baseRintro

November 10, 2017

1 Introduction to Base R

Seija Sirkiä

Hello, and welcome to this introduction to R. The aim of this notebook is to give you a solid understanding of the basics so that you can start learning the more advanced features suitable to your needs.

The intended audience of this notebook are those with no experience with R and very limited or no experience with programming in general. If you do have experience in programming, you will probably find it wordy and sparse on content, but on the other hand quick to go through.

1.1 About Jupyter

Note that R itself is in fact a programming language (accompanied by something called an *interpreter*, of that language), not a full software suite like for example SPSS or Excel. As such, it doesn't *look* like anything on its own. On top of that, R *code* is just text and could be written with anything. What you are usually looking at when you are looking at "R" is some kind of a user interface, nowadays most commonly RStudio (you can go to rstudio.com to have a look). However, what you are currently looking at is Jupyter, a web based interface capable of running R as well as a number of other things. It might be a little confusing at some points to have your code run in the web browser (or more correctly, "in the cloud") and not actually on your own machine. In addition it isn't as powerful for the serious data analysis work you'll be doing later on as RStudio would be. But it does let us focus on just the R language and code for now.

There's no need to learn much about Jupyter itself for the purpose of this R introduction. The main thing is this: there are code cells, such as this one below here:

```
In [ ]: df <- data.frame(x = seq(0,1,length=10))
        df$y <- 2*df$x + rnorm(10,sd=0.3)
        lm(y~x,df)
        rm(df)
```

You can tell the difference between a code cell and a text cell by just their appearance and context, but also, look at the space on the left hand side of the cell: code cells have the text "In [ ]" there to indicate that this is a code input. Now you can run the code! Click on the cell above to make it active, and then click on the toolbar button that looks like the play button (the one that says "run cell, select below" on mouseover). Alternatively, you can press Shift+Enter on your keyboard. Go ahead, run the code!

What should happen is that the text on the left hand side becomes "In [1]" (or possibly some other number), and some new text appears below, talking about Call and Coefficients. If that's the case, you are ready to move on!

By the way, every single character in that tiny snippet of code, as well as what it actually does, should be clear to you by the time you reach the end of this notebook.

1.1.1 Just a bit more about Jupyter

As mentioned above, you don't need to pay too much attention to Jupyter while going through this notebook. However, since the notebook is fully editable, you might e.g. accidentally remove something. You can recover from such an event using checkpoints, which are way to save the contents of the notebook at a given time. You can create a checkpoint using the button on the toolbar, or from the File menu, and revert to a checkpoint from the File menu. **Now would be a good time to create the first checkpoint.**

You can of course change this notebook as much as you like! Change the text, add comments, add new code cells, even create a completely new notebook... but for now you are on your own in figuring out how to do all that.

Accidentally closing the browser tab of the notebook is not a problem. Just reopen it from the Jupyter home tab. If you've closed that too (or your browser crashed, or even if you just want to switch to another browser on another computer) you can start over from notebooks.csc.fi. All of it remains as it were for as long as the virtual machine is running.

But the virtual machine won't run forever. It will stay online for a limited time (such as 4 hours) and once that time is over, **everything you wrote will be gone, including your checkpoints**. You can go back to the notebooks.csc.fi dashboard to see how long is left.

At the end of the day you might want to keep a copy of all of your work. From the file menu you can download the notebook in several formats, of which the following three might interest you for now:

- as a Notebook: use this in case you want to return to the actual complete notebook on another Jupyter instance. As said, the ones on notebooks.csc.fi will only stay up for a limited time, so if you want to work 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) you downloaded earlier, with all of your previous work and continue from there.
- as R code: this will download just the R code parts to be used again e.g. in RStudio
- as pdf: this will create a pdf document which you can read to see what you did, but not actually work on (it's best to do this without the outputs: choose 'Restart & Clear Output' from the Kernel menu)

And now, onwards to learning R

1.2 Expressions and the R interpreter

Doing anything using R comes down to *the R interpreter evaluating expressions*. An expression is a line of code that forms a kind of a question, and evaluation is the process of finding the answer. If you ran the first example in the previous section, you have actually already made R evaluate an expression, or several in fact. But now it's time for you create an expression all on your own.

