COMPUTER LAB 1

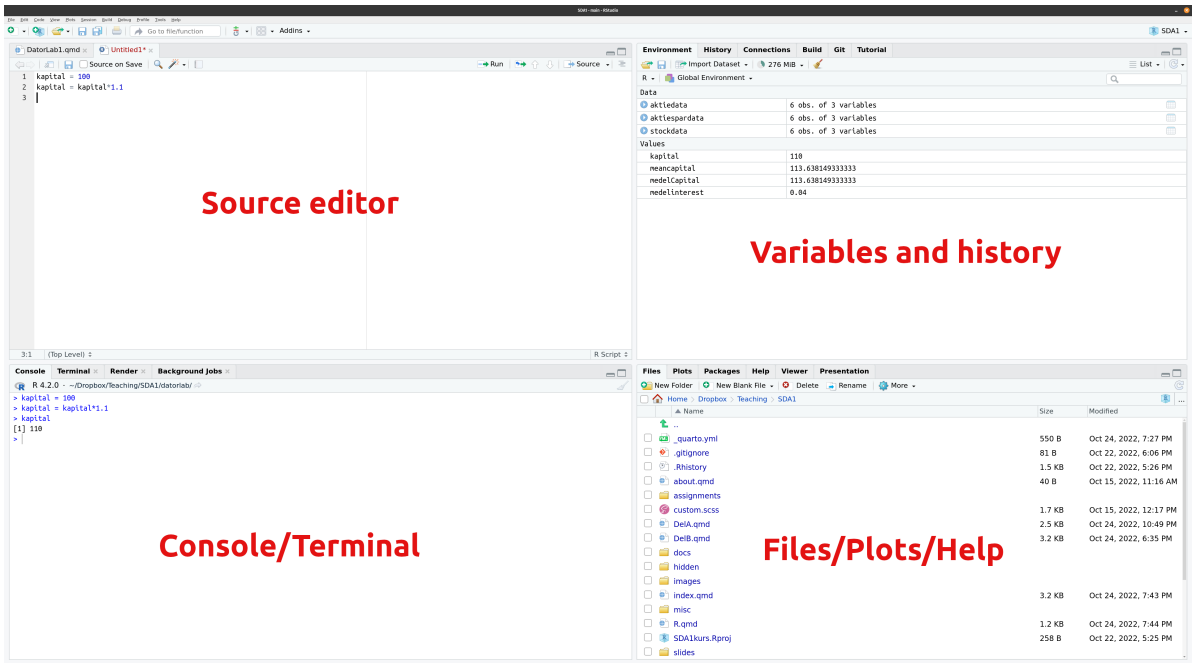
Table of contents

In this first computer lab, you will become familiar with the programming language **R** and its working environment **RStudio**.

1 Using R as a calculator

> **Working with R in your own computer**
>
> If you are not sitting in a computer room on Campus, you must first install R and then RStudio. See the video installing R and Rstudio for instructions.

To launch RStudio, simply locate the program on your computer and click the launch icon. When RStudio starts up, it will look like this (it may look slightly different on different computers):



A good way to get used to R is to use it as a kind of calculator. In the **Console** window in RStudio (usually at the bottom left) you can write different types of **commands** that are sent to R for calculation:

```
2 + 2
```

```
[1] 4
```

The character > is called the **command prompt** (or just **prompt**) and is R's way of saying that it is waiting for a new command to be typed (at the flashing dash).

Multiplication and division

R uses `*` for multiplication and `/` for division. Powers (raised to) are written with the sign `^` , so `3^2` is `9`.

EXERCISE

Try typing 2+2 after the bottom > character in **Console** and then press the key Enter. R should respond **return** the number 4.

EXERCISE

Calculate 3^2 (3 to the power of 2) in the **Console**.

EXERCISE

Type the number (2+3)/(2+5) into the **Console** and check that R responds with 0.7142857.

EXERCISE

You buy shares for SEK 100. The return in the first year is 10%. Use R to calculate the value of your share capital after your first year as a share saver, i.e. enter 100*1.1 and get R return 110.

2 Using variables in R

You are now in your second year as a share saver. The return in year 2 is 6%. How much share capital do you have after year 2? We can calculate this through 100*1.1*1.06 in Console and get the answer 116.6 SEK. But is there any way to reuse our previous calculation 100*1.1 = 110 SEK so we only need to multiply this number by the increase of 1.06 for year 2?

