# Grades analysis in R

70xp

Suppose you're a student in a prestigious academy that only allows the country's best 200 students every year. Just for fun, you decide to analyze the grades of this academy. The highest possible grade is 100, the lowest possible grade is 0.

You have access to your own grades, to anonymized grades for all of your 199 classmates, and for the 200 pupils in the previous 4 years. This data is available in your workspace as `me`, `other_199`, and `previous_4`, respectively. Have a look at these variables in the console.
To format our data so that it'd be easier to work with and analyze later on, let's merge the three datasets to one called `last_5`, with dimensions 200 by 5. That is, `last_5` contains all 200 scores from the last 5 classes.

## Instructions

- Use `c()` to merge the data in `me` and `other_199` (in this order). Store the result in `my_class`.
- Build a 200-by-5 matrix named `last_5`: use `cbind()` to paste together the vector `my_class` and the 200-by-4 matrix `previous_4`.
- Use `colnames()` on `last_5` to give the columns the names `year_1`, up to `year_5`, in this order. To get you started, a character vector `nms` that you can use is already available.

```
me, other_199, and previous_4 are available in your workspace

# Merge me and other_199: my_class

my_class <- c(me,other_199)

# cbind() my_class and previous_4: last_5

last_5 <- cbind(my_class,previous_4)

# Name last_5 appropriately

colnames(last_5) <- c("year_1","year_2","year_3","year_4","year_5")

nms <- paste0("year_", 1:5)
```

## Explore your data

100xp

To get a good feel for your data, it's a good idea to make some visualizations and make some summaries.

The object `me` is still preloaded, as are the objects `my_class` and `last_5` that you built yourself. Up to you to write some exploration code!

## Instructions

- Use `hist()` to create a histogram of `my_class`. Can you tell in which bin your grade (the variable `me`) is?
- Generate a summary of `last_5`. What do these numbers tell you?
- Maybe a boxplot is more informative here; use `boxplot()` on `last_5` and try to read the resulting graph.

```
# me, my_class and last_5 are available in your workspace


# Build histogram of my_class

hist(my_class)


# Generate summary of last_5

summary(last_5)


# Build boxplot of last_5

boxplot(last_5)
```

## Basic queries

100xp

Remember relational operators? Here's a short list to freshen your memory:

- `==` - Equality
- `!=` - Inequality
- `>` - Greater than
- `<` - Less than
- `>=` - Greater than or equal to
- `<=` - Less than or equal to

In this exercise, you'll be writing queries to answer questions about `me`, `my_class` and `last_5`. These variables are available in your workspace. **The result of your queries should be a vector or a matrix of logicals.**

## Instructions

- Is your grade equal to 72?
- Which grades in your class are higher than 75?
- Which grades in the last 5 years are below or equal to 64?

```
# me, my_class and last_5 are preloaded


# Is your grade equal to 72?

me == 72


# Which grades in your class are higher than 75?

my_class > 75


# Which grades in the last 5 years are below or equal to 64?

last_5 <= 64
```

## Build aggregates

100xp

Answering the question *which grades in your class are higher than 75?* with a vector of logicals is not very insightful. It's much better to ask the question *how many grades in your class are higher than 75?* instead.

You can answer such questions with the `sum()` function. Each `TRUE` you passs count as `1`, each `FALSE` as 0. Just in the same way, you can use `mean()`; this will give you the proportion of `TRUE` values in the data structure you pass it.

## Instructions

- How many grades in your class are higher than 75?
- How many students in your class scored strictly higher than you?
- What's the proportion of grades below or equal to 64 in the last five years?

```
# me, my_class and last_5 are preloaded
```

# How many grades in your class are higher than 75?

sum(my_class > 75)

# How many students in your class scored strictly higher than you?

sum(my_class > me)

# What's the proportion of grades below or equal to 64 in the last 5 years?

mean(last_5 <= 64)

## Logical operator

100xp

Next to relational operators, there are also logical operators, to combine logicals:

- `&` - AND
- `|` - OR
- `!` - NOT

Try to answer the following questions such that the output of your code is logical:

## Instructions

- Is your grade greater than 87 and smaller than or equal to 89?
- Which grades in your class are below 60 or above 90?

# me, my_class and last_5 are preloaded

# Is your grade greater than 87 and smaller than or equal to 89?

me > 87 & me <= 89

# Which grades in your class are below 60 or above 90?

my_class < 60 | my_class > 90

## Build aggregates (2)

100xp

This exercise sums it all up: use relational operators, logical operators, and functions such as `sum()` and `mean()` to come up with the solution!

## Instructions

- What's the proportion of grades in your class that is average, i.e. greater than or equal to 70 and lower than or equal to 85?
- How many students in the last 5 years had a grade of exactly 80 or 90?

```
# me, my_class and last_5 are preloaded



# What's the proportion of grades in your class that is average?

mean(my_class >= 70 & my_class <=85)



# How many students in the last 5 years had a grade of 80 or 90?

sum(last_5 ==80 | last_5 ==90)
```

## if, else

50xp

As a refresher of the syntax, have a look at this example:

```
if (me > 80) {
  print("Good student!")
} else {
  print("Better luck next year!")
}
```

Your grade, `me`, equals 89, so the condition in the `if` statement evaluates to `TRUE`, and `print("Good student!")` is executed.

*Note that the `else` part should come on the same line as the closing bracket of the `if` statement! If you don't do this, R will not understand your code!*

What is the output if this control structure is run in case `me` equals 80?

## if, else: DIY

100xp

With the insight acquired from the previous exercise, coding your own `if`-`else` statement will be a walk in the park.

The variables you have been working with all along - `me`, `my_class`, and `last_5` - are available in your workspace.

*Note that the `else` part should come on the same line as the closing bracket of the `if` statement! If you don't do this, R will not understand your code!*

## Instructions

- Assign to `n_smart` the **number** of grades in `my_class` that are greater than or equal to 80.
- Write an `if` statement that prints out `"smart class"` if `n_smart` is greater than 50.
- Extend the `if` statement with an `else` clause that prints out `"rather average"`.

```
# me, my_class and last_5 are preloaded


# Define n_smart

n_smart <- sum(my_class >= 80)


# Code the if-else construct

if (n_smart > 50) {

  print("smart class")

} else {

  print("rather average")

}
```

## else if

100xp

You can further customize your `if`-`else` constructs with an `else if` statement:

```
if (condition) {
  expr
} else if (condition) {
  expr
} else {
  expr
}
```
Remember here that as soon as R encounters a `condition` that evaluates to `TRUE`, the corresponding `expr` is executed and the control structure is abandoned.

*Note that the `else if` and `else` parts should come on the same line as the closing bracket of the previous statement! If you don't do this, R will not understand your code!*

## Instructions

- Assign to `prop_less` the **proportion** of students whose grade - stored in `my_class` - was lower than yours.
- Write a control construct with the following properties:
    - if `prop_less` exceeds 0.9, print out `"you're among the best 10 percent"`.
    - if the above is not the case, but `prop_less` exceeds 0.8, print out `"you're among the best 20 percent"`.
    - if the above two don't hold, print out `"need more analysis"`.

```
# me, my_class and last_5 are preloaded

# Define prop_less

prop_less <- mean(my_class < me)

# Code the control construct
if (prop_less > 0.9) {
 print("you're among the best 10 percent")
 } else if (prop_less > 0.8){
  print("you're among the best 20 percent")
 } else {
  print("need more analysis")
 }
```

## Embed if-else clauses

100xp

An example of embedded control structures is included in the sample code. There's a top-level `if-else` construct and there are other `if-else` constructs inside the statements. However, there's still something wrong here.

## Instructions

Fix the control structure such that:

- The body of the top-level `if` condition is executed if the average score of your class is strictly below 75.
- There are no syntax errors.

## Operations and controls expertise

70xp

Thinking about your time at school, you remember the wide range of students in your class. In this exercise, let's try blending everything together to see whether there were more high achievers than low achievers in your class. You will not receive too much help from the feedback messages... you're pretty much on your own here!

## Instructions

- Create a sub-vector of `my_class` that only contains the grades that are greater than or equal to 85. Call this vector `top_grades`.
- Create a similar sub-vector, but this time with the grades of `my_class` that are strictly under 65. Call this vector `worst_grades`.
- Create a conditional statement that prints out `"top grades prevail"` if the length of `top_grades` exceeds that of `worst_grades`. Don't include an `else` statement.

```
# me, my_class and last_5 are preloaded


# Create top_grades

top_grades <- my_class[my_class >= 85]


# Create worst_grades

worst_grades <- my_class[my_class < 65]


# Write conditional statement

if (length(top_grades) > length(worst_grades)) {
  print("top grades prevail")
}
```

## Scanning Logs in R

100xp

Imagine you're a data scientist in a huge chemical company. Business is going well, but in the last couple of days, some unexpected errors occurred at your plant. You decide to dig through the log files from the last 4 days. You find that approximately every hour, the plant's monitoring system has produced a control message. It is up to you to analyze it in R.

The data is stored as a list `logs`, which is loaded in your workspace. Let's do some exploring.

## Instructions

- Display the structure of `logs`. It appears to be a list of lists, interesting.
- Use list subsetting to print out the `details` component of the 11th element of `logs`.
- Use `class()` to print out the class of the `timestamp` component of the first entry of `logs`.

```
# logs is already available in your workspace


# Print the structure of logs

str(logs)


# Use list subsetting to print the details part of 11th logs entry

logs[[11]]$details

# Print the class of the timestamp component of the first entry

class(logs[[1]][["timestamp"]])
```

## While: start easy

100xp

As a refresher, have a look at this `while` loop recipe:

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

The `expr` part gets executed over and over again, as long as `condition` evaluates to `TRUE`. Remember that `condition` should become `FALSE` at one point, otherwise your loop will go on indefinitely!
As a data scientist, suppose you want to know how many entries there were before the first failure message. One way to go about this is to write a simple `while` loop that goes through each log entry in `logs`, prints out the entry number, and stops when it sees the first failure message, that is, when the `success` element is `FALSE`. The last number printed to the console is the number of entries before the first failure message was logged.

## Instructions

- Start with defining an iterator, `i`, equal to 1, outside of the `while` loop.
- Inside the `while` loop's `condition`, check if the `success` element of `logs[[i]]` is `TRUE`.
- Inside the `while` loop's `condition`:
  - First print out `i` and
  - Next, increase `i` by 1. This is important!

```
# logs is available in your workspace



# Initialize the iterator i to be 1



i <-1

# Code the while loop

while (logs[[i]]$success == TRUE ) {

 print(i)

 i<- i+1



}
```

## Adapt the while loop

100xp

In the previous exercise, you iterated through the logs until you found a log that indicates a failure, i.e. the `success` component is `FALSE`. Here, you'll further extend this `while` loop so that you can read the message for every log entry before the first failure; maybe there were warning messages that we've missed! Instead of simply printing `i`, print out the `message`, available inside the `details` element of each log entry.
The sample code already contains the solution to the previous exercise. It's up to you to make the appropriate changes. You can always use `str(logs)` to have a look at the structure of `logs`.

## Instructions

- Do not change how `i` is initialized or updated.
- Change the `print(i)` call with a call that prints the `message` element of the `details` element of each log entry.

```
# logs is available in your workspace



# Adapt the while loop

i <- 1

while (logs[[i]]$success) {

   print(logs[[i]]$details$message)

 i <- i + 1



}
```

## While: different approach

100xp

To answer more complicated questions, you'll have to work with additional temporary variables that indicate whether you found what you're looking for while going through the logs. This temporary variable comes in addition to the iterator, `i`, that you use to access subsequent `log` entries.

Suppose you have a meeting with your colleagues from other sectors of the company and find there may be problems in the waste department. To investigate this, you need to search through the `while` loop with the iterator `i` again, but this time you need to keep looking until you find a failure that occurred at the waste department, that is, where `location` inside the `details` element of the log equals `"waste"`. Follow the instructions step-by-step to get there.

The ___ parts in the sample code should be replaced with valid R code.

## Instructions

- Initialize two variables: `i`, to 1 as before, and `found`, to `FALSE`.
- Write a `while` loop that:
  - keeps running as long as `found` is `FALSE`.
  - Checks if `logs[[i]]$success` is `FALSE` *and* `logs[[i]]$details$location` equals `"waste"`. You'll need to use the `&&` sign here. This causes the evaluation of the condition to halt as soon as the result of the condition is known; if `logs[[i]]$success` is `TRUE`, it's certain that the condition will be `FALSE`, so the next comparison is not evaluated anymore.

- prints out `"found"` and sets `found` to `TRUE` if the above condition holds.
- prints out `"still looking"` and increment `i` if the above condition does not hold.

```
# logs is available in your workspace


# Initialize i and found

i <- 1

found <- FALSE


# Code the while loop

while (found == FALSE) {

  if ((logs[[i]]$success == FALSE) && (logs[[i]]$details$location == "waste")) {

    print("found")

    found <- TRUE

  } else {

  print("still looking")

  i <- i+1


  }

}
```

## The for loop

100xp

Let's do a quick review. The `while` loop keeps executing code until its condition evaluates to `FALSE`. The `for` loop, on the other hand, iterates over a sequence, where a looping variable changes for each iteration, according to the sequence. Have a look at the following that prints the value of each element in `vec` in two different ways:

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

# Option 1
for (el in vec) {
  print(el)
}
```

```
# Option 2
for (i in 1:length(vec)) {
  print(vec[i])
}
```

In this exercise and in the following exercises, you'll again be working with the chemical plant logs that's loaded in your workspace as `logs`. Here, you'll write a `for` loop that prints when each log entry was logged.

## Instructions

Build a `for` loop that separately prints the `timestamp` element of each log entry in the `logs` list. You can do this in several ways:

- By looping over `logs`: `for(log in logs)`. At each iteration, `log` will be an entry of the `logs` list.
- By using a looping index: `for(i in 1:length(logs))`. You can then use `logs[[i]]` to access the log entry.

```
# Code a for loop that prints the timestamp of each log

for (log in logs) {

  print(log$timestamp)

}
```

## Going through the list

100xp

You can use an `if`-`else` construct inside a `for` loop. This allows you to loop through the log entries and perform more specific tasks.
In the sample code on the right, a `for` loop that solved the previous exercise is included. It's up to you to extend it so that you know when each successful log entry was logged.

## Instructions

Add an `if` statement such that the `for` loop only prints the `timestamp` if the `log` in question represents a success, i.e. if its `success` element is `TRUE`.

