# 5 Dealing with messy data

…or, put differently, *welcome to the real world*. Real datasets are seldom as tidy and clean as those you have seen in the previous examples in this book. On the contrary, real data is messy. Things will be out of place and formatted in the wrong way. You'll need to filter the rows to remove those that aren't supposed to be used in the analysis. You'll need to remove some columns and merge others. You will need to wrestle, clean, coerce, and coax your data until it finally has the right format. Only then will you be able to actually analyse it.

This chapter contains a number of examples that serve as cookbook recipes for common data wrangling tasks. And as with any cookbook, you'll find yourself returning to some recipes more or less every day, until you know them by heart, while you never find the right time to use other recipes. You definitely do not have to know all of them by heart and can always go back and look up a recipe that you need.

After working with the material in this chapter, you will be able to use R to:

- Handle numeric and categorical data,
- Manipulate and find patterns in text strings,
- Work with dates and times,
- Filter, subset, sort, and reshape your data using `data.table` , `dplyr` , and `tidyr` ,
- Split and merge datasets,
- Scrape data from the web, and
- Import data from different file formats.

## 5.1 Changing data types

In Exercise 2.21 you discovered that R implicitly coerces variables into other data types when needed. For instance, if you add a `numeric` to a `logical` , the result is a `numeric` . And if you place them together in a vector, the vector will contain two `numeric` values:

```
TRUE + 5
v1 <- c(TRUE, 5)
v1
```

However, if you add a `numeric` to a `character`, the operation fails. And if you put them together in a vector, both become `character` strings:

```
"One" + 5
v2 <- c("One", 5)
v2
```

There is a hierarchy for data types in R: `logical` < `integer` < `numeric` < `character`. When variables of different types are somehow combined (with addition, put in the same vector, and so on), R will coerce both to the higher ranking type. That is why `v1` contained `numeric` variables (`numeric` is higher ranked than `logical`) and `v2` contained `character` values (`character` is higher ranked than `numeric`).

Automatic coercion is often useful, but will sometimes cause problems. As an example, a vector of numbers may accidentally be converted to a `character` vector, which will confuse plotting functions. Luckily it is possible to convert objects to other data types. The functions most commonly used for this are `as.logical`, `as.numeric` and `as.character`. Here are some examples of how they can be used:

```
as.logical(1)           # Should be TRUE
as.logical("FALSE")     # Should be FALSE
as.numeric(TRUE)        # Should be 1
as.numeric("2.718282")  # Should be numeric 2.718282
as.character(2.718282)  # Should be the string "2.718282"
as.character(TRUE)      # Should be the string "TRUE"
```

A word of warning though – conversion only works if R can find a natural conversion between the types. Here are some examples where conversion fails. Note that only some of them cause warning messages:

```
as.numeric("two")                   # Should be 2
as.numeric("1+1")                   # Should be 2
as.numeric("2,718282")              # Should be numeric 2.718282
as.logical("Vaccines cause autism") # Should be FALSE
```

∼

**Exercise 5.1** The following tasks are concerned with converting and checking data types:

1. What happens if you apply `as.logical` to the `numeric` values 0 and 1? What happens if you apply it to other numbers?
2. What happens if you apply `as.character` to a vector containing `numeric` values?
3. The functions `is.logical` , `is.numeric` and `is.character` can be used to check if a variable is a `logical` , `numeric` or `character` , respectively. What type of object do they return?
4. Is `NA` a `logical` , `numeric` or `character` ?

(Click here to go to the solution.)

## 5.2 Working with lists

A data structure that is very convenient for storing data of different types is `list` . You can think of a `list` as a data frame where you can put different types of objects in each column: like a `numeric` vector of length 5 in the first, a data frame in the second, and a single `character` in the third[31]. Here is an example of how to create a `list` using the function of the same name:

```
my_list <- list(my_numbers = c(86, 42, 57, 61, 22),
                my_data = data.frame(a = 1:3, b = 4:6),
                my_text = "Lists are the best.")
```

To access the elements in the list, we can use the same `$` notation as for data frames:

```
my_list$my_numbers
my_list$my_data
my_list$my_text
```

In addition, we can access them using indices, but using *double* brackets:

```
my_list[[1]]
my_list[[2]]
my_list[[3]]
```

To access elements within the elements of lists, additional brackets can be added. For instance, if you wish to access the second element of the `my_numbers` vector, you can use either of these:

```
my_list[[1]][2]
my_list$my_numbers[2]
```

### 5.2.1 Splitting vectors into lists

Consider the `airquality` dataset, which among other things describes the temperature on each day during a five-month period. Suppose that we wish to split the `airquality$Temp` vector into five separate vectors: one for each month. We could do this by repeated filtering, e.g.,

```
temp_may <- airquality$Temp[airquality$Month == 5]
temp_june <- airquality$Temp[airquality$Month == 6]
# ...and so on.
```

Apart from the fact that this isn't a very good-looking solution, this would be infeasible if we needed to split our vector into a larger number of new vectors. Fortunately, there is a function that allows us to split the vector by month, storing the result as a list - `split` :

```r
temps <- split(airquality$Temp, airquality$Month)
temps

# To access the temperatures for June:
temps$`6`
temps[[2]]

# To give more informative names to the elements in the list:
names(temps) <- c("May", "June", "July", "August", "September")
temps$June
```

Note that, in breach of the rules for variable names in R, the original variable names here were numbers (actually `character` variables that happened to contain numeric characters). When accessing them using `$` notation, you need to put them between backticks ( ` ), e.g., `` temps$`6` ``, to make it clear that `6` is a variable name and not a number.

## 5.2.2 Collapsing lists into vectors

Conversely, there are times where you want to collapse a list into a vector. This can be done using `unlist`:

```r
unlist(temps)
```

~

**Exercise 5.2** Load the `vas.csv` data from Exercise 2.30. Split the `VAS` vector so that you get a list containing one vector for each patient. How can you then access the visual analogue scale (VAS) values for patient 212?

(Click here to go to the solution.)

## 5.3 Working with numbers

A lot of data analyses involve numbers, which typically are represented as `numeric` values in R. We've already seen in Section 2.4.5 that there are numerous mathematical operators that can be applied to numbers in R. But there are also other functions that come in handy when working with numbers.

### 5.3.1 Rounding numbers

At times you may want to round numbers, either for presentation purposes or for some other reason. There are several functions that can be used for this:

```
a <- c(2.1241, 3.86234, 4.5, -4.5, 10000.1001)
round(a, 3)           # Rounds to 3 decimal places
signif(a, 3)          # Rounds to 3 significant digits
ceiling(a)            # Rounds up to the nearest integer
floor(a)              # Rounds down to the nearest integer
trunc(a)              # Rounds to the nearest integer, toward 0
                      # (note the difference in how 4.5
                      #  and -4.5 are treated!)
```

### 5.3.2 Sums and means in data frames

When working with numerical data, you'll frequently find yourself wanting to compute sums or means of either columns or rows of data frames. The `colSums` , `rowSums` , `colMeans` and `rowMeans` functions can be used to do this. Here is an example with an expanded version of the `bookstore` data, where three purchases have been recorded for each customer:

```
bookstore2 <- data.frame(purchase1 = c(20, 59, 2, 12, 22, 160,
                                        34, 34, 29),
                         purchase2 = c(14, 67, 9, 20, 20, 81,
                                        19, 55, 8),
                         purchase3 = c(4, 62, 11, 18, 33, 57,
                                        24, 49, 29))

colSums(bookstore2)   # The total amount for customers' 1st, 2nd and
                      # 3rd purchases
rowSums(bookstore2)   # The total amount for each customer
colMeans(bookstore2)  # Mean purchase for 1st, 2nd and 3rd purchases
rowMeans(bookstore2)  # Mean purchase for each customer
```

Moving beyond sums and means, in Section 6.5 you'll learn how to apply any function to the rows or columns of a data frame.

### 5.3.3 Summaries of series of numbers

When a `numeric` vector contains a series of consecutive measurements, as is the case, e.g., in a time series, it is often of interest to compute various cumulative summaries. For instance, if the vector contains the daily revenue of a business during a month, it may be of value to know the total revenue up to each day, that is, the *cumulative sum* for each day.

Let's return to the `a10` data from Section 4.7, which described the monthly anti-diabetic drug sales in Australia during 1991-2008.

```r
library(fpp2)
a10
```

Elements 7 to 18 contain the sales for 1992. We can compute the total, highest, and smallest monthly sales up to and including each month using `cumsum` , `cummax` , and `cummin` :

```r
a10[7:18]
cumsum(a10[7:18])  # Total sales
cummax(a10[7:18])  # Highest monthly sales
cummin(a10[7:18])  # Lowest monthly sales

# Plot total sales up to and including each month:
plot(1:12, cumsum(a10[7:18]),
     xlab = "Month",
     ylab = "Total sales",
     type = "b")
```

In addition, the `cumprod` function can be used to compute cumulative products.

At other times, we are interested in studying *run lengths* in series, that is, the lengths of runs of equal values in a vector. Consider the `upp_temp` vector defined in the code chunk below, which contains the daily temperatures in Uppsala, Sweden, in February 2020[32].

```r
upp_temp <- c(5.3, 3.2, -1.4, -3.4, -0.6, -0.6, -0.8, 2.7, 4.2, 5.7,
              3.1, 2.3, -0.6, -1.3, 2.9, 6.9, 6.2, 6.3, 3.2, 0.6, 5.5,
              6.1, 4.4, 1.0, -0.4, -0.5, -1.5, -1.2, 0.6)
```

It could be interesting to look at runs of sub-zero days, i.e., consecutive days with sub-zero temperatures. The `rle` function counts the lengths of runs of equal values in a vector. To find the length of runs of temperatures below or above zero we can use the vector defined by the condition `upp_temp < 0`, the values of which are `TRUE` on sub-zero days and `FALSE` when the temperature is 0 or higher. When we apply `rle` to this vector, it returns the length and value of the runs:

```
rle(upp_temp < 0)
```

We first have a 2-day run of above zero temperatures ( `FALSE` ), then a 5-day run of sub-zero temperatures ( `TRUE` ), then a 5-day run of above zero temperatures, and so on.

## 5.3.4 Scientific notation: `1e-03`

When printing very large or very small numbers, R uses *scientific notation*, meaning that 7,000,000 (7 followed by 6 zeroes) is displayed as (the mathematically equivalent) $7 \cdot 10^6$, and $0.0000007$ is displayed as $7 \cdot 10^{-7}$. Well, almost, the *10 raised to the power of x* bit isn't really displayed as $10^x$, but as `e+x`, a notation used in many programming languages and calculators. Here are some examples:

```
7000000
0.0000007
7e+07
exp(30)
```

Scientific notation is a convenient way to display very large and very small numbers, but it's not always desirable. If you just want to print the number, the `format` function can be used to convert it to a character, suppressing scientific notation:

```
format(7000000, scientific = FALSE)
```

If you still want your number to be a `numeric` (as you often do), a better choice is to change the option for when R uses scientific notation. This can be done using the `scipen` argument in the `options` function:

```
options(scipen = 1000)
7000000
0.0000007
7e+07
exp(30)
```

To revert this option back to the default, you can use:

```
options(scipen = 0)
7000000
0.0000007
7e+07
exp(30)
```

Note that this option only affects how R *prints* numbers, and not how they are treated in computations.