We can solve this by saving our first calculation as a variable. We can give this variable (almost) any name we want. Let's call it `capital` and start by setting the value of the variable `capital` to 100, the initial capital. We write

```
capital <- 100
```

or

```
capital = 100
```

in the Console. We can then test that R now actually remembers that capital is 100 by just writing the variable's name followed by Enter in the command prompt in the Console:

```
capital = 100
capital
```

```
[1] 100
```

R simply prints the value (100) that we assigned to the variable `capital`.

We can recall the value 100 from the variable `capital` at any time. But if you shut down RStudio and then restart the program, R no longer remembers the value of `capital`. R and RStudio only remember the variable within a session, i.e. until you exit RStudio. If you want to save data between different sessions, you have to save the variables on the computer's disk (or on some storage on the internet). More on this later.

We can now calculate the capital in year 1 by multiplying the variable `capital` by the number 1.1

```
capital = 100
capital
```

```
[1] 100
```

```
capital*1.1
```

```
[1] 110
```

We can also **overwrite** the value in the variable `capital` with a new value. We may want the variable `capital` to always contain the value of the share capital that we have right now. First, let's change the value of `capital` to the value after year 1:

```
capital = 100
capital = capital*1.1
capital
```

```
[1] 110
```

Note especially the line `capital = capital*1.1`. First R takes the value of `capital` (100) and multiplies it by 1.1, the result of this will now be called `capital`. Thus, the value of the variable `capital` is now 110.

The beauty of this is that we can now continue to change the capital variable after another year has passed, i.e. the value of your share capital after year 2:

```
capital = 100
capital = capital*1.1
capital
```

```
[1] 110
```

```
capital = capital*1.06
capital
```

```
[1] 116.6
```

The variable `capital` is now 116.6 and you are ready for your upcoming return in year 3.

EXERCISE

The return in year 3 was unfortunately minus 4%. Update the variable `capital` above so that it shows that the value of the capital after year 3 is 111,936 SEK. Note that a decrease of 4% means that we must multiply by 0.96 (1-0.04). Multiplication by numbers less than 1 leads to a reduction of the capital.

3 Using the Editor (Source) to save code

Writing commands directly in the Console has a couple of disadvantages. First, R does not remember the commands we have written in a previous session (before we shut down RStudio). Every time we start up RStudio, we have to rewrite our commands if we want to continue our calculations where we left off last time. Second, working in the Console is quite inconvenient if we are writing long codes as it is not easy to see in an organized way what we have written already. Third, correcting errors in the console is not very convenient either.

We would like to write all our commands in a file that we can save on the computer's hard drive and then just re-run all the commands from a previous Session. The **Source Editor** in the upper left part of RStudio is used for exactly this.

If we click on the **File** menu and then under the **New File** menu and finally on **R Script** , an empty text file called Untitled1 or something similar is opened in the **Editor**. Here we can enter the commands that we want to save for future sessions. We can, for example, enter our calculations of the share savings:

```
capital = 100
capital = capital*1.1
capital = capital*1.06
capital = capital*0.96
```

When working with the code, you often want to run one command at a time. This can be done by placing the cursor (the flashing line) on the line that you want to run (execute) and then clicking **Run** the button in the upper corner of the editor. You don't need to highlight the code on the line, just place the cursor somewhere on the line. The keyboard shortcut `Ctrl + Enter` does the same thing.

You can also run multiple lines of code at once by selecting the lines and pressing the **Run** button.

EXERCISE

Run the command on line 1 of your code by using the Run button. Examine what value the variable capital gets in the Console. Repeat this for the remaining rows of the R-script.

Errors

Run the following line of code (note the capital "C"):

```
Capital
```

What did you get? The object we defined before was called `capital` not `Capital`. R is case sensitive, which means that the object `Capital` does not exist. That is what the error message is telling us.
You will get thousands of errors when Working with R. That is completely normal. Every time you get an error, it comes with a message that briefly describes what is the reason for the error. Often it is easy to fix the problem and continue working.

We can save our work by clicking on the **File** menu and then on **Save** and then navigate to the folder where we want to save the file. Name the file with a name that speaks of what the file contains (for example **stock.R**) . Click on Save. The file will automatically have the file extension .R, so RStudio knows it is a file with R commands.

> **Organize yourself with file folders**
>
> It is important to keep your files well organized on your computer and to be able to tell R where your files are so that R can read them. We recommend that you create a folder for each course you take. Do not save all files on the computer's desktop or in the Downloads folder or similar. Go to your computer's file manager and create a new folder.