```
# logs is available in your workspace



# Make the printout conditional: only if success

for (log in logs) {
```

```
if (log$success == TRUE){
  print(log$timestamp)  }
}
```

## Adapt the logs list

100xp

Apart from doing simple printouts in a `for` loop, remember that it's possible to adapt data structures inside a `for` loop.
As a data scientist, you know it's always helpful to convert any time-related data to a `Date` object. So, in this case, we want to convert the `timestamp` to a `Date` object. How could we go about this? Using the `for (log in logs)` approach here won't work, because `log` is a local copy of an element in `logs`. To actually access and change the elements in the `logs` list, you will need to use the *looping index*.
The sample code already includes a `for` loop without a body, can you complete it?

## Instructions

- Finish the `for` loop in the sample code; after running it, each entry in `logs` should contain a `date` element. This should b a `Date` object that you can build from the `timestamp` element with `as.Date()`.
- Print the first 6 elements in `logs` by calling the `head()` function on `logs`. Is the `date` information in there now?

```
# logs is available in your workspace


# Finish the for loop: add date element for each entry
for (i in 1:length(logs)) {
  logs[[i]]$date <- as.Date(logs[[i]]$timestamp)
}


# Print first 6 elements in logs
head(logs)
```

## Collect all failures

100xp

In the previous exercise, you adapted the data structure you were looping over. Remember, it's also possible to build a new data structure altogether inside your `for` loop.

Your plant manager approaches you and asks for a report on all failures that are available in the `logs` list. Instead of the entire list, she is only interested in the failures. Get to work to generate what she askes for!

*Just a tip before you get to it*: If you have a list of lists `a` and want to add a list `b` to it, you can use `c(a, list(b))`.

## Instructions

- Initialize an empty list, `failures` using the `list()` function without arguments.
- Finish the `for` loop such that each log entry that indicates a failure is added to `failures`.
- Display the structure of the `failures` list that results.

```
# logs is available in your workspace


# Intialize empty list: failures

failures <- list()


# Finish the for loop: add each failure to failures

for (log in logs) {

  if (log$success == FALSE) {

    failures <- c(failures, list(log))

  }

}


# Display the structure of failures

str(failures)
```

## Using functions

100xp

Recall that when you call a function, R matches your input parameters with its function arguments, either by value or by position, then executes the function body. Function arguments can have default values: if you do not specify these arguments, R will take the default value.

Remember the chemical plant logs stored as `logs`? `logs` is a list of log entries; each entry itself is also a list containing a bunch of information regarding measurements in the plant. `logs` is still available in your workspace so feel free to refresh your memory.
You wonder when the most recent failure was logged. The `timestamps` vector, which contains the timestamps of logs indicating a failure, is already available in your workspace. It's up to you to use the `max()` function to find the most recent timestamp. You can check the function's documentation by typing `?max` in the console.

## Instructions

- Call `max()` on `timestamps` to find the latest timestamp.
- Write a one-liner that converts the latest timestamp to a `Date` object. You can use `as.Date()` for this.

```
# logs is available in your workspace



# Call max() on timestamps

max(timestamps)



# What is the date of the latest timestamp?

as.Date(max(timestamps))
```

## Optional Arguments

100xp

Due to some irregularities in the logging system, some of the timestamps for failures are missing. The vector `timestamps` that was gathered from all failures is available in your workspace. You will see that some timestamps are `NA`, which is R's way of denoting a missing value.
`max()` has a default way of handling `NA` values in a vector: it simply returns `NA`. You can have `max()` ignore `NA` values by controlling the `na.rm` argument.

## Instructions

- Inspect the contents of `timestamps` by simply printing them.
- Call `max()` on `timestamps` without specifying additional arguments.
- Again call `max()` on `timestamps` but set the `na.rm` argument appropriately. What is the difference?

```
# A faulty version of timestamps is available in your workspace
```

# Print out timestamps

print(timestamps)

# Call max() on timestamps, no additional arguments

max(timestamps)

# Call max() on timestamps, specify na.rm

max(timestamps,na.rm = TRUE)

## Which call is valid?

50xp

Because of the way positional and named arguments work in R, you can call functions in several ways.

Take the `nchar()` function, for example. You can use this function to count the number of characters in a character vector, such as in `vec` that is available in your workspace.
Only one of the calls of `nchar()` below is valid; can you guess which one? Have a look at the function's documentation first with

```
?nchar
```

before trying each command in the console.

## Extract log information (1)

100xp

By now you've seen quite some examples of how data is extracted from the `logs` list with a `for` loop. As the plant's data scientist, you'll often want to extract information from the list, be it the messages, the timestamp, whether or not it was a success, etc. You could write a dedicated `for` loop for each one of them, but this is rather tedious. Why not wrap it in a function? Let's start simple and make your function more awesome step-by-step.
Remember you can define your own function as follows:

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

The `for` loop that extracts the timestamp information for each log is included on the right. This time, the `logs` list doesn't contain any missing information like it did in the previous exercises.

## Instructions

- Skeleton code for `extract_info()` is already provided. This function should take a list of logs as the input (the `x` argument), and `return()` a vector with all the logs' timestamps. Place the `for` loop that has already been coded in the function body. You'll also have to rename variables to make it work. Good luck!
- Call `extract_info()` on `logs`; simply print out the result, no need to store it in a variable.

```
# logs is available in your workspace


# for loop to extract timestamp; put this inside function body below




# Build a function extract_info(): use for loop, add return statement

extract_info <- function(x) {

info <- c()

  for (log in x) {

 info <- c(info, log$timestamp)


}

return(info)

}


# Call extract_info() on logs

extract_info(logs)
```

# Extract log information (2)

100xp

In the previous exercise, you wrote a function `extract_info()`, which is available on the right. To make this function really powerful, you'll want to add an additional argument, `property`, so that you can select any property (ie. `success`, `details`, or `timestamp`) from the log entries.

Next, you can use this argument to subset the list accordingly. You cannot use the `$` notation if the element you want to select is a variable and not the actual name of a list:

```
log$property # won't work
log[[property]] # will work
```

## Instructions

- Add an additional argument to the function, called `property`.
- Adapt the function body such that `property` is used to select the correct information from each log.
- Call `extract_info()` on `logs` and set the `property` argument to `"timestamp"`.
- Call `extract_info()` on `logs` and set the `property` argument to `"success"`.

```
# logs is available in your workspace


# Adapt the extract_info() function.

extract_info <- function(x, property) {

  info <- c()

  for (log in x) {

   info <- c(info, log[[property]])

  }

  return(info)

}


# Call extract_info() on logs, set property to "timestamp"

extract_info(logs,"timestamp")


# Call extract_info() on logs, set property to "success"

extract_info(logs,"success")
```

# Extract log information (3)

100xp

The `property` argument in the previous exercises did not have a default value. This causes R to throw an error if you call `extract_info()` without specifying the `property` argument. Time to add this default value and see how your function behaves.

## Instructions

- In the definition of `extract_info()`, set the default value of the `property` argument to `"success"`.
- Call `extract_info()` on `logs` without specifying the `property` argument.
- Call `extract_info()` on `logs`, and set the `property` argument to `"timestamp"`.

```
# logs is available in your workspace


# Add default value for property argument

extract_info <- function(x, property = "success") {

  info <- c()

  for (log in x) {

   info <- c(info, log[[property]])

  }

  return(info)

}


# Call extract_info() on logs, don't specify property

extract_info(logs)


# Call extract_info() on logs, set property to "timestamp"

extract_info(logs, "timestamp")
```

## Extract log information (4)

100xp

In the `for` loop exercises, you wrote code that extracts information on log entries that indicate a failure. This is something your `extract_info()` function can not yet do. You can already guess what the purpose of this exercise is, right?

## Instructions

- Add an argument to your function `extract_info()`: call it `include_all`, and make it `TRUE` by default. That is, the default is to extract all log entries, whether it indicates a failure or a success.
- Change the body of your function: inside the `for` loop, add an `if` test: if `include_all` *or* if `!log$success`, you want to add the `log[[property]]` to the `info` vector. In all other cases, you're not adding anything to `info`. Use the `||` operator in your condition.
- Call your new `extract_info()` function on `logs`, first without any additional arguments. The default value for `include_all`, which is `TRUE`, will be used.
- Call `extract_info()` on `logs` again; this time set `include_all` to `FALSE`.

```
# logs is available in your workspace


# Adapt extract_info():

# - add argument with default value

# - change function body

extract_info <- function(x, property = "success", include_all = TRUE) {

  info <- c()

  for (log in x) {

   if (include_all || !log$success) {

     info <- c(info, log[[property]])

   }

  }

  return(info)

}


# Call extract_info() on logs, no additional arguments

extract_info(logs)


# Call extract_info() on logs, set include_all to FALSE

extract_info(logs, include_all = FALSE)
```

## Extract log information (5)

100xp

Have another look at `logs`, that is still available in your workspace. Have you noticed that the `details` element of each log entry differs between logs indicating success and failure? For successes, it's a list with a `message` element. For failures, it's a list with two elements: `message` and `location`. We've printed out `str(logs)` for you below. See the differences in structures between a successful log and an unsuccessful log:

```
 $ :List of 3
  ..$ success  : logi TRUE
  ..$ details  :List of 1
  .. ..$ message: chr "all good"
  ..$ timestamp: POSIXct[1:1], format: "2015-09-18 13:45:27"
 $ :List of 3
  ..$ success  : logi FALSE
  ..$ details  :List of 2
  .. ..$ message : chr "human error"
  .. ..$ location: chr "waste"
  ..$ timestamp: POSIXct[1:1], format: "2015-09-17 23:37:18"
```

At first sight, our function only allows the selection of log entry information on the first level, such as `success` and `details`. To get information that's deeper inside the log entries, such as `message`, we'll need another function, right? Nope, your function will work just fine. To select elements from embedded lists, you can use *chained selection*. The following code chunk uses chained selection to return the value 2:

```
x <- list(a = 1, b = list(r = 2, s = 3))
x[[c("b", "r")]]
```

## Instructions

- Use `extract_info()` to build a vector containing the `message` elements of all log entries, irrespective of whether they indicate a failure or not.
- Use `extract_info()` to build a vector containing the `location` information for log entries *indicating a failure*. This means you have to set `include_all = FALSE` now!

```
# logs is available in your workspace

# Defition of the extract_info() function

extract_info <- function(x, property = "success", include_all = TRUE) {

 info <- c()

 for (log in x) {

  if (include_all || !log$success) {

   info <- c(info, log[[property]])

  }

 }
```

```
  return(info)
}

# Generate vector of messages
extract_info(logs, property = c("details", "message"))

# Generate vector of locations for failed log entries
extract_info(logs, property = c("details", "location"), include_all = FALSE)
```

## Over to you

70xp

Now that you've played around with building up a function, making it more powerful step-by-step, you're ready to write your own function from the bottom with minimal help.

As usual, `logs`, a list of lists, is available in your workspace to test your function.

## Instructions

- Write a function, `compute_fail_pct()`, that takes a list of log entries that is formatted as the `logs` list and returns the *percentage*, i.e. a value between 0 and 100, of log entries that indicate a failure. If, for example, your list of logs has a length 50, and contains 5 failures, `compute_fail_pct()` should return $5/50*100=10$ 5/50∗100=10
- Call `compute_fail_pct()` on `logs`.

```
# logs is available in your workspace


# Write the function compute_fail_pct
compute_fail_pct <-function(x){
i <- 0
  for (log in x ){
    if (log$success == FALSE){
      i <- i+1
```

```
   }
  }
  return(100*i/length(x))
}



# Call compute_fail_pct on logs
compute_fail_pct(logs)
```

## lapply refresher

100xp

`lapply()` is the first and arguably the most commonly used function of R's apply family. `lapply()` is short for *list apply*. have a look at this example:

```
x <- list("R", "is", "awesome")
lapply(x, nchar)
```

This piece of code goes over each element in the list `x`, calls the `nchar()` function on it and returns a list with the number of characters in each list. In other words, `nchar()` is applied on each list element.

Still remember the `logs` list? It was a list containing a bunch of log entries on measurements from a chemical plant. Have another look at its structure with `str(head(logs))`, for example. It is already available in your workspace, so you can refresh your memory of the apply functions straight away!

### Instructions

- Use `lapply()` to call the `length()` function on each element of `logs`.
- Use `lapply()` in combination with `class()` to get a list indicating the class of each log entry in `logs`.

## lapply on logs (1)

100xp

Before you've worked with `for` loops to iterate through `logs`. This is fine, but it's pretty tedious to write, and for huge datasets, it can become slow. `lapply()` is a better alternative here.

Remember the `for` loop to get the timestamp from each log entry? Well, you can do that with `lapply` with a single line. `lapply()` should go through each element of `logs`, so through each log entry, which is stored as a list itself.

## Instructions

- Write a function `get_timestamp()` that has one argument, named `x`. `get_timestamp()` returns the `timestamp` element of the named list you pass it.
- Use `lapply()` such that `get_timestamp()` is called on each element in `logs`, so on each log entry in `logs`.

```
# logs is available in your workspace


# Define get_timestamp()

get_timestamp <- function(x) {

return(x$timestamp)



}


# Apply get_timestamp() over all elements in logs

lapply(logs, get_timestamp)
```

## lapply on logs (2)

100xp

Defining a new function simply to get an element from a list sounds like a lot of work. Do anonymous functions ring a bell here?

Without having to give the function a name, you can simply put a function definition inside the `FUN` part of the `lapply()` function.
Take this example, that returns two times the numbers in the list `a`:

```
a <- list(3, 2, 5)
lapply(a, function(x) { 2 * x })
```

The solution to the previous exercise is included on the right. As always, it's up to you to make the appropriate changes.

## Instructions

- Convert the `get_timestamp()` function to an anonymous function that you plug into `lapply()` straight away.
- Make sure to remove the original definition of `get_timestamp()`; you don't need it anymore!

# logs is available in your workspace

# Have lapply() use an anonymous function

lapply(logs, function(x){ x$timestamp})

## lapply on logs (3)

100xp

Actually, even using an anonymous function inside `lapply` is too much. The double square brackets to select an element from a list is also a function itself, as this example reveals:

```
x <- list(a = 1, b = 2, c = 3)
x[["a"]]
`[[`(x, "a")
```

Copy and paste the code snippet above to the console so you gain a better understanding of what's going on. This means that you can assign ``` `[[` ``` to the `FUN` argument to `lapply()`, and add a third argument to `lapply()`, which will be passed as an argument to the ``` `[[` ``` function.

## Instructions

Instead of using an anonymous function, use ``` `[[` ``` to select the `timestamp` element from each log entry. The third argument of `lapply()` should specify which element you want to select from each sublist.

# logs is available in your workspace

# Replace the anonymous function with `[[`

lapply(logs, `[[`, "timestamp")

## sapply refresher

100xp

Remember `sapply()`, `lapply()`'s little brother? R offers this function because many of the operations you perform on lists, will generate a list containing all elements with the same type. These lists can just as well be

represented by a vector. That's exactly what `sapply()` does. It performs an `lapply()`, and sees whether the result can be simplified to a vector.
As usual, you'll be working on `logs`, a list of log entries.

## Instructions

- Call `length()` on each element of `logs` using `sapply()`. Simply print out the result.
- `get_timestamp()` has already been defined for you. Use `sapply()` to print out a vector with all the log entries' timestamps.

