baseRintro_JH_230419

April 23, 2019

1 Introduction to Base R

Jesse Harrison and Seija Sirkiä

Welcome to this introduction to R. The objectives of this notebook are to equip you with a basic understanding of what R is, how R code works and how to manage code of your own. Another purpose is to convince you that learning R can completely transform how you work with and share your data!

The intended audience of this notebook includes those with no experience with R. You may also have very limited or no experience with programming in general. Not to worry - R is quite simple and intuitive once you learn the basics, and it only becomes more so the further you proceed.

2 1. The basics

2.1 1.1 What are R and Jupyter?

R is a programming language for handling, analysing and visualising data. It is freely downloadable and works on all major platforms, including Windows, macOS and Linux. At its core, R code is made of text-based commands that can be run from a command prompt and without a graphical user interface. To make things easier, R can also be run via an open-source interface such as RStudio (see rstudio.com).

What you are currently using is Jupyter, a web-based ("cloud") interface that can run R. While not as powerful as RStudio, it enables us to focus on some of the basics of reading, creating and editing R code.

We do not need to learn much about Jupyter to use it. We will be working with code cells such as the one below:

```
In [ ]: df <- data.frame(x = seq(0,1, length = 10)) # Right now this may look difficult, but w
        df$y <- 2 * df$x + rnorm(10, sd = 0.3)
        lm(y ~ x, df)
        rm(df)
```

Code cells have a distinct appearance and include the text "In [ ]" (on the left) to indicate that this is a field for inputting code. Next, we can try running the code. Click on the cell above to make it active and then press Control + Enter on your keyboard. You can also run the code by clicking on the "Run" button on the Jupyter tool bar above.

When you run the code, some output will appear below the cell and the text on the left should show "In [1]" (or another number). The output includes information on a "Call" and "Coefficients". If you can see this, you're ready to move on!

2.2 1.2 Checkpoints in Jupyter

Jupyter notebooks are fully editable, which means that you can edit both the text and code fields. If you delete something and want to restore a previous version of the notebook, this can be done using *checkpoints*. You can easily create a checkpoint by using the button on the Jupyter tool bar or via the File menu. **Now would be a good time to create your first checkpoint.**

Accidentally closing the browser tab of the notebook is also not a problem. Just reopen it from the Jupyter home tab. If you've closed that too (or e.g. if you wish to switch browsers or computers) you can start over from notebooks.csc.fi. Your notebook can will remain intact for as long as the virtual machine session is running.

To see how long your session is running for, you can go back to the notebooks.csc.fi dashboard. **NOTE: once the session expires, everything you wrote will be deleted, including all checkpoints.**

To keep a copy of your work, you can download the notebook in several formats (via the File menu). The following formats may be useful:

- Notebook: use this in case you want to return to the actual complete notebook on another Jupyter instance. Notebooks on notebooks.csc.fi only stay up for a limited time and if you want to work for longer than that, this is what you need. You can restart the R for Beginners machine and before clicking on the .ipynb file to open the notebook, upload and replace the notebook (the .ipynb file) with the version you downloaded earlier.
- R code: this will download only the parts involving R code (e.g. for importing in RStudio).
- PDF: this will create a PDF document which is possible to read but not work on. It's best to do this without the outputs (choose 'Restart & Clear Output' from the Kernel menu).

2.3 1.3 Simple calculations in R

By running the code above, you already performed several tasks using R. To become more familiar with how this works, let's try some of our own. An example we can try is performing a numeric calculation. For example, try typing `1 + 1` in the field below and running it. Typing code without spaces (e.g. `1+1`) also works, although long sections of code without spaces can become difficult to read.

As you can see, with tasks like these R performs similarly to a calculator. You can also use symbols including parentheses and `^` (for exponentiation). For decimal points, R accepts the full stop symbol (`.`) instead of a comma (`,`). Try out some calculations of your own! You can also try several at once by adding them on separate lines.

```
In [ ]: 1 + 1
        1+1

        60 * 24 * 7
        39.90 * (1 - 0.3)
        65 / 1.5 ^ 2
        (24 - 32) * 5 / 9
```

3 2. Assigning and managing objects

3.1 2.1 Creating objects

One of the key features of R is that your data can be assigned to a named *object* using the operator `<-` (less-than symbol followed by a dash). A quick way to type this is Alt + dash. Try running the code below, which assigns the result of a calculation to an object we have decided to call `answerA`:

```
In [ ]: answerA <- 1 + 1
```

While previously the output appeared below the code cell, here it is not immediately visible. We can access the output by running `answerA` on its own:

```
In [ ]: answerA
```