Do not use å, ä or ö when naning files or folders. There is a big risk that you will eventually have some problems.

> **EXERCISE**
>
> Create a new folder on your computer and save your code there.

> **Note**
>
> If you create a folder on a computer in the computer room, it will unfortunately not be there the next time you log on to another computer, or maybe not even on the same computer. You have to save your files online by yourself (e.g. with a service such as Dropbox or OneDrive) or another way is to always carry a USB stick with you and create a new folder on it.

4 Setting the working directory in R

RStudio works in a directory that is chosen by default (typically *Documents*) this means that whenever we try to save something we have produced in our session or we try to read some file into RStudio, it will save or search for the file in this working directory. However, as we point out before, it is very important to have our files well organized, so it is good practice to change the working directory that has been chosen by default to a different directory in our computer.

There are at least two ways for doing this (of which we only recommend the first if you are an absolute beginner. Pretty soon you should learn to use the second way, which is smoother in the long run):

- Click on **session** at the top menu, then **set working directory**, then **Choose directory** and then navigate to the directory that you want to set as your working directory. (I would recommend this one for beginners.)
- In order to avoid clicking through menus all the time, it is practical to change the working directory at the beginning of the code file you are working with. Then you can just run that code file and the working directory is set automatically. To this end we use the command **setwd**. Here, however, we need to know the path to the folder,

i.e. the computer's way of finding the new folder. The way to write paths differs between Windows and Mac.

- On a Mac, we write the command: `setwd('/Users/username/Documents/SDA1')` where `username` should be replaced with your username on your Mac (the name that comes up when you log in to the computer).
- On a Windows computer, we type the command `setwd('path')`. Windows paths are usually written with a backslash (\), In RStudio you have to use a slash (/).

Note the apostrophe `'` around the file path in the setwd command.

EXERCISE

Set the folder you created in 3.2 as your working directory in RStudio. Check that you were successful by typing the command **getwd()** in the Console, which should print the path to your new folder.

5 Reading datasets in R

In Athena you can find the files `stock.csv` and `stock.xlsx`. Both files contain the same information, the only difference between them is that one has been saved as a comma separated values (csv) file and the another one is an Excel file. It is a dataset with three variables `year`, `capital` and 'return' which, for each year between 2018 and 2023, it indicates the returns and the capital at the end of the year for an investment of 100 in 2018.

EXERCISE

Download the file into the folder you created in Exercise 3.2.

5.1 Reading csv files in R

Now that you have your csv file `stock.csv` and R directed to the same folder, simply try the following line of code in order to read the dataset into R:

```
stockdata = read.csv("stock.csv")
```

This means that we are reading the file `stock.csv` by making use of the function `read.csv()` and we are saving the dataset as the object `stockdata` in R.

EXERCISE

Read the file `stock.csv` in R.

5.2 Reading xlsx files in R

Unlike the case for csv files, R does not include by default a function to read Excel files. Therefore, in order to read the Excel file, we need to run a couple of extra steps.

1. R comes with a number of basic commands pre-installed. For example, we have already used the function `setwd()` to tell R which folder is our working directory. But many commands/functions in R must be loaded via so-called R packages. There are R packages for almost everything you want to do: load data, do different types of statistical analysis, etc. In particular, the package `openxlsx` allows for loading Excel files into R. To install the package we run the command

```
install.packages('openxlsx')
```

either by typing it in a code file and press **Run** or by typing it directly into the console. After that, R will print a lot of mumbo-jumbo in the Console describing the installation. Packages may take a minute or more to install. If no errors occur, the installation message usually ends with something like this:

```
DONE (openxlsx)

The downloaded source packages are in
```

followed by some cryptic path to the location on your computer where the package has been installed.

2. Load the R package `openxlsx` using

```
library(openxlsx)
```

In this way we load all the functions of the package openxlsx into R's working memory. Only after this command we can use the functions of the package.

3. Read the Excel file using the function `read.xlsx`:

```
stockdata = read.xlsx('stock.xlsx')
```

There are a few things to clear up here. First, installation of packages only needs to be done once on your computer. You do not need to reinstall them the next time you restart RStudio. Loading packages, on the other hand, has to be done every time you start a new session in R. In many senses, packages in R are like apps in your phone. For instance, you need install them only once. Once installed you will need to open them every time you want to use them. Just like apps in your phone, packages add new functions to R and just like apps in your phone, there may be several packages for doing the same thing.