```
# logs is available in your workspace


# Call length() on each element of logs using sapply()

sapply(logs,length)


# Definition of get_timestamp

get_timestamp <- function(x) {

  x$timestamp

}


# Get vector of log entries' timestamps

sapply(logs, get_timestamp)
```

## sapply on logs (1)

100xp

With `lapply()`, you could use `` `[[` `` to select specific elements from your list. The same thing is true for `sapply()`!

## Instructions

- Use `sapply()` to select the `success` element from each log and store the logical vector that results in a new variable `results`.
- Call `mean()` on `results` to see the ratio of successes. This works because `TRUE` coerces to 1 and `FALSE` coerces to 0. Print out the result.
- Again use `sapply()`, this time to select the `details` element from each log. Print out the result. Did simplification work?

# logs is available in your workspace

# Use sapply() to select the success element from each log: results

results <- sapply(logs,`[[`, "success")

# Call mean() on results

mean(results)

# Use sapply() to select the details element from each log

sapply(logs,`[[`, "details")

## sapply on logs (2)

100xp

In the previous exercise, you saw that in some cases, `sapply()` can simplify to a vector. In other cases, it can't, and you get the same result `lapply()` would have returned.

Let's experiment some more with this by writing a rather advanced function to be used by `sapply()`.

We're still working with the `logs` data; try the following commands in the console if you want a refresher on the structure of the log entries:

```
str(logs[[1]])
str(logs[[77]])
```

## Instructions

- Create a function `get_failure_loc()`:
    - It takes a single argument `x`, whose format corresponds to a log entry from the `logs` list.
    - If the input `x` (which is a list) has a `success` element that equals `TRUE`, `get_failure_loc()` returns `NULL`.
    - If the input has a `success` element that equals `FALSE`, `get_failure_loc()` returns `x$details$location`.
- Call `get_failure_loc()` on each element in `logs` using `sapply()`. Print the output to the console. Try to interpret the result.

# logs is available in your workspace

# Implement function get_failure_loc

```
get_failure_loc <- function(x) {
  if (x$success == TRUE) {
    return(NULL)
  } else {
    return(x$details$location)
  }
}


# Use sapply() to call get_failure_loc on logs
sapply(logs,get_failure_loc)
```

## vapply refresher

100xp

Recall that next to `lapply()` and `sapply()`, there's also `vapply()`. You can think of `vapply()` as the secure version of `sapply()`. Where `sapply()` *tries* to simplify the result, you have to explicitly mention what the outcome of the function you're applying will be with `vapply()`.

You do this with a third argument in `vapply()`, named `FUN.VALUE`:

```
vapply(X, FUN, FUN.VALUE)
```

`FUN.VALUE` must be a template for the output `FUN` generates. You can use functions such as `integer()`, `numeric()`, `character()` and `logical()` to do this.

The `sapply()` calls that you've coded some exercises ago are available in the script on the right. Up to you to convert them to `vapply()` calls, by changing the function call and adding the `FUN.VALUE` argument.

### Instructions

- Convert the `sapply()` call that gets the length for each log entry in `logs`. You can use `integer(1)` to specify `FUN.VALUE`.
- Convert the `sapply()` function that gets the `success` element from each log entry. You can use `logical(1)` instead of `integer(1)` this time as the template.

```
# logs is available in your workspace


# Convert the sapply call to vapply
vapply(logs, length, FUN.VALUE = integer(1))
```

```
# Convert the sapply call to vapply

vapply(logs, `[[`, "success", FUN.VALUE = logical(1))
```

## vapply on logs (1)

100xp

Once you know about `vapply()`, there's really no reason to use `sapply()` anymore. If the output that `lapply()` would generate can be simplified to an array, you'll want to use `vapply()` to do this securely. If simplification is not possible, simply stick to `lapply()`.

### Instructions

- Convert the `sapply()` call that selects the `message` element from the `details` list inside each log entry. It should use `lapply()` if simplification is not possible and `vapply()` if simplification is possible.
- Convert the `sapply()` call that selects the `details` element from each log entry, again using `lapply()` or `vapply()`.

```
# logs is available in your workspace


# Convert the sapply() call to a vapply() or lapply() call

vapply(logs, `[[`, c("details", "message"), FUN.VALUE = character(1))


# Convert the sapply() call to a vapply() or lapply() call

lapply(logs, function(x) { x$details })
```

## Loop it the way you want it

100xp

`for`, `while`, `lapply()`, `sapply()`, `vapply()`... that's quite a list. All of these can be used to loop through your very own data. Which one works best depends on the situation and your personal preference, but like most things, the functions of the `apply()` family are faster and easier to use with more practice.
To finish this chapter in style, you'll solve this exercise with minimum guidance. You'll be working with the `logs` dataset one last time. Enjoy!

## Instructions

Create a vector that contains the `message` part of each log entry in `logs` in uppercase letters. In your solution, you might want to use the function `toupper()`. Simply print this vector to the console.

```
# logs is available in your workspace

# Return vector with uppercase version of message elements in log entries
extract_caps <- function(x){
  return(toupper(x$details$message))
}
vapply(logs,extract_caps,FUN.VALUE = character(1))
```

```
# logs is available in your workspace

# Return vector with uppercase version of message elements in log entries
extract_caps <- function(x){
  return(toupper(x$details$message))
}
vapply(logs,extract_caps,FUN.VALUE = character(1))
```

## Titanic

100xp

Let's step away from your data science job at the chemical plant and start analyzing a major event in modern history: the sinking of the Titanic.

This terrible accident claimed the life of over 1500 people, making it one of the deadliest maritime disasters in times of peace.

To gain some more context about this event, we've prepared data on the Titanic (Source: Kaggle). The code that imports the data in `titanic.csv` into an R object `titanic` has already been included. Let's start by exploring this `titanic` dataset.

## Instructions

- Call `dim()` on `titanic` to figure out how many observations and variables there are.
- Call `hist()` on the `Age` column of the `titanic` dataset to create a histogram displaying the age distribution of the passengers.

## Exploratory queries

70xp

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

- `abs()`: calculate the absolute value.

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

Can you use the appropriate function to do some exploratory queries on the `titanic` data frame?

## Instructions

- Print out the total of the `Fare` column; this is the total of fares paid by all passengers in the dataset.
- Print out the proportion of passengers that survived.

```
# titanic is available in your workspace

# Print out total value of fares
sum(titanic$Fare)

# Print out proportion of passengers that survived
mean(titanic$Survived )
```

## Infer gender from name (1)

100xp

In the console, have a look at the `Name` column of `titanic`, that contains the names of the passengers. The name of the first passenger in the dataset is `Braund, Mr. Owen Harris`. All these names have a common format. First, we have the family name, next there's the title, followed by the first and middle names. Because this layout is consistent throughout the `Name` column, we're able to infer the gender of the passenger: men have the title `Mr.`, women have the title `Mrs` or `Miss`.
Let's start by finding out which names contain the pattern `"", Mr\\."`. Notice that we need `\\.` because we want to use it as an actual period, not as the wildcard character used in regular expressions.

## Instructions

- Use `grepl()` with the pattern `", Mr\\."` on the predefined `pass_names` vector. Store the resulting logical vector as `is_man`.
- Sum up the logical values in `is_man` to figure out the number of men.
- The last line that counts the number of men based on the `Sex` column in `titanic` is already included, so we can compare the results.

```
# Extract the name column from titanic
pass_names <- titanic$Name

# Create the logical vectror is_man
is_man <-grepl(pattern = ", Mr\\.",x = pass_names)

# Count the number of men
sum(is_man)
```

# Count number of men based on gender
sum(titanic$Sex == "male")

## Infer gender from name (2)

100xp

In the previous exercise, it appeared that the title `Mr.` may not cover all men on board. Instead of manually going through all titles that appear in the `Name` column of `titanic`, we can write a clever `gsub()` command that extracts the title part.
The pattern we'll need is the following:

```
"^.*, (.*?)\\..*$"
```

With `^` and `$` we signify the start and end of the string. Next, we have two `.*` parts in there: wildcards for the last name and first names. With `, (.*?)\\.` we use a similar pattern as before, but the parentheses allow us to re-use whatever is matched inside the parentheses in our replacement.

## Instructions

- Fill in the pattern into the `gsub()` call to create the `titles` vector. The `"\\1"` part tells R to replace the entire string with whatever is matched inside the parentheses.
- Call `unique()` on `titles` to get an overview of all different titles that are found in the `name` vector. Simply print out the result.

# Extract the name column from titanic
pass_names <- titanic$Name

# Create titles
titles <- gsub("^.*, (.*?)\\..*$", "\\1", pass_names)

# Call unique() on titles
unique(titles)

## Infer gender from name (3)

100xp

After a close look at the different titles that appeared in the previous exercise, we made a selection of titles that can be linked to male passengers. Patterns for these titles are in the `titles` vector on the right.
To figure out which passenger has which title, we can create a matrix. In this matrix, each passenger is a row, and each column represents a title. If a certain matrix element is `TRUE`, this means that the passenger has the title. This also means that every row can only contain one `TRUE`, the rest being `FALSE`, because titles are mutually exclusive. That is, nobody is titled both `Mr.` and `Major`, for instance. To end up with this matrix, we could use the following `for` loop:

```
res <- matrix(nrow = length(pass_names),
              ncol = length(titles))

for (i in seq_along(titles)) {
  res[, i] <- grepl(titles[i], pass_names)
}
```

There's a way more concise way to do this, however. Remember the `vapply()` function from the third chapter? You can use it to call `grepl()` over all titles in the `titles` vector, with `pass_names` as an additional argument. If you do this properly, you'll end up with the exact same matrix described above. Simply taking the sum of this matrix should give us the total number of hits for each title, and thus the total count of males inferred from their respective titles.

## Instructions

- Carefully reread the last paragraph in the assignment text above. Finish the `vapply()` call accordingly to calculate the `hits` matrix.
- Call `sum()` on `hits` to print out the total number of elements in `hits` that are `TRUE` (Remember? `TRUE` coerces to 1).
- The code that counts the number of males based on `Sex` is already included, so you can compare.
- Input string
- SQL code or error code you are receiving (if you don't have error details send input string to be run in data studio for replicate the error code.)
  Without SQL code it is not possible to analyze the issue.

```
pass_names <- titanic$Name
titles <- paste(",", c("Mr\\.", "Master", "Don", "Rev", "Dr\\.", "Major", "Sir", "Col", "Capt", "Jonkheer"))

# Finish the vapply() command
hits <- vapply(titles,
            FUN = grepl,
            FUN.VALUE = logical(length(pass_names)),
            pass_names)

# Calculate the sum() of hits
sum(hits)

# Count number of men based on gender
sum(titanic$Sex == "male")
```

## Reformat passenger names

100xp

Now that you had some practice on regular expressions, let's try to clean up the names.

The `pass_names` vector that you worked with before is already preloaded. For men, the overall format is like before. For women, however, there is only a female title, but then the name of her spouse. The first two elements in `name` show this:

```
> pass_names[1:2]
[1] "Braund, Mr. Owen Harris"
[2] "Cumings, Mrs. John Bradley (Florence Briggs Thayer)"
```

Suppose we want to change men's names to a modern format, without a title, and change the women's names to only include their own name, like this:

```
> clean_pass_names[1:2]
[1] "Owen Harris"
[2] "Florence Briggs Thayer"
```

To make this conversion, we've started a function `convert_name()` that converts the name depending on the case (male or female). The first `gsub()` function uses `\\1` as the replacement argument. This is a reference to the matched characters that are captured inside the parentheses of the pattern. To see how it works, try the following example in the console:

```
gsub("(a|b|c)", "_\\1_", "all cool brother")
```

Once you finish this function, you can use it inside `vapply()` to apply it to all elements in the `pass_names` vector.

## Instructions

- Finish the second `gsub()` function inside `convert_name()`; replace the `___` parts with `\\1` or `\\2` to referencing the matched characters contained in the first and second set of parentheses in the pattern.
- Complete the `vapply()` call so that it applies the `convert_name` function to each element in `pass_names`.
- Finally, print out `clean_pass_names` to check if the result makes sense.

```
# pass_names is available in your workspace

# Finish the convert_name() function
convert_name <- function(name) {
  # women: take name from inside parentheses
  if (grepl("\\(.*?\\)", name)) {
    gsub("^.*?\\((.*?)\\)$", "\\1", name)
  # men: take name before comma and after title
  } else {
    gsub("^(.*?),\\s[a-zA-Z\\.]*?\\s(.*?)$", "\\2 \\1", name)
  }
}

# Call convert_name on name
clean_pass_names <- vapply(pass_names, FUN = convert_name,
                           FUN.VALUE = character(1), USE.NAMES = FALSE)

# Print out clean_pass_names
clean_pass_names
```

# Choose the pattern

50xp

Regular expression are extremely useful, but it is sometimes hard to figure out which pattern you should use for a certain problem.

The only way to get better at writing them is practice!

Suppose you want to see if a string `x` contains 3 or more digits in a row. Which `grepl()` call do you need? *You can check out the documentation on regular expressions in R by typing `?regex` in the console. Online tools exist to experiment with regular expressions for different programming language. https://regex101.com,* *for example, has a very clear interface that shows which parts in your regular expressions match with which elements in the string you're trying to match. Make sure to check it out some day!*

## Add birth dates

100xp

Another look at the structure of the `titanic` data frame (preloaded) reminds us that it contains a column named `Age`, that contains the age of the passengers when the Titanic disaster took place.

To get more details on people's age, you decided to look up the actual dates of birth of all passengers that are in your dataset. After a lot of digging around, you end up with two character vectors: `dob1` and `dob2`. The first vector corresponds to the first 400 observations in your dataset, the second vector corresponds to the remaning observations.

You can convert these character vectors into `Date` vectors with `as.Date()`, but there's a problem: not all the dates may be in the standard format, so you might have to specify the `format` argument manually... To refresh your memory on this, have a look at the documentation:

```
?strptime
```

## Instructions

- Have a look at the `head()` of both `dob1` and `dob2`.
- Convert `dob1` to a Date vector, named `dob1d`. Do you have to specify the `format` argument?
- Convert `dob2` to a Date vector, named `dob2d`. Do you have to specify the `format` argument?
- Paste `dob1d` and `dob2d` together in a single Date vector `birth_dates`. You can use `c()` or `append()` for this.

