lapply) to compute the the maximum ([**max()**](http://www.rdocumentation.org/packages/base/functions/Extremes)) temperature for each day.
* Again, use [**sapply()**](http://www.rdocumentation.org/packages/base/functions/lapply) to solve the same question and see how [**lapply()**](http://www.rdocumentation.org/packages/base/functions/lapply) and [**sapply()**](http://www.rdocumentation.org/packages/base/functions/lapply) differ.

[**Take Hint (-30 XP)**](javascript:void(0))

> temp

[[1]]

[1] 3 7 9 6 -1

[[2]]

[1] 6 9 12 13 5

[[3]]

[1] 4 8 3 -1 -3

[[4]]

[1] 1 4 7 2 -2

[[5]]

[1] 5 7 9 4 2

[[6]]

[1] -3 5 8 9 4

[[7]]

[1] 3 6 9 4 1

>

> # temp has already been defined in the workspace

>

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

> lapply(temp, min)

[[1]]

[1] -1

[[2]]

[1] 5

[[3]]

[1] -3

[[4]]

[1] -2

[[5]]

[1] 2

[[6]]

[1] -3

[[7]]

[1] 1

>

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

> sapply(temp, min)

[1] -1 5 -3 -2 2 -3 1

>

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

> lapply(temp, max)

[[1]]

[1] 9

[[2]]

[1] 13

[[3]]

[1] 8

[[4]]

[1] 7

[[5]]

[1] 9

[[6]]

[1] 9

[[7]]

[1] 9

>

> # Use sapply() to find each day's maximum temperature

> sapply(temp, max)

[1] 9 13 8 7 9 9 9

>

>

 +100 XP

Nice! Can you tell the difference between the output of lapply() and sapply()? The former returns a list, while the latter returns a vector that is a simplified version of this list. Notice that this time, unlike in the cities example of the instructional video, the vector is not named.

**Exercise**

**Exercise**

**sapply with your own function**

Like [**lapply()**](http://www.rdocumentation.org/packages/base/functions/lapply), [**sapply()**](http://www.rdocumentation.org/packages/base/functions/lapply) 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.

**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()**](http://www.rdocumentation.org/packages/base/functions/lapply) to apply it over the vectors inside temp.
* Use the same function over temp with [**lapply()**](http://www.rdocumentation.org/packages/base/functions/lapply) and see how the outputs differ.

[**Take Hint (-30 XP)**](javascript:void(0))

> # 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)

[1] 4.0 9.0 2.5 2.5 5.5 3.0 5.0

>

> # Apply extremes\_avg() over temp using lapply()

> lapply(temp, extremes\_avg)

[[1]]

[1] 4

[[2]]

[1] 9

[[3]]

[1] 2.5

[[4]]

[1] 2.5

[[5]]

[1] 5.5

[[6]]

[1] 3

[[7]]

[1] 5

>

 +100 XP

Great job! Of course, you could have solved this exercise using an anonymous function, but this would require you to use the code inside the definition of extremes\_avg() twice. Duplicating code should be avoided as much as possible!

**Exercise**

**Exercise**

**sapply with function returning vector**

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

**Instructions**

**100 XP**

* Finish the definition of the extremes() function. It takes a vector of numerical values and returns a vector containing the minimum and maximum values of a given vector, with the names "min" and "max", respectively.
* Apply this function over the vector temp using [**sapply()**](http://www.rdocumentation.org/packages/base/functions/lapply).
* Finally, apply this function over the vector temp using [**lapply()**](http://www.rdocumentation.org/packages/base/functions/lapply) as well.

[**Take Hint (-30 XP)**](javascript:void(0))

> # 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)

[,1] [,2] [,3] [,4] [,5] [,6] [,7]

min -1 5 -3 -2 2 -3 1

max 9 13 8 7 9 9 9

>

> # Apply extremes() over temp with lapply()

> lapply(temp, extremes)

[[1]]

min max

-1 9

[[2]]

min max

5 13

[[3]]

min max

-3 8

[[4]]

min max

-2 7

[[5]]

min max

2 9

[[6]]

min max

-3 9

[[7]]

min max

1 9

>

 +100 XP

Wonderful! Have a final look at the console and see how sapply() did a great job at simplifying the rather uninformative 'list of vectors' that lapply() returns. It actually returned a nicely formatted matrix!