The object is now stored in your R workspace. This is extremely useful because it makes it much easier to organise, manipulate and analyse your data. As part of writing R code, we can create as many objects as needed. For example, try creating a second object for a calculation of your own, called `answerB`, and printing the results below the code cell.

```
In [ ]: answerB <- 2 * 5
        answerB
```

We can also modify the contents of an object. For example, we could multiply the contents of `answerA` by two (`answerA * 2`). If we decided to assign this back to the same object (`answerA <- answerA * 2`), we would overwrite the original contents. In case we wanted to avoid this, the results can also be assigned to a new object:

```
In [ ]: answerC <- answerA * 2
        answerC
```

We now have three objects in our workspace. When working with real data sets, the number can of course be much higher. To keep track of all the objects in your workspace, you can use the following command:

```
In [ ]: ls()
```

A great thing about working with objects in R is that you can easily manipulate your data while keeping the original values intact. This is often much more difficult to achieve with other environments for data analysis. In other words, learning R can help you **keep your raw data raw.**

3.2 2.2 Removing objects

Sometimes we may wish to remove an object from the workspace. This can be done using `rm()` (for example, typing `rm(answerA)` would remove `answerA`. There is also a command for completely clearing your workspace: `rm(list = ls())`. This deletes **everything** in your current workspace. Be sure to use these commands with caution - there is no 'Undo' button in R.

If you accidentally remove something, you will need to run your code again. In Jupyter, you can also start over with an empty workspace by choosing 'Restart & Clear Output' from the Kernel menu. For now, however, let's keep the objects you created in the previous section. You may still need them!

4 3. Code annotation

This is very easy to do, but also one of the most important parts to learn. It is possible to annotate your code using the # character. R will ignore the entire line after # (without this, it would attempt to run your text as a command, resulting in an error message). You can also use # to inactivate parts of your code, for example if you want to try a calculation while omitting some specific step preceding it.

Always annotate your code! This will enable you to keep track of things that you've done and it also helps you become more familiar with different commands in R. Not only this, but it will be easier for your colleagues to interpret your code. The principle is the same as keeping a thorough laboratory notebook: well-annotated code will make your work more reproducible and easier to communicate.

5 4. Recap

Take a moment to try out different expressions in the cell below until you get the hang of it. For example, you can use the objects you already created as initial values or as intermediates. You can also try adding annotations to your code and removing (as well as re-creating) your objects. Name the new objects as you wish, but keep the following guidelines in mind:

- The name must begin with a letter.
- It can contain numbers, full stops and underscores, and is case-sensitive.
- While it is possible to include spaces in object names, this requires extra effort and is not recommended.
  - In general, it is good practice to keep your code as simple and easy to read as possible.
- Letters with accents and umlauts are best to avoid.
  - They may not work on all systems, which can become problematic when sharing your code with others.

```
In [ ]: hour <- 60
        day <- 24 * hour
        7 * day
```

6 5. Numeric vectors

You have now learned the basic logic of how R works with simple expressions and individual numbers. The next step toward doing some real data analysis in R involves getting to grasp with *vectors*. There are three types of vectors in R: numeric, character-based and logical. We will start with numeric vectors.

- Numeric vectors are ordered sequences of numbers (the order is data-dependent and the sequence can be of any length)
  - An example: 3, 67, 2, 500, 0.5, 10

You can do calculations with vectors just like with individual numbers. The calculations are separately applied to each value. Actually, single numbers **are** vectors with a length of 1!

- Most mathematical functions also work on the same principle when applied to vectors
  - `exp()`, `log()`, `abs()` and so on
- However, some functions treat the vector as a whole
  - `length()`, `sum()`, `mean()`, `median()`, `sd()`, `var()`, `min()`, `max()`
- `summary()` tells us several things about a vector at once

6.1 5.1 Initial exercises

Let's consider a data set where we have numbers of steps and the estimated number of calories burned per day (from an activity bracelet worn for one week). First, let's create vectors of the data using `c()` and assign them to two separate objects. Then let's perform some calculations:

```
In [ ]: steps <- c(106, 0, 12775, 8287, 9222, 7080, 6055)
        kcal <- c(1356, 1341, 2109, 1882, 1970, 1938, 1851)

        # average hourly no. of steps each day
        steps / 24

        # energy burned per step each day
        kcal / steps
```

Note that the results are given as vectors. We also see a strange value called `Inf` (more on this later). For now, let's assign the average number of steps to a new object and calculate their overall sum:

```
In [ ]: hourlysteps <- steps / 24
        tot_hourlysteps <- sum(hourlysteps) # One example of how we could calculate the sum
        tot_hourlysteps
```

