programming_basics_hardcode

March 2, 2021

Quantitative Methods II
Méthodes Quantitative II
Quantitative Methoden II
SGG.00273

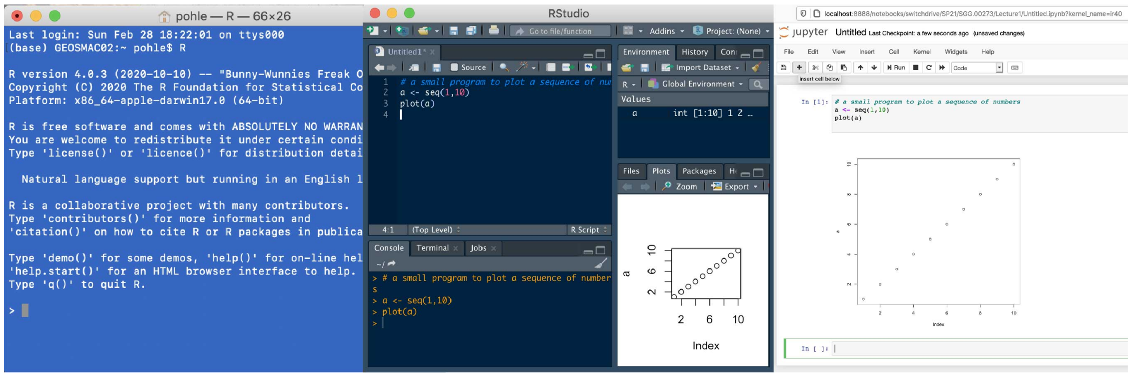
# Programming basics

Programming aims to **find solutions to our problems** that we cannot (or do not want to) solve manually.

0.1 Programming language R

As in QMI (SA 2020), we will use `R` as our programming language. Key points on `R`: - Statistical software - Large user numbers and user-made packages - Fast for statistics - Slow for images (but easy to use)

`R` is a stand-alone software. There are different user interfaces available to make it easier to work with it, e.g.: - (base version / terminal) - Rstudio - **Jupyter Notebooks**



```
Last login: Sun Feb 28 18:22:01 on ttys000
(base) GEOSMAC02:~ pohle$ R

R version 4.0.3 (2020-10-10) -- "Bunny-Wunnies Freak O
Copyright (C) 2020 The R Foundation for Statistical Co
Platform: x86_64-apple-darwin17.0 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRAN
You are welcome to redistribute it under certain condi
Type 'license()' or 'licence()' for distribution detai

  Natural language support but running in an English l

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publica

Type 'demo()' for some demos, 'help()' for on-line hel
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

```
# a small program to plot a sequence of nu
a <- seq(1,10)
plot(a)
```

```
a         int [1:10] 1 2 …
```

```
> # a small program to plot a sequence of number
s
> a <- seq(1,10)
> plot(a)
>
```

```
localhost:8888/notebooks/switchdrive/SP21/SGG.00273/Lecture1/Untitled.ipynb?kernel_name=ir40
```

```
In [1]: # a small program to plot a sequence of numbers
        a <- seq(1,10)
        plot(a)