Second, note that we need apostrophes around the package name when we install, but not when we load the package.

When you have loaded a dataset, it is good to take a quick look at the loaded table to see that it was loaded correctly. Just as we did with variable capital to see its value (e.g. 100) we can write stockdata in Console to see its value (which is a table after all). If you have a table with many rows, it quickly becomes pointless to print all the rows. The command `head()` is then practical, which only prints the first 6 lines of the table.

```
#Show the first six rows
head(stockdata)
```

Comments

In the chunk of code above we wrote "#Show the first six rows". All text that comes after the character `#` is a so-called **comment** in R. Comments are ignored by R and are therefore a good way to write small things that make the code easier to read to humans. You must have a `#` sign in front of each line that you want to "comment out".

EXERCISE

Read the file `stock.xlsx` in R.

EXERCISE

Change the value of the variable `returns` in 2023 to 0.05 in the stock data table by typing

```
stockdata[6,3] = 0.05
```

The command above indicates to R that the cell in the sixth row and third column of the dataset `stockdata` will take the value 0.05. Try typing stockdata in the Console to verify that the value was actually changed.

EXERCISE

Save the changed stock dataset to disk by typing the following command in the Console:

```
save(stockdata, file="stockdata2.Rdata")
```

which saves the object `stockdata` to the file `stockdata2.Rdata` in your working directory. The file format `Rdata` is R's own file format. In a later session, you can load the stock data easily through the command

```
load("stockdata2.Rdata")
```

which will read the table `stockdata` into R.

6 Analyze the data

Once you have read the data into memory, it is time to analyze it. Let's make a simple graph of the variable capital over the years (year):

```
plot(capital ~ year, data = stockdata)
```

Here we have used the function `plot()` with two arguments. The argument `capital ~ year` tells to `plot()` to graph the variable `capital` on the y-axis and the variable `year` on the x-axis. We say we "plot capital against year". But `capital` and `year` are not actually variables in R's working memory. There is only one table (dataframe) named `stockdata` with columns named `capital` and `year`. So we need to specify that we want to get the variables capital and year from the dataset `stockdata`, which is exactly what happens in the second argument `data = stockdata`. The character `~` is usually read as a tilde.

If we want to calculate the mean value of a variable, we can use the function `mean()`. But if we want to calculate the mean value of the variable `capital` with the command `mean(capital)`, we get the problem that R does not have a variable `capital` in memory; `capital` is just a column in the dataset `stockdata`. The solution is to use the `$` sign to extract the column capital as a separate variable as follows

```
capital <- mean(stockdata$capital)
capital
```

The previous code thus asks R to calculate the `mean` of the variable `capital` in the dataset `stockdata`. By explicitly picking out columns in this way, we can make the graph with a slightly different command:

```
plot(x = stockdata$year, y = stockdata$capital)
```

where we no longer have to use the argument `data = stockdata` to tell where these variables come from. But the previous command might be still easier to read.

> **EXERCISE**
>
> Make a graph where you plot the variable capital against year.

Here is the entire code in one paragraph:

```
library(openxlsx)
stockdata <- read.xlsx('stock.xlsx', sheet = 1)    #import data
stockdata <- data.frame(stockdata)
meancapital <- mean(stockdata$capital)
plot(capital ~ year, data = stockdata)
```

7 Some extra tips

- If you press the 'up arrow' key in the Console, you can step back to old commands that you have typed.
- If you want to clean up the Console, you can type `Ctrl + l` (small L) to clear the Console of text. Your variables remain in working memory.
- The `History` tab at the top right of RStudio shows all previous commands.
- The `Environment` tab at the top right of RStudio gives you information about the datasets and variables that R has in the working memory. Try defining a new variable `a = 100` in the Console and see how it appears in the `Environment` tab. Try clicking on the blue arrow in front of stock data, you will see the columns of that dataset.
- The `Files` tab at the bottom right of RStudio is a built-in file manager. It shows the files in your working directory and you can also manage files there (delete, rename, etc.).
- In the `Help` tab, you can search for help on various commands. Try entering `mean`. You can also get help by writing, for example, `?mean` in the Console.