### 5.3.5 Floating point arithmetics

Some numbers cannot be written in finite decimal forms. Take $1/3$ for example, the decimal form of which is

$$0.333333333333333333333333333333\ldots.$$

Clearly, the computer cannot store this number exactly, as that would require an infinite memory[33]. Because of this, numbers in computers are stored as *floating point numbers*, which aim to strike a balance between *range* (being able to store both very small and very large numbers) and *precision* (being able to represent numbers accurately). Most of the time, calculations with floating points yield exactly the results that we'd expect, but sometimes these non-exact representations of numbers will cause unexpected problems. If we wish to compute $1.5 - 0.2$ and $1.1 - 0.2$, say, we could of course use R for that. Let's see if it gets the answers right:

```
1.5 - 0.2
1.5 - 0.2 == 1.3  # Check if 1.5-0.2=1.3
1.1 - 0.2
1.1 - 0.2 == 0.9  # Check if 1.1-0.2=0.9
```

The limitations of floating point arithmetics cause the second calculation to fail. To see what has happened, we can use `sprintf` to print numbers with 30 decimals (by default, R prints a rounded version with fewer decimals):

```
sprintf("%.30f", 1.1 - 0.2)
sprintf("%.30f", 0.9)
```

The first 12 decimals are identical, but after that the two numbers `1.1 - 0.2` and `0.9` diverge. In our other example, $1.5 - 0.2$, we don't encounter this problem – both `1.5 - 0.2` and `0.3` have the same floating point representation:

```
sprintf("%.30f", 1.5 - 0.2)
sprintf("%.30f", 1.3)
```

The order of the operations also matters in this case. The following three calculations would all yield identical results if performed with real numbers, but in floating point arithmetics the results differ:

```
1.1 - 0.2 - 0.9
1.1 - 0.9 - 0.2
1.1 - (0.9 + 0.2)
```

In most cases, it won't make a difference whether a variable is represented as $0.90000000000000013\ldots$ or $0.90000000000000002\ldots$, but in some cases tiny differences like that can propagate and cause massive problems. A famous example of this involves the US Patriot surface-to-air defence system, which at the end of the first Gulf War missed an incoming missile due to an error in floating point arithmetics[34]. It is important to be aware of the fact that floating point arithmetics occasionally will yield incorrect results. This can happen for numbers of any size, but it is more likely to occur when very large and very small numbers appear in the same computation.

So, `1.1 - 0.2` and `0.9` may not be the same thing in floating point arithmetics, but at least they are *nearly* the same thing. The `==` operator checks if two numbers are exactly equal, but there is an alternative that can be used to check if two numbers are nearly equal: `all.equal`. If the two numbers are (nearly) equal, it returns `TRUE`, and if they are not, it returns a description of how they differ. In order to avoid the latter, we can use the `isTRUE` function to return `FALSE` instead:

```
1.1 - 0.2 == 0.9
all.equal(1.1 - 0.2, 0.9)
all.equal(1, 2)
isTRUE(all.equal(1, 2))
```

∼

**Exercise 5.3** These tasks showcase some problems that are commonly faced when working with numeric data:

1. The vector `props <- c(0.1010, 0.2546, 0.6009, 0.0400, 0.0035)` contains proportions (which, by definition, are between 0 and 1). Convert the proportions to percentages with one decimal place.
2. Compute the highest and lowest temperatures up to and including each day in the `airquality` dataset.
3. What is the longest run of days with temperatures above 80 in the `airquality` dataset?

(Click here to go to the solution.)

**Exercise 5.4** These tasks are concerned with floating point arithmetics:

1. Very large numbers, like `10e500`, are represented by `Inf` (infinity) in R. Try to find out what the largest number that can be represented as a floating point number in R is.
2. Due to an error in floating point arithmetics, `sqrt(2)^2 - 2` is not equal to `0`. Change the order of the operations so that the results is `0`.

(Click here to go to the solution.)

## 5.4 Working with categorical data and factors

In Sections 2.6.2 and 2.8 we looked at how to analyse and visualise categorical data, i.e., data where the variables can take a fixed number of possible values that somehow correspond to groups or categories. But so far we haven't really gone into how to handle categorical variables in R.

Categorical data is stored in R as `factor` variables. You may ask why a special data structure is needed for categorical data, when we could just use `character` variables to represent the categories. Indeed, the latter is what R does by default, e.g., when creating a `data.frame` object or reading data from `.csv` and `.xlsx` files.

Let's say that you've conducted a survey on students' smoking habits. The possible responses are *Never*, *Occasionally*, *Regularly*, and *Heavy*. From 10 students, you get the following responses:

```
smoke <- c("Never", "Never", "Heavy", "Never", "Occasionally",
           "Never", "Never", "Regularly", "Regularly", "No")
```

Note that the last answer is invalid – `No` was not one of the four answers that were allowed for the question.

You could use `table` to get a summary of how many answers of each type that you got:

```
table(smoke)
```

But the categories are not presented in the correct order! There is a clear order between the different categories, *Never* < *Occasionally* < *Regularly* < *Heavy*, but `table` doesn't present the results in that way. Moreover, R didn't recognise that `No` was an invalid answer and treats it just the same as the other categories.

This is where `factor` variables come in. They allow you to specify which values your variable can take, and the ordering between them (if any).

### 5.4.1 Creating factors

When creating a `factor` variable, you typically start with a `character` , `numeric` or `logical` variable, the values of which are turned into categories. To turn the `smoke` vector that you created in the previous section into a `factor` , you can use the `factor` function:

```
smoke2 <- factor(smoke)
```

You can inspect the elements, and *levels*, i.e., the values that the categorical variable takes, as follows:

```
smoke2
levels(smoke2)
```

So far, we haven't solved neither the problem of the categories being in the wrong order nor that invalid `No` value. To fix both these problems, we can use the `levels` argument in `factor`:

```
smoke2 <- factor(smoke, levels = c("Never", "Occasionally",
                                   "Regularly", "Heavy"),
                        ordered = TRUE)

# Check the results:
smoke2
levels(smoke2)
table(smoke2)
```

You can control the order in which the levels are presented by choosing which order we write them in the `levels` argument. The `ordered = TRUE` argument specifies that the order of the variables is *meaningful*. It can be excluded in cases where you wish to specify the order in which the categories should be presented purely for presentation purposes (e.g., when specifying whether to use the order `Male/Female/Other` or `Female/Male/Other`). Also note that the `No` answer now became an `NA`, which in the case of `factor` variables represents both missing observations and invalid observations. To find the values of `smoke` that became `NA` in `smoke2`, you can use `which` and `is.na`:

```
smoke[which(is.na(smoke2))]
```

By checking the original values of the `NA` elements, you can see if they should be excluded from the analysis or recoded into a proper category (`No` could for instance be recoded into `Never`). In Section 5.5.3 you'll learn how to replace values in larger datasets automatically using regular expressions.

### 5.4.2 Changing factor levels

When we created `smoke2`, one of the elements became an `NA`. `NA` was however not included as a level of the `factor`. Sometimes, it is desirable to include `NA` as a level, for instance when you want to analyse rows with missing data. This is easily done using the `addNA` function:

```
smoke2 <- addNA(smoke2)
```

If you wish to change the name of one or more of the `factor` levels, you can do it directly via the `levels` function. For instance, we can change the name of the `NA` category, which is the 5th level of `smoke2`, as follows:

```
levels(smoke2)[5] <- "Invalid answer"
```

The above solution is a little brittle in that it relies on specifying the index of the level name, which can change if we're not careful. More robust solutions using the `data.table` and `dplyr` packages are presented in Section 5.7.6.

Finally, if you've added more levels than what are actually used, these can be dropped using the `droplevels` function:

```
smoke2 <- factor(smoke, levels = c("Never", "Occasionally",
                                   "Regularly", "Heavy",
                                   "Constantly"),
                        ordered = TRUE)
levels(smoke2)
smoke2 <- droplevels(smoke2)
levels(smoke2)
```

### 5.4.3 Changing the order of levels

Now suppose that we'd like the levels of the `smoke2` variable to be presented in the reverse order: *Heavy*, *Regularly*, *Occasionally*, and *Never*. This can be done by a new call to `factor`, where the new level order is specified in the `levels` argument:

```r
smoke2 <- factor(smoke2, levels = c("Heavy", "Regularly",
                                    "Occasionally", "Never"))

# Check the results:
levels(smoke2)
```

## 5.4.4 Combining levels

Finally, `levels` can be used to merge categories by replacing their separate names with a single name. For instance, we can combine the smoking categories *Occasionally*, *Regularly*, and *Heavy* to a single category named *Yes*. Assuming that these are first, second, and third in the list of names (as will be the case if you've run the last code chunk above), here's how to do it:

```r
levels(smoke2)[1:3] <- "Yes"

# Check the results:
levels(smoke2)
```

Alternative ways to do this are presented in Section 5.7.6.

∼

**Exercise 5.5** In Exercise 2.27 you learned how to create a `factor` variable from a `numeric` variable using `cut`. Return to your solution (or the solution at the back of the book) and do the following:

1. Change the category names to `Mild`, `Moderate`, and `Hot`.
2. Combine `Moderate` and `Hot` into a single level named `Hot`.

(Click here to go to the solution.)

**Exercise 5.6** Load the `msleep` data from the `ggplot2` package. Note that categorical variable `vore` is stored as a `character`. Convert it to a `factor` by running `msleep$vore <- factor(msleep$vore)`.

1. How are the resulting factor levels ordered? Why are they ordered in that way?

2. Compute the mean value of `sleep_total` for each `vore` group.
3. Sort the factor levels according to their `sleep_total` means. Hint: this can be done manually, or more elegantly using, e.g., a combination of the functions `rank` and `match` in an intermediate step.

(Click here to go to the solution.)

## 5.5 Working with strings

Text in R is represented by `character` strings. These are created using double or single quotes. I recommend double quotes for three reasons. First, it is the default in R and is the recommended style (see, e.g., `?Quotes`). Second, it improves readability – code with double quotes is easier to read because double quotes are easier to spot than single quotes. Third, it will allow you to easily use apostrophes in your strings, which single quotes don't (because apostrophes will be interpreted as the end of the string). Single quotes can however be used if you need to include double quotes inside your string:

```
# This works:
text1 <- "An example of a string. Isn't this great?"
text2 <- 'Another example of a so-called "string".'

# This doesn't work:
text1_fail <- 'An example of a string. Isn't this great?'
text2_fail <- "Another example of a so-called "string"."
```

If you check what these two strings look like, you'll notice something funny about `text2`:

```
text1
text2
```

R has put backslash characters, `\`, before the double quotes. The backslash is called an *escape character*, which invokes a different interpretation of the character that follows it. In fact, you can use this to put double quotes inside a string that you define using double quotes:

```
text2_success <- "Another example of a so-called \"string\"."
```

There are a number of other special characters that can be included using a backslash: `\n` for a line break (a new line) and `\t` for a tab (a long whitespace) being the most important[35]:

```
text3 <- "Text...\n\tWith indented text on a new line!"
```

To print your string in the Console in a way that shows special characters instead of their escape character-versions, use the function `cat`:

```
cat(text3)
```

You can also use `cat` to print the string to a text file…

```
cat(text3, file = "new_findings.txt")
```

…and to append text at the end of a text file:

```
cat("Let's add even more text!", file = "new_findings.txt",
    append = TRUE)
```

(Check the output by opening `new_findings.txt`!)

## 5.5.1 Concatenating strings

If you wish to concatenate multiple strings, `cat` will do that for you:

```
first <- "This is the beginning of a sentence"
second <- "and this is the end."
cat(first, second)
```

By default, `cat` places a single white space between the two strings, so that `"This is the beginning of a sentence"` and `"and this is the end."` are concatenated to `"This is the beginning of a sentence and this is the end."`. You can change that using the `sep` argument in `cat`. You can also add as many strings as you like as input:

```
cat(first, second, sep = "; ")
cat(first, second, sep = "\n")
cat(first, second, sep = "")
cat(first, second, "\n", "And this is another sentence.")
```

At other times, you want to concatenate two or more strings without printing them. You can then use `paste` in exactly the same way as you'd use `cat`, the exception being that `paste` returns a string instead of printing it.

```
my_sentence <- paste(first, second, sep = "; ")
my_novel <- paste(first, second, "\n",
                  "And this is another sentence.")

# View results:
my_sentence
my_novel
cat(my_novel)
```

Finally, if you wish to create a number of similar strings based on information from other variables, you can use `sprintf`, which allows you to write a string using `%s` as a placeholder for the values that should be pulled from other variables:

```
names <- c("Irma", "Bea", "Lisa")
ages <- c(5, 59, 36)

sprintf("%s is %s years old.", names, ages)
```

There are many more uses of `sprintf` (we've already seen some in Section 5.3.5), but this is enough for us for now.

