Intermediate R

1. Conditionals and Control Flow

Equality

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

```
3 == (2 + 1)
```

```
"intermediate" != "r"
```

```
TRUE != FALSE
```

```
"Rchitect" != "rchitect"
```

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

Greater and less than

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

```
(1 + 2) > 4
```

```
"dog" < "Cats"
```

```
TRUE <= FALSE
```

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

Compare vectors

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

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

Compare matrices

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

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

& and |

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

- `TRUE & TRUE`
- `FALSE | TRUE`
- `5 <= 5 & 2 < 3`
- `3 < 4 | 7 < 6`

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

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

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

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

Blend it all together

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

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

The if statement

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

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

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

Add an else

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

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

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

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

Customize further: else if

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

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

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

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

```
if (number < 10) {
  if (number < 5) {
    result <- "extra small"
  } else {
    result <- "small"
  }
```

```
} else if (number < 100) {
  result <- "medium"
} else {
  result <- "large"
}
print(result)
```

Have a look at the following statements:

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

Select the option that lists all the true statements.

2. Loops

Write a while loop

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

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

Remember that the `condition` part of this recipe should become `FALSE` at some point during the execution. Otherwise, the `while` loop will go on indefinitely. In DataCamp's learning interface, your session will be disconnected in this case.

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

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

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

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

Stop the while loop: break

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

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

Loop over a vector

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

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

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

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

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

Loop over a list

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

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

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

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

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

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

Loop over a matrix

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

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

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

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

Mix it up with control flow

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

Next, you break it

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

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

The `break` statement abandons the active loop: the remaining code in the loop is skipped and the loop is not iterated over anymore.

The `next` statement skips the remainder of the code in the loop, but continues the iteration.

Build a for loop from scratch

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

In the editor on the right, we already went ahead and defined a variable `rquote`. This variable has been split up into a vector that contains separate letters and has been stored in a vector `chars`with the **`strsplit()`** function.

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

3. Functions

Function documentation

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

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

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

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

```
args(sample)
```

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

Use a function

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Functions inside functions

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

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

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

make sure to specify `na.rm` to treat missing values correctly! => `na.rm = TRUE`

Required, or optional?

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

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

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

The following statements about the **`read.table()`** function are true:

1. `header`, `sep` and `quote` are all optional arguments.
2. `read.table("myfile.txt", "-", TRUE)` will throw an error.

Write your own function

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

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

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

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

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

throw_die()
```

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

Do you still remember the difference between an argument with and without default values? Have another look at the **`sd()`** function by typing `?sd` in the console. The usage section shows the following information:

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

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

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

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

Function scoping

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

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

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

R passes arguments by value

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

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

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

The following statements are true about this piece of code?

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

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

R you functional?

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

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

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

Load an R Package

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

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

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

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

Before starting, execute the following commands in the console:

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

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

4. lapply = list apply

Use lapply with a built-in R function

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

```
lapply(X, FUN, ...)
```

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

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

Use lapply with your own function

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

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

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

lapply and anonymous functions

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

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

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

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

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

Use lapply with additional arguments

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

```
multiply <- function(x, factor) {
  x * factor
}
```

```
lapply(list(1,2,3), multiply, factor = 3)
```

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

Apply functions that return NULL

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

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

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