In [ ]:
```

Figure 1: The forms/flavors of `R`. From left to right: R in the console, Rstudio, Jupyter Notebook.

0.1.1 Jupyter and Renkulab

- Jupyter Notebooks have a simple, and intuitive design (website).
- Allow saving the output in form of a website with outputs and images inside (good for sharing your work)

Jupyter normally runs on your own computer, as was the case for the course SGG.00272. Because this requires the installation and configuration of two programs (Anaconda, and R) this was already troublesome. For SGG.00273, we are going to work with external `libaries` or `packages`, which require additional installation. In order to make sure everything is running, we will use a `JupyterLab/RenkuLab` environment (a special form developped by the Swiss Data Science Center).

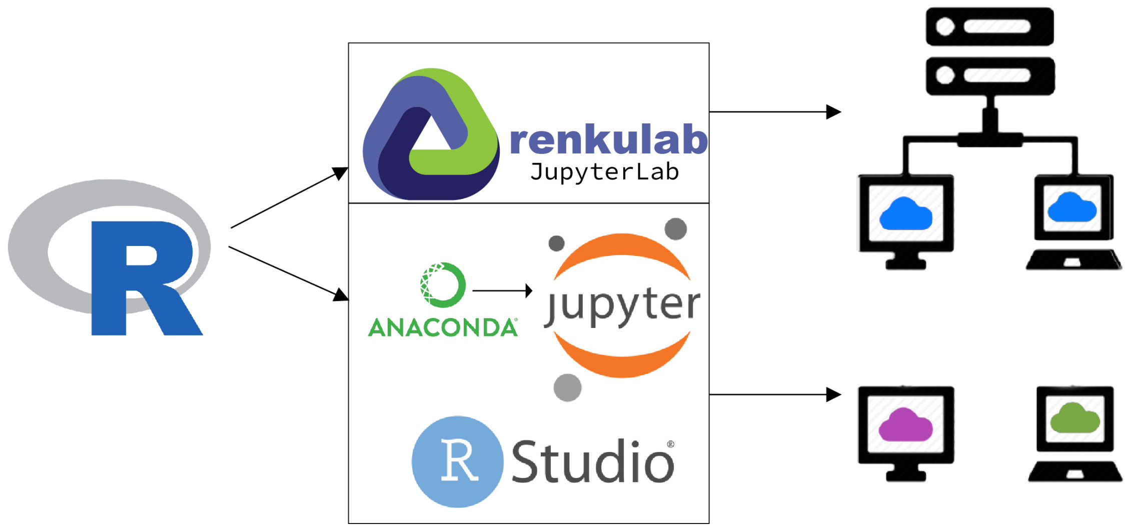


Figure 2: Layercake environment. `R` *is the basis to do the programming. If used in Rstudio or Jupyter directly, users have individual installations with slight variations (and problems). Renkulab is running on a dedicated server and provides a "copy" of a running R-version and Jupyter/Rstudio environment that allows every user to work with the same software.*

RenkuLab is a centralized server that runs Jupyter for us. In our case, this will be the UniFr RenkuLab (https://renku.unifr.ch/). On this server we have created a working environment for you. This working environment is an installation of Jupyter, R, and all needed packages. You can log in to (https://renku.unifr.ch/) and use it as if it was your own Jupyter installation.

> Make sure to save your jupyter notebooks **and copy them to your personal computers** after the classes.

0.2 Working with online resources

From SGG.00272 Lecture 2

Programming aims to **find solutions for our problems**! It is not of central scope, or even impossible, to know all the programming syntax. It is more important to know that most solutions for programming-related issues have already been dealt with by others. These solutions often result from someone asking in online forums "How can I solve this particular problem?". It might be a challenge to find these solutions because you will have to know the search terms. (**note:** *searching in English will often have a higher success rate because the English speaking community is the biggest*)

You can also find plenty of R online books, courses and tutorials.

- In English: A large variety of online courses and tutorials exists, check out the following link for an (very incomplete) overview: https://r-dir.com/learn/e-books.html (for python a good start is here: https://wiki.python.org/moin/BeginnersGuide/Programmers)
- In German: "Angewandte Statistik - Methodensammlung mit R" Hedderich and Sachs (https://link.springer.com/book/10.1007%2F978-3-662-45691-0)

A good source to look for specific questions is the website of stackexchange:

- https://stackoverflow.com/questions/tagged/r

⇒ You will come accross the terms IDE (integrated development environment) and GUI (graphical user interface). If you are going to program more seriously in the future it is recommended to download such an IDE/GUI platform. They will help to organize bigger programming task. However, for this course it is absolutely not neccessary because Jupyter is providing us with all neccessary functionalities for the course.