It is also possible to create `tot_hourlysteps` using just a single line of code. There are often many ways to achieve the same result in R. Choosing which way to write your code may depend on the situation or personal preference:

```
In [ ]: sum(steps / 24) # Another way to achieve the same thing
```

Next, try both of these ways to calculate the average hourly energy consumption over the week (that is, the total energy consumed over the week divided by the number of hours in a week). Now might also be a good time to create another checkpoint in case you already haven't!

```
In [ ]: totenergy <- sum(kcal)
        weekh <- 24 * 7
        totenergy / weekh

        # or like this:
        sum(kcal) / (24 * 7)
```

6.2 5.2 Accessing specific values within a vector

As we have seen, the data within a numeric vector are always arranged in a specific order. We can refer to this ordering using *indices*. For example, the first value within a vector has an index of 1, the second an index of 2 (and so on). Sometimes it is of interest to extract or access only certain values within a vector. This can be done using square brackets and the index:

```
In [ ]: steps[1]
```

It is also easy to specify a range of elements using this notation:

```
In [ ]: steps[1:5]
        kcal[6:7]
```

Again, the results are given as vectors. Using the object `steps`, try calculating the average numbers of steps separately for weekdays and the weekend. Hint: have a look at the list of functions given at the beginning of Section 5!

```
In [ ]: mean(steps) # average for the whole week
        mean(steps[1:5]) # average for weekdays
        mean(steps[6:7]) # average for the weekend
```

What if you wanted to extract values corresponding to Monday, Wednesday and Friday? Remember that a single index value is also a vector (with a length of 1). By extension, we can use a vector with a length of n to extract these values:

```
In [ ]: steps[c(1,3,5)]
```

Note that the following gives an error (because `1,3,5` is not a vector while `c(1,3,5)` is). If you were wondering why the error message talks about dimensions, this is because the code below treats `steps` as a 3D array (as opposed to a one-dimensional array, that is, a vector).

```
In [ ]: steps[1,3,5]
```

7 6. Logical vectors

Logical (also known as Boolean) vectors have two possible values: `TRUE` and `FALSE`. They come up as results from logical operations, such as comparing one value to another. Here we are told, element by element, whether the number of steps taken was above 1000:

```
In [ ]: steps > 1000
```

The possible comparisons are:

- less than `<`
- greater than `>`
- less or equal `<=`
- greater or equal `>=`
- equal to `==`
- not equal to `!=`

There are also logical operators: - and `&` - or `|` - not `!`

A convenience operator for querying whether a value is equal to one of several in a given set exists: `x %in% 1:3`.

NOTE: If you have never played with logical values before, do not worry. For basic usage, the situations where they are needed are quite limited. We will cover two common scenarios in the following sections.

7.1 6.1 Initial exercises

One scenario where logical vectors are useful is when you want to subset your data (for example, extracting part of a vector). For example, we can calculate the mean energy use on days where the number of steps exceeded 1000, presumably when the bracelet was worn all day long:

```
In [ ]: allday_steps <- steps > 1000 # select days where the number of steps exceeds 1000
        allday_steps

        allday_kcal <- kcal[allday_steps] # extract kcal values for those days
        allday_kcal

        mean(allday_kcal) # calculate the mean
```

Previously we tried to calculate the mean energy consumption per step, but we did not account for energy consumption during rest (this is called the basal metabolic rate or BMR). Indeed, on one of the days had zero steps recorded while some energy was still consumed. For the purposes of this exercise, let's call this level of energy use the BMR.

Let's attempt to recalculate the mean energy consumption per step using only that part of the data that exceeded the BMR. We can do this step by step:

```
In [ ]: # NOTE: To complete this exercise, replace the points marked with '??' with the correc

        # First we need to find the BMR value in our data.
        # Hint: It's the element that coincides with the 'TRUE' value of the logical vector "s

        steps0 <- steps == 0
        BMR <- kcal[steps0]
        BMR

        # You should see the BMR value come out as 1341

In [ ]: # Next we can subtract the BMR value from the kcal values

        aboveBMR <- kcal - BMR
        aboveBMR

        # You should now see smaller kcal values, with 0 as the second element

In [ ]: # Then we can recalculate the energy consumption per step:

        kcalperstep <- aboveBMR / steps
```

```
kcalperstep

# You should see values between 0.060 and 0.085 for the full days,
# something a little higher for the first day, and "NaN" for the second
```

This result seems to make more sense! The `NaN` value will be explained in the next section.