```
# titanic, dob1 and dob2 are preloaded

# Have a look at head() of dob1 and dob2

head(dob1)
head(dob2)

# Convert dob1 to dob1d, convert dob2 to dob2d
dob1d <- as.Date(dob1,format = "%Y-%m-%d")
dob2d <- as.Date(dob2,format = "%B %d,%Y")


# Combine dob1d and dob2d into single vector: birth_dates
birth_dates <- c(dob1d,dob2d)
```

## Average age

100xp

Now that you have an actual `Date` vector with the birth dates of most passengers - some dates are missing - you can go ahead and add this information to the `titanic` data frame you started with.

## Instructions

- Add `birth_dates` to `titanic` as a new column named `Birth`.
- Finish the `subset()` call to create `survivors`, containing all observations that survived. That is, where `Survived` equals 1.
- Use the `Birth` column and `disaster_date` to calculate the average age of survivors at the day of the disaster. Simply print out the result. If you're using math functions like `sum()` or `mean()`, make sure to set `na.rm = TRUE` to ignore missing values

```
ob1d <- as.Date(dob1)
dob2d <- as.Date(dob2, format = "%B %d, %Y")
birth_dates <- c(dob1d, dob2d)
disaster_date <- as.Date("1912-04-15")

# Add birth_dates to titanic (column Birth)
titanic$Birth <- birth_dates

# Create subset: survivors
survivors <- subset(titanic, Survived == 1)

# Calculate average age of survivors
mean(disaster_date - survivors$Birth, na.rm = TRUE)
```

Regular expression (Regex)

## read.csv

100xp

The `utils` package, which is automatically loaded in your R session on startup, can import CSV files with the `read.csv()` function.

In this exercise, you'll be working with `swimming_pools.csv`; it contains data on swimming pools in Brisbane, Australia (Source: data.gov.au). The file contains the column names in the first row. It uses a comma to separate values within rows.

Type `dir()` in the console to list the files in your working directory. You'll see that it contains `swimming_pools.csv`, so you can start straight away.

## Instructions

- Use `read.csv()` to import `"swimming_pools.csv"` as a data frame with the name `pools`.
- Print the structure of `pools` using `str()`.

```
# Import swimming_pools.csv correctly: pools
pools <- read.csv("swimming_pools.csv",stringsAsFactors = FALSE)
```

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

## read.delim

100xp

Aside from `.csv` files, there are also the `.txt` files which are basically text files. You can import these functions with `read.delim()`. By default, it sets the `sep` argument to `"\t"` (fields in a record are delimited by tabs) and the `header` argument to `TRUE` (the first row contains the field names).
In this exercise, you will import `hotdogs.txt`, containing information on sodium and calorie levels in different hotdogs (Source: UCLA). The dataset has 3 variables, but the variable names are *not* available in the first line of the file. The file uses tabs as field separators.

### Instructions

- Import the data in `"hotdogs.txt"` with `read.delim()`. Call the resulting data frame `hotdogs`. The variable names are **not** on the first line, so make sure to set the `header` argument appropriately.
- Call `summary()` on `hotdogs`. This will print out some summary statistics about all variables in the data frame.

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

# Summarize hotdogs
summary(hotdogs)
```

## read.table

100xp

If you're dealing with more exotic flat file formats, you'll want to use `read.table()`. It's the most basic importing function; you can specify tons of different arguments in this function. Unlike `read.csv()` and `read.delim()`, the `header` argument defaults to `FALSE` and the `sep` argument is `""` by default.
Up to you again! The data is still `hotdogs.txt`. It has no column names in the first row, and the field separators are tabs. This time, though, the file is in the `data` folder inside your current working directory. A variable `path` with the location of this file is already coded for you.

### Instructions

- Finish the `read.table()` call that's been prepared for you. Use the `path` variable, and make sure to set `sep` correctly.
- Call `head()` on `hotdogs`; this will print the first 6 observations in the data frame.

```
# Path to the hotdogs.txt file: path
path <- file.path("data", "hotdogs.txt")

# Import the hotdogs.txt file: hotdogs
hotdogs <- read.table(path,
                      sep = "",
                      col.names = c("type", "calories", "sodium"))

# Call head() on hotdogs
head(hotdogs)
```

## Arguments

100xp

Lily and Tom are having an argument because they want to share a hot dog but they can't seem to agree on which one to choose. After some time, they simply decide that they will have one each. Lily wants to have the one with the fewest calories while Tom wants to have the one with the most sodium.

Next to `calories` and `sodium`, the hotdogs have one more variable: `type`. This can be one of three things: `Beef`, `Meat`, or `Poultry`, so a categorical variable: a factor is fine.

## Instructions

- Finish the `read.delim()` call to import the data in `"hotdogs.txt"`. It's a tab-delimited file without names in the first row.
- The code that selects the observation with the lowest calorie count and stores it in the variable `lily` is already available. It uses the function `which.min()`, that returns the index the smallest value in a vector.
- Do a similar thing for Tom: select the observation with the *most sodium* and store it in `tom`. Use `which.max()` this time.
- Finally, print both the observations `lily` and `tom`.

```
# Finish the read.delim() call

hotdogs <- read.delim("hotdogs.txt", header = FALSE, col.names = c("type", "calories", "sodium"))



# Select the hot dog with the least calories: lily

lily <- hotdogs[which.min(hotdogs$calories), ]



# Select the observation with the most sodium: tom

tom <- hotdogs[which.max(hotdogs$sodium),]
```

# Print lily and tom

lily

tom

## Column classes

100xp

Next to column names, you can also specify the column types or column classes of the resulting data frame. You can do this by setting the `colClasses` argument to a vector of strings representing classes:

```
read.delim("my_file.txt",
           colClasses = c("character",
                          "numeric",
                          "logical"))
```

This approach can be useful if you have some columns that should be factors and others that should be characters. You don't have to bother with `stringsAsFactors` anymore; just state for each column what the class should be.

If a column is set to `"NULL"` in the `colClasses` vector, this column will be skipped and will not be loaded into the data frame.

## Instructions

- The `read.delim()` call from before is already included and creates the `hotdogs` data frame. Go ahead and display the structure of `hotdogs`.
- **Edit** the second `read.delim()` call. Assign the correct vector to the `colClasses` argument. `NA` should be replaced with a character vector: `c("factor", "NULL", "numeric")`.
- Display the structure of `hotdogs2` and look for the difference.

# Previous call to import hotdogs.txt

hotdogs <- read.delim("hotdogs.txt", header = FALSE, col.names = c("type", "calories", "sodium"))

# Display structure of hotdogs

# Edit the colClasses argument to import the data correctly: hotdogs2

```
hotdogs2 <- read.delim("hotdogs.txt", header = FALSE,

                       col.names = c("type", "calories", "sodium"),

                       colClasses = c("factor", "NULL", "numeric"))




# Display structure of hotdogs2

str(hotdogs2)
```

## read_csv

100xp

CSV files can be imported with `read_csv()`. It's a wrapper function around `read_delim()` that handles all the details for you. For example, it will assume that the first row contains the column names.

The dataset you'll be working with here is `potatoes.csv`. It gives information on the impact of storage period and cooking on potatoes' flavor. It uses commas to delimit fields in a record, and contains column names in the first row. The file is available in your workspace. Remember that you can inspect your workspace with `dir()`.

## Instructions

- Load the `readr` package with `library()`. It's already installed on DataCamp's servers.
- Import `"potatoes.csv"` using `read_csv()`. Assign the resulting data frame to the variable `potatoes`.

```
# Load the readr package

library(readr)



# Import potatoes.csv with read_csv(): potatoes

potatoes <- read_csv("potatoes.csv")
```

## read_tsv

100xp

Where you use `read_csv()` to easily read in CSV files, you use `read_tsv()` to easily read in TSV files. TSV is short for tab-seperated values.

This time, the potatoes data comes in the form of a tab-separated values file; `potatoes.txt` is available in your workspace. In contrast to `potatoes.csv`, this file does **not** contain columns names in the first row, though.

There's a vector `properties` that you can use to specify these column names manually.

## Instructions

- Use `read_tsv()` to import the potatoes data from `potatoes.txt` and store it in the data frame `potatoes`. In addition to the path to the file, you'll also have to specify the `col_names` argument; you can use the `properties` vector for this.
- Call `head()` on `potatoes` to show the first observations of your dataset.

```
# readr is already loaded


# Column names

properties <- c("area", "temp", "size", "storage", "method",

                "texture", "flavor", "moistness")


# Import potatoes.txt: potatoes

potatoes <- read_tsv("potatoes.txt",col_names = properties)


# Call head() on potatoes

head(potatoes)
```

## read_delim

100xp

Just as `read.table()` was the main `utils` function, `read_delim()` is the main `readr` function. `read_delim()` takes two mandatory arguments:

- `file`: the file that contains the data
- `delim`: the character that separates the values in the data file

You'll again be working `potatoes.txt`; the file uses tabs (`"\t"`) to delimit values and does **not** contain column names in its first line. It's available in your working directory so you can start right away. As before, the vector `properties` is available to set the `col_names`.

## Instructions

- Import all the data in `"potatoes.txt"` using `read_delim()`; store the resulting data frame in `potatoes`.
- Print out `potatoes`.

## skip and n_max

100xp

Through `skip` and `n_max` you can control *which part* of your flat file you're actually importing into R.

- `skip` specifies the number of lines you're ignoring in the flat file before actually starting to import data.
- `n_max` specifies the number of lines you're actually importing.

Say for example you have a CSV file with 20 lines, and set `skip = 2` and `n_max = 3`, you're only reading in lines 3, 4 and 5 of the file.
Watch out: Once you `skip` some lines, you also skip the first line that can contain column names!
`potatoes.txt`, a flat file with tab-delimited records and without column names, is available in your workspace.

## Instructions

Finish the first `read_tsv()` call to import observations 7, 8, 9, 10 and 11 from `potatoes.txt`.

```
# readr is already loaded

# Column names

properties <- c("area", "temp", "size", "storage", "method",
```

"texture", "flavor", "moistness")

# Import 5 observations from potatoes.txt: potatoes_fragment

potatoes_fragment <- read_tsv("potatoes.txt", skip = 6, n_max = 5, col_names = properties

## col_types

100xp

You can also specify which types the columns in your imported data frame should have. You can do this with `col_types`. If set to `NULL`, the default, functions from the `readr` package will try to find the correct types themselves. You can manually set the types with a string, where each character denotes the class of the column: `c`haracter, `d`ouble, `i`nteger and `l`ogical. `_` skips the column as a whole.

`potatoes.txt`, a flat file with tab-delimited records and without column names, is again available in your workspace.

### Instructions

- In the second `read_tsv()` call, edit the `col_types` argument to import *all* columns as characters (`c`). Store the resulting data frame in `potatoes_char`.
- Print out the structure of `potatoes_char` and verify whether all column types are `chr`, short for `character`.

# readr is already loaded

# Column names

properties <- c("area", "temp", "size", "storage", "method",

"texture", "flavor", "moistness")

# Import all data, but force all columns to be character: potatoes_char

potatoes_char <- read_tsv("potatoes.txt", col_types = "cccccccc", col_names = properties)

# Print out structure of potatoes_char

str(potatoes_char)

```
# readr is already loaded


# Column names

properties <- c("area", "temp", "size", "storage", "method",

                "texture", "flavor", "moistness")


# Import potatoes.txt using read_delim(): potatoes

potatoes <- read_delim("potatoes.txt",delim = "\t",col_names = properties)


# Print out potatoes

Potatoes
```

## col_types with collectors

100xp

Another way of setting the types of the imported columns is using **collectors**. Collector functions can be passed in a `list()` to the `col_types` argument of `read_` functions to tell them how to interpret values in a column.
For a complete list of collector functions, you can take a look at the `collector` documentation. For this exercise you will need two collector functions:

- `col_integer()`: the column should be interpreted as an integer.
- `col_factor(levels, ordered = FALSE)`: the column should be interpreted as a factor with `levels`.

In this exercise, you will work with `hotdogs.txt`, which is a tab-delimited file without column names in the first row.

## Instructions

- `hotdogs` is created for you without setting the column types. Inspect its summary using the `summary()` function.
- Two collector functions are defined for you: `fac` and `int`. Have a look at them, do you understand what they're collecting?
- In the second `read_tsv()` call, edit the `col_types` argument: Pass a `list()` with the elements `fac`, `int` and `int`, so the first column is importead as a factor, and the second and third column as integers.
- Create a `summary()` of `hotdogs_factor`. Compare this to the summary of `hotdogs`.

```
# readr is already loaded



# Import without col_types

hotdogs <- read_tsv("hotdogs.txt", col_names = c("type", "calories", "sodium"))



# Display the summary of hotdogs

summary(hotdogs)



# The collectors you will need to import the data

fac <- col_factor(levels = c("Beef", "Meat", "Poultry"))

int <- col_integer()



# Edit the col_types argument to import the data correctly: hotdogs_factor

hotdogs_factor <- read_tsv("hotdogs.txt",

                           col_names = c("type", "calories", "sodium"),

                           col_types = list(fac, int, int))



# Display the summary of hotdogs_factor

summary(hotdogs_factor)
```

## fread

100xp

You still remember how to use `read.table()`, right? Well, `fread()` is a function that does the same job with very similar arguments. It is extremely easy to use and blazingly fast! Often, simply specifying the path to the file is enough to successfully import your data.

Don't take our word for it, try it yourself! You'll be working with the `potatoes.csv` file, that's available in your workspace. Fields are delimited by commas, and the first line contains the column names.

## Instructions

- Load the `data.table` package using `library()`; it is already installed on DataCamp's servers.
- Import `"potatoes.csv"` with `fread()`. Simply pass it the file path and see if it worked. Store the result in a variable `potatoes`.
- Print out `potatoes`.

# load the data.table package

library(data.table)

# Import potatoes.csv with fread(): potatoes

potatoes <-fread("potatoes.csv")

# Print out potatoes

Potatoes

## fread: more advanced use

100xp

Now that you know the basics about `fread()`, you should know about two arguments of the function: `drop` and `select`, to drop or select variables of interest.

Suppose you have a dataset that contains 5 variables and you want to keep the first and fifth variable, named "a" and "e". The following options will all do the trick:

```
fread("path/to/file.txt", drop = 2:4)
fread("path/to/file.txt", select = c(1, 5))
fread("path/to/file.txt", drop = c("b", "c", "d")
fread("path/to/file.txt", select = c("a", "e"))
```

Let's stick with potatoes since we're particularly fond of them here at DataCamp. The data is again available in the file `potatoes.csv`, containing comma-separated records.

## Instructions

- Using `fread()` and `select` or `drop` as arguments, only import the `texture` and `moistness` columns of the flat file. They correspond to the columns 6 and 8 in `"potatoes.csv"`. Store the result in a variable `potatoes`.
- `plot()` 2 columns of the `potatoes` data frame: `texture` on the x-axis, `moistness` on the y-axis. Use the dollar sign notation twice. Feel free to name your axes and plot.