0.3 Programming

SGG.00272 Lecture 2:

The individual pieces of a program:

- **input:** Get data from the keyboard, a file, or some other device.
- **output:** Display data on the screen or send data to a file or other device.
- **math:** Perform basic mathematical operations like addition and multiplication.
- **conditional execution:** Check for certain conditions and execute the appropriate code.
- **repetition:** Perform some action repeatedly, usually with some variation.

(free after: Allen Downey - Think Python www.thinkpython.com)

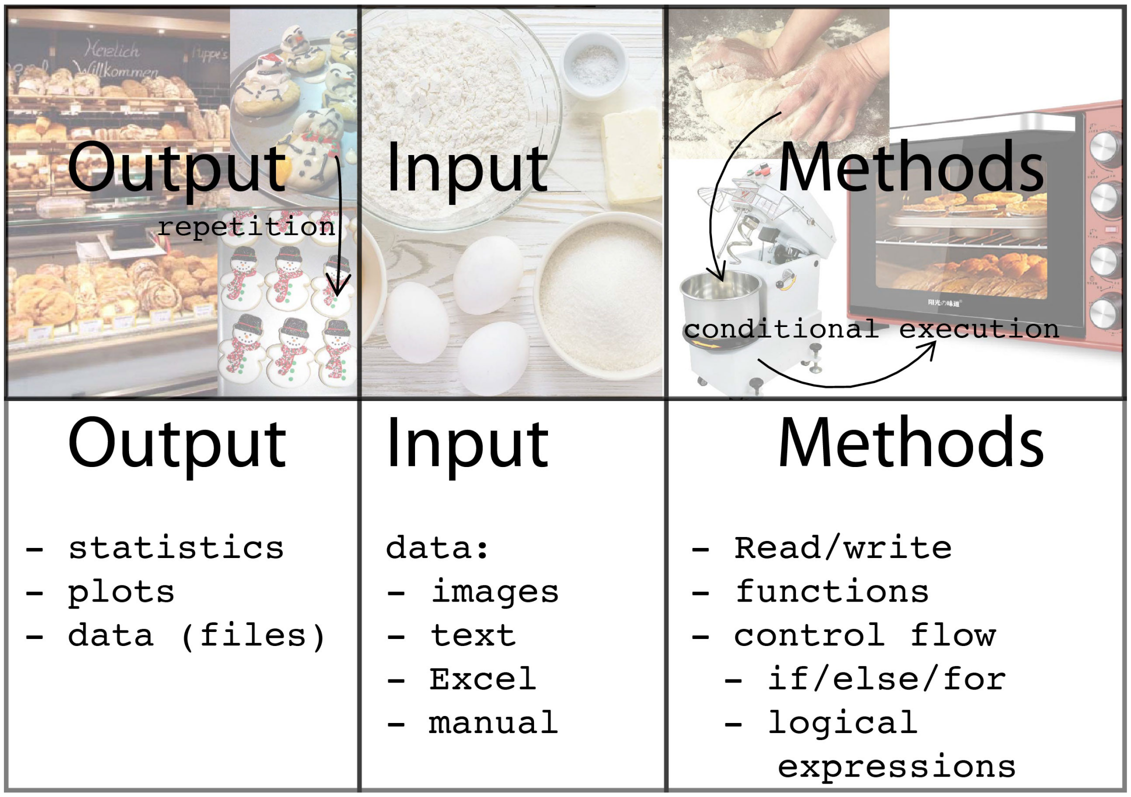


Figure 3: Programming analogy: A bakery where inputs (ingredients) used with the right methods (kneading, baking) produces the output (bread, pastries).

- Why?

    - Automated problem solving
    - Recurring application of same operations on different data (same format)
    - Large data amounts
    - Dealing with data of different formats and bringing these together in analysis
- How?
    - Concept / Idea !
    - Single case solution
    - Upscaling (application to the general case / lots of data)
        * Loops
        * Functions (self-made)

For newcomers (and those who forgot):

This course is build up on Quantitative Methods I. In QM I, an introduction to programming was given and we cannot repeat all of this here. You should be able to understand the programming examples that will follow.

In order to "catch up", a deatailed tutorial script that goes through the most important aspects is provided on moodle. It is highly recommended that you practice by yourself so that at the end you can understand the code of this Lecture. You have three weeks time. Do little by little. Do not hesitate to play around in the script (change things and see what happens). If everything "breaks" you can just download the file again.

The renkulab is at the time this is written not running. Therefore, the practice script is provided in a plain `R` script. This can be opened in `R` or `Rstudio`. If you have `Jupyter` installed on your personal computer, you can copy the content of the script (plain text format) and paste it into a multiple "coding" cells within a new `Jupyter` Notebook. The script is very long, so it would be difficult to read it if all is in one cell. It is recommended to use `Rstudio`.

Instructions on how to install `R` and `Rstudio` on your own personal computer are provided in a separate document. ***

0.3.1 Why?

In quantitative analysis we deal with data and a question that we aim to answer. Examples from SGG.00272 are: - *Did the COVID deaths increase in the second wave of the pandemic?* - *Did air temperatures rise significantly over the last 50 years in Switzerland?*

We used statistical analysis, such as statistical tests and linear regression, as well as graphical respresentations of the data to answer these questions. The datasets that were the basis for these analyses are sometimes very large (too large in many cases to open e.g. in Excel). In `R` we can open many different data formats (e.g. tables in plain text format, images in various formats, satellite data in special formats).

Data The methods to open and access data are often the first step –> data basis. In the following we recap two methods to get started with data.

[4]:
```r
# Create a vector with data that we typed in by hand
mydata1 = c(1,4,6,7.8)
```