While the same calculation could have been performed by directly typing in the BMR value (1341), the approach we took is more useful because it illustrates the principle behind extracting specific values from a data set. Real data often need some form of subsetting (or other pre-treatment steps) and this is a good skill to learn!

7.2 6.2 Missing values

7.2.1 6.2.1 Data with 'NA' values

The first two values of the `steps` vector, 106 and 0, were probably due to the bracelet not being worn for the whole day or at all. In other words, we don't know the true step count because we are missing data. The R language provides an explicit way of stating this, the `NA` symbol ("Not Available").

We could create an object with the step counts for Monday and Tuesday listed as `NA`. Let's also try calculating the overall sum of steps for this vector:

```
In [ ]: steps_na <- c(NA, NA, 12775, 8287, 9222, 7080, 6055)
        sum(steps_na)
```

When a vector contains values denoted as `NA`, the sum of this vector is also reported as NA. This makes sense because we cannot know the result! However, what we can do is to calculate the sum of the remaining values while removing the missing data using a single additional *parameter*. This is another scenario where you commonly encounter logical values:

```
In [ ]: sum(steps_na, na.rm = TRUE)
```

The default behaviour of `sum()` and many other functions is to include missing values in the calculation. However, we have just changed how `sum()` behaves in this particular instance by adding the parameter `na.rm = TRUE`. We will learn more about functions and parameters later in this notebook.

Sometimes we do not directly know which values are missing in a data set. A convenient way to find out is by using the *test* function `is.na()`. One might also be tempted to try a logical comparison using `==`. Let's try both (and think for a moment why the second result does not tell us much!):

```
In [ ]: is.na(steps_na)
        steps_na == NA
```

In the previous example there was a related symbol, `NaN`, which stands for 'Not a Number'. This resulted from the calculation 0/0, which has no well-defined mathematical meaning. For our purposes, the difference between `NA` and `NaN` can safely be ignored. We also saw a value called `Inf` (or 'Infinity'). This resulted from calculating 1341/0.

7.2.2 6.2.2 Converting values to 'NA'

Sometimes one needs to deal with data sets where missing values are indicated by a number such as 0. This is problematic because such values will influence the outcome of calculations and other analyses. While it is best to avoid this type of notation from the outset, it is possible to rename such values as `NA` directly in R:

```
In [ ]: steps[steps == 0] <- NA
        steps
```

This converted Tuesday's measurement to `NA`, but we also know the first value in the steps vector is unreliable. How would you convert the values from both Monday and Tuesday? One way would be to use indices just like before:

```
In [ ]: steps[c(1, 2)] <- NA
        steps
```

8 7. Character vectors and factors

Vectors come in one more flavour: character. The elements of character vectors are snippets of text, such as here:

```
In [ ]: treatg_char <- c("Trt1", "Trt2", "Ctrl", "Trt1", "Ctrl", "Trt2") # Two treatment group
```

Note that each element needs to be surrounded by quotation marks. Both the ' and " symbols can be used.

Data like this often represent group membership. While some the elements could belong to specific groups (such as different treatments or an experimental control), character vectors do not take these interrelations into account because each element is treated as entirely independent. However, there is a specific type of object in R that can be used to group the data: *factors*. The defining feature of a factor is that it can have several *levels*. For example, in our case we have six elements belonging to three levels: Trt1, Trt2 and Ctrl.

To clarify things further, let's describe the character vector using the `summary()` function:

```
In [ ]: summary(treatg_char)
```

Then let's change the character vector into a factor:

```
In [ ]: treatg <- factor(treatg_char)
        summary(treatg)
```

In the first case, nothing is told about the contents beyond the length and type. The second summary looks more useful: we can see that the factor has three levels, each with two observations. Here are two other functions that give us the titles and number of levels:

```
In [ ]: levels(treatg)
        nlevels(treatg)
```

Now that you are familiar with vectors and factors, you are ready for the most important data type: data frames.

9 8. Data frames

9.1 8.1 Creating and accessing a data frame

So far, the exercises in this notebook have dealed with individual vectors that weren't actually connected to each other. We treated the `kcal` and `steps` vectors as if their elements corresponded to the same days of the week, but the data were assigned to separate objects. In the following the same data are assembled into a *data frame*. Data frames are where all data are usually kept and manipulated. They can be considered **the most important data structure in R**.