Here's a code cell for you. Type out a question, and run the cell (using the toolbar button, or Shift+Enter).

```
In [ ]:
```

It's quite likely that the answer you get is some kind of an error. This trick exercise serves to make a couple of points:

- errors like this will happen and they aren't dangerous
- the R interpreter can only evaluate proper R code
- in Jupyter you can go back to a code cell, edit the code and re-run it

And that's what you should do next. The simplest form of an expression that does make sense and can be evaluated and gives out a well defined answer is a numeric calculation. So go ahead, type 1+1 in the code cell and re-run it, and make sure you get the familiar answer.

Ordinary calculations use the common symbols, including ^ for exponentiation. Parentheses work as usual. The decimal separator is the point/dot `.` (and not comma `,`). Try out calculations in the code cell below! Invent your own, or use these questions as inspiration.

- How many minutes are there in a week?
- A glass vase of 39.90€ is put on 30% sale. What is the new price?
- Body mass index is calculated as weight (in kg) per height (in m) squared. Find yours! Or use Google to find out the numbers for some celebrity. Or think up an imaginary person.
- If the current outside temperature in Fahrenheit degrees is 24, what should you wear on a trip to downtown?

Note that you can put many expressions (calculations) in the same code cell! Just put them on separate lines.

```
In [ ]:
```

1.3 Assignments

Previously, the results of the evaluations were printed out for viewing. It is also possible to *assign the result value to a name* using the symbol `<-` (that's the less-than sign and dash; on both Jupyter and RStudio you can also use alt-dash to type it out easily and with the nice whitespaces around) like this:

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

Now if you run this code cell, it seems that nothing happens but in fact it ***creates an object in the workspace*** called `answer` which can afterwards be used again in other expressions, like this:

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

You can always check the names of the objects currently in the workspace using this command:

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

The workspace is shared between all code cells, and it is changed only if you run or re-run some code. It doesn't matter in which order the code is typed out, only the order in which it is run. Remember, the code itself is just text until it's run through the interpreter, and after that it is again just text. If you get confused, it's possible to start over from empty workspace and clean notebook by choosing 'Restart & Clear output' from the Kernel menu.

Please take a moment of trying out assignments and expressions in the cell below until you get the hang of it. You can for example revisit the calculations from before and use objects for the initial values, and/or intermediate and final results. The object names can be anything but there are some rules to keep in mind:

- The name must begin with a letter
- It can contain numbers, dots and underscores, and is case sensitive
- It can't contain whitespaces
  - (In fact this is not entirely true! It can but that requires special effort. Don't go there.)
- Letters with accents and umlauts may or may not work or be portable; try to avoid

```
In [ ]:
```

1.3.1 The comment character

One more thing about the R interpreter: the comment character. You can add non-code text to your R code, for example to explain (mostly to yourself!) what you are doing, using the `#` character. The interpreter will completely ignore the whole line after that character. It is also a quick way to temporarily "remove" a line of code from a code snippet. You can try this out now, or wait for an example in the next section.

1.4 Numeric vectors

You have now learned the basic logic of how the R interpreter works using simple expressions and assignments. These have so far involved just single numbers. In order to do some proper data analysis you of course need to learn how to deal with actual data. The next stop on that route is *vectors*. They come in three basic flavors: numeric, character and logical. We start with the numeric ones.

- Numeric vectors are ordered sequences of numbers
- They are created with the *function* `c()`
  - Or as a result of some other function (we will learn more about functions in due time)
- You can do calculations with them just like with single numbers, they happen element by element
  - Actually, single numbers **are** vectors, just of length 1!
- Most mathematical functions also work elementwise
  - `exp()`, `log()`, `abs()` and so on
- Some functions treat the vector as a whole
  - `length()`, `sum()`, `mean()`, `median()`, `sd()`, `var()`, `min()`, `max()`
- `summary()` tells several things about a vector at once

Below are some numbers - steps taken and estimated calories burned per day - from an activity bracelet from a single week, and a few calculations concerning them.