```
mydata1
```

1. 1 2. 4 3. 6 4. 7.8

[8]:
```
# read in external data (here in form of a text-file table)
mydata2 = read.table('https://drive.switch.ch/index.php/s/HrfiyDdguZOx8t5/
↪download', sep=';', header=TRUE)
mydata2
```

A data.frame: 12 × 6

Name <chr>	Duration <int>	Ranking <int>	Genre <chr>	Mood <chr>	year <int>
The lord of the rings I	219	4	Fantastique	Travel	2001
The lord of the rings II	234	5	Fantastique	Travel	2003
The lord of the rings III	262	5	Fantastique	Travel	2005
Fast and furious 1	107	1	Action	Entertained	1995
Skyfall	144	5	Action	Entertained	2012
Star wars 3	146	5	Sci	Travel	2006
Star wars 7	135	3	Sci	Travel	2014
Le prénom	109	5	Comedy	Laugh	2002
Tenet	150	4	Action	Entertained	2020
Ad astra	124	2	Sci	Entertained	2019
Le meilleur reste a venir	118	4	Drama	Emotions	2019
Crazy Stupid Love	158	3	Romantique	Sad	2007

For the purpose of demonstration, the file is small. Much bigger files, like the Worldbank data (SGG.00272) can be read in as well.

`mydata2` showcases a dataset that consists of different **data types**. The entire data object, saved as the variable `mydata2` is a `data.frame` object. The columns have different data types. In Jupyter Notebooks these are indicated with the `<chr>` and `<int>` descriptions underneath the column names.

`<chr>` stands for character, which is simply speaking any kind of text. The character data type - sometimes also called **string** - is non-numeric. We cannot perform numerical operations with it. We can, however, attach it to other characters, e.g. with the `paste()` function.

[2]:
```
paste('Some text', 'and some more text')
```

'Some text and some more text'

The other data type in `mydata2` is **integer** (`<int>`) - whole numbers. Integers can be used in numerical operations. There are more data types like **floating point numbers** `<float>`. Integers and floats (short for floating point numbers) can be used in numerical operations, like addition, subtraction, etc. Integers, unlike floats, are **also used as indices** to get a value at a certain **position**.

[63]:
```
# example numerical operation with an integer and a float
100 + 10.4