For clarity, let's empty the workspace first:

```
In [ ]: rm(list = ls()) # You may remember this from Section 2.2!
```

Now we are ready to create a data frame:

```
In [ ]: bracelet <- data.frame(day = c("Monday", "Tuesday", "Wednesday", "Thursday", "Friday",
                                 steps = c(106, 0, 12775, 8287, 9222, 7080, 6055),
                                 kcal = c(1356, 1341, 2109, 1882, 1970, 1938, 1851))
        bracelet
        summary(bracelet)
        ls()
```

You can see that the whole data set is now shown as a table with rows and columns. This is the proper way to think about data frames: they are tables where columns are variables (in the statistical sense) and rows are observations. The columns of a data frame can still be treated like vectors or factors. The only difference is that you have to refer to the data frame where the data are stored, using the character `$`:

```
In [ ]: bracelet$steps # typing just 'steps' would now give an error message!
```

9.2 8.2 Initial exercise

Armed with this knowledge, try and see if you can redo some of the previous calculations related to `steps` and `kcal`, using the code cell below. Let's focus on calculating the mean energy consumption per step while accounting for the BMR (as originally described in Section 6.1).

NOTE: This could be challenging and you may run into some pitfalls. If something goes wrong, take your time and try to understand what happened. Remember that you can use `ls()` to see the current objects and that you can restart from the previous code cells if you get confused.

```
In [ ]: steps0 <- bracelet$steps == 0
        steps0

        BMR <- bracelet$kcal[steps0]
        BMR

        aboveBMR <- bracelet$kcal - BMR
        aboveBMR

        kcalperstep <- aboveBMR / bracelet$steps
        kcalperstep
```

9.3 8.3 Working with the iris data set

A key step in learning how to analyse data with R would naturally involve importing your own data to work with. We will soon cover this, but first we need to cover a few more topics including functions, working directories and CSV files. To do this, let's still make use of some of the example data that come as part of a default R installation.

The example data sets in R can be imported into the workspace using a function called `data()`. First, let's look at a classic example, the iris data set:

```
In [ ]: data(iris)
        summary(iris)
        head(iris)
```

The summary shows us that there are five variables: four numeric ones and one factor with three levels (each with 50 observations). The `head(iris)` call is used to look at a few first lines of the data frame, instead of looking at all 150 of them. Next, we can look at the documentation page of this data set. This will open a new pop-up window in Jupyter. You can close it afterwards or you can move it to its own tab using the button in its top right corner.

```
In [ ]: ?iris
```

It is said that, when doing data analysis, at least 80% of the time is spent on cleaning, rearranging and otherwise preparing the data. R is great for that, particularly when combined with a good user interface (RStudio) and some dedicated *packages* (more on those in Section 12). Most of that is beyond this basic introduction, but a few simple tricks will get us started. The first one is how to add new variables into the data frame. One way to do this is by simple assignment. Let's pretend that the sepal leaves of the iris flowers are rectangular, and add their area to the data:

```
In [ ]: iris$Sepal.Area <- iris$Sepal.Width * iris$Sepal.Length
        summary(iris)
```

Now try the same with petal leaves:

```
In [ ]: iris$Petal.Area <- iris$Petal.Width * iris$Petal.Length
        summary(iris)
```

We can also try taking subsets of the data frame. Recall how we used brackets and indices for extracting parts of vectors. The same kind of syntax works for data frames, except that there are now two sets of indices, one for rows and another for columns. Consider this:

```
In [ ]: corner <- iris[1:5, 1:4]
        corner
```

This created a new data frame, which consists of the rows 1 to 5 and columns 1 to 4 of the iris data set. The indices before the comma refer to rows and the indices after it refer to columns. Leaving the other index out completely means "all of them". This means that, in addition to the $ notation from before, you can also extract the data from an entire column like this:

```
In [ ]: iris[,4]
```

Earlier we also learned about using logical vectors to extract parts of vectors that satisfy a condition. This works for data frames too, in an analogous manner. However, there is also a function called `subset()` that is more convenient to use. Here are two examples:

```
In [ ]: smallones <- subset(iris, Sepal.Length < 5)
        setosas <- subset(iris, Species == "setosa")
```

This function works so that you first supply the data frame you want to subset and you then specify the logical rule for the rows you want to keep. Note that inside this function call you only need to mention the data frame once, and then all its columns can be referred to without the `$` symbol.

Try now to calculate the average sepal length for each species separately. You should know (at least) two ways to do this. First, you can exploit the fact that the species observations are nicely in order, 50 observations at a time. Second, because things are usually not this nicely arranged, you should try splitting the data frame into three single-species parts with `subset()`. This solution was actually started in the previous code cell already.