**Exercise**

**Exercise**

**sapply can't simplify, now what?**

It seems like we've hit the jackpot with [**sapply()**](http://www.rdocumentation.org/packages/base/functions/lapply). On all of the examples so far, [**sapply()**](http://www.rdocumentation.org/packages/base/functions/lapply) was able to nicely simplify the rather bulky output of [**lapply()**](http://www.rdocumentation.org/packages/base/functions/lapply). But, as with life, there are things you can't simplify. How does [**sapply()**](http://www.rdocumentation.org/packages/base/functions/lapply) react?

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

**Instructions**

**100 XP**

* Apply below\_zero() over temp using [**sapply()**](http://www.rdocumentation.org/packages/base/functions/lapply) and store the result in freezing\_s.
* Apply below\_zero() over temp using [**lapply()**](http://www.rdocumentation.org/packages/base/functions/lapply). Save the resulting list in a variable freezing\_l.
* Compare freezing\_s to freezing\_l using the [**identical()**](http://www.rdocumentation.org/packages/base/functions/identical) function.

[**Take Hint (-30 XP)**](javascript:void(0))

> # 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)

[1] TRUE

> freezing\_s

[[1]]

[1] -1

[[2]]

numeric(0)

[[3]]

[1] -1 -3

[[4]]

[1] -2

[[5]]

numeric(0)

[[6]]

[1] -3

[[7]]

numeric(0)

> freezing\_l

[[1]]

[1] -1

[[2]]

numeric(0)

[[3]]

[1] -1 -3

[[4]]

[1] -2

[[5]]

numeric(0)

[[6]]

[1] -3

[[7]]

numeric(0)

>

 +100 XP

Nice one! Given that the length of the output of below\_zero() changes for different input vectors, sapply() is not able to nicely convert the output of lapply() to a nicely formatted matrix. Instead, the output values of sapply() and lapply() are exactly the same, as shown by the TRUE output of identical().

**Exercise**

**Exercise**

**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()**](http://www.rdocumentation.org/packages/base/functions/lapply) 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()**](http://www.rdocumentation.org/packages/base/functions/cat) function.

**Instructions**

**100 XP**

* Apply print\_info() over the contents of temp with [**sapply()**](http://www.rdocumentation.org/packages/base/functions/lapply).
* Repeat this process with [**lapply()**](http://www.rdocumentation.org/packages/base/functions/lapply). Do you notice the difference?

[**Take Hint (-30 XP)**](javascript:void(0))

 +100 XP

Great! Notice here that, quite surprisingly, sapply() does not simplify the list of NULL's. That's because the 'vector-version' of a list of NULL's would simply be a NULL, which is no longer a vector with the same length as the input. Proceed to the next exercise.

**Exercise**

**Exercise**

**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()**](http://www.rdocumentation.org/packages/base/functions/lapply) can't simplify the result that [**lapply()**](http://www.rdocumentation.org/packages/base/functions/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()**](http://www.rdocumentation.org/packages/base/functions/lapply) 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)

(1) and (4)

(2), (3) and (4)

[**Take Hint (-15 XP)**](javascript:void(0))

> runif

function (n, min = 0, max = 1)

.Call(C\_runif, n, min, max)

<bytecode: 0x2409218>

<environment: namespace:stats>

> runif(10)

[1] 0.93277014 0.63282340 0.86256250 0.74089066 0.09967479 0.17300498

[7] 0.74439825 0.93683439 0.24972539 0.84588585

> runif(100)