8 Data structures

R can save data in different data structures. We previously defined the variable `capital` and assigned the value 100 to this variable through the command `capital <- 100`. Often you want to save more than one number in the same data structure. One way is to create a `vector` with the command `c()`

```
capital_all_years = c(100, 110, 116.6, 111.936, 117.532, 125.760)
```

The new variable `capital_all_years` thus contains the capital for all years. But if we would like to 'pick out' the value of the capital at the third year? Then you can index the vector through brackets `[]`, like this:

```
capital_all_years[3]
```

```
[1] 116.6
```

You can also extract several values at the same time, e.g. years 3 and 5, with the command

```
capital_all_years[c(3,5)]
```

```
[1] 116.600 117.532
```

Note that we indexed with a vector, i.e. we used the `c()` command because we want to tell R to pick more than one value.

We have already seen one of R's most important data types: a so-called `data frame`. A `data frame` is a table with information about what the columns are called, and sometimes also what the rows are called. An example is our stockdata which is a data frame with three columns: `year`, `capital` and `stock`. We can see that it is a dataframe by using the command `class()`:

```
class(stockdata)
```

We have actually already used the idea of extracting values by indexing into data frames before when we wrote `stockdata[6,3]` to extract the value of row 6 and column 3 of the dataframe. And we saw that we could also change the values in the table by assignment: `stockdata[6,3] = 0.05`. The same thing can be done with a vector, for example the command `capital_all_years[5] = 200` makes us change the capital in year 5 to 200 in our vector which contains the capital of all years.

There is another kind of table that can be good to know in R, a so-called `matrix`. Matrices are like data frames, but do not contain information about the names of the columns. On the other hand, you can do mathematical calculations with matrices, but that is not something we do in this course. The reason we mention matrices here is that sometimes you have to convert a matrix into a dataframe or vice versa. Some R functions will only work with dataframes and will return errors if you try to get it to work with a matrix. Let us create a matrix that we call `A`:

```
A = matrix(c(11, 2, 4, 5, 62, 3),nrow = 3, ncol = 2)
A
```

```
     [,1] [,2]
[1,]   11    5
[2,]    2   62
[3,]    4    3
```

which is a matrix (table) with 3 rows and 2 columns (which we create from a vector of 6 numbers). If we use the `class()` command, we see that A is indeed a matrix:

```
class(A)
```

```
[1] "matrix" "array"
```

If we want to turn a matrix into a dataframe, we use the command

```
B = as.data.frame(A)
B
```

```
  V1 V2
1 11  5
2  2 62
3  4  3
```

where R determines that the columns should be named `V1` and `V2` (a dataframe must have names for the columns in the table). `B` is thus now a dataframe, which can be tested by:

```
class(B)
```

```
[1] "data.frame"
```

Another important data type we want to mention is a so-called `string`, which is called character in R and is a variable that contains text. The value of a string variable is written inside quotation marks:

```
my.name <- "mattias"
class(my.name)
```

```
[1] "character"
```

You can also use simple apostrophes around the value of the string i.e. `my.name = 'mattias'`.

For example, if we use `names()` the command on our stockdata dataframe, we get a vector of strings (a vector because there is more than one column name in stockdata):

```
names(stockdata)
```

Another data type in R is `list`. A `list` is a vector of different data structures. A normal vector must contain data of the same type , for example a vector of numbers `x = c(1,4,5)` or a vector of strings (text): `c("mattias","adam","elma")`. A list is more general and can contain data of various kinds:

```
my_list = list(a = 2, b = "hej", d = c(4,5,2))
my_list
```

```
$a
[1] 2

$b
[1] "hej"

$d
[1] 4 5 2
```

We can name the elements in the list whatever we want, here we called them, `a`, `b` and `d` (we want to avoid the letter `c` because it is the symbol for creating vectors). Note that the element `a` contains a single number, `b` a string, and `d` a vector of three numbers. In the printout, you also see the various list elements with a `$` sign in front. In order to select the list element, `d` for example, you can write

```
my_list$d
```

```
[1] 4 5 2
```

We have already used the symbol `$` before when we learned that you can extract a column from a data frame (e.g. `stockdata$capital`). A data frame is actually a kind of list.

Finally, we will introduce another important data type in R, namely, a `factor`. Try typing the command `str(stockdata)` into the Console. The command returns the names and data types of the variables, which is numeric for all our three columns `year`, `capital` and `returns` in the table. But table columns can consist of letters (strings) or categorical variables (e.g. man and woman). Categorical variables are called factor variables in R.