(Note that while these approaches suit us currently, there are other ways to work with the data that could be used 'in the real world'. We will not cover those here, but for example one could use the `aggregate()` function in base R or `summarise()` available via the package `dplyr`. Packages are discussed later in more detail. For illustrative purposes, the solution using `aggregate()` is given in the code cell below.)

```
In [ ]: setosas <- subset(iris, Species = "setosa")
        summary(setosas)

        # Or take the mean specifically:
        mean(setosas$Sepal.Length)

        # It's possible to do it in one line:
        mean(subset(iris, Species == "setosa")$Sepal.Length)
        mean(subset(iris, Species == "versicolor")$Sepal.Length)
        mean(subset(iris, Species == "virginica")$Sepal.Length)

        # The solution using 'aggregate()'
        # aggregate(Sepal.Length ~ Species, data = iris, FUN = mean)
```

The correct answers are 5.006, 5.936 and 6.588 for setosa, versicolor and virginica, respectively.

10 9. More about functions and their documentation

Everything in R is accomplished by using functions. In fact, once you know the basics, learning to do data analysis using R often revolves around developing a working understanding of different functions and how they might be applied in different situations. Luckily there is an active and helpful R community online (one example is stackoverflow.com). Another useful resource comes in the form of documents - each R function comes with its own documentation with details on how exactly it can be used.

You can bring up the documentation of a function by typing the question mark `?` in front of the function name (without the parentheses), like before with the iris data set. Try this now with the function `mean`:

```
In [ ]: ?mean
```

The documentation pages have a common structure, although not all have every section present. First there is **Description**, an outline of what the function does. **Usage** and **Arguments** list all the arguments the function accepts (including their type, e.g. numeric or logical) and their default values. There is often a section on **Details**, a longer explanation of how the function works and is intended for use. **Value** explains what kind of output the function will produce. At the end, there is usually a list of related functions under **See also** and finally some practical **Examples**.

Try now to figure out how to use a completely new function based only on its documentation. Use `sample()` to produce

- a Finnish lottery ticket: 7 randomly drawn numbers out of 40
- a vector of 0s and 1s, representing 10 coin tosses
  - you can also try using a factor with the levels 'heads' and 'tails', if you feel like it!

```
In [ ]: sample(1:40, 7)
        # sample(40, 7)

        sample(c(0, 1), 10, replace = TRUE) # sampling with replacement
        # sample(0:1, 10, replace = TRUE)
        # sample(2, 10, replace = TRUE)

        cointoss <- factor(c("heads", "tails"))
        sample(cointoss, 10 , replace = TRUE)
```

11 10. Importing data from CSV files

Data can come in many formats, but we will cover one of the most typical: CSV files. A CSV ('Comma-separated values') file is a text file containing data in a tabular format. The first row is usually a header row containing column (variable) names, with the rest of the rows containing the actual data. Columns are separated by commas. Software packages such as Excel, SPSS, SAS and many others are able to export data in this format. Here's an example of what such data look like (these are the first few rows of a CSV file called weather-kumpula.csv we will soon use):

```
ts,year,month,day,dp,rmm,wdir,ws,t,rh,p
2014-01-01,2014,1,1,0.7526754690757457,1.640000000000001,158.0,4.61702571230021,3.0535788742182
2014-01-02,2014,1,2,-2.2381944444444573,0.6000000000000004,118.0,3.627777777777778,0.4078472222
2014-01-03,2014,1,3,-1.4593055555555527,0.5000000000000002,141.0,3.3281944444444385,1.0606944
2014-01-04,2014,1,4,0.3417361111111121,3.6999999999999966,219.0,4.28805555555555,1.7476388888888
```

The function for importing data in this format is called `read.csv()`.

Were we to check its documentation, we would see that `read.csv()` is an alias for a function called `read.table()`. This function can read many other tabular data formats than just CSV files, although we don't need to do this now. One thing to note, however: Finnish locale settings and some others use commas as the decimal (rather than column) separator. Because of this, there also exists a non-standard version of the CSV format that uses the semicolon (`;`) as the column separator. For these situations, the function `read.csv2()` will work.

All you need to tell `read.csv()` is where to find the file. It is done by giving the name of the file, as a character string, like this:

read.csv("filename.csv")

This only works if the file is in your current R session *working directory*. To see what the current working directory is, use this command:

```
In [ ]: getwd() # This will likely say: '/home/jovyan/work/R-for-beginners'
```

To set a new working directory or to access a file outside it, you can:

- change the current working directory to something else using the function `setwd()`
- give the whole path to the file
  - note to Windows users: instead of the \ character, you have to use / or \\
- use interactive file choosing: `read.csv(file.choose())`
  - note: this does not work on this notebook!
- use the import wizard in RStudio (although the above options also work in RStudio)

In the case of this notebook, we can import the weather data and assign it to an object using the following code:

```
In [ ]: weather <- read.csv("data/weather-kumpula.csv") # The data are imported as a data fram
```

Now you should be able to both import data *and* understand what's going on when you do that. If you have any data of your own, now would be a great time to try and import it. You just need to upload it to the notebook first (from the Home page). If not, let's get familiar with the weather data.