### 5.5.2 Changing case

If you need to translate characters from lowercase to uppercase or vice versa, that can be done using `toupper` and `tolower`:

```
my_string <- "SOMETIMES I SCREAM (and sometimes I whisper)."
toupper(my_string)
tolower(my_string)
```

If you only wish to change the case of some particular element in your string, you can use `substr`, which allows you to access substrings:

```
months <- c("january", "february", "march", "aripl")

# Replacing characters 2-4 of months[4] with "pri":
substr(months[4], 2, 4) <- "pri"
months

# Replacing characters 1-1 (i.e. character 1) of each element of month
# with its uppercase version:
substr(months, 1, 1) <- toupper(substr(months, 1, 1))
months
```

### 5.5.3 Finding patterns using regular expressions

*Regular expressions* (regexps) are special strings that describe patterns. They are extremely useful if you need to find, replace, or otherwise manipulate a number of strings depending on whether or not a certain pattern exists in each one of them. For instance, you may want to find all strings containing only numbers and convert them to `numeric`, or find all strings that contain an email address and remove said addresses (for censoring purposes, say). Regular expressions are incredibly useful but can be daunting. Not everyone will need them, and if this all seems a bit too much, you can safely skip this section or just skim through it and return to it at a later point.

To illustrate the use of regular expressions, we will use a sheet from the `projects-email.xlsx` file from the book's web page. In Exercise 2.32, you explored the second sheet in this file, but here we'll use the third instead. Set `file_path` to the path to the file, and then run the following code to import the data:

```
library(openxlsx)
contacts <- read.xlsx(file_path, sheet = 3)
str(contacts)
```

There are now three variables in `contacts`. We'll primarily be concerned with the third one: `Address`. Some people have email addresses attached to them, others have postal addresses, and some have no address at all:

```
contacts$Address
```

You can find loads of guides on regular expressions online, but few of them are easy to use with R, the reason being that regular expressions in R sometimes require escape characters that aren't needed in some other programming languages. In this section we'll take a look at regular expressions, *as they are written in R*.

The basic building blocks of regular expressions are patterns consisting of one or more characters. If, for instance, we wish to find all occurrences of the letter `y` in a vector of strings, the regular expression describing that "pattern" is simply `"y"`. The functions used to find occurrences of patterns are called `grep` and `grepl`. They differ only in the output they return: `grep` returns the indices of the strings containing the pattern, and `grepl` returns a `logical` vector with `TRUE` at indices matching the patterns and `FALSE` at other indices.

To find all addresses containing a lowercase `y`, we use `grep` and `grepl` as follows:

```
grep("y", contacts$Address)
grepl("y", contacts$Address)
```

Note how both outputs contain the same information presented in different ways.

In the same way, we can look for word or substrings. For instance, we can find all addresses containing the string `"Edin"`:

```
grep("Edin", contacts$Address)
grepl("Edin", contacts$Address)
```

Similarly, we can also look for special characters. Perhaps we can find all email addresses by looking for strings containing the `@` symbol:

```
grep("@", contacts$Address)
grepl("@", contacts$Address)

# To display the addresses matching the pattern:
contacts$Address[grep("@", contacts$Address)]
```

Interestingly, this includes two rows that aren't email addresses. To separate the email addresses from the other rows, we'll need a more complicated regular expression, describing the pattern of an email address in more general terms. Here are four examples or regular expressions that'll do the trick:

```
grep(".+@.+[.].+", contacts$Address)
grep(".+@.+\\..+", contacts$Address)
grep("[[:graph:]]+@[[:graph:]]+[.][[:alpha:]]+", contacts$Address)
grep("[[:alnum:]._-]+@[[:alnum:]._-]+[.][[:alpha:]]+",
     contacts$Address)
```

To try to wrap our head around what these mean, we'll have a look at the building blocks of regular expressions. These are:

- Patterns describing a single character.
- Patterns describing a class of characters, e.g., letters or numbers.
- Repetition quantifiers describing how many repetitions of a pattern to look for.
- Other operators.

We've already looked at single character expressions, as well as the multicharacter expression `"Edin"` which simply is a combination of four single-character expressions. Patterns describing classes of characters, e.g., characters with certain properties, are denoted by brackets `[]` (for manually defined classes) or double brackets `[[]]` (for pre-defined classes). One example of the latter is `"[[:digit:]]`, which is a pattern that matches all digits: `0 1 2 3 4 5 6 7 8 9`. Let's use it to find all addresses containing a number:

```
grep("[[:digit:]]", contacts$Address)
contacts$Address[grep("[[:digit:]]", contacts$Address)]
```

Some important pre-defined classes are:

- `[[:lower:]]` matches lowercase letters,

- `[[:upper:]]` matches UPPERCASE letters,
- `[[:alpha:]]` matches both lowercase and UPPERCASE letters,
- `[[:digit:]]` matches digits: `0 1 2 3 4 5 6 7 8 9`,
- `[[:alnum:]]` matches alphanumeric characters (alphabetic characters and digits),
- `[[:punct:]]` matches punctuation characters: `! " # $ % & ' ( ) * + , - . / : ; < = > ? @ [ \ ] ^ _ {|}~``,
- `[[:space:]]` matches space characters: space, tab, newline, and so on,
- `[[:graph:]]` matches letters, digits, and punctuation characters,
- `[[:print:]]` matches letters, digits, punctuation characters, and space characters,
- `.` matches *any* character.

Examples of manually defined classes are:

- `[abcd]` matches `a`, `b`, `c`, and `d`,
- `[a-d]` matches `a`, `b`, `c`, and `d`,
- `[aA12]` matches `a`, `A`, `1` and `2`,
- `[.]` matches `.`,
- `[.,]` matches `.` and `,`,
- `[^abcd]` matches anything except `a`, `b`, `c`, or `d`.

So, for instance, we can find all addresses that don't contain at least one of the letters `y` and `z` using:

```
grep("[^yz]", contacts$Address)
contacts$Address[grep("[^yz]", contacts$Address)]
```

All of these patterns can be combined with patterns describing a single character:

- `gr[ea]y` matches `grey` and `gray` (but not `greay`!),
- `b[^o]g` matches `bag`, `beg`, and similar strings, but not `bog`,
- `[.]com` matches `.com`.

When using the patterns above, you only look for a single occurrence of the pattern. Sometimes you may want a pattern like *a word of 2-4 letters* or *any number of digits in a row*. To create these, you add repetition patterns to your regular expression:

- `?` means that the preceding pattern is matched *at most once*, i.e., 0 or 1 time,
- `*` means that the preceding pattern is matched *0 or more* times,
- `+` means that the preceding pattern is matched *at least once*, i.e., 1 time or more,
- `{n}` means that the preceding pattern is matched *exactly* `n` times,

- `{n,}` means that the preceding pattern is matched *at least* `n` times, i.e., `n` times or more,
- `{n,m}` means that the preceding pattern is matched *at least* `n` times *but not more than* `m` times.

Here are some examples of how repetition patterns can be used:

```
# There are multiple ways of finding strings containing two n's
# in a row:
contacts$Address[grep("nn", contacts$Address)]
contacts$Address[grep("n{2}", contacts$Address)]

# Find strings with words beginning with an uppercase letter, followed
# by at least one lowercase letter:
contacts$Address[grep("[[:upper:]][[:lower:]]+", contacts$Address)]

# Find strings with words beginning with an uppercase letter, followed
# by at least six lowercase letters:
contacts$Address[grep("[[:upper:]][[:lower:]]{6,}", contacts$Address)]

# Find strings containing any number of letters, followed by any
# number of digits, followed by a space:
contacts$Address[grep("[[:alpha:]]+[[:digit:]]+[[:space:]]",
                      contacts$Address)]
```

Finally, there are some other operators that you can use to create even more complex patterns:

- `|` alteration, picks one of multiple possible patterns. For example, `ab|bc` matches `ab` or `bc` .
- `()` parentheses are used to denote a subset of an expression that should be evaluated separately. For example, `colo|our` matches `colo` or `our` while `col(o|ou)r` matches `color` or `colour` .
- `^` , when used outside of brackets `[]` , means that the match should be found at the start of the string. For example, `^a` matches strings beginning with `a` , but not `"dad"` .
- `$` means that the match should be found at the end of the string. For example, `a$` matches strings ending with `a` , but not `"dad"` .
- `\\` escape character that can be used to match special characters like `.` , `^` and `$` ( `\\.` , `\\^` , `\\$` ).

This may seem like a lot (and it is!), but there are in fact many more possibilities when working with regular expression. For the sake of some sort of brevity, we'll leave it at this for now though.

Let's return to those email addresses. We saw three regular expressions that could be used to find them:

```
grep(".+@.+[.].+", contacts$Address)
grep(".+@.+\\..+", contacts$Address)
grep("[[:graph:]]+@[[:graph:]]+[.][[:alpha:]]+", contacts$Address)
grep("[[:alnum:]._-]+@[[:alnum:]._-]+[.][[:alpha:]]+",
     contacts$Address)
```

The first two of these specify the same pattern: *any number of any characters, followed by an* `@` *, followed by any number of any characters, followed by a period* `.` *, followed by any number of characters*. This will match email addresses but would also match strings like `"?=)(/x@!.a??"` , which isn't a valid email address. In this case, that's not a big issue, as our goal was to find addresses that looked like email addresses and not to verify that the addresses were valid.

The third alternative has a slightly different pattern: *any number of letters, digits, and punctuation characters, followed by an* `@` *, followed by any number of letters, digits, and punctuation characters, followed by a period* `.` *, followed by any number of letters*. This too would match `"?=)(/x@!.a??"` as it allows punctuation characters that don't usually occur in email addresses. The fourth alternative, however, won't match `"?=)(/x@!.a??"` as it only allows letters, digits and the symbols `.` , `_` and `-` in the name and domain name of the address.

### 5.5.4 Substitution

An important use of regular expressions is in substitutions, where the parts of strings that match the pattern in the expression are replaced by another string. There are two email addresses in our data that contain `(a)` instead of `@` :

```
contacts$Address[grep("[(]a[])]", contacts$Address)]
```

If we wish to replace the `(a)` by `@`, we can do so using `sub` and `gsub`. The former replaces only the *first* occurrence of the pattern in the input vector, whereas the latter replaces *all* occurrences.

```
contacts$Address[grep("[(]a[])]", contacts$Address)]
sub("[(]a[])]", "@", contacts$Address)  # Replace first occurrence
gsub("[(]a[])]", "@", contacts$Address) # Replace all occurrences
```

### 5.5.5 Splitting strings

At times you want to extract only a part of a string, for example if measurements recorded in a column contain units, e.g. `66.8 kg` instead of `66.8`. To split a string into different parts, we can use `strsplit`.

As an example, consider the email addresses in our `contacts` data. Suppose that we want to extract the user names from all email addresses, i.e., remove the `@domain.topdomain` part. First, we store all email addresses from the data in a new vector, and then we split them at the `@` sign:

```
emails <- contacts$Address[grepl(
             "[[:alnum:]._-]+@[[:alnum:]._-]+[.][[:alpha:]]+",
             contacts$Address)]
emails_split <- strsplit(emails, "@")
emails_split
```

`emails_split` is a *list*. In this case, it seems convenient to convert the split strings into a matrix using `unlist` and `matrix` (you may want to have a quick look at Exercise 2.23 to re-familiarise yourself with `matrix`):

```
emails_split <- unlist(emails_split)

# Store in a matrix with length(emails_split)/2 rows and 2 columns:
emails_matrix <- matrix(emails_split,
                        nrow = length(emails_split)/2,
                        ncol = 2,
                        byrow = TRUE)

# Extract usernames:
emails_matrix[,1]
```

Similarly, when working with data stored in data frames, it is sometimes desirable to split a column containing strings into two columns. Some convenience functions for this are discussed in Section 5.11.3.

### 5.5.6 Variable names

Variable names can be very messy, particularly when they are imported from files. You can access and manipulate the variable names of a data frame using `names`:

```
names(contacts)
names(contacts)[1] <- "ID number"
grep("[aA]", names(contacts))
```

∼

**Exercise 5.7** Download the file `handkerchief.csv` from the book's web page. It contains a short list of prices of Italian handkerchiefs from the 1769 publication *A list of prices in those branches of the weaving manufactory, called the black branch, and the fancy branch*. Load the data in a data frame in R and then do the following:

1. Read the documentation for the function `nchar`. What does it do? Apply it to the `Italian.handkerchief` column of your data frame.
2. Use `grep` to find out how many rows of the `Italian.handkerchief` column that contain numbers.
3. Find a way to extract the prices in shillings (S) and pence (D) from the `Price` column, storing these in two new `numeric` variables in your data frame.