R has a number of built-in datasets, for example, the dataset `warpbreaks` that contains information on the number of errors (breaks) when weaving for two different types of yarn (wool) and three different tensions of the loom (tension). We start by assigning the dataset to the `yarndata` variable by typing `yarndata = warpbreaks` in the Console. Write `head(garndata)` for a quick look at the first observations. As you can see, `breaks` is a numeric variable (or rather a column in the table) but both `wool` and `tension` are categorical factor variables (`wool` can only be of yarn type A or B, and tension can be low (L), medium (M) or high (H). We can see this more precisely by typing `str(garndata)` in the Console:

```
yarndata = warpbreaks
str(yarndata)
```

```
'data.frame':   54 obs. of  3 variables:
 $ breaks : num  26 30 54 25 70 52 51 26 67 18 ...
 $ wool   : Factor w/ 2 levels "A","B": 1 1 1 1 1 1 1 1 1 1 ...
 $ tension: Factor w/ 3 levels "L","M","H": 1 1 1 1 1 1 1 1 1 2 ...
```

where we see that `wool` is a factor variable with two different values (levels), while `tension` has three levels. We also see that `breaks` is a numeric variable. We can also see this information in the upper right panel of R under the `Environment` tab by clicking on the small arrow on `yarndata`.

9 Functions

We've seen the idea of functions before: a function does something with data. In the world of mathematics a function f is a kind of machine that uses an input, does some math calculations, and returns an output. We usually write the function as $y = f(x)$.

A function in R is a similar thing: it takes one or more inputs and returns one or more outputs. It can be a pure mathematical function, such as the square root of a number, which in R is calculated by the function `sqrt`:

```
x = 4
y = sqrt(x)
y
```

```
[1] 2
```

but it can also be something more complicated, such as reading in data from an Excel file:

```
stockdata <- read.xlsx('stock.xlsx')
```

there `read.xlsx` is a function as that takes one input argument, `filename` (as a string!), its output is a dataframe with the data.

The beauty of programming languages is that you can create your own functions that do exactly the job you need. For example, if we want to create a function that calculates the interest-on-interest effect on a savings capital (with the same interest every year), we can write the following function:

```
capital_interest = function(capital, interest, years){
  newcapital = capital*interest^years
  return(newcapital)
}
```

A few things to note: - You always use the word function there to define a function. - The function's input arguments are written in parentheses. Which names you choose is completely free, but it must be the same variable name that you then use inside the so-called curly brackets {} that contain the actual code that does the work. - The function is assigned to a variable that we have given the name capital_interest, which is a name you choose entirely by yourself. - The code uses the interest-on-interest formula, i.e. the interest rate, stated as 1.1 if you get 10% interest, is increased in the number of years of savings. - The function ends with the word `return` and the variable you want the function to return as output.

If we then want to use the function to see how an investment of 100 SEK will develop over 20 years with 5% interest, we write:

```
capital_interest(100, 1.05, 20)
```

```
[1] 265.3298
```

If we examine what type of variable `capital_interest`is, it's actually a function:

```
class(capital_interest)
```

```
[1] "function"
```

When we wrote the variable `capital` in Console, it printed the value 100. What happens if we do the same thing with our function `capital_interest`? The entire function is printed as text!

```
capital_interest
```

```
function (capital, interest, years)
{
    newcapital = capital * interest^years
    return(newcapital)
}
```

We will not write our own functions in the course. But you will use functions. And then it can be good to know that someone has written these functions exactly the way we created the function `capital_interest` above.

One last thing: some functions have no input arguments and sometimes no outputs either. This does not mean that the function does nothing, but instead it has certain side effects that are not always visible. `setwd` is, for example, a function that has the path as input, but has no output. However it has an effect: it changes the working directory. The function `getwd()` has no input arguments but returns your current working directory as output.

10 Three dialects of R

You can divide R's language into three kinds of dialects, i.e. three different commands (syntax) to do roughly the same thing:

- Base-R - the original syntax in R.
- Formula syntax - developed via the package Mosaic for teaching statistics.
- Tidyverse - a modern syntax developed by the people behind RStudio.

In this course we will try to use Base-R syntax as much as possible. Tidyverse code can often be extremely efficient but takes too long to learn in a basic statistics course.

11 Summary

In this lab you have learned to:

- Use R as a calculator.
- Use variables as a way to save values in a session.
- Set working directory so R can find your files.
- Write code both in the Console and the Editor.
- Load data from file and make initial graphs and mean calculations for data analysis.
- We have also looked at data structures in R and got some insight into R functions.