[1] 0.72854436 0.33491759 0.69691231 0.20636863 0.62186097 0.94269579

[7] 0.02579893 0.02686489 0.84490080 0.26265311 0.09662584 0.22868754

[13] 0.05141822 0.94827979 0.76678023 0.45104774 0.95886581 0.07392000

[19] 0.62507170 0.83370425 0.44775552 0.07100136 0.54527032 0.66102949

[25] 0.61795954 0.30087265 0.70180299 0.38462690 0.31987001 0.83237166

[31] 0.27714795 0.93153617 0.98004739 0.16329586 0.72154521 0.27191951

[37] 0.10160338 0.19686961 0.64386328 0.37731527 0.59602648 0.94711100

[43] 0.73866033 0.22899984 0.64140697 0.64885843 0.05062476 0.50882671

[49] 0.29759676 0.97619080 0.44284957 0.69468744 0.19869553 0.60028460

[55] 0.52570103 0.64500257 0.47132591 0.65205861 0.22266466 0.94110331

[61] 0.45950394 0.53620552 0.93164804 0.18788986 0.13517597 0.70334031

[67] 0.60622174 0.42277213 0.40004872 0.56007741 0.47683994 0.20441276

[73] 0.69880554 0.41866396 0.84103483 0.38216539 0.44529870 0.41649422

[79] 0.34017239 0.10072628 0.35973530 0.22475850 0.71529661 0.72613912

[85] 0.52549326 0.08594019 0.61024016 0.90968105 0.89029050 0.18839616

[91] 0.11999440 0.43164625 0.92199782 0.07999758 0.45160846 0.16884265

[97] 0.59411173 0.08539806 0.31294214 0.82843814

>

> sapply(list(runif (10), runif (10)),

+ function(x) c(min = min(x), mean = mean(x), max = max(x)))

[,1] [,2]

min 0.1353514 0.02524469

mean 0.5871551 0.41749332

max 0.9973402 0.89536998

>

>

 +50 XP

Great! This concludes the exercise set on sapply(). Head over to another video to learn all about vapply()!

**Exercise**

**Exercise**

**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()**](http://www.rdocumentation.org/packages/base/functions/lapply), 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()**](http://www.rdocumentation.org/packages/base/functions/lapply) tries to generate a named array, if possible.

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

**Instructions**

**100 XP**

* Apply the function basics() over the list of temperatures, temp, using [**vapply()**](http://www.rdocumentation.org/packages/base/functions/lapply). This time, you can use numeric(3) to specify the FUN.VALUE argument.

[**Take Hint (-30 XP)**](javascript:void(0))

> # 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))

[,1] [,2] [,3] [,4] [,5] [,6] [,7]

min -1.0 5 -3.0 -2.0 2.0 -3.0 1.0

mean 4.8 9 2.2 2.4 5.4 4.6 4.6

max 9.0 13 8.0 7.0 9.0 9.0 9.0

>

 +100 XP

Perfect! Notice how, just as with sapply(), vapply() neatly transfers the names that you specify in the basics() function to the row names of the matrix that it returns.

**Use vapply (2)**

So far you've seen that [**vapply()**](http://www.rdocumentation.org/packages/base/functions/lapply) mimics the behavior of [**sapply()**](http://www.rdocumentation.org/packages/base/functions/lapply) 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()**](http://www.rdocumentation.org/packages/base/functions/lapply) will throw an error that informs you about the misalignment between expected and actual output.

**Instructions**

**100 XP**

* Inspect the code on the right and try to run it. If you haven't changed anything, an error should pop up. That's because [**vapply()**](http://www.rdocumentation.org/packages/base/functions/lapply) 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()**](http://www.rdocumentation.org/packages/base/functions/lapply) command.