(Click here to go to the solution.)

**Exercise 5.8** Download the `oslo-biomarkers.xlsx` data from the book's web page. It contains data from a medical study about patients with disc herniations, performed at the Oslo University Hospital, Ullevål (this is a modified[36] version of the data analysed by Moen et al. (2016)). Blood samples were collected from a number of patients with disc herniations at three time points: 0 weeks (first visit at the hospital), 6 weeks, and 12 months. The levels of some biomarkers related to inflammation were measured in each blood sample. The first column in the spreadsheet contains information about the patient ID and the time point of sampling. Load the data and check its structure. Each patient is uniquely identified by their ID number. How many patients were included in the study?

(Click here to go to the solution.)

**Exercise 5.9** What patterns do the following regular expressions describe? Apply them to the `Address` vector of the `contacts` data to check that you interpreted them correctly.

1. `"$g"`
2. `"^[^[[:digit:]]"`
3. `"a(s|l)"`
4. `"[[:lower:]]+[.][[:lower:]]+"`

(Click here to go to the solution.)

**Exercise 5.10** Write code that, given a string, creates a vector containing all words from the string, with one word in each element and no punctuation marks. Apply it to the following string to check that it works:

```r
x <- "This is an example of a sentence, with 10 words. Here are 4 more!"
```

(Click here to go to the solution.)

## 5.6 Working with dates and times

Data describing dates and times can be complex, not least because they can be written is so many different formats. The date 1 April 2020 can be written as `2020-04-01`, `20/04/01`, `200401`, `1/4 2020`, `4/1/20`, `1 Apr 20`, and a myriad of other ways. The time 5 past 6 in the evening can be written as `18:05` or `6.05 pm`. In addition to this ambiguity, time zones, daylight saving time, leap years, and even leap seconds make working with dates and times even more complicated.

The default in R is to use the ISO8601 standards, meaning that dates are written as YYYY-MM-DD, and that times are written using the 24-hour hh:mm:ss format. In order to avoid confusion, you should always use these, unless you have *very* strong reasons not to.

Dates in R are represented as `Date` objects, and dates with times as `POSIXct` objects. The examples below are concerned with `Date` objects, but you will explore `POSIXct` too, in Exercise 5.12.

### 5.6.1 Date formats

The `as.Date` function tries to coerce a `character` string to a date. For some formats, it will automatically succeed, whereas for others, you have to provide the format of the date manually. To complicate things further, what formats work automatically will depend on your system settings. Consequently, the safest option is always to specify the format of your dates, to make sure that the code still will run if you at some point have to execute it on a different machine. To help describe date formats, R has a number of tokens to describe days, months, and years:

- `%d` - day of the month as a number (01-31).
- `%m` - month of the year as a number (01-12).
- `%y` - year without century (00-99).
- `%Y` - year with century (e.g., 2020).

Here are some examples of date formats, all describing 1 April 2020 – try them both with and without specifying the format to see what happens:

```r
as.Date("2020-04-01")
as.Date("2020-04-01", format = "%Y-%m-%d")
as.Date("4/1/20")
as.Date("4/1/20", format = "%m/%d/%y")

# Sometimes dates are expressed as the number of days since a
# certain date. For instance, 1 April 2020 is 43,920 days after
# 1 January 1900:
as.Date(43920, origin = "1900-01-01")
```

If the date includes month or weekday names, you can use tokens to describe that as well:

- `%b` - abbreviated month name, e.g., `Jan` , `Feb` .
- `%B` - full month name, e.g., `January` , `February` .
- `%a` - abbreviated weekday, e.g., `Mon` , `Tue` .
- `%A` - full weekday, e.g., `Monday` , `Tuesday` .

Things become a little more complicated now though, because R will interpret the names as if they were written in the language set in your *locale*, which contains a number of settings related to your language and region. To find out what language is in your locale, you can use:

```r
Sys.getlocale("LC_TIME")
```

I'm writing this on a machine with Swedish locale settings (my output from the above code chunk is `"sv_SE.UTF-8"` ). The Swedish word for *Wednesday* is *onsdag*[37], and therefore the following code doesn't work on my machine:

```r
as.Date("Wednesday 1 April 2020", format = "%A %d %B %Y")
```

However, if I translate it to Swedish, it runs just fine:

```r
as.Date("Onsdag 1 april 2020", format = "%A %d %B %Y")
```

You may at times need to make similar translations of dates. One option is to use `gsub` to translate the names of months and weekdays into the correct language (see Section 5.5.4). Alternatively, you can change the locale settings. On most systems, the following setting will allow you to read English months and days properly:

```
Sys.setlocale("LC_TIME", "C")
```

The locale settings will revert to the defaults the next time you start R.

Conversely, you may want to extract a substring from a `Date` object, for instance the day of the month. This can be done using `strftime`, using the same tokens as above. Here are some examples, including one with the token `%j`, which can be used to extract the day of the year:

```
dates <- as.Date(c("2020-04-01", "2021-01-29", "2021-02-22"),
                   format = "%Y-%m-%d")

# Extract the day of the month:
strftime(dates, format = "%d")

# Extract the month:
strftime(dates, format = "%m")

# Extract the year:
strftime(dates, format = "%Y")

# Extract the day of the year:
strftime(dates, format = "%j")
```

Should you need to, you can of course convert these objects from `character` to `numeric` using `as.numeric`.

For a complete list of tokens that can be used to describe date patterns, see `?strftime`.

~

**Exercise 5.11** Consider the following `Date` vector:

```
dates <- as.Date(c("2015-01-01", "1984-03-12", "2012-09-08"),
                 format = "%Y-%m-%d")
```

1. Apply the functions `weekdays`, `months`, and `quarters` to the vector. What do they do?
2. Use the `julian` function to find out how many days passed between 1970-01-01 and the dates in `dates`.

**Exercise 5.12** Consider the three `character` objects created below:

```
time1 <- "2020-04-01 13:20"
time2 <- "2020-04-01 14:30"
time3 <- "2020-04-03 18:58"
```

1. What happens if you convert the three variables to `Date` objects using `as.Date` without specifying the date format?
2. Convert `time1` to a `Date` object and add `1` to it. What is the result?
3. Convert `time3` and `time1` to `Date` objects and subtract them. What is the result?
4. Convert `time2` and `time1` to `Date` objects and subtract them. What is the result?
5. What happens if you convert the three variables to `POSIXct` date and time objects using `as.POSIXct` without specifying the date format?
6. Convert `time3` and `time1` to `POSIXct` objects and subtract them. What is the result?
7. Convert `time2` and `time1` to `POSIXct` objects and subtract them. What is the result?
8. Use the `difftime` to repeat the calculation in task 6, but with the result presented in hours.

(Click here to go to the solution.)

**Exercise 5.13** In some fields, e.g., economics, data is often aggregated on a quarter-year level, as in these examples:

```
qvec1 <- c("2020 Q4", "2021 Q1", "2021 Q2")
qvec2 <- c("Q4/20", "Q1/21", "Q2/21")
qvec3 <- c("Q4-2020", "Q1-2021", "Q2-2021")
```

To convert `qvec1` to a `Date` object, we can use `as.yearqtr` from the `zoo` package in two ways:

```r
library(zoo)
as.Date(as.yearqtr(qvec1, format = "%Y Q%q"))
as.Date(as.yearqtr(qvec1, format = "%Y Q%q"), frac = 1)
```

1. Describe the results. What is the difference? Which do you think is preferable?
2. Convert `qvec2` and `qvec3` to `Date` objects in the same way. Make sure that you get the `format` argument, which describes the date format, right.

(Click here to go to the solution.)

### 5.6.2 Plotting with dates

`ggplot2` automatically recognises `Date` objects and will usually plot them in a nice way. That only works if it actually has the dates though. Consider the following plot, which we created in Section 4.7.7 – it shows the daily electricity demand in Victoria, Australia in 2014:

```r
library(plotly)
library(fpp2)

# Create the plot object
myPlot <- autoplot(elecdaily[,"Demand"])

# Create the interactive plot
ggplotly(myPlot)
```

When you hover over the points, the formatting of the dates looks odd. We'd like to have proper dates instead. In order to do so, we'll use `seq.Date` to create a sequence of dates, ranging from 2014-01-01 to 2014-12-31:

```r
# Create a data frame with better formatted dates
elecdaily2 <- as.data.frame(elecdaily)
elecdaily2$Date <- seq.Date(as.Date("2014-01-01"),
                            as.Date("2014-12-31"),
                            by = "day")

# Create the plot object
myPlot <- ggplot(elecdaily2, aes(Date, Demand)) +
      geom_line()

# Create the interactive plot
ggplotly(myPlot)
```

`seq.Date` can be used analogously to create sequences where there is a week, month, quarter, or year between each element of the sequence, by changing the `by` argument.

∼

**Exercise 5.14** Return to the plot from Exercise 4.12, which was created using

```r
library(fpp2)
autoplot(elecdaily, facets = TRUE)
```

You'll notice that the x-axis shows week numbers rather than dates (the dates in the `elecdaily` time series object are formatted as weeks with decimal numbers). Make a time series plot of the `Demand` variable with dates (2014-01-01 to 2014-12-31) along the x-axis (your solution is likely to rely on standard R techniques rather than `autoplot`).

(Click here to go to the solution.)

**Exercise 5.15** Create an interactive version time series plot of the `a10` anti-diabetic drug sales data, as in Section 4.7.7. Make sure that the dates are correctly displayed.

(Click here to go to the solution.)

## 5.7 Data manipulation with `data.table` , `dplyr` ,

## and `tidyr`

In the remainder of this chapter, we will use three packages that contain functions for fast and efficient data manipulation: `data.table` and the tidyverse packages `dplyr` and `tidyr`. To begin with, it is therefore a good idea to install `data.table` and `tidyr`, which we haven't used before. And while you wait for the installation to finish, read on.

```
install.packages(c("tidyr", "data.table"))
```

There is almost always more than one way to solve a problem in R. We now know how to access vectors and elements in data frames, e.g., to compute means. We also know how to modify and add variables to data frames. Indeed, you can do just about anything using the functions in base R. Sometimes, however, those solutions become rather cumbersome, as they can require a fair amount of programming and verbose code. `data.table` and the tidyverse packages offer simpler solutions and speed up the workflow for these types of problems. Both can be used for the same tasks. You can learn one of them or both. The syntax used for `data.table` is often more concise and arguably more consistent than that in `dplyr` (it is in essence an extension of the `[i, j]` notation that we have already used for data frames). Second, it is fast and memory-efficient, which makes a huge difference if you are working with big data (you'll see this for yourself in Section 6.6). On the other hand, many people prefer the syntax in `dplyr` and `tidyr`, which lends itself exceptionally well for usage with pipes. If you work with small- or medium-sized datasets, the difference in performance between the two packages is negligible. `dplyr` is also much better suited for working directly with databases, which is a huge selling point if your data already is in a database[38].

In the sections below, we will see how to perform different operations using both `data.table` and the tidyverse packages. Perhaps you already know which one you want to use (`data.table` if performance is important to you, `dplyr` + `tidyr` if you like to use pipes or will be doing a lot of work with databases). If not, you can use these examples to guide your choice. Or not choose at all! I regularly use all three packages myself, to harness the strength of them all. There is no harm in knowing how to use both a hammer and a screwdriver.

### 5.7.1 `data.table` and tidyverse syntax basics

`data.table` relies heavily on the `[i, j]` notation that is used for data frames in R. It also adds a third element: `[i, j, by]`. Using this, R selects the rows indicated by `i`, the columns indicated by `j`, and groups them by `by`. This makes it easy, e.g., to compute grouped summaries.

With the tidyverse packages, you will instead use specialised functions with names like `filter` and `summarise` to perform operations on your data. These are typically combined using the pipe operator, `|>`, which makes the code flow nicely from left to right.