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

        # estimate of distance walked each day:
        steps*0.7

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

- Most mathematical functions also work elementwise
  - `exp(),log(),abs()` and so on
- Some functions treat the vector as a whole
  - `length(),sum(),mean(),median(),sd(),var(),min(),max()`
- `summary()` tells several things about a vector at once (but doesn't make much sense with vectors as short as the ones in this example)

Note that the result of the calculation is a new vector. You can always save it to a new variable if you want to do something with the result later, like this:

```
In [ ]: distances <- steps*0.7
        tot_dist <- sum(distances)
        tot_dist
```

However, since the result of the calculation is a new vector, you can also use it directly where ever a vector is expected. So, the same result as above could have been obtained on a single line as well:

```
In [ ]: sum(steps*0.7)
```

It is up to the situation and your personal preference which way to do it. Go ahead and try both ways here to calculate the average hourly energy consumption over the week, that is, total energy consumed over the week divided by the number of hours in a week.

```
In [ ]:
```

(Have you remembered to make a checkpoint? Maybe now would be a good time to make another?)

Next we have to talk about indices. As we have seen, the elements of the vector reside in a fixed order (and not in a random pile, for example) so that you can talk about the first element, the second element etc. You can also extract, or access, only certain elements of a vector, using square brackets and the index of the element, for example like this:

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

The first element has the index 1, the second 2, etc. It is also possible to talk about a whole range of elements, using this notation:

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

Again, the result of such operation is a new vector. Therefore, they can be combined with calculations. Let's assume that the step and energy numbers here are from a full week from Monday to Sunday. Calculate here the average step amount and energy consumption for weekdays, and for weekend separately. Remember what you learned previously about assigning intermediate results to names/objects!

```
In [ ]:
```

You may have asked now, what if you wanted to extract something like Monday, Wednesday and Friday, instead of a range? If this never occured to you, feel free to skip this part and continue from logical vectors. It's not a difficult idea but you can just as well do without. In essence, what you're supposed to put inside the brackets when you are extracting elements by their index, is an index *vector*. A single index is a vector (of length one). The : operator also creates a vector. So, you are fine as long as you put a vector of indices inside the brackets. Like this:

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

Note that this is something very different, and gives you an error (because `1,3,5` is not a vector, while `c(1,3,5)` is):

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

(In case you are wondering, this last bit would make sense if `steps` were a 3-dimensional array and not a 1-dimensional array or in other words, a vector.) Ok, now onwards to logical vectors.

1.5 Logical vectors

Besides numeric, vectors come in another flavor: logical. Another name for these is Boolean values. They are pretty much like numbers, but there are only two possible values: `TRUE` and `FALSE`. They come up as results from logical operations, such as comparing values to each other. Here we are told, element by element, whether the number of steps taken was above 1000:

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

The possible comparison are

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

and the logical operations are - 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`.

If you have never played with logical values before, worry not. For basic usage there are essentially just two situations were you need them. One is to use a comparison, such as above to extract a subset of data (so far, that means a part of a vector). For example, here we calculate the average of the steps taken and the energy consumed only on the days that the amount was above 1000 (presumably, on days that the bracelet was worn all day):

```
In [ ]: allday <- steps > 1000
        fullsteps <- steps[allday]
        fullkcal <- kcal[allday]
        mean(fullsteps)
```

(If you feel like it, see if you can again do this on just one line!) In fact, even this is a bit more complicated than is usually really needed, but the idea is clearly shown here. And we'll get back to it. The second case where logical values are needed, is to control the behaviour of certain functions. We'll try that soon in the context of missing values. First, let's just play a bit with what we have learned so far.