These data consist of daily weather measurements for one year at Kumpula, Helsinki. The names of the variables are:

```
In [ ]: names(weather)
```

and they stand for:

- timestamp
- year
- month
- day
- dewpoint (řC)
- rainfall (mm)
- wind direction (ř)
- wind speed
- temperature (řC)
- relative humidity
- air pressure (millibars)

You can now use the knowledge you have to get a better handle of this data set. For example:

- Use the function `dim()` on the whole data frame. What does this function tell you?

- Use `summary()` on the whole data frame get an overall feeling of the values.
  - For example, what is the range of temperatures observed over the year?
- Take a look at a function called `complete.cases()`. Use it, possibly together with `subset()` and/or `dim()`, to see on how many different days there are missing values.
  - There are different ways to do this, but it is possible to make use of the logical operators list in Section 6.
  - (Answer: three)
- How much did it rain in September, October and November in total?
  - Remember the `%in%` operator mentioned back in Section 6!
  - (Answer: 195.48 mm)

```
In [ ]: dim(weather) # this tells us the data frame has 365 rows and 11 columns

        summary(weather)

        # complete.cases will give a logical vector with FALSE for the rows that have even a s
        allthere <- complete.cases(weather)
        weather_complete <- subset(weather, allthere)
        dim(weather_complete) # 362 rows, so on three days

        # another way to do this
        weather_missing <- subset(weather, !allthere)
        weather_missing

        autumn <- subset(weather, month %in% 9:11)
        sum(autumn$rmm)
```

12 11. Data analysis using formulas

You are now almost ready to start working with real data: you have learned about vectors, factors and data frames, are able to import data into R and understand how the language works. There is one more concept that we'll cover in this notebook and which is useful to learn: *formulas*. Here's an example:

```
In [ ]: # let's make sure you have your iris data set with you still:
        data(iris)
        # create a scatter plot of sepal length vs sepal width:
        plot(Sepal.Length ~ Sepal.Width, data = iris)
```

`plot()` is a general-purpose plotting function. What we have typed inside the brackets is a *formula* specifying the type of plot we want, and a data frame including the required variables. What characterises formulas in R is the `~` character in the middle. Its purpose is best revealed if you think of the `~` character as something similar to the word "by". In this case, we plotted `Sepal.Length` "by" `Sepal.Width`. When both variables are numeric vectors, `plot()` creates a scatter plot.

Here's another example:

```
In [ ]: lm(Sepal.Length ~ Sepal.Width, data = iris)
```

Same formula but a different function: this time we used `lm()` to fit a linear model corresponding to the scatter plot above. The left-hand side of the formula contains the response (dependent) variable whereas on the right we have the explanatory variable. The output of `lm()` is minimal, showing only the coefficient estimates of the fitted model. Use `summary()` on the result too see something more enlightening:

```
In [ ]: model <- lm(Sepal.Length ~ Sepal.Width, data = iris)
        summary(model)
```

(Note: while the output of `lm()` seemed to be just a bit of text, it is actually a *list* object. Lists are a data type we have skipped in this notebook, but you can visit the documenation of `lm()` to get a better idea of what the result list actually is. Also note that data frames are in fact a special case of lists. There is also a an object type called a *matrix* that we have not covered in this notebook. A matrix is similar to a vector with the exception of being two-dimensional. Unlike a data frame, a matrix usually only houses numeric information.)

Here is one more example of where you might use formulas. We can see if there's a difference in sepal length between two of the species using a t-test:

```
In [ ]: t.test(Sepal.Length ~ Species, data = iris, subset = Species != "versicolor")
```

Further to specifying the formula and the data frame, we have also included an argument for subsetting the data frame. Functions that take a formula as an argument usually also accept a subset argument. It works exactly like the `subset()` function we encountered earlier, but skips creating a separate subsetted data frame.