It's almost time to look at some examples of what this actually looks like in practice. First though, now that you've installed `data.table` and `dplyr`, it's time to load them (we'll get to `tidyr` a little later). We'll also create a `data.table` version of the `airquality` data, which we'll use in the examples below. This is required in order to use `data.table` syntax, as it only works on `data.table` objects. Luckily, `dplyr` works perfectly when used on `data.table` objects, so we can use the same object for the examples for both packages.

```
library(data.table)
library(dplyr)

aq <- as.data.table(airquality)
```

When importing data from csv files, you can import them as `data.table` objects instead of `data.frame` objects by replacing `read.csv` with `fread` from the `data.table` package. The latter function also has the benefit of being substantially faster when importing large (several megabytes) csv files.

Note that, similar to what we saw in Section 5.2.1, variables in imported data frames can have names that would not be allowed in base R, for instance including forbidden characters like `-`. `data.table` and `dplyr` allow you to work with such variables by wrapping their names in apostrophes: referring to the illegally named variable as `illegal-character-name` won't work, but `` `illegal-character-name` `` will.

## 5.7.2 Modifying a variable

As a first example, let's consider how to use `data.table` and `dplyr` to modify a variable in a data frame. The wind speed in `airquality` is measured in miles per hour (mph). We can convert that to metres per second (m/s) by multiplying the speed by 0.44704. Using only base R, we'd do this using `airquality$Wind <- airquality$Wind * 0.44704`. With `data.table` we can instead do this using `[i, j]` notation, and with `dplyr` we can do it by using a function called `mutate` (because it "mutates" your data).

Change wind speed to m/s instead of mph:

With `data.table`:

```
aq[, Wind := Wind * 0.44704]
```

With `dplyr`:

```
aq |> mutate(Wind =
          Wind * 0.44704) -> aq
```

Note that when using `data.table`, there is not an explicit assignment. We don't use `<-` to assign the new data frame to `aq` – instead, the assignment happens automatically. This means that you have to be a bit careful, so that you don't inadvertently make changes to your data when playing around with it.

In this case, using `data.table` or `dplyr` doesn't make anything easier. Where these packages really shine is when we attempt more complicated operations. Before that though, let's look at a few more simple examples.

## 5.7.3 Computing a new variable based on existing variables

What if we wish to create new variables based on other variables in the data frame? For instance, maybe we want to create a *dummy variable* called `Hot`, containing a `logical` that describes whether a day was hot (temperature above 90 - `TRUE`) or not (`FALSE`). That is, we wish to check the condition `Temp > 90` for each row and put the resulting `logical` in the new variable `Hot`.

Add a dummy variable describing whether it is hotter than 90:

With `data.table`:

```
aq[, Hot := Temp > 90]
```

With `dplyr`:

```
aq |> mutate(Hot = Temp > 90) -> aq
```

## 5.7.4 Renaming a variable

To change the name of a variable, we can use `setnames` from `data.table` or `rename` from `dplyr`. Let's change the name of the variable `Hot` that we created in the previous section, to `HotDay`:

With `data.table`:

With `dplyr`:

```r
setnames(aq, "Hot", "HotDay")
```

5

.

```r
aq |> rename(HotDay = Hot) -> aq
```

## 7.5 Removing a variable

Maybe adding `Hot` to the data frame wasn't such a great idea after all. How can we remove it?

Removing `Hot` :

With `data.table` :

```r
aq[, Hot := NULL]
```

With `dplyr` :

```r
aq |> select(-Hot) -> aq
```

If we wish to remove multiple columns at once, the syntax is similar:

Removing multiple columns:

With `data.table` :

```r
aq[, c("Month", "Day") := NULL]
```

With `dplyr` :

```r
aq |> select(-Month, -Day) -> aq
```

∼

**Exercise 5.16** Load the VAS pain data `vas.csv` from Exercise 2.30. Then do the following:

1. Remove the columns `X` and `X.1` .
2. Add a dummy variable called `highVAS` that indicates whether a patient's `VAS` is 7 or greater on any given day.

(Click here to go to the solution.)

## 5.7.6 Recoding `factor` levels

Changing the names of `factor` levels in base R typically relies on using indices of level names, as in Section 5.4.2. This can be avoided using `data.table` or the `recode` function in `dplyr` . We return to the `smoke` example from Section 5.4 and put it in a `data.table` :

```
library(data.table)
library(dplyr)

smoke <- c("Never", "Never", "Heavy", "Never", "Occasionally",
           "Never", "Never", "Regularly", "Regularly", "No")

smoke2 <- factor(smoke, levels = c("Never", "Occasionally",
                                   "Regularly", "Heavy"),
                        ordered = TRUE)

smoke3 <- data.table(smoke2)
```

Suppose that we want to change the levels' names to abbreviated versions: *Nvr*, *Occ*, *Reg*, and *Hvy*. Here's how to do this:

With `data.table`:

```
new_names = c("Nvr", "Occ", "Reg", "Hvy")
smoke3[.(smoke2 = levels(smoke2), to = new_names),
         on = "smoke2",
         smoke2 := i.to]
smoke3[, smoke2 := droplevels(smoke2)]
```

With `dplyr`:

```
# Version 1:
smoke3 |> mutate(smoke2 = case_match(smoke2,
                  "Never" ~ "Nvr",
                  "Occasionally" ~ "Occ",
                  "Regularly" ~ "Reg",
                  "Heavy" ~ "Hvy")) -> smoke3

# Version 2:
smoke3 |> mutate(smoke2 = recode(smoke2,
                  "Never" = "Nvr",
                  "Occasionally" = "Occ",
                  "Regularly" = "Reg",
                  "Heavy" = "Hvy")) -> smoke3
```

Next, we can combine the *Occ*, *Reg*, and *Hvy* levels into a single level called *Yes*:

With `data.table`:

```
smoke3[.(smoke2 = c("Occ", "Reg", "Hvy"), to = "Yes"),
       on = "smoke2",
       smoke2 := i.to]
```

With `dplyr`:

```
# Version 1:
smoke3 |> mutate(smoke2 = case_match(smoke2,
                  "Nvr" ~ "Nvr",
                  c("Occ", "Reg", "Hvy") ~ "Yes")) -> smoke3

# Version 2:
smoke3 |> mutate(smoke2 = recode(smoke2,
                  "Occ" = "Yes",
                  "Reg" = "Yes",
                  "Hvy" = "Yes")) -> smoke3
```

∼

**Exercise 5.17** In Exercise 2.27 you learned how to create a `factor` variable from a `numeric` variable using `cut`. Return to your solution (or the solution at the back of the book) and do the following using `data.table` and/or `dplyr`:

1. Change the category names to `Mild`, `Moderate`, and `Hot`.
2. Combine `Moderate` and `Hot` into a single level named `Hot`.

(Click here to go to the solution.)

## 5.7.7 Grouped summaries

We've already seen how we can use `aggregate` and `by` to create grouped summaries. However, in many cases it is as easy or easier to use `data.table` or `dplyr` for such summaries.

To begin with, let's load the packages again (in case you don't already have them loaded), and let's recreate the `aq` `data.table`, which we made a bit of a mess of by removing some important columns in the previous section:

```
library(data.table)
library(dplyr)

aq <- data.table(airquality)
```

Now, let's compute the mean temperature for each month. Both `data.table` and `dplyr` will return a data frame with the results. In the `data.table` approach, assigning a name to the summary statistic (`mean`, in this case) is optional, but not in `dplyr`.

With `data.table`:

```
aq[, mean(Temp), Month]
# or, to assign a name:
aq[, .(meanTemp = mean(Temp)),
   Month]
```

With `dplyr`:

```
aq |> group_by(Month) |>
       summarise(meanTemp =
                     mean(Temp))
```

You'll recall that if we apply `mean` to a vector containing `NA` values, it will return `NA`:

With `data.table`:

With `dplyr`:

```
aq[, mean(Ozone), Month]
```

```
aq |> group_by(Month) |>
      summarise(meanTemp =
                  mean(Ozone))
```

In order to avoid this, we can pass the argument `na.rm = TRUE` to `mean`, just as we would in other contexts. To compute the mean ozone concentration for each month, ignoring `NA` values:

With `data.table`:

```
aq[, mean(Ozone, na.rm = TRUE),
   Month]
```

With `dplyr`:

```
aq |> group_by(Month) |>
      summarise(meanTemp =
                  mean(Ozone,
                   na.rm = TRUE))
```

What if we want to compute a grouped summary statistic involving two variables? For instance, the correlation between temperature and wind speed for each month?

With `data.table`:

```
aq[, cor(Temp, Wind), Month]
```

With `dplyr`:

```
aq |> group_by(Month) |>
      summarise(cor =
                 cor(Temp, Wind))
```

The syntax for computing multiple grouped statistics is similar. We compute both the mean temperature and the correlation for each month:

With `data.table`:

```
aq[, .(meanTemp = mean(Temp),
   cor = cor(Temp, Wind)),
   Month]
```

With `dplyr`:

```
aq |> group_by(Month) |>
      summarise(meanTemp =
                 mean(Temp),
                cor =
                 cor(Temp, Wind))
```

At times, you'll want to compute summaries for all variables that share some property. As an example, you may want to compute the mean of all `numeric` variables in your data frame. In `dplyr` there is a convenience function called `across` that can be used for this: `summarise(across(where(is.numeric), mean))` will compute the mean of all numeric variables. In `data.table`, we can instead utilise the `apply` family of functions from base R, that we'll study in Section 6.5. To compute the mean of all `numeric` variables:

With `data.table`:

```
aq[, lapply(.SD, mean),
   Month,
   .SDcols = is.numeric]
```

With `dplyr`:

```
aq |> group_by(Month) |>
       summarise(across(
         where(is.numeric),
         mean, na.rm = TRUE))
```

Both packages have special functions for counting the number of observations in groups: `.N` for `data.table` and `n` for `dplyr`. For instance, we can count the number of days in each month:

With `data.table`:

```
aq[, .N, Month]
```

With `dplyr`:

```
aq |> group_by(Month) |>
      summarise(days = n())
```

Similarly, you can count the number of unique values of variables using `uniqueN` for `data.table` and `n_distinct` for `dplyr`:

With `data.table`:

```
aq[, uniqueN(Month)]
```

With `dplyr`:

```
aq |> summarise(months =
                  n_distinct(Month))
```

∼

**Exercise 5.18** Load the VAS pain data `vas.csv` from Exercise 2.30. Then do the following using `data.table` and/or `dplyr`:

1. Compute the mean VAS for each patient.

2. Compute the lowest and highest VAS recorded for each patient.

3. Compute the number of high-VAS days, defined as days with where the VAS was at least 7, for each patient.

(Click here to go to the solution.)

**Exercise 5.19** We return to the `datasauRus` package and the `datasaurus_dozen` dataset from Exercise 2.28. Check its structure and then do the following using `data.table` and/or `dplyr`:

1. Compute the mean of `x`, mean of `y`, standard deviation of `x`, standard deviation of `y`, and correlation between `x` and `y`, grouped by `dataset`. Are there any differences between the 12 datasets?
2. Make a scatterplot of `x` against `y` for each dataset. Are there any differences between the 12 datasets?

(Click here to go to the solution.)

### 5.7.8 Filling in missing values

In some cases, you may want to fill in missing values of a variable with the previous non-missing entry. To see an example of this, let's create a version of `aq` where the value of `Month` are missing for some days:

```
aq$Month[c(2:3, 36:39, 70)] <- NA

# Some values of Month are now missing:
head(aq)
```

To fill in the missing values with the last non-missing entry, we can now use `nafill` or `fill` as follows:

With `data.table`:

```
aq[, Month := nafill(
      Month, "locf")]
```

With `tidyr`:

```
aq |> fill(Month) -> aq
```

To instead fill in the missing values with the *next* non-missing entry:

With `data.table`:

With `tidyr`:

```
aq[, Month := nafill(
      Month, "nocb")]
```

~

```
aq |> fill(Month,
        .direction = "up") -> aq
```

E
x
e
**rcise 5.20** Load the VAS pain data `vas.csv` from Exercise 2.30. Fill in the missing values in the `Visit` column with the last non-missing value.

(Click here to go to the solution.)