We've calculated before the average energy consumption per step, but we did it a bit wrong: a person consumes energy to run the body even if she laid in bed all day. This is called the basal metabolic rate. One day was marked with 0 steps, while there was a positive energy consumption on that day as well. That positive kcal value is the bracelet's estimate of the wearer's basal metabolic rate, based on height, weight etc. Let's calculate the average energy consumption per step again, this time based only on the part that was above the basal metabolic rate. Let's do it step by step. Here you should write the correct code at the points marked with ??.

```
In [ ]: # Here you should write the correct code at the points marked with ??

        # First we need to find the bmr value:
        # it's the element that coincides with the 'TRUE' value of the
        # logical vector "steps is equal to 0":
        steps0 <- ??
        bmr  <- kcal[??]
        bmr
        # You should see the bmr value come out as 1341
```

```
In [ ]: # Next we subtract the bmr value from the kcal values
        abovebmr <- ??
        abovebmr
        # You should now see smaller kcal values, with 0 as the second element
```

```
In [ ]: # Finally, we get to do the energy consumed per step calculation again
        kcalperstep <- ??
        kcalperstep

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

There! This makes more sense. The first day is still a bit off, but the other weekdays are similar with each other, and so are the weekend days, which is consistent with the wearer's workpattern. The `NaN` value will be explained in the next section. Meanwhile, if this approach felt unnecessarily complicated it is because it was just that. We could have just done the same calculation by typing in the number 1341. The point here was to give you a feeling of what it is like to do things in the abstract, programming style.

1.6 Missing values

In the `steps` vector of the previous examples the first two values, 106 and 0, were probably due to the bracelet not being worn the whole day or at all, and do not reflect the true amount. In other words, we don't actually know the step number: those data point are missing. The R language provides an explicit way of saying this, the `NA` symbol ("not available"). Here is the steps vector with missing values explicitly stated:

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

Another common way of expressing missingness is some specific numeric value that's supposed to be apparent but this causes problems. An actual number is always going to be treated as an actual number in calculations, such as total sums etc, skewing results unless they are filtered out manually (which is easy to forget). An explicit missing value will behave properly and can be dealt with automatically. For example:

```
In [ ]: sum(steps_na)
```

The sum of a vector containing NAs is also NA. This makes sense: if you don't know what number to add, you also don't know what the result will be. On the other hand, you might be interested to see the sum of the numbers that are there. This is easily done:

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

Here an additional *parameter* was given to the sum-function, telling it to skip the missing values ("remove NAs"). This is the other case where you commonly encounter logical values: the default behaviour of sum (and many other functions) is to include missing values in the calculation and it can be changed by setting the parameter `na.rm` to `TRUE`. We will learn more about functions and parameters later in this notebook.

One more thing about NAs: if you need to know which values are missing, a direct logical comparison with NA will not work. You need to use a *testing* function `is.na()` instead. See here (and think for a moment what the first result means!):

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

In the previous example there was a related symbol, `NaN` which stands for "not a number". This is another kind of missingness that resulted from the calculation 0/0 which has no well defined mathematical meaning (not even infinity, which in turn has its own symbol `Inf` which we also saw earlier). The difference between NA and NaN can be easily ignored for now.

1.7 Character vectors and factors

There's one more flavor of vectors: character. The elements of character vectors are pieces of text, such as here:

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

Note that the pieces of text have to be enclosed in quotation marks. Single ' and double " both work the same.

There's not much you can do with character vectors as such. They often show up in data representing group membership, or in other words, a categorical variable. However, R has a special object type reserved exactly for that: *factors*. The difference between a character vector and a factor is a bit technical, and in many situations, a character vector is quietly and implicitly changed in to a factor when needed, so it's that more difficult to make the distinction. However, the defining features of a factor is its *levels*: the specific collection of the different values it might get. In contrast, the elements of character vector could be thought to be anything, potentially distinct values for every element. To clarify, here the character vector version of the treatment groups is described using the `summary` function:

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

Here the character vector is changed in to a factor and described again:

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

In the first case, nothing is told about the contents beyond the length and type, in the second, a useful summary of the variable is given: it has three categories (levels) each with 2 observations. Here are two other functions about the levels of a factor:

```
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.

1.8 Data frames

The examples so far have dealt with single 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 this was not explicitly stated anywhere. In the following the same data are put in a *data frame*. Data frames are where all data is usually kept and manipulated and therefore they can be considered **the most important data structure in R**.

For clarity, let's empty the workspace first:

```
In [ ]: ls()
        # you probably see the names of all the objects from before, including step
```

```
In [ ]: # this command removes everything:
        rm(list=ls())

        # now the same listing command from before gives no output:
        ls()
```

In other words, there should now be no objects in the workspace. We'll go right ahead and create one new one:

```
In [ ]: bracelet <- data.frame(day=c("Monday","Tuesday","Wednesday","Thursday","Fri
                               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 dataset is now visually represented 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 of those variables that go together. On the other hand, the columns of a data frame are still vectors, or factors, and you can do anything with them that you would do with vectors, or factors. The only difference is that you have to refer to the data frame where it comes from too. See here:

```
In [ ]: # this will give an error message:
        steps
```

The error message should not come as a surprise: assuming you ran the previous two code cells just before, it should be clear to you that the only object in the workspace is now `bracelet`, and that there is nothing called `steps`. A small change will make a difference:

```
In [ ]: bracelet$steps
```

Using the name of the data frame and the `$` character it is possible to talk about the column vector called steps of the data frame called bracelet. Armed with this knowledge, please try and see if you can redo the calculations related to steps and kcal that were done before in the next code cell.

Note! This should be very educating and perhaps include a few pitfalls. If you get error messages, or surprising behavior, 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 [ ]:
```

1.9 Working with data frames

The next step in learning data analysis with R would naturally be how to import your own data to work with. We will get to that but it requires knowledge of a few more concepts: functions, working directories, csv files... So before we do that, let's play around a bit with some example data sets that come with R itself.

All of the available example data sets are taken in to use with the function called `data`. First, let's look at a classic example, the iris data set:

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

So now we have a new data frame in the workspace, called iris. Its summary shows us that there are 5 variables, four numeric ones, and one that is a 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 like 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
```

You might also consider searching the Wikipedia page of this data set to get an idea of what kind of flowers these are, and what the petal and sepal leaves are.

It is said that when doing data analysis, at least 80% of the time is spent on cleaning, rearranging and otherwise preparing the data, and only 20% on the actual analysis. R is great for that, particularly when combined with a good user interface (RStudio) and some dedicated packages. 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 in to 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 you try the same with petal leaves:

```
In [ ]:
```

Another simple trick we look at is taking subsets of the data frame. Recall how we used the 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
        ls()
```

This created a new data frame, which consists of the rows 1 to 5 and columns 1 to 4 of the iris. So, the indices before the comma refer to rows, the indices after the comma refer to columns. Leaving the other index out completely means "all of them". So, this means that in addition to the `$` notation from before, you can extract a single column vector also 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, and in fact makes even more sense here. 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 state the data frame you want to subset, and then you give 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 are known without the $ notation.

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 nice you should also try splitting the data frame in three single-species parts with `subset`. This solution was actually started in the previous code cell already.

(Note that neither of these two ways is in fact viable in the real world. In a real situation one would use the `aggregate` function from base R, or `summarise` from the `dplyr` package. But it's too early to cover those.)

```
In [ ]: 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.

1.10 Understanding function calls and documentation

Everything in R is accomplished by using functions. In fact, learning to do data analysis using R after the basics is mostly about learning which functions out of the millions available will do the tasks you need to do. That is a completely different story from what we are looking at now. Once you know which function you want to use though, you can look at its documentation to see how exactly it is 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` and remember again that you can detach the page as its own tab.

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

The documentation pages have a common structure, although not all have every section present. First there is **Description**, a short explanation of what the function (or functions, if several are described in the same page) does. Then there are the parts called **Usage** and **Arguments**, which together list all the possible arguments the function could take and their default values. The arguments are also explained in more detail: most importantly, the expected type of the arguments (such as numeric, logical, single value, a vector...). After this there is often **Details**, a longer and hopefully enlightening explanation of how the function works and is intended to be used. Here though, `mean` is assumed to be simple enough to not need further explanation. Next one is usually **Value**, explaining what kind of output the function will produce. At the end, there is usually a list of related functions under **See also** and finally **Examples**, which are often very educating.

The arguments are always named in the Usage list, but when calling the function you don't have to use the names, as long as the order you give the values matches the order listed in Usage. If you want to skip giving some (and use their default values), or give them in some other order, you do need to use the names. A common custom is to give the first argument unnamed, and the rest with names.

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

- a Finnish Lotto (lottery) ticket: 7 randomly drawn numbers out of 40
- a vector of 0's and 1's, representing 10 coin tosses
  - as an extra: make a factor with levels "heads" and "tails" instead, if you feel like it. It is more appropriate but requires a bit more tweaking
- a sample of 5 random sepal length measurements from the iris data set. This is a bit trickier than the previous 2 parts: first produce a vector of random indices using `sample`, then use that to pick the measurements (and yes, once more, a function for picking random samples out of data frames exists, this is for educational purposes!)