[**Take Hint (-30 XP)**](javascript:void(0))

> # 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))

[,1] [,2] [,3] [,4] [,5] [,6] [,7]

min -1.0 5 -3.0 -2.0 2.0 -3.0 1.0

mean 4.8 9 2.2 2.4 5.4 4.6 4.6

median 6.0 9 3.0 2.0 5.0 5.0 4.0

max 9.0 13 8.0 7.0 9.0 9.0 9.0

>

**Exercise**

**Exercise**

**From sapply to vapply**

As highlighted before, [**vapply()**](http://www.rdocumentation.org/packages/base/functions/lapply) can be considered a more robust version of [**sapply()**](http://www.rdocumentation.org/packages/base/functions/lapply), because you explicitly restrict the output of the function you want to apply. Converting your [**sapply()**](http://www.rdocumentation.org/packages/base/functions/lapply) expressions in your own R scripts to [**vapply()**](http://www.rdocumentation.org/packages/base/functions/lapply) expressions is therefore a good practice (and also a breeze!).

**Instructions**

**100 XP**

Convert all the [**sapply()**](http://www.rdocumentation.org/packages/base/functions/lapply) expressions on the right to their [**vapply()**](http://www.rdocumentation.org/packages/base/functions/lapply) counterparts. Their results should be exactly the same; you're only adding robustness. You'll need the templates numeric(1) and logical(1).

[**Take Hint (-30 XP)**](javascript:void(0))

> # temp is already defined in the workspace

>

> # Convert to vapply() expression

> vapply(temp, max, numeric(1))

[1] 9 13 8 7 9 9 9

>

> # Convert to vapply() expression

> vapply(temp, function(x, y) { mean(x) > y }, y = 5, logical(1))

[1] FALSE TRUE FALSE FALSE TRUE FALSE FALSE

>

 +100 XP

Great! You've got no more excuses to use sapply() in the future!

**Exercise**

**Exercise**

**Mathematical utilities**

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

* [**abs()**](http://www.rdocumentation.org/packages/base/functions/MathFun): Calculate the absolute value.
* [**sum()**](http://www.rdocumentation.org/packages/base/functions/sum): Calculate the sum of all the values in a data structure.
* [**mean()**](http://www.rdocumentation.org/packages/base/functions/mean): Calculate the arithmetic mean.
* [**round()**](http://www.rdocumentation.org/packages/base/functions/round): Round the values to 0 decimal places by default. Try out ?round in the console for variations of [**round()**](http://www.rdocumentation.org/packages/base/functions/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. The error values are already defined in the workspace on the right (errors).

**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.

[**Take Hint (-30 XP)**](javascript:void(0))

**Incorrect Submission**

Have a look at the highlighted code in the editor. Don't forget to close this parenthesis.

Incorrect. Have a look at the hint if you're not sure about the order of operations. Also, make sure that R prints the result!

Your code contains an error that you should fix:

Error: object 'errors' of mode 'function' was not found

Error: invalid 'type' (list) of argument

**Hint**

To know the order of operations, you should read the sentence in the instructions backwards: put [**round()**](http://www.rdocumentation.org/packages/base/functions/round) inside [**abs()**](http://www.rdocumentation.org/packages/base/functions/MathFun), and put the result of [**abs()**](http://www.rdocumentation.org/packages/base/functions/MathFun) in [**sum()**](http://www.rdocumentation.org/packages/base/functions/sum).

Parsing error in script.R:6:0: unexpected end of input

4: # Sum of absolute rounded values of errors

5: sum(abs(errors)

^

> # 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(errors))

[1] 28.7

> # 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(lapply(abs, errors))

Error: object 'errors' of mode 'function' was not found

> # 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(lapply(errors, abs))

Error: invalid 'type' (list) of argument

> abs(3)

[1] 3

> abs(-3)

[1] 3

> abs(-3)

[1] 3

> 3

[1] 3

> lapply(abs, errors)

Error: object 'errors' of mode 'function' was not found

> vapply(abs, errors)

Error: object 'errors' of mode 'function' was not found

> sapply(abs, errors)

Error: object 'errors' of mode 'function' was not found

> bapply(abs, errors)

Error: could not find function "bapply"

> sum(lapply(round, errors))

Error: object 'errors' of mode 'function' was not found

> # 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(lapply(round, errors))

Error: object 'errors' of mode 'function' was not found

> errors

[1] 1.9 -2.6 4.0 -9.5 -3.4 7.3

Parsing error in script.R:6:0: unexpected end of input

4: # Sum of absolute rounded values of errors

5: sum(abs(round(errors)

^

> # 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)))

[1] 29

>

**Find the error**

We went ahead and included some code on the right, but there's still an error. Can you trace it and fix it?