### 5.7.9 Chaining commands together

When working with tidyverse packages, commands are usually chained together using `|>` pipes. When using `data.table`, commands are chained by repeated use of `[]` brackets on the same line. This is probably best illustrated using an example. Assume again that there are missing values in `Month` in `aq`:

```
aq$Month[c(2:3, 36:39, 70)] <- NA
```

To fill in the missing values with the last non-missing entry (Section 5.7.8) and then count the number of days in each month (Section 5.7.7), we can do as follows:

With `data.table`:

```
aq[, Month := nafill(Month, "locf")][, .N, Month]
```

With `tidyr` and `dplyr`:

```
aq |> fill(Month) |>
       group_by(Month) |>
       summarise(days = n())
```

## 5.8 Filtering: select rows

You'll frequently want to filter away some rows from your data. Perhaps you only want to select rows where a variable exceeds some value, or you want to exclude rows with `NA` values. This can be done in several different ways: using row numbers, using conditions, at random, or using regular expressions. Let's have a look at them, one by one. We'll use `aq`, the `data.table` version of `airquality` that we created before, for the examples.

```
library(data.table)
library(dplyr)

aq <- data.table(airquality)
```

### 5.8.1 Filtering using row numbers

If you know the row numbers of the rows that you wish to remove (perhaps you've found them using `which`, as in Section 2.11.3?), you can use those numbers for filtering. Here are four examples.

To select row 3:

With `data.table`:

```
aq[3,]
```

With `dplyr`:

```
aq |> slice(3)
```

To select rows 3 to 5:

With `data.table`:

```
aq[3:5,]
```

With `dplyr`:

```
aq |> slice(3:5)
```

To select rows 3, 7, and 15:

With `data.table`:

```
aq[c(3, 7, 15),]
```

With `dplyr`:

```
aq |> slice(c(3, 7, 15))
```

To select all rows except rows 3, 7, and 15:

With `data.table`:

```
aq[-c(3, 7, 15),]
```

With `dplyr`:

```
aq |> slice(-c(3, 7, 15))
```

### 5.8.2 Filtering using conditions

Filtering is often done using conditions, e.g., to select observations with certain properties. Here are some examples:

To select rows where `Temp` is greater than 90:

With `data.table`:

```
aq[Temp > 90,]
```

With `dplyr`:

```
aq |> filter(Temp > 90)
```

To select rows where `Month` is 6 (June):

With `data.table`:

```
aq[Month == 6,]
```

With `dplyr`:

```
aq |> filter(Month == 6)
```

To select rows where `Temp` is greater than 90 and `Month` is 6 (June):

With `data.table`:

```
aq[Temp > 90 & Month == 6,]
```

With `dplyr`:

```
aq |> filter(Temp > 90,
             Month == 6)
```

To select rows where `Temp` is between 80 and 90 (including 80 and 90):

With `data.table`:

```
aq[Temp %between% c(80, 90),]
```

With `dplyr`:

```
aq |> filter(between(Temp,
                     80, 90))
```

To select the five rows with the highest `Temp`:

With `data.table`:

With `dplyr`:

```
aq |> top_n(5, Temp)
```

```
aq[frankv(-Temp,
          ties.method = "min") <= 5,
   ]
```

In this case, the above code returns more than 5 rows because of ties.

To remove duplicate rows:

With `data.table`:

```
unique(aq)
```

With `dplyr`:

```
aq |> distinct()
```

To remove rows with missing data ( `NA` values) in at least one variable:

With `data.table` :

```r
na.omit(aq)
```

With `tidyr` :

```r
library(tidyr)
aq |> drop_na()
```

To remove rows with missing `Ozone` values:

With `data.table` :

```r
na.omit(aq, "Ozone")
```

With `tidyr` :

```r
library(tidyr)
aq |> drop_na("Ozone")
```

At times, you want to filter your data based on whether the observations are connected to observations in a different dataset. Such filters are known as semijoins and antijoins. They are discussed in Section 5.12.4.

## 5.8.3 Selecting rows at random

In some situations, for instance when training and evaluating machine learning models, you may wish to draw a random sample from your data. This is done using the `sample` ( `data.table` ) and `sample_n` ( `dplyr` ) functions.

To select five rows at random:

With `data.table` :

```r
aq[sample(.N, 5),]
```

With `dplyr` :

```r
aq |> sample_n(5)
```

If you run the code multiple times, you will get different results each time. See Section 7.1 for more on random sampling and how it can be used.

### 5.8.4 Using regular expressions to select rows

In some cases, particularly when working with text data, you'll want to filter using regular expressions (see Section 5.5.3). `data.table` has a convenience function called `%like%` that can be used to call `grepl` in an alternative (less opaque?) way. With `dplyr` we use `grepl` in the usual fashion. To have some text data to try this out on, we'll use this data frame, which contains descriptions of some dogs:

```
dogs <- data.table(Name = c("Bianca", "Bella", "Mimmi", "Daisy",
                             "Ernst", "Smulan"),
                   Breed = c("Greyhound", "Greyhound", "Pug", "Poodle",
                            "Bedlington Terrier", "Boxer"),
                   Desc = c("Fast, playful", "Fast, easily worried",
                           "Intense, small, loud",
                           "Majestic, protective, playful",
                           "Playful, relaxed",
                           "Loving, cuddly, playful"))
View(dogs)
```

To select all rows with names beginning with B:

With `data.table`:

```
dogs[Name %like% "^B",]
# or:
dogs[grepl("^B", Name),]
```

With `dplyr`:

```
dogs |> filter(grepl("B[a-z]",
                     Name))
```

To select all rows where `Desc` includes the word `playful`:

With `data.table`:

```
dogs[Desc %like% "[pP]layful",]
```

With `dplyr`:

```
dogs |> filter(grepl("[pP]layful",
                     Desc))
```

~

**Exercise 5.21** Download the `ucdp-onesided-191.csv` data file from the book's web page. It contains data about international attacks on civilians by governments and formally organised armed groups during the period 1989-2018, collected as part of the Uppsala Conflict Data Program (Eck & Hultman, 2007; Petterson et al., 2019). Among other things, it contains information about the actor (attacker), fatality rate, and attack location. Load the data and check its structure.

1. Filter the rows so that only conflicts that took place in Colombia are retained. How many different actors were responsible for attacks in Colombia during the period?
2. Using the `best_fatality_estimate` column to estimate fatalities, calculate the number of worldwide fatalities caused by government attacks on civilians during 1989-2018.

(Click here to go to the solution.)

**Exercise 5.22** Load the `oslo-biomarkers.xlsx` data from Exercise 5.8. Use `data.table` and/or `dplyr` to do the following:

1. Select only the measurements from blood samples taken at 12 months.
2. Select only the measurements from the patient with ID number 6.

(Click here to go to the solution.)

## 5.9 Subsetting: select columns

Another common situation is that you want to remove some variables from your data. Perhaps the variables aren't of interest in a particular analysis that you're going to perform, or perhaps you've simply imported more variables than you need. As with rows, this can be done using numbers, names, or regular expressions. Let's look at some examples using the `aq` data:

```r
library(data.table)
library(dplyr)

aq <- data.table(airquality)
```

### 5.9.1 Selecting a single column

When selecting a single column from a data frame, you sometimes want to extract the column as a vector and sometimes as a single-column data frame (for instance, if you are going to pass it to a function that takes a data frame as input). You should be a bit careful when doing this, to make sure that you get the column in the correct format:

With `data.table`:

```
# Return a vector:
aq$Temp
# or
aq[, Temp]

# Return a data.table:
aq[, "Temp"]
```

With `dplyr`:

```
# Return a vector:
aq$Temp
# or
aq |> pull(Temp)

# Return a tibble:
aq |> select(Temp)
```

### 5.9.2 Selecting multiple columns

Selecting multiple columns is more straightforward, as the object that is returned always will be a data frame. Here are some examples.

To select `Temp`, `Month`, and `Day`:

With `data.table`:

```
aq[, .(Temp, Month, Day)]
```

With `dplyr`:

```
aq |> select(Temp, Month, Day)
```

To select all columns between `Wind` and `Month`:

With `data.table`:

```
aq[, Wind:Month]
```

With `dplyr`:

```
aq |> select(Wind:Month)
```

To select all columns except `Month` and `Day`:

With `data.table`:

With `dplyr`:

```
aq[, -c("Month", "Day")]
```

```
aq |> select(-Month, -Day)
```

T
o
s

elect all numeric variables (which for the `aq` data is all variables!):

With `data.table`:

```
aq[, .SD, .SDcols = is.numeric]
```

With `dplyr`:

```
aq |> select_if(is.numeric)
```

To remove columns with missing ( `NA` ) values:

With `data.table`:

```
aq[, .SD,
   .SDcols = !anyNA]
```

With `dplyr`:

```
aq |> select_if(~all(!is.na(.)))
```

### 5.9.3 Using regular expressions to select columns

In `data.table`, using regular expressions to select columns is done using `grep`. `dplyr` differs in that it has several convenience functions for selecting columns, like `starts_with`, `ends_with`, `contains`. As an example, we can select variables the name of which contains the letter `n`:

With `data.table`:

```
vars <- grepl("n", names(aq))
aq[, ..vars]
```

With `dplyr`:

```
# contains is a convenience function for checking if a name contains a string:
aq |> select(contains("n"))
# matches can be used with any regular expression:
aq |> select(matches("n"))
```

### 5.9.4 Subsetting using column numbers

It is also possible to subset using column numbers, but you need to be careful if you want to use that approach. Column numbers can change, for instance if a variable is removed from the data frame. More importantly, however, using column numbers can yield different results depending on what type of data table you're using. Let's have a look at what happens if we use this approach with different types of data tables:

```
# data.frame:
aq <- as.data.frame(airquality)
str(aq[,2])

# data.table:
aq <- as.data.table(airquality)
str(aq[,2])

# tibble:
aq <- as_tibble(airquality)
str(aq[,2])
```

As you can see, `aq[, 2]` returns a vector, a data table or a tibble, depending on what type of object `aq` is. Unfortunately, this approach is used by several R packages and can cause problems because it may return the wrong type of object.

A better approach is to use `aq[[2]]`, which works the same for data frames, data tables, and tibbles, returning a vector:

```
# data.frame:
aq <- as.data.frame(airquality)
str(aq[[2]])

# data.table:
aq <- as.data.table(airquality)
str(aq[[2]])

# tibble:
aq <- as_tibble(airquality)
str(aq[[2]])
```

**Exercise 5.23** Return to the `ucdp-onesided-191.csv` data from Exercise 5.21. To have a cleaner and less bloated dataset to work with, it can make sense to remove some columns. Select only the `actor_name` , `year` , `best_fatality_estimate` and `location` columns.

(Click here to go to the solution.)

## 5.10 Sorting

Sometimes you don't want to filter rows but rearrange their order according to their values for some variable. Similarly, you may want to change the order of the columns in your data. I often do this after merging data from different tables (as we'll do in Section 5.12). This is often useful for presentation purposes but can at times also aid in analyses.

### 5.10.1 Changing the column order

It is straightforward to change column positions using `setcolorder` in `data.table` and `relocate` in `dplyr` .

To put `Month` and `Day` in the first two columns, without rearranging the other columns:

With `data.table` :

```r
setcolorder(aq, c("Month", "Day"))
```

With `dplyr` :

```r
aq |> relocate("Month", "Day")
```

### 5.10.2 Changing the row order

In `data.table` , `order` is used for sorting rows, and in `dplyr` , `arrange` is used (sometimes in combination with `desc` ). The syntax differs depending on whether you wish to sort your rows in ascending or descending order. We will illustrate this using the `airquality` data.

```r
library(data.table)
library(dplyr)

aq <- data.table(airquality)
```

First of all, if you're just looking to sort a single vector, rather than an entire data frame, the quickest way to do so is to use `sort`:

```
sort(aq$Wind)
sort(aq$Wind, decreasing = TRUE)
sort(c("C", "B", "A", "D"))
```

If you're looking to sort an entire data frame by one or more variables, you need to move beyond `sort`. To sort rows by `Wind` (*ascending* order):

With `data.table`:

```
aq[order(Wind),]
```

With `dplyr`:

```
aq |> arrange(Wind)
```

To sort rows by `Wind` (*descending* order):

With `data.table`:

```
aq[order(-Wind),]
```

With `dplyr`:

```
aq |> arrange(-Wind)
# or
aq |> arrange(desc(Wind))
```

To sort rows, first by `Temp` (ascending order) and then by `Wind` (descending order):

With `data.table`:

```
aq[order(Temp, -Wind),]
```

With `dplyr`:

```
aq |> arrange(Temp, desc(Wind))
```

∼

**Exercise 5.24** Load the `oslo-biomarkers.xlsx` data from Exercise 5.8. Note that it is not ordered in a natural way. Reorder it by patient ID instead.