# or
```

```
mean(mydata1)
```

110.4

4.7

[5]:
```
# example of using an integer as index (pl. indices)
mydata1
mydata1[2]
```

1. 1 2. 4 3. 6 4. 7.8

4

[29]:
```
# example of using integers as indices in 2 dimensions
mydata2[3,1]  # 3rd row, 1st column
```

'The lord of the rings III'

0.3.2 How?

In SGG.00272, you learned already how to write programs in the Jupyter Notebook. The two types of **cells** (this one is a markdown cell, and the cell above is a program code cell) serve to document and calculate.

Concept/Idea At the beginning of a program is your idea of what you want to do. If you have a concept of how to solve the problem, you can work on translating it into program code. The more concrete your idea is on what a program shall do, the better.

A simple example:

You want to find the smallest number in a vector. We use the vector `mydata_1` as an example. The concept could look like this: 1. Start by assuming that the first value is the smallest number 2. Go through all the values, one by one, and test if maybe another value is smaller than the first value 3. If that is the case, mark this value as the smallest value unless we find another value that is even smaller. 4. Continue until we went through all the values

The computer does of course not understand our instructions like this. We have to use operations to control the - **flow** and to do the - **logic**

Single case solution The basis to solve if a value is smaller than another is a logical operator - `<` smaller - `<=` smaller or equal - `>` greater - `>=` greater or equal - `!=` not equal - `==` equal

⇒ The single equal sign `=` is used to assign a variable, not for logical operations!

[6]:
```
# single case solution
smallest_number = 1  # for example
value_to_compare = 4
value_to_compare < smallest_number
```

FALSE

The basis for our program will be this comparison. However, we have to now think of the **flow** so that we can do a comparison with all the values.

Upscaling Our concecpt requires to go bigger, because we do not just want to compare two values, but find the smallest value of all of the values in our vector `mydata_1`.

We will translate all the individual parts of our concept into code, one by one.

[30]:
```
# --- translating our concept --- #
# 1 - take first value as smallest number to start with
smallest_number = mydata1[1]

# 2 - there are two things we want to do:
# 2.1 - go though all values
for(value in mydata1){

}
# 2.2 - test if a values is smaller than the one we think is the smallest one
value < smallest_number

# 3 - if we find a smaller value, then we assign it as the new smallest value
if(value < smallest_number){
    smallest_number = value
}

# 4 - Point 4 is actually covered by point 2.1 (go through all the values)
```

FALSE

The program is not ready yet. So far, each individual step stands by itself. We have to add them together in a way that it corresponds to our concept. In particular, we have to do the test (2.2) for every value. So it must be in the `for`-loop (2.1). And point (3) needs to be also inside the loop.

[23]:
```
smallest_number = mydata1[1]
for(value in mydata1){
    if(value < smallest_number){
        smallest_number = value
    }
}

paste('The smallest number in our dataset is:',smallest_number)
```

'The smallest number in our dataset is: 1'

The **logic** in this example is the test, whether `value` is smaller than the value of the variable `smallest_number`.

The **flow** in this example is controled by the `for` loop to go through each value of the vector `mydata_1`, and by the `if` statement, that only allows to overwrite the variable `smallest_number` **if** a value is smaller.

The first category is hence called: - logical operation/statement

And the second one: - control flow operation/statement

0.3.3 Example / Exercise

Find all the movies in `mydata2` that are good to watch while traveling.

What do we need? Follow the link and type your answer: https://tinyurl.com/r6h24edc

[9]:
```
mydata2
```

A data.frame: 12 × 6

Name <chr>	Duration <int>	Ranking <int>	Genre <chr>	Mood <chr>	year <int>
The lord of the rings I	219	4	Fantastique	Travel	2001
The lord of the rings II	234	5	Fantastique	Travel	2003
The lord of the rings III	262	5	Fantastique	Travel	2005
Fast and furious 1	107	1	Action	Entertained	1995
Skyfall	144	5	Action	Entertained	2012
Star wars 3	146	5	Sci	Travel	2006
Star wars 7	135	3	Sci	Travel	2014
Le prénom	109	5	Comedy	Laugh	2002
Tenet	150	4	Action	Entertained	2020
Ad astra	124	2	Sci	Entertained	2019
Le meilleur reste a venir	118	4	Drama	Emotions	2019
Crazy Stupid Love	158	3	Romantique	Sad	2007

[9]:
```
mydata2
```

A data.frame: 12 × 6

Name <chr>	Duration <int>	Ranking <int>	Genre <chr>	Mood <chr>	year <int>
The lord of the rings I	219	4	Fantastique	Travel	2001
The lord of the rings II	234	5	Fantastique	Travel	2003
The lord of the rings III	262	5	Fantastique	Travel	2005
Fast and furious 1	107	1	Action	Entertained	1995
Skyfall	144	5	Action	Entertained	2012
Star wars 3	146	5	Sci	Travel	2006
Star wars 7	135	3	Sci	Travel	2014
Le prénom	109	5	Comedy	Laugh	2002
Tenet	150	4	Action	Entertained	2020
Ad astra	124	2	Sci	Entertained	2019
Le meilleur reste a venir	118	4	Drama	Emotions	2019
Crazy Stupid Love	158	3	Romantique	Sad	2007

Part 1 Start with your concept:

1. Select the data that describes if the movie is a **potential Travel** category movie

2. Test if the category is actually **Travel**
3. Remember **which** (indices!) of the movies is such a category
4. **Select** the data that will tell us the movie name
5. Use the indices to **select** the movie name

Part 2 Translating the concept into code:

[35]:
```
# 1 - where is the information stored?
# you can have also a look at the data again to find the respective column by␣
↪calling the variable
# mydata2  # outcomment(remove first '#') and run
category = mydata2$Mood
category_to_find = 'Travel'