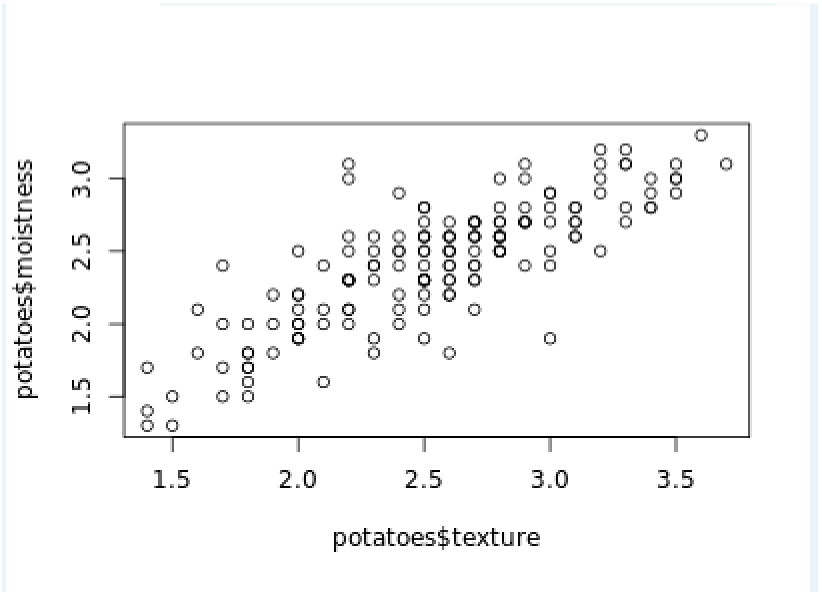
# fread is already loaded

# Import columns 6 and 8 of potatoes.csv: potatoes

potatoes <-fread("potatoes.csv", select = c("texture","moistness"))

# Plot texture (x) and moistness (y) of potatoes

plot(potatoes$texture,potatoes$moistness)



## List the sheets of an Excel file

100xp

Before you can start importing from Excel, you should find out which sheets are available in the workbook. You can use the `excel_sheets()` function for this.

You will find the Excel file `urbanpop.xlsx` in your working directory (type `dir()` to see it). This dataset contains urban population metrics for practically all countries in the world throughout time (Source: Gapminder). It contains three sheets for three different time periods. In each sheet, the first row contains the column names.

### Instructions

- Load the `readxl` package using `library()`. It's already installed on DataCamp's servers.
- Use `excel_sheets()` to print out the names of the sheets in `urbanpop.xlsx`.

Load the readxl package

library(readxl)

# Print out the names of both spreadsheets

excel_sheets("urbanpop.xlsx")

## Import an Excel sheet

100xp

Now that you know the names of the sheets in the Excel file you want to import, it is time to import those sheets into R. You can do this with the `read_excel()` function. Have a look at this recipe:

```
data <- read_excel("data.xlsx", sheet = "my_sheet")
```

This call simply imports the sheet with the name `"my_sheet"` from the `"data.xlsx"` file. You can also pass a number to the `sheet` argument; this will cause `read_excel()` to import the sheet with the given sheet number. `sheet = 1` will import the first sheet, `sheet = 2` will import the second sheet, and so on.

In this exercise, you'll continue working with the `urbanpop.xlsx` file.

### Instructions

- The code to import the first and second sheets is already included. Can you add a command to also import the third sheet, and store the resulting data frame in `pop_3`?
- Store the data frames `pop_1`, `pop_2` and `pop_3` in a list, that you call `pop_list`.
- Display the structure of `pop_list`

# The readxl package is already loaded

```
# Read the sheets, one by one
pop_1 <- read_excel("urbanpop.xlsx", sheet = 1)
pop_2 <- read_excel("urbanpop.xlsx", sheet = 2)
pop_3 <- read_excel("urbanpop.xlsx", sheet = 3)

# Put pop_1, pop_2 and pop_3 in a list: pop_list
pop_list <- list(pop_1,pop_2,pop_3)

# Display the structure of pop_list
str(pop_list)
```

## Reading a workbook

100xp

In the previous exercise you generated a list of three Excel sheets that you imported. However, loading in every sheet manually and then merging them in a list can be quite tedious. Luckily, you can automate this with `lapply()`. If you have no experience with `lapply()`, feel free to take Chapter 4 of the Intermediate R course.

Have a look at the example code below:

```
my_workbook <- lapply(excel_sheets("data.xlsx"),
                      read_excel,
                      path = "data.xlsx")
```

The `read_excel()` function is called multiple times on the `"data.xlsx"` file and each sheet is loaded in one after the other. The result is a list of data frames, each data frame representing one of the sheets in `data.xlsx`.

You're still working with the `urbanpop.xlsx` file.

## Instructions

- Use `lapply()` in combination with `excel_sheets()` and `read_excel()` to read all the Excel sheets in `"urbanpop.xlsx"`. Name the resulting list `pop_list`.
- Print the structure of `pop_list`.

```
# The readxl package is already loaded

# Read all Excel sheets with lapply(): pop_list
pop_list <- lapply(excel_sheets("urbanpop.xlsx"),read_excel, path = "urbanpop.xlsx")

# Display the structure of pop_list
str(pop_list)
```

## The col_names argument

100xp

Apart from `path` and `sheet`, there are several other arguments you can specify in `read_excel()`. One of these arguments is called `col_names`.
By default it is `TRUE`, denoting whether the first row in the Excel sheets contains the column names. If this is not the case, you can set `col_names` to `FALSE`. In this case, R will choose column names for you. You can also choose to set `col_names` to a character vector with names for each column. It works exactly the same as in the `readr` package.
You'll be working with the `urbanpop_nonames.xlsx` file. It contains the same data as `urbanpop.xlsx` but has no column names in the first row of the excel sheets.

## Instructions

- Import the *first* Excel sheet of `"urbanpop_nonames.xlsx"` and store the result in `pop_a`. Have R set the column names of the resulting data frame itself.
- Import the first Excel sheet of `urbanpop_nonames.xlsx`; this time, use the `cols` vector that has already been preparedfor you to specify the column names. Store the resulting data frame in `pop_b`.
- Print out the summary of `pop_a`.
- Print out the summary of `pop_b`. Can you spot the difference with the other summary?

```
# The readxl package is already loaded

# Import the the first Excel sheet of urbanpop_nonames.xlsx (R gives names): pop_a
pop_a <- read_excel("urbanpop_nonames.xlsx", sheet = 1, col_names = FALSE)

# Import the the first Excel sheet of urbanpop_nonames.xlsx (specify col_names): pop_b
cols <- c("country", paste0("year_", 1960:1966))
pop_b <- read_excel("urbanpop_nonames.xlsx", sheet = 1, col_names = cols)

# Print the summary of pop_a
summary(pop_a)

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

## The skip argument

100xp

Another argument that can be very useful when reading in Excel files that are less tidy, is `skip`. With `skip`, you can tell R to ignore a specified number of rows inside the Excel sheets you're trying to pull data from.
Have a look at this example:

```
read_excel("data.xlsx", skip = 15)
```

In this case, the first 15 rows in the first sheet of `"data.xlsx"` are ignored.
If the first row of this sheet contained the column names, this information will also be ignored by `readxl`. Make sure to set `col_names` to `FALSE` or manually specify column names in this case!
The file `urbanpop.xlsx` is available in your directory; it has column names in the first rows.

## Instructions

- Import the *second* sheet of `"urbanpop.xlsx"`, but skip the first 21 rows. Make sure to set `col_names = FALSE`. Store the resulting data frame in a variable `urbanpop_sel`.
- Select the first observation from `urbanpop_sel` and print it out.

```
# The readxl package is already loaded

# Import the second sheet of urbanpop.xlsx, skipping the first 21 rows: urbanpop_sel
urbanpop_sel <- read_excel("urbanpop.xlsx",sheet = 2, skip = 21, col_names = FALSE)

# Print out the first observation from urbanpop_sel
urbanpop_sel[1,]
```

## Import a local file

100xp

In this part of the chapter you'll learn how to import `.xls` files using the `gdata` package. Similar to the `readxl` package, you can import single Excel sheets from Excel sheets to start your analysis in R.
You'll be working with the `urbanpop.xls` dataset, the `.xls` version of the Excel file you've been working with before. It's available in your current working directory.

### Instructions

- Load the `gdata` package with `library()`. `gdata` and Perl are already installed on DataCamp's Servers.
- Import the second sheet, named `"1967-1974"`, of `"urbanpop.xls"` with `read.xls()`. Store the resulting data frame as `urban_pop`.
- Print the first 11 observations of `urban_pop` with `head()`.

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

# Import the second sheet of urbanpop.xls: urban_pop
urban_pop <- read.xls("urbanpop.xls", sheet = "1967-1974")

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

## read.xls() wraps around read.table()

100xp

Remember how `read.xls()` actually works? It basically comes down to two steps: converting the Excel file to a `.csv` file using a Perl script, and then reading that `.csv` file with the `read.csv()` function that is loaded by default in R, through the `utils` package.
This means that all the options that you can specify in `read.csv()`, can also be specified in `read.xls()`.
The `urbanpop.xls` dataset is already available in your workspace. It's still comprised of three sheets, and has column names in the first row of each sheet.

## Instructions

- Finish the `read.xls()` call that reads data from the second sheet of `urbanpop.xls`: skip the first 50 rows of the sheet. Make sure to set `header` appropriately and that the country names are not imported as factors.
- Print the first 10 observations of `urban_pop` with `head()`.

```
# The gdata package is alreaded loaded

# Column names for urban_pop
columns <- c("country", paste0("year_", 1967:1974))

# Finish the read.xls call
urban_pop <- read.xls("urbanpop.xls", sheet = 2,
                    skip = 50, header = FALSE, stringsAsFactors = FALSE,
                    col.names = columns)

# Print first 10 observation of urban_pop
head(urban_pop, n = 10)
```

## Work that Excel data!

100xp

Now that you can read in Excel data, let's try to clean and merge it. You already used the `cbind()` function some exercises ago. Let's take it one step further now.
The `urbanpop.xls` dataset is available in your working directory. The file still contains three sheets, and has column names in the first row of each sheet.

## Instructions

- Add code to read the data from the third sheet in `"urbanpop.xls"`. You want to end up with three data frames: `urban_sheet1`, `urban_sheet2` and `urban_sheet3`.
- Extend the `cbind()` call so that it also includes `urban_sheet3`. Make sure the first column of `urban_sheet2` and `urban_sheet3` are removed, so you don't have duplicate columns. Store the result in `urban`.
- Use `na.omit()` on the `urban` data frame to remove all rows that contain `NA` values. Store the cleaned data frame as `urban_clean`.
- Print a summary of `urban_clean` and assert that there are no more `NA` values.

## Work that Excel data!

100xp

Now that you can read in Excel data, let's try to clean and merge it. You already used the `cbind()` function some exercises ago. Let's take it one step further now.
The `urbanpop.xls` dataset is available in your working directory. The file still contains three sheets, and has column names in the first row of each sheet.

## Instructions

- Add code to read the data from the third sheet in `"urbanpop.xls"`. You want to end up with three data frames: `urban_sheet1`, `urban_sheet2` and `urban_sheet3`.
- Extend the `cbind()` call so that it also includes `urban_sheet3`. Make sure the first column of `urban_sheet2` and `urban_sheet3` are removed, so you don't have duplicate columns. Store the result in `urban`.
- Use `na.omit()` on the `urban` data frame to remove all rows that contain `NA` values. Store the cleaned data frame as `urban_clean`.
- Print a summary of `urban_clean` and assert that there are no more `NA` values.

```
# Add code to import data from all three sheets in urbanpop.xls
path <- "urbanpop.xls"
urban_sheet1 <- read.xls(path, sheet = 1, stringsAsFactors = FALSE)
urban_sheet2 <- read.xls(path, sheet = 2, stringsAsFactors = FALSE)
urban_sheet3 <- read.xls(path, sheet = 3, stringsAsFactors = FALSE)
```

Remove header while combining

```
# Extend the cbind() call to include urban_sheet3: urban
urban <- cbind(urban_sheet1, urban_sheet2[-1],urban_sheet3[-1])

# Remove all rows with NAs from urban: urban_clean
urban_clean <- na.omit(urban)

# Print out a summary of urban_clean
summary(urban_clean)
```

## Connect to a workbook

100xp

When working with `XLConnect`, the first step will be to load a workbook in your R session with `loadWorkbook()`; this function will build a "bridge" between your Excel file and your R session.
In this and the following exercises, you will continue to work with `urbanpop.xlsx`, containing urban population data throughout time. The Excel file is available in your current working directory.

### Instructions

- Load the `XLConnect` package using `library()`; it is already installed on DataCamp's servers.
- Use `loadWorkbook()` to build a connection to the `"urbanpop.xlsx"` file in R. Call the workbook `my_book`.
- Print out the class of `my_book`. What does this tell you?

```
# urbanpop.xlsx is available in your working directory

# Load the XLConnect package
library(XLConnect)

# Build connection to urbanpop.xlsx: my_book
my_book <- loadWorkbook("urbanpop.xlsx")

# Print out the class of my_book
Class(my_book)
```

## List and read Excel sheets

100xp

Just as `readxl` and `gdata`, you can use `XLConnect` to import data from Excel file into R.
To list the sheets in an Excel file, use `getSheets()`. To actually import data from a sheet, you can use `readWorksheet()`. Both functions require an XLConnect workbook object as the first argument.
You'll again be working with `urbanpop.xlsx`. The `my_book` object that links to this Excel file has already been created.

## Instructions

- Print out the sheets of the Excel file that `my_book` links to.
- Import the second sheet in `my_book` as a data frame. Print it out.

# XLConnect is already available

# Build connection to urbanpop.xlsx

my_book <- loadWorkbook("urbanpop.xlsx")

# List the sheets in my_book

getSheets(my_book)

# Import the second sheet in my_book

readWorksheet(my_book, sheet = 2)

## Customize readWorksheet

100xp

To get a clear overview about `urbanpop.xlsx` without having to open up the Excel file, you can execute the following code:

```
my_book <- loadWorkbook("urbanpop.xlsx")
sheets <- getSheets(my_book)
all <- lapply(sheets, readWorksheet, object = my_book)
str(all)
```

Suppose we're only interested in urban population data of the years 1968, 1969 and 1970. The data for these years is in the columns 3, 4, and 5 of the second sheet. Only selecting these columns will leave us in the dark about the actual countries the figures belong to,

## Instructions

- Extend the `readWorksheet()` command with the `startCol` and `endCol` arguments to only import the columns 3, 4, and 5 of the second sheet.

- `urbanpop_sel` no longer contains information about the countries now. Can you write another `readWorksheet()` command that imports only the first column from the second sheet? Store the resulting data frame as `countries`.
- Use `cbind()` to paste together `countries` and `urbanpop_sel`, in this order. Store the result as `selection`.