```
In [ ]:
```

1.11 Reading in data from CSV files

Data can come in many formats and R is probably able to read all of them with some effort and the correct package. Here, we will only consider one: CSV files. Feel free to have a look now to see what they are like: https://en.wikipedia.org/wiki/Comma-separated_values. The short version, suitable for current purposes, is that a CSV file is a text file, containing data in a tabular format much like the data frame you would like it to become. The first row is usually a header row, which contains the names for the columns (variables), and the rest of the rows will have the actual data. Columns are separated with the comma character. Programs such as Excel, SPSS, SAS, and many other sources are able to export their data in this format. Here's an example of what such data looks like (in fact, these are the first few rows of a CSV file called weather-kumpula.csv we will soon read):

```
ts,year,month,day,dp,rmm,wdir,ws,t,rh,p
2014-01-01,2014,1,1,0.7526754690757457,1.640000000000001,158.0,4.61702571230021,3.0
2014-01-02,2014,1,2,-2.2381944444444573,0.6000000000000004,118.0,3.627777777777778,
2014-01-03,2014,1,3,-1.4593055555555527,0.5000000000000002,141.0,3.328194444444438
2014-01-04,2014,1,4,0.3417361111111121,3.6999999999999966,219.0,4.288055555555555,1.
```

The function for reading in data like this is called `read.csv`. If you go ahead and open its documentation…

```
In [ ]: ?read.csv
```

… you will find out - if you read beyond the dauntingly long list of arguments and other information - that `read.csv` is an alias for a function called `read.table`. This function can in fact read many more kinds of tabular data than just CSV files but you can safely ignore most of this information now. If your data is in CSV format, `read.csv` will get the job done. One exception: some locales (Finnish, for example) will use comma instead of dot as the decimal separator, and have to use something else as the column separator. Therefore there exists a non-standard version of CSV with semicolon ; as the column separator. For this case, `read.csv2` will work.

All you really need to tell the `read.csv` function is where to find the data file. It is done by giving the name of the file, as a character string, like this: read.csv("weather-kumpula.csv") This works if the file is in the current *working directory* of your R session. To see what the current working directory is, use this command:

```
In [ ]: getwd()
```

(This notebook session probably says /home/jovyan/work/R-for-beginners.) If the file is not in that directory, 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 \ character, you have to use / or double \\
- use interactive file choosing: `read.csv(file.choose())`
  - note: this does not work on this notebook!

(Further note: RStudio has an interactive import wizard, which makes life a lot easier. But this way still works on RStudio as well.)

This notebook is set so that you have the weather-kumpula.csv available in a directory called data under the working directory, so we are ready to read the weather data in:

```
In [ ]: weather  <- read.csv("data/weather-kumpula.csv")
```

And that's it. If you go back to the documentation of the read.csv, and see the Value section, it tells you that this function produces a data frame. In order to be able to actually use the result, it needs to be assigned to a name. Just like any other result.

And now you should both be able to 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. You just need to upload it to the notebook first (from the Home page). If not, let's get familiar with the weather data.

This data contains the 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 in Celsius degrees
- rain in millimeters
- wind direction in degrees
- wind speed
- temperature in Celsius degrees
- relative humidity
- air pressure in millibars

You can now use the knowledge you have, and these new hints, to get a handle of this data set. For example,

- Use the function `dim` on the whole data set. What does this function do? What does it tell you?

- Use `summary` on the whole data frame get an overall feeling of the values, such as, what is the range of temperature over the year? Are there any missing values? (Answer: yes)
- 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 the missing values happen. (Answer: 3) (Which days?)
- How much did it rain in September, October and November in total? Remember the `%in%` operator mentioned back in the section on logical values. (Answer: 195.48 mm)