```
str(TRUE)
```

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

Did you notice that lapply() always returns a list, no matter the input? This can be kind of annoying. In the next video tutorial you'll learn about sapply() to solve this.

How to use sapply SAPPLY = Simplified Apply

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

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

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

sapply with your own function

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

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

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

sapply with function returning vector

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

Use vapply

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

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

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

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

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

From sapply to vapply

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

5. useful functions

Mathematical utilities

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

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

As a data scientist in training, you've estimated a regression model on the sales data for the past six months. After evaluating your model, you see that the training error of your model is quite regular, showing both positive and negative values. The error values are already defined in the workspace on the right (`errors`).

Find the error

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

In times of despair, help with functions such as `sum()` and `rev()` are a single command away; simply use `?sum` and `?rev` in the console.

Data Utilities

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

- `seq()`: Generate sequences, by specifying the `from`, `to`, and `by` arguments.
- `rep()`: Replicate elements of vectors and lists.
- `sort()`: Sort a vector in ascending order. Works on numerics, but also on character strings and logicals.
- `rev()`: Reverse the elements in a data structures for which reversal is defined.
- `str()`: Display the structure of any R object.
- `append()`: Merge vectors or lists.
- `is.*()`: Check for the class of an R object.
- `as.*()`: Convert an R object from one class to another.
- `unlist()`: Flatten (possibly embedded) lists to produce a vector.

Beat Gauss using R

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

grepl & grep

In their most basic form, regular expressions can be used to see whether a pattern exists inside a character string or a vector of character strings. For this purpose, you can use:

- `grepl()`, which returns `TRUE` when a pattern is found in the corresponding character string.
- `grep()`, which returns a vector of indices of the character strings that contains the pattern.

Both functions need a `pattern` and an `x` argument, where `pattern` is the regular expression you want to match for, and the `x` argument is the character vector from which matches should be sought.

In this and the following exercises, you'll be querying and manipulating a character vector of email addresses! The vector `emails` has already been defined in the editor on the right so you can begin with the instructions straight away!

You can use the caret, `^`, and the dollar sign, `$` to match the content located in the start and end of a string, respectively. This could take us one step closer to a correct pattern for matching only the ".edu" email addresses from our list of emails. But there's more that can be added to make the pattern more robust:

- `@`, because a valid email must contain an at-sign.
- `.*`, which matches any character (.) zero or more times (*). Both the dot and the asterisk are metacharacters. You can use them to match any character between the at-sign and the ".edu" portion of an email address.
- `\\.edu$`, to match the ".edu" part of the email at the end of the string. The `\\` part *escapes*the dot: it tells R that you want to use the `.` as an actual character.

sub & gsub

While `grep()` and `grepl()` were used to simply check whether a regular expression could be matched with a character vector, `sub()` and `gsub()` take it one step further: you can specify a `replacement` argument. If inside the character vector `x`, the regular expression `pattern` is found, the matching element(s) will be replaced with `replacement`.`sub()` only replaces the first match, whereas `gsub()` replaces all matches.

Suppose that `emails` vector you've been working with is an excerpt of DataCamp's email database. Why not offer the owners of the .edu email addresses a new email address on the datacamp.edu domain? This could be quite a powerful marketing stunt: Online education is taking over traditional learning institutions! Convert your email and be a part of the new generation!

Regular expressions are a typical concept that you'll learn by doing and by seeing other examples. Before you rack your brains over the regular expression in this exercise, have a look at the new things that will be used:

- `.*`: A usual suspect! It can be read as "any character that is matched zero or more times".
- `\\s`: Match a space. The "s" is normally a character, escaping it (`\\`) makes it a metacharacter.
- `[0-9]+`: Match the numbers 0 to 9, at least once (+).
- `([0-9]+)`: The parentheses are used to make parts of the matching string available to define the replacement. The `\\1` in the `replacement` argument of `sub()` gets set to the string that is captured by the regular expression `[0-9]+`.

Right here, right now

In R, dates are represented by `Date` objects, while times are represented by `POSIXct` objects. Under the hood, however, these dates and times are simple numerical values. `Date` objects store the number of days since the 1st of January in 1970. `POSIXct` objects on the other hand, store the number of seconds since the 1st of January in 1970.

The 1st of January in 1970 is the common origin for representing times and dates in a wide range of programming languages. There is no particular reason for this; it is a simple convention. Of course, it's also possible to create dates and times before 1970; the corresponding numerical values are simply negative in this case.

Create and format dates

To create a `Date` object from a simple character string in R, you can use the **`as.Date()`** function. The character string has to obey a format that can be defined using a set of symbols (the examples correspond to 13 January, 1982):

- `%Y`: 4-digit year (1982)
- `%y`: 2-digit year (82)
- `%m`: 2-digit month (01)
- `%d`: 2-digit day of the month (13)
- `%A`: weekday (Wednesday)
- `%a`: abbreviated weekday (Wed)
- `%B`: month (January)
- `%b`: abbreviated month (Jan)

The following R commands will all create the same `Date` object for the 13th day in January of 1982:

```
as.Date("1982-01-13")
as.Date("Jan-13-82", format = "%b-%d-%y")
as.Date("13 January, 1982", format = "%d %B, %Y")
```

Notice that the first line here did not need a format argument, because by default R matches your character string to the formats `"%Y-%m-%d"` or `"%Y/%m/%d"`.

Create and format times

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

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

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

```
?strptime
```

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

Calculations with Dates

Both `Date` and `POSIXct` R objects are represented by simple numerical values under the hood. This makes calculation with time and date objects very straightforward: R performs the calculations using the underlying numerical values, and then converts the result back to human-readable time information again.

You can increment and decrement `Date` objects, or do actual calculations with them (try it out in the console!):

```
today <- Sys.Date()
today + 1
today - 1

as.Date("2015-03-12") - as.Date("2015-02-27")
```

Calculations with Times

Calculations using `POSIXct` objects are completely analogous to those using `Date` objects. Try to experiment with this code to increase or decrease `POSIXct` objects:

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

Adding or substracting time objects is also straightforward:

```
birth <- as.POSIXct("1879-03-14 14:37:23")
death <- as.POSIXct("1955-04-18 03:47:12")
einstein <- death - birth
einstein
```

Time is of the essence

The dates when a season begins and ends can vary depending on who you ask. People in Australia will tell you that spring starts on September 1st. The Irish people in the Northern hemisphere will swear that spring starts on February 1st, with the celebration of St. Brigid's Day. Then there's also the difference between astronomical and meteorological seasons: while astronomers are used to equinoxes and solstices, meteorologists divide the year into 4 fixed seasons that are each three months long. (source: www.timeanddate.com)