```
# XLConnect is already available

# Build connection to urbanpop.xlsx
my_book <- loadWorkbook("urbanpop.xlsx")

# Import columns 3, 4, and 5 from second sheet in my_book: urbanpop_sel
urbanpop_sel <- readWorksheet(my_book, sheet = 2,startCol = 3, endCol = 5)

# Import first column from second sheet in my_book: countries
countries <- readWorksheet(my_book, sheet = 2,startCol = 1, endCol = 1)

# cbind() urbanpop_sel and countries together: selection
selection <- cbind(countries, urbanpop_sel)
```

## Add worksheet

100xp

Where `readxl` and `gdata` were only able to import Excel data, `XLConnect`'s approach of providing an actual interface to an Excel file makes it able to edit your Excel files from inside R. In this exercise, you'll create a new sheet. In the next exercise, you'll populate the sheet with data, and save the results in a new Excel file.

You'll continue to work with `urbanpop.xlsx`. The `my_book` object that links to this Excel file is already available.

## Instructions

- Use `createSheet()`, to create a new sheet in `my_book`, named `"data_summary"`.
- Use [`getSheets()`] to verify that `my_book` now represents an Excel file with four sheets.

```
# XLConnect is already available

# Build connection to urbanpop.xlsx
my_book <- loadWorkbook("urbanpop.xlsx")

# Add a worksheet to my_book, named "data_summary"
createSheet(my_book, "data_summary")
# Use getSheets() on my_book
getSheets(my_book)
```

## Populate worksheet

100xp

The first step of creating a sheet is done; let's populate it with some data now! `summ`, a data frame with some summary statistics on the two Excel sheets is already coded so you can take it from there.

## Instructions

- Use `writeWorksheet()` to populate the `"data_summary"` sheet with the `summ` data frame.
- Call `saveWorkbook()` to store the adapted Excel workbook as a new file, `"summary.xlsx"`.

```
# XLConnect is already available

# Build connection to urbanpop.xlsx
my_book <- loadWorkbook("urbanpop.xlsx")

# Add a worksheet to my_book, named "data_summary"
createSheet(my_book, "data_summary")

# Create data frame: summ
sheets <- getSheets(my_book)[1:3]
dims <- sapply(sheets, function(x) dim(readWorksheet(my_book, sheet = x)), USE.NAMES = FALSE)
summ <- data.frame(sheets = sheets,
           nrows = dims[1, ],
           ncols = dims[2, ])

# Add data in summ to "data_summary" sheet
writeWorksheet(my_book,summ,"data_summary")

# Save workbook as summary.xlsx
saveWorkbook("summary.xlsx")
```

```
# XLConnect is already available

# Build connection to urbanpop.xlsx
my_book <- loadWorkbook("urbanpop.xlsx")

# Add a worksheet to my_book, named "data_summary"
createSheet(my_book, "data_summary")

# Create data frame: summ
sheets <- getSheets(my_book)[1:3]
dims <- sapply(sheets, function(x) dim(readWorksheet(my_book, sheet = x)), USE.NAMES = FALSE)
summ <- data.frame(sheets = sheets,
           nrows = dims[1, ],
           ncols = dims[2, ])

# Add data in summ to "data_summary" sheet
```

```
writeWorksheet(my_book, summ, "data_summary")

# Save workbook as summary.xlsx
saveWorkbook(my_book, "summary.xlsx")
```

## Renaming sheets

100xp

Come to think of it, `"data_summary"` is not an ideal name. As the summary of these excel sheets is always data-related, you simply want to name the sheet `"summary"`.
The workspace already contains a workbook, `my_book`, that refers to an Excel file with 4 sheets: the three data sheets, and the `"data_summary"` sheet.

### Instructions

- Use `renameSheet()` to rename the fourth sheet to `"summary"`.
- Next, call `getSheets()` on `my_book` to print out the sheet names.
- Finally, make sure to actually save the `my_book` object to a new Excel file, `"renamed.xlsx"`.

```
# my_book is available

# Rename "data_summary" sheet to "summary"
renameSheet(my_book, sheet = 4, "summary")

# Print out sheets of my_book
getSheets(my_book)

# Save workbook to "renamed.xlsx"
saveWorkbook(my_book, "renamed.xlsx")
```

## Removing sheets

100xp

After presenting the new Excel sheet to your peers, it appears not everybody is a big fan. Why summarize sheets and store the info in Excel if all the information is implicitly available? To hell with it, just remove the entire fourth sheet!

### Instructions

- Load the `XLConnect` package.
- Build a connection to `"renamed.xlsx"`, the Excel file that you've built in the previous exercise; it's available in your working directory. Store this connection as `my_book`.
- Use `removeSheet()` to remove the fourth sheet from `my_book`. The sheet name is `"summary"`.
- Save the resulting workbook, `my_book`, to a file `"clean.xslx"`.

# Load the XLConnect package

library(XLConnect)

# Build connection to renamed.xlsx: my_book

my_book <- loadWorkbook("renamed.xlsx")

# Remove the fourth sheet

removeSheet(my_book, sheet = 4)

# Save workbook to "clean.xlsx"

saveWorkbook(my_book, "clean.xlsx")

Connecting databases and importing various formats

---

## Establish a connection

100xp

The first step to import data from a SQL database is creating a connection to it. As Filip explained, you need different packages depending on the database you want to connect to. All of these packages do this in a uniform way, as specified in the `DBI` package.

`dbConnect()` creates a connection between your R session and a SQL database. The first argument has to be a `DBIdriver` object, that specifies how connections are made and how data is mapped between R and the database. Specifically for MySQL databases, you can build such a driver with `RMySQL::MySQL()`.

If the MySQL database is a remote database hosted on a server, you'll also have to specify the following arguments in `dbConnect()`: `dbname`, `host`, `port`, `user` and `password`. Most of these details have already been provided.

## Instructions

- Load the `DBI` library, which is already installed on DataCamp's servers.
- Edit the `dbConnect()` call to connect to the MySQL database. Change the `port` argument (`3306`) and `user` argument (`"student"`).

```
# Load the DBI package

library(DBI)


# Edit dbConnect() call

con <- dbConnect(RMySQL::MySQL(),

                 dbname = "tweater",

                 host = "courses.csrrinzqubik.us-east-1.rds.amazonaws.com",

                 port = 3306,

                 user = "student",

                 password = "datacamp")
```

## List the database tables

100xp

After you've successfully connected to a remote MySQL database, the next step is to see what tables the database contains. You can do this with the `dbListTables()` function. As you might remember from the video, this function requires the connection object as an input, and outputs a character vector with the table names.

## Instructions

- Add code to create a vector `tables`, that contains the tables in the tweater database. You can connect to this database through the `con` object.
- Display the structure of `tables`; what's the class of this vector?

```
# Load the DBI package

library(DBI)


# Connect to the MySQL database: con

con <- dbConnect(RMySQL::MySQL(),

                 dbname = "tweater",

                 host = "courses.csrrinzqubik.us-east-1.rds.amazonaws.com",

                 port = 3306,
                 user = "student",

                 password = "datacamp")


# Build a vector of table names: tables

tables <- dbListTables(con)


# Display structure of tables

str(tables)
```

## Import users

100xp

As you might have guessed by now, the database contains data on a more tasty version of Twitter, namely Tweater. Users can post tweats with short recipes for delicious snacks. People can comment on these tweats. There are three tables: **users**, **tweats**, and **comments** that have relations among them. Which ones, you ask? You'll discover in a moment!

Let's start by importing the data on the users into your R session. You do this with the `dbReadTable()` function. Simply pass it the connection object (`con`), followed by the name of the table you want to import. The resulting object is a standard R data frame.

## Instructions

- Add code that imports the `"users"` table from the tweater database and store the resulting data frame as `users`.
- Print the `users` data frame.

```
# Load the DBI package

library(DBI)


# Connect to the MySQL database: con

con <- dbConnect(RMySQL::MySQL(),

                 dbname = "tweater",

                 host = "courses.csrrinzqubik.us-east-1.rds.amazonaws.com",

                 port = 3306,

                 user = "student",
            password = "datacamp")

# Import the users table from tweater: users
users <- dbReadTable(con, "users")

# Print users
Users
```

## Import all tables

100xp

Next to the `users`, we're also interested in the `tweats` and `comments` tables. However, separate `dbReadTable()` calls for each and every one of the tables in your database would mean a lot of code duplication. Remember about the `lapply()` function? You can use it again here! A connection is already coded for you, as well as a vector `table_names`, containing the names of all the tables in the database.

## Instructions

- Finish the `lapply()` function to import the `users`, `tweats` and `comments` tables in a single call. The result, a list of data frames, will be stored in the variable `tables`.
- Print `tables` to check if you got it right.

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


# Connect to the MySQL database: con
con <- dbConnect(RMySQL::MySQL(),
            dbname = "tweater",
            host = "courses.csrrinzqubik.us-east-1.rds.amazonaws.com",
            port = 3306,
            user = "student",
            password = "datacamp")
```

# Get table names

table_names <- dbListTables(con)

# Import all tables

tables <- lapply(table_names, dbReadTable, conn = con)

# Print out tables

Tables

## Query tweater (1)

100xp

In your life as a data scientist, you'll often be working with huge databases that contain tables with millions of rows. If you want to do some analyses on this data, it's possible that you only need a fraction of this data. In this case, it's a good idea to send SQL queries to your database, and only import the data you actually need into R.

`dbGetQuery()` is what you need. As usual, you first pass the connection object to it. The second argument is an SQL query in the form of a character string. This example selects the `age` variable from the `people` dataset where `gender` equals `"male"`:

```
dbGetQuery(con, "SELECT age FROM people WHERE gender = 'male'")
```

A connection to the `tweater` database has already been coded for you.

### Instructions

- Use `dbGetQuery()` to create a data frame, `elisabeth`, that **selects** the `tweat_id` column **from** the `comments` table **where** elisabeth is the commenter, her `user_id` is 1
- Print out `elisabeth` so you can see if you queried the database correctly.

# Connect to the database

library(DBI)

con <- dbConnect(RMySQL::MySQL(),

        dbname = "tweater",

```
                 host = "courses.csrrinzqubik.us-east-1.rds.amazonaws.com",

                 port = 3306,

                 user = "student",

                 password = "datacamp")



# Import tweat_id column of comments where user_id is 1: elisabeth

elisabeth <- dbGetQuery(con, "SELECT tweat_id FROM comments WHERE user_id = 1")




# Print elisabeth

Elisabeth
```

## Query tweater (2)

100xp

Apart from checking equality, you can also check for *less than* and *greater than* relationships, with `<` and `>`, just like in R.

`con`, a connection to the `tweater` database, is again available.

### Instructions

- Create a data frame, `latest`, that **selects** the `post` column **from** the `tweats` table observations **where** the `date` is higher than `'2015-09-21'`.
- Print out `latest`.

```
# Connect to the database

library(DBI)

con <- dbConnect(RMySQL::MySQL(),

                 dbname = "tweater",
```

host = "courses.csrrinzqubik.us-east-1.rds.amazonaws.com",

port = 3306,

user = "student",

password = "datacamp")

# Import post column of tweats where date is higher than '2015-09-21': latest

latest <-

# Print latest

Latest

Suppose that you have a `people` table, with a bunch of information. This time, you want to find out the `age` and `country` of married males. Provided that there is a `married` column that's 1 when the person in question is married, the following query would work.

```
SELECT age, country
  FROM people
    WHERE gender = "male" AND married = 1
```

Can you use a similar approach for a more specialized query on the `tweater` database?

## Instructions

- Create an R data frame, `specific`, that **selects** the `message` column **from** the `comments` table **where** the `tweat_id` is 77 **and** the `user_id` is greater than 4.
- Print `specific`.

# Connect to the database

library(DBI)

con <- dbConnect(RMySQL::MySQL(),

dbname = "tweater",

host = "courses.csrrinzqubik.us-east-1.rds.amazonaws.com",

port = 3306,

user = "student",

password = "datacamp")

# Create data frame specific

specific <- dbGetQuery(con, "SELECT message FROM comments WHERE tweat_id = 77 AND user_id > 4")

# Print specific

Specific

## Query tweater (4)

100xp

There are also dedicated SQL functions that you can use in the `WHERE` clause of an SQL query. For example, `CHAR_LENGTH()` returns the number of characters in a string.

## Instructions

- Create a data frame, `short`, that **selects** the `id` and `name` columns **from** the `users` table **where** the number of characters in the `name` is strictly less than 5.
- Print `short`.

# Connect to the database

library(DBI)

con <- dbConnect(RMySQL::MySQL(),

dbname = "tweater",

host = "courses.csrrinzqubik.us-east-1.rds.amazonaws.com",

port = 3306,

user = "student",

password = "datacamp")

# Create data frame short

short <- dbGetQuery(con, "SELECT id, name FROM users WHERE CHAR_LENGTH(name) < 5")

# Print short

Short

## Send - Fetch - Clear

100xp

You've used `dbGetQuery()` multiple times now. This is a virtual function from the `DBI` package, but is actually implemented by the `RMySQL` package. Behind the scenes, the following steps are performed:

- Sending the specified query with `dbSendQuery()`;
- Fetching the result of executing the query on the database with `dbFetch()`;
- Clearing the result with `dbClearResult()`.

Let's not use `dbGetQuery()` this time and implement the steps above. This is tedious to write, but it gives you the ability to fetch the query's result in chunks rather than all at once. You can do this by specifying the `n` argument inside `dbFetch()`.

## Instructions

- Inspect the `dbSendQuery()` call that has already been coded for you. It selects the comments for the users with an id above 4.
- Use `dbFetch()` twice. In the first call, import only two records of the query result by setting the `n` argument to `2`. In the second call, import all remaining queries (don't specify `n`). In both calls, simply print the resulting data frames.
- Clear `res` with `dbClearResult()`.

# Connect to the database

library(DBI)

con <- dbConnect(RMySQL::MySQL(),

        dbname = "tweater",

        host = "courses.csrrinzqubik.us-east-1.rds.amazonaws.com",

        port = 3306,

        user = "student",

```
                 password = "datacamp")


# Send query to the database

res <- dbSendQuery(con, "SELECT * FROM comments WHERE user_id > 4")


# Use dbFetch() twice

dbFetch(res,n = 2)

dbFetch(res)


# Clear res

dbClearResult(res)
```

## Import all tables

100xp

Next to the `users`, we're also interested in the `tweats` and `comments` tables. However, separate `dbReadTable()` calls for each and every one of the tables in your database would mean a lot of code duplication. Remember about the `lapply()` function? You can use it again here! A connection is already coded for you, as well as a vector `table_names`, containing the names of all the tables in the database.

## Instructions

- Finish the `lapply()` function to import the `users`, `tweats` and `comments` tables in a single call. The result, a list of data frames, will be stored in the variable `tables`.
- Print `tables` to check if you got it right.

```
# Load the DBI package
```

```
library(DBI)


# Connect to the MySQL database: con

con <- dbConnect(RMySQL::MySQL(),

                 dbname = "tweater",

                 host = "courses.csrrinzqubik.us-east-1.rds.amazonaws.com",

                 port = 3306,

                 user = "student",

                 password = "datacamp")


# Get table names

table_names <- dbListTables(con)


# Import all tables

tables <- lapply(table_names, dbReadTable, conn = con)


# Print out tables

Tables
```

## Send - Fetch - Clear

100xp

You've used `dbGetQuery()` multiple times now. This is a virtual function from the `DBI` package, but is actually implemented by the `RMySQL` package. Behind the scenes, the following steps are performed:

- Sending the specified query with `dbSendQuery()`;
- Fetching the result of executing the query on the database with `dbFetch()`;

- Clearing the result with `dbClearResult()`.