```
In [ ]:
```

1.12 A glimpse at actual analysis; formulas

You are now ready to start actually working with real data: you have learned about vectors, factors and data frames, are able to read in data, and understand the general logic of how R works. This is as far as this introduction was meant to go. From now on, learning how to do things in R involves learning which functions to use to which tasks and how exactly that happens, and since there are so many possible directions to go, this is a good place to draw the line - almost. There's one more concept that is in general use in R: a formula. Here's a first example:

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

`plot` is a general purpose plotting function, and what we have given it as arguments is one *formula* describing the kind of plot we want, and a data frame where to find the needed variables. (If at this point you feel like checking the documentation for the plot function, good for you! Unfortunately, because of its generality, the documentation is rather unhelpful.) The formula is the part in the function call with the ~ character in the middle: it has a left hand side and a right hand side. Its purpose is best revealed if you think of the ~ character as the word "by". Here, as stated, we plotted Sepal.Length "by" Sepal.Width. When both are numeric vectors, `plot` makes a scatter plot. Here's another example:

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

Same formula, different function: this time we fitted a linear model, corresponding to the scatter plot before. On the left hand side of the formula there is the response (dependent) variable, and on the right hand side there is 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 [ ]: summary(lm(Sepal.Length~Sepal.Width, data=iris))
```

(Note: while the output of the `lm` function seemed to be just a bit of text, it is actually a *list* object, just returned silently, and the text output displayed instead. Lists are a data type we have skipped talking about, but you can go to the documenation of `lm` to get an idea of what the result list actually is. Also note that data frames are in fact a special case of lists.)

Here is one more example of where you might use formulas, a classical T-test:

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

There's one more argument in there in addition to the formula and the data frame: a subset argument. Functions that take a formula as an argument usually also accept a subset argument, which is used to indicate that only a subset of the data frame is to be used, namely, the part for which the given logical vector is `TRUE`. It works exactly like the `subset` function we encountered earlier, but just skips creating an explicit subset data frame. In the T-test function here, we took one species out of the data because the T-test makes no sense with three groups, only with two.

Formulas are a case where the difference between factors and vectors becomes rather clear. Consider this example data set:

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

As described in the documentation, this data set contains the results from an experiment concerning the effect of vitamin C on the tooth growth of guinea pigs. There are two supplement methods and three dose levels, a total of 6 groups with 10 animals in each. The `supp` variable is a factor, but `dose` is numeric. However, it is a valid question to ask, whether the doses should be looked at as actual numeric values, or as group labels. The difference should become apparent if we create a factor version of it, and fit models and plot graphs with both.

```
In [ ]: ToothGrowth$dose.factor <- factor(ToothGrowth$dose)
        summary(lm(len~dose,data=ToothGrowth))
        summary(lm(len~dose.factor,data=ToothGrowth))
        plot(len~dose,data=ToothGrowth)
        plot(len~dose.factor,data=ToothGrowth)
```

One final case where formulas could come in to use: earlier, when calculating groupwise means with the iris dataset, it was mentioned in a real situation those would not be calculated by manually splitting the vector. Instead one would use for example the function `aggregate`. That can also be used with a formula. Try now if you can figure out how to do that:

```
In [ ]:
```

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

- Make boxplots of monthly temperatures. Which month seems to be most variable?
- Is there any relationship between air pressure and wind speed?
- How about between the air pressure and temperature? Is the answer different if winter and summer are looked at separately?

```
In [ ]:
```

1.13 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. So far in this notebook we have been able to get by with things that come with a base installation and a fresh untouched R session. Only the most basic packages are automatically enabled, others have to be brought in to use with the command such as

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

Here, `spatial` is the name of a package containing functions for spatial statistics methods. It is one of the handful of recommended packages that come with a default R installation. More (roughly 12 000 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 same `library` command.

The abundance of packages is overwhelming. To avoid getting lost in the jungle you can use the traditional methods of asking colleagues and searching the internet for advice. In addition, one good source to keep in mind are the Task Views on the CRAN home page https://cran.r-project.org/. These are curated lists of packages related to a certain field of study, with short explanations of which functions do which tasks.

1.14 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.

And of course, consider installing R on your own computer. Also consider installing RStudio as well. Note that they are two different things: R is provided by CRAN and comes with its own user interface. RStudio is an alternative user interface provided by the RStudio company. Only the commercial version comes with a price tag, you can download the open source version for free.

Learning how to use RStudio efficiently is a separate learning objective in addition to learning how to read and write R code, and is in fact intertwined with the learning how to use a collection of packages called **tidyverse** also provided by the RStudio company. See www.tidyverse.org for more information and the book 'R for Data Science' linked there. These packages have an underlying philosophy that is slightly different from the base R way, 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!