(Click here to go to the solution.)

## 5.11 Reshaping data

The `gapminder` dataset from the `gapminder` package contains information about life expectancy, population size and GDP per capita for 142 countries for 12 years from 1952 to 2007. To begin with, let's have a look at the data[39]:

```
library(gapminder)
?gapminder
View(gapminder)
```

Each row contains data for one country and one year, meaning that the data for each country is spread over 12 rows. This is known as *long data* or *long format*. As another option, we could store it in *wide format*, where the data is formatted so that all observations corresponding to a country are stored on the same row:

```
Country        Continent   lifeExp1952 lifeExp1957 lifeExp1962 ...
Afghanistan    Asia         28.8        30.2        32.0        ...
Albania        Europe       55.2        59.3        64.8        ...
```

Sometimes, it makes sense to spread an observation over multiple rows (long format), and sometimes it makes more sense to spread a variable across multiple columns (wide format). Some analyses require long data, whereas others require wide data. And if you're unlucky, data will arrive in the wrong format for the analysis you need to do. In this section, you'll learn how to transform your data from long to wide, and back again.

### 5.11.1 From long to wide

When going from a long format to a wide format, you choose columns to group the observations by (in the `gapminder` case: `country` and maybe also `continent`), columns to take value names from ( `lifeExp` , `pop` , and `gdpPercap` ), and columns to create variable names from ( `year` ).

In `data.table` , the transformation from long to wide is done using the `dcast` function. `dplyr` does not contain functions for such transformations, but its sibling, the tidyverse package `tidyr` , does.

The `tidyr` function used for long-to-wide formatting is `pivot_wider` . First, we convert the `gapminder` data frame to a `data.table` object:

```
library(data.table)
library(tidyr)

gm <- as.data.table(gapminder)
```

To transform the `gm` data from long to wide and store it as `gmw`:

With `data.table`:

```
gmw <- dcast(gm, country + continent ~ year,
             value.var = c("pop", "lifeExp", "gdpPercap"))
```

With `tidyr`:

```
gm |> pivot_wider(id_cols = c(country, continent),
                  names_from = year,
                  values_from =
                   c(pop, lifeExp, gdpPercap)) -> gmw
```

### 5.11.2 From wide to long

We've now seen how to transform the long format `gapminder` data to the wide format `gmw` data. But what if we want to go from wide format to long? Let's see if we can transform `gmw` back to the long format.

In `data.table`, wide-to-long formatting is done using `melt`, and in `dplyr` it is done using `pivot_longer`.

To transform the `gmw` data from long to wide:

With `data.table`:

```
gm <- melt(gmw, id.vars = c("country", "continent"),
           measure.vars = 2:37)
```

With `tidyr`:

```
gmw |> pivot_longer(names(gmw)[2:37],
                    names_to = "variable",
                    values_to = "value") -> gm
```

The resulting data frames are perhaps *too* long, with each variable ( `pop` , `lifeExp` , and `gdpPercapita` ) being put on a different row. To make it look like the original dataset, we must first split the `variable` variable (into a column with variable names and column with years) and then make the data frame a little wider again. That is the topic of the next section.

### 5.11.3 Splitting columns

In the too long `gm` data that you created at the end of the last section, the observations in the `variable` column look like `pop_1952` and `gdpPercap_2007` , i.e., are of the form `variableName_year` . We'd like to split them into two columns: one with variable names and one with years. `dplyr` has a function called `tstrsplit` for this purpose, and `tidyr` has `separate` .

To split the `variable` column at the underscore `_` , and then reformat `gm` to look like the original `gapminder` data:

With `data.table` :

```
gm[, c("variable", "year") := tstrsplit(variable,
                                 "_", fixed = TRUE)]
gm <- dcast(gm, country + year ~  variable,
            value.var = c("value"))
```

With `tidyr` :

```
gm |> separate(variable,
               into = c("variable", "year"),
               sep = "_") |>
      pivot_wider(id_cols = c(country, continent, year),
                  names_from = variable,
                  values_from = value) -> gm
```

### 5.11.4 Merging columns

Similarly, you may at times want to merge two columns, for instance if one contains the day+month part of a date and the other contains the year. An example of such a situation can be found in the `airquality` dataset, where we may want to merge the `Day` and `Month` columns into a new `Date` column. Let's re-create the `aq` `data.table` object one last time:

```
library(data.table)
library(tidyr)

aq <- as.data.table(airquality)
```

If we wanted to create a `Date` column containing the year (1973), month and day for each observation, we could use `paste` and `as.Date` :

```
as.Date(paste(1973, aq$Month, aq$Day, sep = "-"))
```

The natural `data.table` approach is just this, whereas `tidyr` offers a function called `unite` to merge columns, which can be combined with `mutate` to paste the year to the date. To merge the `Month` and `Day` columns with a year and convert it to a `Date` object:

With `data.table` :

```
aq[, Date := as.Date(paste(1973,
                     aq$Month,
                     aq$Day,
                     sep = "-"))]
```

With `tidyr` and `dplyr` :

```
aq |> unite("Date", Month, Day,
            sep = "-") |>
      mutate(Date = as.Date(
             paste(1973,
                   Date,
                   sep = "-")))
```

~

**Exercise 5.25** Load the `oslo-biomarkers.xlsx` data from Exercise 5.8. Then do the following using `data.table` and/or `dplyr` / `tidyr` :

1. Split the `PatientID.timepoint` column in two parts: one with the patient ID and one with the timepoint.
2. Sort the table by patient ID, in numeric order.

3. Reformat the data from long to wide, keeping only the IL-8 and VEGF-A measurements.

Save the resulting data frame – you will need it again in Exercise 5.26!
(Click here to go to the solution.)

## 5.12 Merging data from multiple tables

It is common that data is spread over multiple tables: different sheets in Excel files, different `.csv` files, or different tables in databases. Consequently, it is important to be able to merge data from different tables.

As a first example, let's study the sales datasets available from the book's web page: `sales-rev.csv` and `sales-weather.csv`. The first dataset describes the daily revenue for a business in the first quarter of 2020, and the second describes the weather in the region (somewhere in Sweden) during the same period[40]. Store their respective paths as `file_path1` and `file_path2` and then load them:

```
rev_data <- read.csv(file_path1, sep = ";")
weather_data <- read.csv(file_path2, sep = ";")

str(rev_data)
View(rev_data)

str(weather_data)
View(weather_data)
```

### 5.12.1 Binds

The simplest types of merges are *binds*, which can be used when you have two tables where either the rows or the columns *match each other exactly*. To illustrate what this may look like, we will use `data.table` / `dplyr` to create subsets of the business revenue data. First, we format the tables as `data.table` objects and the `DATE` columns as `Date` objects:

```r
library(data.table)
library(dplyr)

rev_data <- as.data.table(rev_data)
rev_data$DATE <- as.Date(rev_data$DATE)

weather_data <- as.data.table(weather_data)
weather_data$DATE <- as.Date(weather_data$DATE)
```

Next, we wish to subtract three subsets: the revenue in January ( `rev_jan` ), the revenue in February ( `rev_feb` ), and the weather in January ( `weather_jan` ).

With `data.table` :

```r
rev_jan <- rev_data[DATE %between%
                  c("2020-01-01",
                  "2020-01-31"),]
rev_feb <- rev_data[DATE %between%
                  c("2020-02-01",
                  "2020-02-29"),]
weather_jan <- weather_data[DATE
                  %between%
                  c("2020-01-01",
                  "2020-01-31"),]
```

With `dplyr` :

```r
rev_data |> filter(between(DATE,
            as.Date("2020-01-01"),
            as.Date("2020-01-31"))
                   ) -> rev_jan
rev_data |> filter(between(DATE,
            as.Date("2020-02-01"),
            as.Date("2020-02-29"))
                   ) -> rev_feb
weather_data |> filter(between(
            DATE,
            as.Date("2020-01-01"),
            as.Date("2020-01-31"))
                ) -> weather_jan
```

A quick look at the structure of the data reveals some similarities:

```r
str(rev_jan)
str(rev_feb)
str(weather_jan)
```

The rows in `rev_jan` correspond one-to-one to the *rows* in `weather_jan` , with both tables being sorted in exactly the same way. We could therefore *bind their columns*, i.e., add the columns of `weather_jan` to `rev_jan` .

`rev_jan` and `rev_feb` contain the same *columns*. We could therefore *bind their rows*, i.e., add the rows of `rev_feb` to `rev_jan`. To perform these operations, we can use either base R or `dplyr`:

With base R:

```
# Join columns of datasets that have the same rows:
cbind(rev_jan, weather_jan)

# Join rows of datasets that have the same columns:
rbind(rev_jan, rev_feb)
```

With `dplyr` :

```
# Join columns of datasets that have the same rows:
bind_cols(rev_jan, weather_jan)

# Join rows of datasets that have the same columns:
bind_rows(rev_jan, rev_feb)
```

### 5.12.2 Merging tables using keys

A closer look at the business revenue data reveals that `rev_data` contains observations from 90 days, whereas `weather_data` only contains data for 87 days; revenue data for 2020-03-01 is missing, and weather data for 2020-02-05, 2020-02-06, 2020-03-10, and 2020-03-29 are missing.

Suppose that we want to study how weather affects the revenue of the business. In order to do so, we must merge the two tables. We cannot use a simple column bind, because the two tables have different numbers of rows. If we attempt a bind, R will produce a merged table by recycling the first few rows from `rev_data` – note that the two `DATE` columns aren't properly aligned:

```
tail(cbind(rev_data, weather_data))
```

Clearly, this is not the desired output! We need a way to connect the rows in `rev_data` with the right rows in `weather_data` . Put differently, we need something that allows us to connect the observations in one table to those in another. Variables used to connect tables are known as *keys*, and must in some way uniquely identify observations. In this case the `DATE` column gives us the key – each observation is uniquely determined by its `DATE` . So to combine the two tables, we can combine rows from `rev_data` with the rows from `weather_data` that have the same `DATE` values. In the following sections, we'll look at different ways of merging tables using `data.table` and `dplyr` .

But first, a word of warning: finding the right keys for merging tables is not always straightforward. For a more complex example, consider the `nycflights13` package, which contains five separate but connected datasets:

```
library(nycflights13)
?airlines  # Names and carrier codes of airlines.
?airports  # Information about airports.
?flights   # Departure and arrival times and delay information for
           # flights.
?planes    # Information about planes.
?weather   # Hourly meteorological data for airports.
```

Perhaps you want to include weather information with the flight data, to study how weather affects delays. Or perhaps you wish to include information about the longitude and latitude of airports (from `airports` ) in the `weather` dataset. In `airports` , each observation can be uniquely identified in three different ways: either by its airport code `faa` , its name `name` , or its latitude and longitude, `lat` and `lon` :

```
?airports
head(airports)
```

If we want to use either of these options as a key when merging with `airports` data with another table, that table should also contain the same key.

The `weather` data requires no less than four variables to identify each observation: `origin` , `month` , `day` and `hour` :

```
?weather
head(weather)
```

It is not perfectly clear from the documentation, but the `origin` variable is actually the FAA airport code of the airport corresponding to the weather measurements. If we wish to add longitude and latitude to the weather data, we could therefore use `faa` from `airports` as a key.

### 5.12.3 Inner and outer joins

An operation that combines columns from two tables is called a *join*. There are two main types of joins: *inner joins* and *outer joins*.

- *Inner joins*: create a table containing all observations for which the key appeared in both tables. So if we perform an inner join on the `rev_data` and `weather_data` tables using `DATE` as the key, it won't contain data for the days that are missing from either the revenue table or the weather table.

In contrast, outer joins create a table retaining rows, even if there is no match in the other table. There are three types of outer joins:

- *Left join*: retains all rows from the first table. In the revenue example, this means all dates present in `rev_data`.
- *Right join*: retains all rows from the second table. In the revenue example, this means all dates present in `weather_data`.
- *Full join*: retains all rows present in at least one of the tables. In the revenue example, this means all dates present in at least one of `rev_data` and `weather_data`.