Formulas establish the difference between factors and vectors rather clearly. Consider this example data set:

```
In [ ]: data(ToothGrowth)
        summary(ToothGrowth)
```

This data set contains the results from an experiment concerning the effect of vitamin C on the tooth length of guinea pigs. There are two supplement methods and three dose levels, making up a total of six groups with 10 animals in each. The `supp` variable is a factor while `len` and `dose` are numeric. However, it is a valid question to ask whether the doses should be treated as numeric values or as group labels. The difference should become apparent if we convert `dose` into a factor. We can try fitting models and plotting the data for both scenarios:

```
In [ ]: # Version with 'dose' as numeric

        summary(lm(len ~ dose, data = ToothGrowth))
        plot(len ~ dose,data = ToothGrowth)
```

```
In [ ]: # Version with 'dose' as a factor

        ToothGrowth$dose.factor <- factor(ToothGrowth$dose)

        summary(lm(len ~ dose.factor, data = ToothGrowth))
        plot(len ~ dose.factor, data = ToothGrowth)
```

One final case where formulas could be useful: earlier, when calculating groupwise means with the iris dataset, it was mentioned that these could also be calculated using the function `aggregate()`. That function is also compatible with formulas. Try if you can figure out how to use `aggregate()` in combination with a formula to calculate the mean tooth length for each dose (you can use either the numeric or factorial version of `dose`):

```
In [ ]: aggregate(Sepal.Length ~ Species, iris, FUN = mean)
```

Finally, feel free and try any of these plots and other methods using the weather data. Here are a few questions for inspiration:

- Make box plots of monthly temperatures. Which month seems to show the most fluctuation?
- Is there any relationship between air pressure and wind speed?
- How about between air pressure and temperature? Does the answer differ between summer and winter?

```
In [ ]: # If 'month' was a factor, you could just do as follows:
        # plot(t ~ month, weather)
        # However, since it's not, that would just create a scatter plot.

        # You'd either have to make a factor version of 'month' or use boxplot() instead of pl
        boxplot(t ~ month, weather)
        # January and May seem to have a wide range of temperatures

        plot(ws ~ p, weather)
        summary(lm(ws ~ p, weather))
        # looks like higher wind speed at lower pressure

        plot(t ~ p, weather)
        summary(lm(t ~ p, weather))
        # this is harder to interpret

        plot(t ~ p, weather, subset = month %in% 6:8)
        plot(t ~ p, weather, subset = month %in% c(1, 2, 12))
        # sort of looks like higher pressure coincides with higher temperature in summer and t
```

13 12. Packages

A big part of R's success in the world of data analysis is that the scientific community is able to endlessly expand the collection of available methods through *packages*. Packages are user-developed open-source collections of functions and their associated documentation. So far we have been able to get by with functions that come with a default installation and a fresh R session. Only the most basic packages are automatically enabled - these include what we call 'base' R functions (i.e. what we have focused on in this notebook).

To enable packages, one needs to use the `library()` command, for example:

```
In [ ]: library(stats)
```

Here, `stats` is one of the packages that come as part of a default R installation. It also contains the function `lm()` we used in Section 11). More (roughly 14 000 at the time of writing this) are available on CRAN, the main repository of R packages. These need to be installed first using for example the function `install.packages()` (or one of the menu items when using some UI), and again taken in to use with the `library()` command.

The abundance of packages is overwhelming. In practice, however, we end up using only a limited selection that matches our needs. To avoid getting lost, you can use the traditional methods of asking colleagues, and searching the Internet for advice and practical examples. For another good resource, you can have a look at Task Views on the CRAN home page. These are curated lists of packages related to a certain field of study, with short explanations of which functions perform which tasks.

13.1 What next?

You have reached the end of this notebook. Hopefully you have learned enough of the basics to start learning more on your own, or be able to tell what is going on if you are faced with a colleague's R script. You can actually test this: scroll back to the very beginning of this notebook, to the first code cell. You should be able to tell a function from an assignment etc. You will see some unfamiliar function calls but you should know how to bring up their documentation to see what they do. Another thing you can try: search for a method or task familiar to you and add "with R" to the search, see what you find… and see if you can understand the R code part of the content.

Of course, consider installing both R and RStudio on your own computer. Note that they are two different things: R is provided by CRAN and comes with its own (very basic) interface. RStudio is an alternative user interface provided by the RStudio company. The open-source version of RStudio can be downloaded for free.

Using RStudio efficiently is its own separate learning objective and is intertwined with learning how to use a collection of packages called **tidyverse**, also provided by the RStudio company. See the tidyverse website for more information and the book 'R for Data Science' linked there. These packages have an underlying philosophy that slightly differs from base R, so prepare to relearn a few things. Watch videos, read through tutorials, take a course online or offline. It'll be worth it. Good luck!