In times of despair, help with functions such as [**sum()**](http://www.rdocumentation.org/packages/base/functions/sum) and [**rev()**](http://www.rdocumentation.org/packages/base/functions/rev) are a single command away; simply use ?sum and ?rev in the console.

**Instructions**

**100 XP**

Fix the error by *including* code on the last line. Remember: you want to call [**mean()**](http://www.rdocumentation.org/packages/base/functions/mean) only once!

[**Take Hint (-30 XP)**](javascript:void(0))

**Incorrect Submission**

Your code contains an error that you should fix:

Error: 'trim' must be numeric of length one

> # 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), abs(vec2))

Error: 'trim' must be numeric of length one

> # 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(vec2))

[1] 4.48

>

+100 XP

Nice work! If you check out the documentation of mean(), you'll see that only the first argument, x, should be a vector. If you also specify a second argument, R will match the arguments by position and expect a specification of the trim argument. Therefore, merging the two vectors is a must!

**Exercise**

**Exercise**

**Data Utilities**

R features a bunch of functions to juggle around with data structures::

* [**seq()**](http://www.rdocumentation.org/packages/base/functions/seq): Generate sequences, by specifying the from, to, and by arguments.
* [**rep()**](http://www.rdocumentation.org/packages/base/functions/rep): Replicate elements of vectors and lists.
* [**sort()**](http://www.rdocumentation.org/packages/base/functions/sort): Sort a vector in ascending order. Works on numerics, but also on character strings and logicals.
* [**rev()**](http://www.rdocumentation.org/packages/base/functions/rev): Reverse the elements in a data structures for which reversal is defined.
* [**str()**](http://www.rdocumentation.org/packages/utils/functions/str): Display the structure of any R object.
* [**append()**](http://www.rdocumentation.org/packages/base/functions/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()**](http://www.rdocumentation.org/packages/base/functions/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 are already defined as lists on the right.

**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.

[**Take Hint (-30 XP)**](javascript:void(0))

**Incorrect Submission**

Your code contains an error that you should fix:

Error: could not find function "li\_vec.append"

Check your call of sort(). Did you specify the argument decreasing?

Your code contains an error that you should fix:

Error: 'decreasing' must be a length-1 logical vector. Did you intend to set 'partial'? ```

> linkedin

Error: object 'linkedin' not found

> linkedin <- list(16, 9, 13, 5, 2, 17, 14)

> c(linkedin <- list(16, 9, 13, 5, 2, 17, 14))

[[1]]

[1] 16

[[2]]

[1] 9

[[3]]

[1] 13

[[4]]

[1] 5

[[5]]

[1] 2

[[6]]

[1] 17

[[7]]

[1] 14

> c(linkedin)

[[1]]

[1] 16

[[2]]

[1] 9

[[3]]

[1] 13

[[4]]

[1] 5

[[5]]

[1] 2

[[6]]

[1] 17

[[7]]

[1] 14

> linkedin

[[1]]

[1] 16

[[2]]

[1] 9

[[3]]

[1] 13

[[4]]

[1] 5

[[5]]

[1] 2

[[6]]

[1] 17

[[7]]

[1] 14

> c(16, 9, 13, 5, 2, 17, 14)

[1] 16 9 13 5 2 17 14

> as.c(linkedin)

Error: could not find function "as.c"

> unlist(linkedin)

[1] 16 9 13 5 2 17 14

> # The linkedin and facebook lists have already been created for you

> linkedin <- list(16, 9, 13, 5, 2, 17, 14)

> facebook <- list(17, 7, 5, 16, 8, 13, 14)

>

> # Convert linkedin and facebook to a vector: li\_vec and fb\_vec

> li\_vec <- unlist(linkedin)

> fb\_vec <- unlist(facebook)