We will use the `rev_data` and `weather_data` datasets to exemplify the different types of joins. To begin with, we convert them to `data.table` objects (which is optional if you wish to use `dplyr`):

```r
library(data.table)
library(dplyr)

rev_data <- as.data.table(rev_data)
weather_data <- as.data.table(weather_data)
```

Remember that revenue data for 2020-03-01 is missing, and weather data for 2020-02-05, 2020-02-06, 2020-03-10, and 2020-03-29 are missing. This means that out of the 91 days in the period, only 86 have complete data. If we perform an inner join, the resulting table should therefore have 86 rows.

To perform an inner join of `rev_data` and `weather_data` using `DATE` as key:

With `data.table`:

With `dplyr`:

```r
merge(rev_data, weather_data,
      by = "DATE")

# Or:
setkey(rev_data, DATE)
rev_data[weather_data, nomatch = 0]
```

```r
rev_data |> inner_join(
                weather_data,
                by = "DATE")
```

A left join will retain the 90 dates present in `rev_data`. To perform a(n outer) left join of `rev_data` and `weather_data` using `DATE` as key:

With `data.table`:

```r
merge(rev_data, weather_data,
      all.x = TRUE, by = "DATE")

# Or:
setkey(weather_data, DATE)
weather_data[rev_data]
```

With `dplyr`:

```r
rev_data |> left_join(
                weather_data,
                by = "DATE")
```

A right join will retain the 87 dates present in `weather_data`. To perform a(n outer) right join of `rev_data` and `weather_data` using `DATE` as key:

With `data.table`:

```r
merge(rev_data, weather_data,
      all.y = TRUE, by = "DATE")

# Or:
setkey(rev_data, DATE)
rev_data[weather_data]
```

With `dplyr`:

```r
rev_data |> right_join(
                  weather_data,
                  by = "DATE")
```

A full join will retain the 91 dates present in at least one of `rev_data` and `weather_data`. To perform a(n outer) full join of `rev_data` and `weather_data` using `DATE` as key:

With `data.table`:

```r
merge(rev_data, weather_data,
      all = TRUE, by = "DATE")
```

With `dplyr`:

```r
rev_data |> full_join(
                weather_data,
                by = "DATE")
```

### 5.12.4 Semijoins and antijoins

Semijoins and antijoins are similar to joins but work on observations rather than variables. That is, they are used for filtering one table using data from another table:

- *Semijoin*: retains all observations in the first table that have a match in the second table.
- *Antijoin*: retains all observations in the first table that *do not* have a match in the second table.

The same thing can be achieved using the filtering techniques of Section 5.8, but semijoins and antijoins are simpler to use when the filtering relies on conditions from another table.

Suppose that we are interested in the revenue of our business for days in February with subzero temperatures. First, we can create a table called `filter_data` listing all such days:

With `data.table`:

```
rev_data$DATE <- as.Date(rev_data$DATE)
weather_data$DATE <- as.Date(weather_data$DATE)
filter_data <- weather_data[TEMPERATURE < 0 &
                            DATE %between%
                            c("2020-02-01",
                              "2020-02-29"),]
```

With `dplyr`:

```
rev_data$DATE <- as.Date(rev_data$DATE)
weather_data$DATE <- as.Date(weather_data$DATE)
weather_data |> filter(TEMPERATURE < 0,
                       between(DATE,
                       as.Date("2020-02-01"),
                       as.Date("2020-02-29"))
                               ) -> filter_data
```

Next, we can use a semijoin to extract the rows of `rev_data` corresponding to the days of `filter_data`:

With `data.table`:

```
setkey(rev_data, DATE)
rev_data[rev_data[filter_data,
                  which = TRUE]]
```

With `dplyr`:

```
rev_data |> semi_join(
                filter_data,
                by = "DATE")
```

If instead we wanted to find all days *except* the days in February with subzero temperatures, we could perform an antijoin:

With `data.table`:

```
setkey(rev_data, DATE)
rev_data[!filter_data]
```

With `dplyr`:

```
rev_data |> anti_join(
                filter_data,
                by = "DATE")
```

~

**Exercise 5.26** We return to the `oslo-biomarkers.xlsx` data from Exercises 5.8 and 5.25. Load the data frame that you created in Exercise 5.25 (or copy the code from its solution). You should also load the `oslo-covariates.xlsx` data from the book's web page; it contains information about the patients, such as age, gender, and smoking habits.

Then do the following using `data.table` and/or `dplyr` / `tidyr`:

1. Merge the wide data frame from Exercise 5.25 with the `oslo-covariates.xlsx` data, using patient ID as key.

2. Use the `oslo-covariates.xlsx` data to select data for smokers from the wide data frame from Exercise 5.25.

(Click here to go to the solution.)

## 5.13 Scraping data from websites

*Web scraping* is the process of extracting data from a webpage. For instance, let's say that we'd like to download the list of Nobel laureates from the Wikipedia page https://en.wikipedia.org/wiki/List_of_Nobel_laureates. As with most sites, the text and formatting of the page is stored in an HTML file. In most browsers, you can view the HTML code by right-clicking on the page and choosing *View page source*. As you can see, all the information from the table can be found there, albeit in a format that is only just human-readable:

```
...
<tbody><tr>
<th>Year
</th>
<th width="18%"><a href="/wiki/List_of_Nobel_laureates_in_Physics"
  title="List of Nobel laureates in Physics">Physics</a>
</th>
<th width="16%"><a href="/wiki/List_of_Nobel_laureates_in_Chemistry"
title="List of Nobel laureates in Chemistry">Chemistry</a>
</th>
<th width="18%"><a href="/wiki/List_of_Nobel_laureates_in_Physiology_
or_Medicine" title="List of Nobel laureates in Physiology or Medicine
">Physiology<br />or Medicine</a>
</th>
<th width="16%"><a href="/wiki/List_of_Nobel_laureates_in_Literature"
title="List of Nobel laureates in Literature">Literature</a>
</th>
<th width="16%"><a href="/wiki/List_of_Nobel_Peace_Prize_laureates"
title="List of Nobel Peace Prize laureates">Peace</a>
</th>
<th width="15%"><a href="/wiki/List_of_Nobel_laureates_in_Economics"
class="mw-redirect" title="List of Nobel laureates in Economics">
  Economics</a><br />(The Sveriges Riksbank Prize)<sup id="cite_ref-
11" class="reference"><a href="#cite_note-11">&#91;11&#93;</a></sup>
</th></tr>
<tr>
<td align="center">1901
</td>
<td><span data-sort-value="Röntgen, Wilhelm"><span class="vcard"><span
class="fn"><a href="/wiki/Wilhelm_R%C3%B6ntgen" title="Wilhelm
Röntgen">  Wilhelm Röntgen</a></span></span></span>
</td>
<td><span data-sort-value="Hoff, Jacobus Henricus van &#39;t"><span
class="vcard"><span class="fn"><a href="/wiki/Jacobus_Henricus_van_
%27t_Hoff" title="Jacobus Henricus van &#39;t Hoff">Jacobus Henricus
van 't Hoff</a></span></span></span>
</td>
<td><span data-sort-value="von Behring, Emil Adolf"><span class=
"vcard">
<span class="fn"><a href="/wiki/Emil_Adolf_von_Behring" class="mw-
redirect" title="Emil Adolf von Behring">Emil Adolf von Behring</a>
</span></span></span>
</td>
...
```

To get hold of the data from the table, we could perhaps select all rows, copy them, and paste them into a spreadsheet software such as Excel. But it would be much more convenient to be able to just import the table to R straight from the HTML file. Because tables written in HTML follow specific formats, it is possible to write code that automatically converts them to data frames in R. The `rvest` package contains a number of functions for that. Let's install it:

```
install.packages("rvest")
```

To read the entire Wikipedia page, we use:

```
library(rvest)
url <- "https://en.wikipedia.org/wiki/List_of_Nobel_laureates"
wiki <- read_html(url)
```

The object `wiki` now contains all the information from the page – you can have a quick look at it by using `html_text` :

```
html_text(wiki)
```

That is more information than we need. To extract all tables from `wiki` , we can use `html_nodes` :

```
tables <- html_nodes(wiki, "table")
tables
```

The first table, starting with the HTML code

```
<table class="wikitable sortable"><tbody>\n<tr>\n<th>Year\n</th>
```

is the one we are looking for. To transform it to a data frame, we use `html_table` as follows:

```r
laureates <- html_table(tables[[1]], fill = TRUE)
View(laureates)
```

The `rvest` package can also be used for extracting data from more complex website structures using the SelectorGadget tool in the web browser Chrome. The SelectorGadget lets you select the page elements that you wish to scrape in your browser, and helps you create the code needed to import them to R. For an example of how to use it, run `vignette("selectorgadget")`.

$\sim$

**Exercise 5.27** Scrape the table containing different keytar models from https://en.wikipedia.org/wiki/List_of_keytars. Perform the necessary operations to convert the `Dates` column to `numeric`.

(Click here to go to the solution.)

## 5.14 Other commons tasks

### 5.14.1 Deleting variables

If you no longer need a variable, you can delete it using `rm`:

```r
my_variable <- c(1, 8, pi)
my_variable
rm(my_variable)
my_variable
```

This can be useful for instance if you have loaded a data frame that is no longer needed and takes up a lot of memory. If you, for some reason, want to wipe all your variables, you can use `ls`, which returns a vector containing the names of all variables, in combination with `rm`:

```r
# Use this at your own risk! This deletes all currently loaded
# variables.

# Uncomment to run:
# rm(list = ls())
```

Variables are automatically deleted when you exit R (unless you choose to save your workspace). On the rare occasions where I want to wipe all variables from memory, I usually do a restart instead of using `rm`.

### 5.14.2 Importing data from other statistical packages

The `foreign` library contains functions for importing data from other statistical packages, such as Stata ( `read.dta` ), Minitab ( `read.mtp` ), SPSS ( `read.spss` ), and SAS (XPORT files, `read.xport` ). They work just like `read.csv` (see Section 2.15), with additional arguments specific to the file format used for the statistical package in question.

### 5.14.3 Importing data from databases

R and RStudio have excellent support for connecting to databases. However, this requires some knowledge about databases and topics like ODBC drivers and is therefore beyond the scope of the book. More information about using databases with R can be found at https://db.rstudio.com/.

### 5.14.4 Importing data from JSON files

JSON is a common file format for transmitting data between different systems. It is often used in web server systems where users can request data. One example of this is found in the JSON file at: https://opendata-download-metobs.smhi.se/api/version/1.0/parameter/2/station/98230/period/latest-months/data.json. It contains daily mean temperatures from Stockholm, Sweden, during the last few months, accessible from the Swedish Meteorological and Hydrological Institute's server. Have a look at it in your web browser, and then install the `jsonlite` package:

```r
install.packages("jsonlite")
```

We'll use the `fromJSON` function from `jsonlite` to import the data:

```
library(jsonlite)
url <- paste("https://opendata-download-metobs.smhi.se/api/version/",
             "1.0/parameter/2/station/98230/period/latest-months/",
             "data.json",
             sep = "")
stockholm <- fromJSON(url)
stockholm
```

By design, JSON files contain lists, and so `stockholm` is a `list` object. The temperature data that we were looking for is (in this particular case) contained in the list element called `value`:

```
stockholm$value
```

31. In fact, the opposite is true: under the hood, a data frame is a list of vectors of equal length.↩
32. Courtesy of the Department of Earth Sciences at Uppsala University.↩
33. This is not strictly speaking true; if we use base 3, $1/3$ is written as $0.1$ which can be stored in a finite memory. But then other numbers become problematic instead.↩
34. Not in R though.↩
35. See `?Quotes` for a complete list.↩
36. For patient confidentiality purposes.↩
37. The Swedish *onsdag* and English *Wednesday* both derive from the proto-Germanic *Wodensdag*, Odin's day, in honour of the old Germanic god of that name.↩
38. There is also a package called `dtplyr`, which allows you to use the fast functions from `data.table` with `dplyr` syntax. It is useful if you are working with big data, already know `dplyr`, and don't want to learn `data.table`. If that isn't an accurate description of you, you can safely ignore `dtplyr` for now.↩
39. You may need to install the package first, using `install.packages("gapminder")`.↩

40. I’ve intentionally left out the details regarding the business – these are real sales data from a client, which can be sensitive information.↩