Let's not use `dbGetQuery()` this time and implement the steps above. This is tedious to write, but it gives you the ability to fetch the query's result in chunks rather than all at once. You can do this by specifying the `n` argument inside `dbFetch()`.

## Instructions

- Inspect the `dbSendQuery()` call that has already been coded for you. It selects the comments for the users with an id above 4.
- Use `dbFetch()` twice. In the first call, import only two records of the query result by setting the `n` argument to `2`. In the second call, import all remaining queries (don't specify `n`). In both calls, simply print the resulting data frames.
- Clear `res` with `dbClearResult()`.

```
# Connect to the database

library(DBI)

con <- dbConnect(RMySQL::MySQL(),

            dbname = "tweater",

            host = "courses.csrrinzqubik.us-east-1.rds.amazonaws.com",

            port = 3306,

            user = "student",

            password = "datacamp")



# Send query to the database

res <- dbSendQuery(con, "SELECT * FROM comments WHERE user_id > 4")



# Use dbFetch() twice

dbFetch(res, n = 2)

dbFetch(res)
```

```
# Clear res

dbClearResult(res)
```

## Be polite and ...

100xp

Every time you connect to a database using `dbConnect()`, you're creating a new connection to the database you're referencing. `RMySQL` automatically specifies a maximum of open connections and closes some of the connections for you, but still: it's always polite to manually disconnect from the database afterwards. You do this with the `dbDisconnect()` function.
The code that connects you to the database is already available, can you finish the script?

## Instructions

- Using the technique you prefer, build a data frame `long_tweats`. It **selects** the `post` and `date` columns **from** the observations in `tweats` **where** the character length of the `post` variable exceeds `40`.
- Print `long_tweats`.
- Disconnect from the database by using `dbDisconnect()`.

```
library(DBI)

con <- dbConnect(RMySQL::MySQL(),

                 dbname = "tweater",

                 host = "courses.csrrinzqubik.us-east-1.rds.amazonaws.com",

                 port = 3306,

                 user = "student",

                 password = "datacamp")



# Create the data frame  long_tweats

long_tweats <- dbGetQuery(con, "SELECT post, date  FROM tweats WHERE CHAR_LENGTH(post) > 40")
```

# Print long_tweats

print(long_tweats)

# Disconnect from the database

dbDisconnect(con)

## Import flat files from the web

100xp

In the video, you saw that the `utils` functions to import flat file data, such as `read.csv()` and `read.delim()`, are capable of automatically importing from URLs that point to flat files on the web. You must be wondering whether Hadley Wickham's alternative package, `readr`, is equally potent. Well, figure it out in this exercise! The URLs for both a `.csv` file as well as a `.delim` file are already coded for you. It's up to you to actually import the data. If it works, that is...

## Instructions

- Load the `readr` package. It's already installed on DataCamp's servers.
- Use `url_csv` to read in the `.csv` file it is pointing to. Use the `read_csv()` function. The `.csv` contains column names in the first row. Save the resulting data frame as `pools`.
- Similarly, use `url_delim` to read in the online `.txt` file. Use the `read_tsv()` function and store the result as `potatoes`.
- Print `pools` and `potatoes`. Looks correct?

# Load the readr package

library(readr)

# Import the csv file: pools

url_csv <- "http://s3.amazonaws.com/assets.datacamp.com/production/course_1478/datasets/swimming_pools.csv"

pools <- read_csv(url_csv)

# Import the txt file: potatoes

url_delim <- ("http://s3.amazonaws.com/assets.datacamp.com/production/course_1478/datasets/potatoes.txt")

potatoes <- read_tsv(url_delim)

# Print pools and potatoes

pools

potatoes

## Secure importing

100xp

In the previous exercises, you have been working with URLs that all start with `http://`. There is, however, a safer alternative to HTTP, namely HTTPS, which stands for HypterText Transfer Protocol Secure. Just remember this: HTTPS is relatively safe, HTTP is not.
Luckily for us, you can use the standard importing functions with `https://` connections since R version 3.2.2.

## Instructions

- Take a look at the URL in `url_csv`. It uses a secure connection, `https://`.
- Use `read.csv()` to import the file at `url_csv`. The `.csv` file it is referring to contains column names in the first row. Call it `pools1`.
- Load the `readr` package. It's already installed on DataCamp's servers.
- Use `read_csv()` to read in the same `.csv` file in `url_csv`. Call it `pools2`.
- Print out the structure of `pools1` and `pools2`. Looks like the importing went equally well as with a normal `http` connection!

# https URL to the swimming_pools csv file.

url_csv <- "https://s3.amazonaws.com/assets.datacamp.com/production/course_1478/datasets/swimming_pools.csv"

# Import the file using read.csv(): pools1

pools1 <- read.csv(url_csv)

# Load the readr package

library(readr)

# Import the file using read_csv(): pools2

pools2 <- read_csv(url_csv)

# Print the structure of pools1 and pools2

str(pools1)

## Import Excel files from the web

100xp

When you learned about `gdata`, it was already mentioned that `gdata` can handle `.xls` files that are on the internet. `readxl` can't, at least not yet. The URL with which you'll be working is already available in the sample code. You will import it once using `gdata` and once with the `readxl` package via a workaround.

## Instructions

- Load the `readxl` and `gdata` packages. They are already installed on DataCamp's servers.
- Import the `.xls` file located at the URL `url_xls` using `read.xls()` from `gdata`. Store the resulting data frame as `excel_gdata`.
- You can not use `read_excel()` directly with a URL. Complete the following instructions to work around this problem:
    - Use `download.file()` to download the `.xls` file behind the URL and store it locally as `"local_latitude.xls"`.
    - Call `read_excel()` to import the local file, `"local_latitude.xls"`. Name the resulting data frame `excel_readxl`.

library(readxl)

library(gdata)

# Specification of url: url_xls

```
url_xls <- "http://s3.amazonaws.com/assets.datacamp.com/production/course_1478/datasets/latitude.xls"

# Import the .xls file with gdata: excel_gdata
excel_gdata <- read.xls(url_xls)

# Download file behind URL, name it local_latitude.xls
download.file(url_xls,destfile = "local_latitude.xls")

# Import the local .xls file with readxl: excel_readxl
excel_readxl <- read_excel("local_latitude.xls")
```

## Downloading any file, secure or not

100xp

In the previous exercise you've seen how you can read excel files on the web using the `read_excel` package by first downloading the file with the `download.file()` function.
There's more: with `download.file()` you can download any kind of file from the web, using HTTP and HTTPS: images, executable files, but also `.RData` files. An `RData` file is very efficient format to store R data.
You can load data from an `RData` file using the `load()` function, but this function does not accept a URL string as an argument. In this exercise, you'll first download the `RData` file securely, and then import the local data file.

## Instructions

- Take a look at the URL in `url_rdata`. It uses a secure connection, `https://`. This URL points to an `RData` file containing a data frame with some metrics on different kinds of wine.
- Download the file at `url_rdata` using `download.file()`. Call the file `"wine_local.RData"` in your working directory.
- Load the file you created, `wine_local.RData`, using the `load()` function. It takes one argument, the path to the file, which is just the filename in our case. After running this command, the variable `wine` will automatically be available in your workspace.
- Print out the `summary()` of the `wine` dataset

```
# https URL to the wine RData file.

url_rdata <- "https://s3.amazonaws.com/assets.datacamp.com/production/course_1478/datasets/wine.RData"



# Download the wine file to your working directory

download.file(url_rdata, destfile = "wine_local.RData")



# Load the wine data into your workspace using load()

load("wine_local.RData")



# Print out the summary of the wine data

summary(wine)
```

## HTTP? httr! (1)

100xp

Downloading a file from the Internet means sending a GET request and receiving the file you asked for. Internally, all the previously discussed functions use a GET request to download files.

`httr` provides a convenient function, `GET()` to execute this GET request. The result is a `response` object, that provides easy access to the status code, content-type and, of course, the actual content.
You can extract the content from the request using the `content()` function. At the time of writing, there are three ways to retrieve this content: as a raw object, as a character vector, or an R object, such as a list. If you don't tell `content()` how to retrieve the content through the `as` argument, it'll try its best to figure out which type is most appropriate based on the content-type.

## Instructions

- Load the `httr` package. It's already installed on DataCamp's servers.
- Use `GET()` to get the URL stored in `url`. Store the result of this `GET()` call as `resp`.
- Print the `resp` object. What information does it contain?
- Get the content of `resp` using `content()` and set the `as` argument to `"raw"`. Assign the resulting vector to `raw_content`.
- Print the first values in `raw_content` with `head()`.

## HTTP? httr! (1)

100xp

Downloading a file from the Internet means sending a GET request and receiving the file you asked for. Internally, all the previously discussed functions use a GET request to download files.

`httr` provides a convenient function, `GET()` to execute this GET request. The result is a `response` object, that provides easy access to the status code, content-type and, of course, the actual content.
You can extract the content from the request using the `content()` function. At the time of writing, there are three ways to retrieve this content: as a raw object, as a character vector, or an R object, such as a list. If you don't tell `content()` how to retrieve the content through the `as` argument, it'll try its best to figure out which type is most appropriate based on the content-type.

## Instructions

- Load the `httr` package. It's already installed on DataCamp's servers.
- Use `GET()` to get the URL stored in `url`. Store the result of this `GET()` call as `resp`.
- Print the `resp` object. What information does it contain?
- Get the content of `resp` using `content()` and set the `as` argument to `"raw"`. Assign the resulting vector to `raw_content`.
- Print the first values in `raw_content` with `head()`.

```
# Load the httr package

library(httr)



# Get the url, save response to resp

url <- "http://www.example.com/"

resp <- GET(url)



# Print resp

resp



# Get the raw content of resp: raw_content

raw_content <- content(resp, as = "raw")
```

# Print the head of raw_content

head(raw_content)

## HTTP? httr! (2)

100xp

Web content does not limit itself to HTML pages and files stored on remote servers such as DataCamp's Amazon S3 instances. There are many other data formats out there. A very common one is JSON. This format is very often used by so-called Web APIs, interfaces to web servers with which you as a client can communicate to get or store information in more complicated ways.

You'll learn about Web APIs and JSON in the video and exercises that follow, but some experimentation never hurts, does it?

## Instructions

- Use `GET()` to get the `url` that has already been specified in the sample code. Store the response as `resp`.
- Print `resp`. What is the content-type?
- Use `content()` to get the content of `resp`. Set the `as` argument to `"text"`. Simply print out the result. What do you see?
- Use `content()` to get the content of `resp`, but this time do not specify a second argument. R figures out automatically that you're dealing with a JSON, and converts the JSON to a named R list.

```
# httr is already loaded

library(httr)

# Get the url

url <- "http://www.omdbapi.com/?apikey=ff21610b&t=Annie+Hall&y=&plot=short&r=json"

resp <- GET(url)



# Print resp

resp
```

# Print content of resp as text

content(resp, as = "text")

# Print content of resp

content(resp)

## From JSON to R

100xp

In the simplest setting, `fromJSON()` can convert character strings that represent JSON data into a nicely structured R list. Give it a try!

### Instructions

- Load the `jsonlite` package. It's already installed on DataCamp's servers.
- `wine_json` represents a JSON. Use `fromJSON()` to convert it to a list, named `wine`.
- Display the structure of `wine`

# Load the jsonlite package

library(jsonlite)

# wine_json is a JSON

wine_json <- '{"name":"Chateau Migraine", "year":1997, "alcohol_pct":12.4, "color":"red", "awarded":false}'

# Convert wine_json into a list: wine

wine <- fromJSON(wine_json)

# Print structure of wine

str(wine)

## Quandl API

100xp

As Filip showed in the video, `fromJSON()` also works if you pass a URL as a character string or the path to a local file that contains JSON data. Let's try this out on the Quandl API, where you can fetch all sorts of financial and economical data.

## Instructions

- `quandl_url` represents a URL. Use `fromJSON()` directly on this URL and store the result in `quandl_data`.
- Display the structure of `quandl_data`.

# jsonlite is preloaded

library(jsonlite)

# Definition of quandl_url

quandl_url <- "https://www.quandl.com/api/v3/datasets/WIKI/FB/data.json?auth_token=i83asDsiWUUyfoypkgMz"

# Import Quandl data: quandl_data

quandl_data <- fromJSON(quandl_url)

# Print structure of quandl_data

str(quandl_url)

# jsonlite is preloaded

# Definition of quandl_url

quandl_url <- "https://www.quandl.com/api/v3/datasets/WIKI/FB/data.json?auth_token=i83asDsiWUUyfoypkgMz"

# Import Quandl data: quandl_data

quandl_data <- fromJSON(quandl_url)

# Print structure of quandl_data

str(quandl_data)

## OMDb API

100xp

In the video, you saw how easy it is to interact with an API once you know how to formulate requests. You also saw how to fetch all information on Rain Man from OMDb. Simply perform a `GET()` call, and next ask for the contents with the `content()` function. This `content()` function, which is part of the `httr` package, uses `jsonlite` behind the scenes to import the JSON data into R.
However, by now you also know that `jsonlite` can handle URLs itself. Simply passing the request URL to `fromJSON()` will get your data into R. In this exercise, you will be using this technique to compare the release year of two movies in the Open Movie Database.

## Instructions

- Two URLs are included in the sample code, as well as a `fromJSON()` call to build `sw4`. Add a similar call to build `sw3`.
- Print out the element named `Title` of both `sw4` and `sw3`. You can use the `$` operator. What movies are we dealing with here?
- Write an expression that evaluates to `TRUE` if `sw4` was released later than `sw3`. This information is stored in the `Year` element of the named lists.

# The package jsonlite is already loaded

# Definition of the URLs

url_sw4 <- "http://www.omdbapi.com/?apikey=ff21610b&i=tt0076759&r=json"

url_sw3 <- "http://www.omdbapi.com/?apikey=ff21610b&i=tt0121766&r=json"

# Import two URLs with fromJSON(): sw4 and sw3

sw4 <- fromJSON(url_sw4)

sw3 <- fromJSON(url_sw3)

# Print out the Title element of both lists

sw4$Title

sw3$Title

# Is the release year of sw4 later than sw3?

sw4$Year > sw3$Year

## JSON practice (1)

70xp

JSON is built on two structures: objects and arrays. To help you experiment with these, two JSON strings are included in the sample code. It's up to you to change them appropriately and then call `jsonlite`'s `fromJSON()` function on them each time.

## Instructions

- Change the assignment of `json1` such that the R vector after conversion contains the numbers 1 up to 6, in ascending order. Next, call `fromJSON()` on `json1`.
- Adapt the code for `json2` such that it's converted to a named list with two elements: `a`, containing the numbers 1, 2 and 3 and `b`, containing the numbers 4, 5 and 6. Next, call `fromJSON()` on `json2`.