>

>

> # Append fb\_vec to li\_vec: social\_vec

> social\_vec <- li\_vec.append(fb\_vec)

Error: could not find function "li\_vec.append"

>

> # Sort social\_vec

> sort(social\_vec)

Error: object 'social\_vec' not found

> # The linkedin and facebook lists have already been created for you

> linkedin <- list(16, 9, 13, 5, 2, 17, 14)

> facebook <- list(17, 7, 5, 16, 8, 13, 14)

>

> # Convert linkedin and facebook to a vector: li\_vec and fb\_vec

> li\_vec <- unlist(linkedin)

> fb\_vec <- unlist(facebook)

>

>

> # Append fb\_vec to li\_vec: social\_vec

> social\_vec <- append(li\_vec, fb\_vec)

>

> # Sort social\_vec

> sort(social\_vec)

[1] 2 5 5 7 8 9 13 13 14 14 16 16 17 17

> decreasing

Error: object 'decreasing' not found

> # The linkedin and facebook lists have already been created for you

> linkedin <- list(16, 9, 13, 5, 2, 17, 14)

> facebook <- list(17, 7, 5, 16, 8, 13, 14)

>

> # Convert linkedin and facebook to a vector: li\_vec and fb\_vec

> li\_vec <- unlist(linkedin)

> fb\_vec <- unlist(facebook)

>

>

> # Append fb\_vec to li\_vec: social\_vec

> social\_vec <- append(li\_vec, fb\_vec)

>

> # Sort social\_vec

> sort(social\_vec, decreasing = 1)

Error: 'decreasing' must be a length-1 logical vector.

Did you intend to set 'partial'?

>

> # The linkedin and facebook lists have already been created for you

> linkedin <- list(16, 9, 13, 5, 2, 17, 14)

> facebook <- list(17, 7, 5, 16, 8, 13, 14)

>

> # Convert linkedin and facebook to a vector: li\_vec and fb\_vec

> li\_vec <- unlist(linkedin)

> fb\_vec <- unlist(facebook)

>

>

> # Append fb\_vec to li\_vec: social\_vec

> social\_vec <- append(li\_vec, fb\_vec)

>

> # Sort social\_vec

> sort(social\_vec, decreasing = TRUE)

[1] 17 17 16 16 14 14 13 13 9 8 7 5 5 2

>

+100 XP

Wonderful! These instructions required you to solve this challenge in a step-by-step approach. If you're comfortable with the functions, you can combine some of these steps into powerful one-liners.

**Exercise**

**Exercise**

**Find the error (2)**

Just as before, let's switch roles. It's up to you to see what unforgivable mistakes we've made. Go fix them!

**Instructions**

**100 XP**

Correct the expression. Make sure that your fix still uses the functions [**rep()**](http://www.rdocumentation.org/packages/base/functions/rep) and [**seq()**](http://www.rdocumentation.org/packages/base/functions/seq).

[**Take Hint (-30 XP)**](javascript:void(0))

**Incorrect Submission**

Your code contains an error that you should fix:

Error: 'from' must be of length 1

> # Fix me

> seq(rep(1, 7, by = 2), times = 7)

Warning message: In seq.default(rep(1, 7, by = 2), times = 7) :

extra argument 'times' will be disregarded

Error: 'from' must be of length 1

> # Fix me

> rep(seq(1, 7, by = 2), times = 7)

[1] 1 3 5 7 1 3 5 7 1 3 5 7 1 3 5 7 1 3 5 7 1 3 5 7 1 3 5 7

>

+100 XP

Wonderful! Debugging code is also a big part of the daily routine of a data scientist, and you seem to be great at it!

**Exercise**

**Exercise**

**Beat Gauss using R**

There is a popular story about young Gauss. As a pupil, he had a lazy teacher who wanted to keep the classroom busy by having them add up the numbers 1 to 100. Gauss came up with an answer almost instantaneously, 5050. On the spot, he had developed a formula for calculating the sum of an arithmetic series. There are more general formulas for calculating the sum of an arithmetic series with different starting values and increments. Instead of deriving such a formula, why not use R to calculate the sum of a sequence?