# jsonlite is already loaded

# Challenge 1

json1 <- '[1, 2, 3,4,5, 6]'

```
fromJSON(json1)

# Challenge 2
json2 <- '{"a": [1, 2, 3], "b": [4,5,6]}'
fromJSON(json2)
```

## JSON practice (2)

100xp

We prepared two more JSON strings in the sample code. Can you change them and call `jsonlite`'s `fromJSON()` function on them, similar to the previous exercise?

## Instructions

- Remove characters from `json1` to build a 2 by 2 matrix containing only 1, 2, 3 and 4. Call `fromJSON()` on `json1`.
- Add characters to `json2` such that the data frame in which the json is converted contains an additional observation in the last row. For this observations, `a` equals 5 and `b` equals 6. Call `fromJSON()` one last time, on `json2`

```
# jsonlite is already loaded

# Challenge 1
json1 <- '[[1, 2], [3, 4]]'
fromJSON(json1)

# Challenge 2
json2 <- '[{"a": 1, "b": 2}, {"a": 3, "b": 4}, {"a": 5, "b": 6}]'
fromJSON(json2)
```

## toJSON()

100xp

Apart from converting JSON to R with `fromJSON()`, you can also use `toJSON()` to convert R data to a JSON format. In its most basic use, you simply pass this function an R object to convert to a JSON. The result is an R object of the class `json`, which is basically a character string representing that JSON.

For this exercise, you will be working with a `.csv` file containing information on the amount of desalinated water that is produced around the world. As you'll see, it contains a lot of missing values. This data can be found on the URL that is specified in the sample code.

## Instructions

- Use a function of the `utils` package to import the `.csv` file directly from the URL specified in `url_csv`. Save the resulting data frame as `water`. Make sure that strings are *not* imported as factors.
- Convert the data frame `water` to a JSON. Call the resulting object `water_json`.
- Print out `water_json`.

```
# jsonlite is already loaded


# URL pointing to the .csv file

url_csv <-
"http://s3.amazonaws.com/assets.datacamp.com/production/course_1478/datasets/water.csv"


# Import the .csv file located at url_csv


water <- read.csv(url_csv, stringsAsFactors = FALSE )

# Convert the data file according to the requirements

water_json <- toJSON(water)


# Print out water_json

water_json
```

## Minify and prettify

100xp

JSONs can come in different formats. Take these two JSONs, that are in fact exactly the same: the first one is in a minified format, the second one is in a pretty format with indentation, whitespace and new lines:

```
# Mini
{"a":1,"b":2,"c":{"x":5,"y":6}}

# Pretty
{
  "a": 1,
  "b": 2,
  "c": {
    "x": 5,
    "y": 6
  }
}
```

Unless you're a computer, you surely prefer the second version. However, the standard form that `toJSON()` returns, is the minified version, as it is more concise. You can adapt this behavior by setting the `pretty` argument inside `toJSON()` to `TRUE`. If you already have a JSON string, you can use `prettify()` or `minify()` to make the JSON pretty or as concise as possible.

## Instructions

- Convert the `mtcars` dataset, which is available in R by default, to a *pretty* `JSON`. Call the resulting JSON `pretty_json`.
- Print out `pretty_json`. Can you understand the output easily?
- Convert `pretty_json` to a minimal version using `minify()`. Store this version under a new variable, `mini_json`.
- Print out `mini_json`. Which version do you prefer, the pretty one or the minified one?

```
# jsonlite is already loaded

# Convert mtcars to a pretty JSON: pretty_json

pretty_json <- toJSON(mtcars, pretty = TRUE)

# Print pretty_json

pretty_json

# Minify pretty_json: mini_json

mini_json <- minify(pretty_json)

# Print mini_json
```

mini_json

# Import SAS data with haven

100xp

haven is an extremely easy-to-use package to import data from three software packages: SAS, STATA and SPSS. Depending on the software, you use different functions:

- SAS: `read_sas()`
- STATA: `read_dta()` (or `read_stata()`, which are identical)
- SPSS: `read_sav()` or `read_por()`, depending on the file type.

All these functions take one key argument: the path to your local file. In fact, you can even pass a URL; haven will then automatically download the file for you before importing it.
You'll be working with data on the age, gender, income, and purchase level (0 = low, 1 = high) of 36 individuals (Source: SAS). The information is stored in a SAS file, `sales.sas7bdat`, which is available in your current working directory. You can also download the data here.

## Instructions

- Load the `haven` package; it's already installed on DataCamp's servers.
- Import the data file `"sales.sas7bdat"`. Call the imported data frame `sales`.
- Display the structure of `sales` with `str()`. Some columns represent categorical variables, so they should be factors.

```
library(haven)


# Import sales.sas7bdat: sales

sales <- read_sas("sales.sas7bdat")


# Display the structure of sales

str(sales)
```

haven is an extremely easy-to-use package to import data from three software packages: SAS, STATA and SPSS. Depending on the software, you use different functions:

- SAS: `read_sas()`
- STATA: `read_dta()` (or `read_stata()`, which are identical)

- SPSS: `read_sav()` or `read_por()`, depending on the file type.

All these functions take one key argument: the path to your local file. In fact, you can even pass a URL; `haven` will then automatically download the file for you before importing it.
You'll be working with data on the age, gender, income, and purchase level (0 = low, 1 = high) of 36 individuals (Source: SAS). The information is stored in a SAS file, `sales.sas7bdat`, which is available in your current working directory. You can also download the data here.

## Instructions

- Load the `haven` package; it's already installed on DataCamp's servers.
- Import the data file `"sales.sas7bdat"`. Call the imported data frame `sales`.
- Display the structure of `sales` with `str()`. Some columns represent categorical variables, so they should be factors.

```
# Load the haven package

library(haven)



# Import sales.sas7bdat: sales

sales <- read_sas("sales.sas7bdat")



# Display the structure of sales

str(sales)
```

## Import STATA data with haven

100xp

Next up are STATA data files; you can use `read_dta()` for these.
When inspecting the result of the `read_dta()` call, you will notice that one column will be imported as a `labelled` vector, an R equivalent for the common data structure in other statistical environments. In order to effectively continue working on the data in R, it's best to change this data into a standard R class. To convert a variable of the class `labelled` to a factor, you'll need `haven`'s `as_factor()` function.
In this exercise, you will work with data on yearly import and export numbers of sugar, both in USD and in weight. The data can be found at: http://assets.datacamp.com/production/course_1478/datasets/trade.dta

## Instructions

- Import the data file directly from the URL using `read_dta()`, and store it as `sugar`.

- Print out the structure of `sugar`. The `Date` column has class `labelled`.
- Convert the values in the `Date` column of `sugar` to dates, using `as.Date(as_factor(___))`.
- Print out the structure of `sugar` once more. Looks better now?

```
# haven is already loaded

library(haven)

# Import the data from the URL: sugar

sugar <-read_dta("http://assets.datacamp.com/production/course_1478/datasets/trade.dta")


# Structure of sugar

str(sugar)


# Convert values in Date column to dates

sugar$Date <- as.Date(as_factor(sugar$Date))


# Structure of sugar again

str(sugar)
```

## Import SPSS data with haven

100xp

The `haven` package can also import data files from SPSS. Again, importing the data is pretty straightforward. Depending on the SPSS data file you're working with, you'll need either `read_sav()` - for `.sav` files - or `read_por()` - for `.por` files.
In this exercise, you will work with data on four of the Big Five personality traits for 434 persons (Source: University of Bath). The Big Five is a psychological concept including, originally, five dimensions of personality to classify human personality. The SPSS dataset is called `person.sav` and is available in your working directory.

## Instructions

- Use `read_sav()` to import the SPSS data in `"person.sav"`. Name the imported data frame `traits`.
- `traits` contains several missing values, or `NA`s. Run `summary()` on it to find out how many `NA`s are contained in each variable.

- Print out a subset of those individuals that scored high on Extroversion *and* on Agreeableness, i.e. scoring higher than 40 on each of these two categories. You can use `subset()` for this

```
# haven is already loaded

library(haven)

# Import person.sav: traits

traits <- read_sav("person.sav")


# Summarize traits

summary(traits)


# Print out a subset

subset(traits, Extroversion > 40 & Agreeableness > 40)
```

## Factorize, round two

100xp

In the last exercise you learned how to import a data file using the command `read_sav()`. With SPSS data files, it can also happen that some of the variables you import have the `labelled` class. This is done to keep all the labelling information that was originally present in the `.sav` and `.por` files. It's advised to coerce (or change) these variables to factors or other standard R classes.
The data for this exercise involves information on employees and their demographic and economic attributes (Source: QRiE). The data can be found on the following URL:

http://s3.amazonaws.com/assets.datacamp.com/production/course_1478/datasets/employee.sav

## Instructions

- Import the SPSS data straight from the URL and store the resulting data frame as `work`.
- Display the summary of the `GENDER` column of `work`. This information doesn't give you a lot of useful information, right?
- Convert the `GENDER` column in `work` to a factor, the class to denote categorical variables in R. Use `as_factor()`.
- Once again display the summary of the `GENDER` column. This time, the printout makes much more sense.

```
# haven is already loaded

library(haven)

# Import SPSS data from the URL: work

work <-
read_sav("http://s3.amazonaws.com/assets.datacamp.com/production/course_1478/datasets/employee.sav")


# Display summary of work$GENDER

summary(work$GENDER)




# Convert work$GENDER to a factor


work$GENDER <- as_factor(work$GENDER)


# Display summary of work$GENDER again

summary(work$GENDER)
```

## Import STATA data with foreign (1)

100xp

The `foreign` package offers a simple function to import and read *STATA* data: `read.dta()`.
In this exercise, you will import data on the US presidential elections in the year 2000. The data in `florida.dta` contains the total numbers of votes for each of the four candidates as well as the total number of votes per election area in the state of Florida (Source: Florida Department of State). The file is available in your working directory, you can download it here if you want to experiment some more.

## Instructions

- Load the `foreign` package; it's already installed on DataCamp's servers.
- Import the data on the elections in Florida, `"florida.dta"`, and name the resulting data frame `florida`. Use `read.dta()` without specifying extra arguments.
- Check out the last 6 observations of `florida` with `tail()`

```
# Load the foreign package


library(foreign)

# Import florida.dta and name the resulting data frame florida

florida <- read.dta("florida.dta")


# Check tail() of florida

tail(florida)
```

## Import STATA data with foreign (2)

100xp

Data can be very diverse, going from character vectors to categorical variables, dates and more. It's in these cases that the additional arguments of `read.dta()` will come in handy.
The arguments you will use most often are `convert.dates`, `convert.factors`, `missing.type` and `convert.underscore`. Their meaning is pretty straightforward, as Filip explained in the video. It's all about correctly converting STATA data to standard R data structures. Type `?read.dta` to find out about about the default values.
The dataset for this exercise contains socio-economic measures and access to education for different individuals (Source: World Bank). This data is available as `edequality.dta`, which is located in the `worldbank` folder in your working directory.

## Instructions

- Specify the path to the file using `file.path()`. Call it `path`. Remember the `"edequality.dta"` file is located in the `"worldbank"` folder.
- Use the `path` variable to import the data file in three different ways; each time show its structure with `str()`:
    - `edu_equal_1`: By passing only the file `path` to `read.dta()`.
    - `edu_equal_2`: By passing the file `path`, and setting `convert.factors` to `FALSE`.
    - `edu_equal_3`: By passing the file `path`, and setting `convert.underscore` to `TRUE`.

```
# foreign is already loaded

library(foreign)

# Specify the file path using file.path(): path

path <- file.path("worldbank" ,"edequality.dta")
```

# Create and print structure of edu_equal_1

edu_equal_1 <- read.dta(path)

str(edu_equal_1)

# Create and print structure of edu_equal_2

edu_equal_2 <- read.dta(path,convert.factors = FALSE)

str(edu_equal_2)

# Create and print structure of edu_equal_3

edu_equal_3 <- read.dta(path, convert.underscore = TRUE)

str(edu_equal_3)

## Do you know your data?

50xp

The previous exercise dealt about socio-economic indicators and access to education of different individuals. The `edu_equal_1` dataset that you've built is already available in the workspace. Now that you have it in R, it's pretty easy to get some basic insights.
For example, you can ask yourself *how many observations (e.g. how many people) have an `age` higher than 40 and are `literate`?* When you call

```
str(edu_equal_1)
```

You'll see that `age` is an integer, and `literate` is a factor, with the levels "yes" and "no". The following expression thus answers the question:

```
nrow(subset(edu_equal_1, age > 40 & literate == "yes"))
```

Up to you to answer a similar question now:

*How many observations/individuals from Bulgaria have an income above 1000?*

```
nrow(subset(edu_equal_1, ethnicity_head == “Bulgaria” & income > 1000))
```

## Import SPSS data with foreign (1)

100xp

All great things come in pairs. Where `foreign` provided `read.dta()` to read Stata data, there's also `read.spss()` to read SPSS data files. To get a data frame, make sure to set `to.data.frame = TRUE` inside `read.spss()`.
In this exercise, you'll be working with socio-economic variables from different countries (Source: Quantative Data Analysis in Education). The SPSS data is in a file called `international.sav`, which is in your working directory. You can also download it here if you want to play around with it some more.

## Instructions

- Import the data file `"international.sav"` and have R convert it to a data frame. Store this data frame as `demo`.
- Create a boxplot of the `gdp` variable of `demo`.

```
# foreign is already loaded

library(foreign)

# Import international.sav as a data frame: demo

demo <- read.spss("international.sav", to.data.frame = TRUE )



# Create boxplot of gdp variable of demo

boxplot(demo$gdp)
```

## Import SPSS data with foreign (2)

100xp

In the previous exercise, you used the `to.data.frame` argument inside `read.spss()`. There are many other ways in which to customize the way your SPSS data is imported.
In this exercise you will experiment with another argument, `use.value.labels`. It specifies whether variables with value labels should be converted into R factors with levels that are named accordingly. The argument is `TRUE` by default which means that so called labelled variables inside SPSS are converted to factors inside R.
You'll again be working with the international.sav data, which is available in your current working directory.

## Instructions

- Import the data file `"international.sav"` as a data frame, `demo_1`.
- Print the first few rows of `demo_1` using the `head()` function.
- Import the data file `"international.sav"` as a data frame, `demo_2`, but this time in a way such that variables with value labels are *not* converted to R factors.
- Again, print the first few rows of `demo_2`. Can you tell the difference between the two data frames?

```
# foreign is already loaded
```

```
library(foreign)

# Import international.sav as demo_1

demo_1 <- read.spss("international.sav",to.data.frame = TRUE )


# Print out the head of demo_1

head(demo_1)


# Import international.sav as demo_2

demo_2 <- read.spss("international.sav",to.data.frame = TRUE ,use.value.labels = FALSE)


# Print out the head of demo_2

head(demo_2)
```