**Instructions**

**100 XP**

* Using the function [**seq()**](http://www.rdocumentation.org/packages/base/functions/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()**](http://www.rdocumentation.org/packages/base/functions/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()**](http://www.rdocumentation.org/packages/base/functions/sum) function twice and adding the two results, or by first concatenating the sequences and then using the [**sum()**](http://www.rdocumentation.org/packages/base/functions/sum) function once. Print the result to the console.

[**Take Hint (-30 XP)**](javascript:void(0))

> # Create first sequence: seq1

> seq1 <- seq(1, 500, 3)

>

> # Create second sequence: seq2

> seq2 <- seq(1200, 900, -7)

>

> # Calculate total sum of the sequences

> sum(seq1, seq2)

[1] 87029

> seq1

[1] 1 4 7 10 13 16 19 22 25 28 31 34 37 40 43 46 49 52

[19] 55 58 61 64 67 70 73 76 79 82 85 88 91 94 97 100 103 106

[37] 109 112 115 118 121 124 127 130 133 136 139 142 145 148 151 154 157 160

[55] 163 166 169 172 175 178 181 184 187 190 193 196 199 202 205 208 211 214

[73] 217 220 223 226 229 232 235 238 241 244 247 250 253 256 259 262 265 268

[91] 271 274 277 280 283 286 289 292 295 298 301 304 307 310 313 316 319 322

[109] 325 328 331 334 337 340 343 346 349 352 355 358 361 364 367 370 373 376

[127] 379 382 385 388 391 394 397 400 403 406 409 412 415 418 421 424 427 430

[145] 433 436 439 442 445 448 451 454 457 460 463 466 469 472 475 478 481 484

[163] 487 490 493 496 499

> seq2

[1] 1200 1193 1186 1179 1172 1165 1158 1151 1144 1137 1130 1123 1116 1109 1102

[16] 1095 1088 1081 1074 1067 1060 1053 1046 1039 1032 1025 1018 1011 1004 997

[31] 990 983 976 969 962 955 948 941 934 927 920 913 906

>

 +100 XP

Nice! Head over to the next video and learn more about regular expressions!

**Exercise**

**Exercise**

**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()**](http://www.rdocumentation.org/packages/base/functions/grep), which returns TRUE when a pattern is found in the corresponding character string.
* [**grep()**](http://www.rdocumentation.org/packages/base/functions/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 already been defined in the editor on the right so you can begin with the instructions straight away!

**Instructions**

**100 XP**

* Use [**grepl()**](http://www.rdocumentation.org/packages/base/functions/grep) to generate a vector of logicals that indicates whether these email addressess contain "edu". Print the result to the output.
* Do the same thing with [**grep()**](http://www.rdocumentation.org/packages/base/functions/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".

[**Take Hint (-30 XP)**](javascript:void(0))

**Incorrect Submission**

Have you correctly selected the emails that match the pattern? You can use emails[hits]. Simply print out the result!

> # 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)

[1] TRUE TRUE FALSE TRUE TRUE FALSE

>

> # Use grep() to match for "edu", save result to hits

> hits <- grep('edu', emails)

>

> # Subset emails using hits

> sub('edu', emails, hits)

Warning message: argument 'replacement' has length > 1 and only the first element will be used

[1] "1" "2" "4" "5"

> # 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)

[1] TRUE TRUE FALSE TRUE TRUE FALSE

>

> # Use grep() to match for "edu", save result to hits

> hits <- grep('edu', emails)

>

> # Subset emails using hits

> emails[hits]

[1] "john.doe@ivyleague.edu" "education@world.gov"

[3] "invalid.edu" "quant@bigdatacollege.edu"

>

+100 XP

Bellissimo! You can probably guess what we're trying to achieve here: select all the emails that end with “.edu”. However, the strings education@world.gov and invalid.edu were also matched. Let's see in the next exercise what you can do to improve our pattern and remove these false positives.