# 2 - test
category == category_to_find

# 3 - remember the position
index_to_remember = which(category == category_to_find)

# 4 - select movie name data
movie_names = mydata2$Name

# 5 - select the movie names using the indetified indices
movie_names[index_to_remember]
```

1. TRUE 2. TRUE 3. TRUE 4. FALSE 5. FALSE 6. TRUE 7. TRUE 8. FALSE 9. FALSE 10. FALSE 11. FALSE 12. FALSE

1. 'The lord of the rings I' 2. 'The lord of the rings II' 3. 'The lord of the rings III' 4. 'Star wars 3' 5. 'Star wars 7'

Unlike the first example, we do not need a `loop` or `conditional` control flow statements. We directly get the results. If we had two movie files, we could need a loop. Imagine the data was split into two parts:

[40]:
```
mydata3 = mydata2[1:4,] # rows 1 to 4 of mydata2
mydata4 = mydata2[5:NROW(mydata2),]  # rows 5 to the end of mydata2
mydata3
mydata4

# creates a list object that has mydata3 in position 1, and mydata4 in position␣
↪2
movie_list = list(mydata3, mydata4)
```

A data.frame: 4 × 6

	Name <chr>	Duration <int>	Ranking <int>	Genre <chr>	Mood <chr>	year <int>
1	The lord of the rings I	219	4	Fantastique	Travel	2001
2	The lord of the rings II	234	5	Fantastique	Travel	2003
3	The lord of the rings III	262	5	Fantastique	Travel	2005
4	Fast and furious 1	107	1	Action	Entertained	1995

A data.frame: 8 × 6

	Name <chr>	Duration <int>	Ranking <int>	Genre <chr>	Mood <chr>	year <int>
5	Skyfall	144	5	Action	Entertained	2012
6	Star wars 3	146	5	Sci	Travel	2006
7	Star wars 7	135	3	Sci	Travel	2014
8	Le prénom	109	5	Comedy	Laugh	2002
9	Tenet	150	4	Action	Entertained	2020
10	Ad astra	124	2	Sci	Entertained	2019
11	Le meilleur reste a venir	118	4	Drama	Emotions	2019
12	Crazy Stupid Love	158	3	Romantique	Sad	2007

[42]:
```
# using the same code as before but with a loop that goes though the different␣
↪movie files

# What we are looking for does not change, so we can have it outside the loop
category_to_find = 'Travel'

for(data in movie_list){
    # we have to change the variable name, because we are not looking in␣
↪mydata2 anymore
    category = data$Mood  # <-- we now look in "data", a variable that␣
↪corresponds to either mydata3 or mydata4
    category == category_to_find
    index_to_remember = which(category == category_to_find)
    movie_names = data$Name  # <-- changed mydata2 to "data"

    print(movie_names[index_to_remember])
}
```

```
[1] "The lord of the rings I"   "The lord of the rings II"
[3] "The lord of the rings III"
[1] "Star wars 3" "Star wars 7"
```

[ ]:
```
```

0.4 Indices

Finding indices is very important. We need these to create subsets (data we are interested in), like the movies that are good for traveling, or if we want to inspect data of a country but the original file has data for many countries.

Two steps are required: 1) a **logical** operation, like comparing a character string (“Travel”) or numerical value (snow height > 20 cm) with the entries of the dataset that contains the relevant information (e.g. column “Mood” for the movies, or a dataset with multiple snow heights). 2) Identify **where** the logical expression is TRUE.

Step 1 gives TRUE and FALSE value. Step 2 only gives back the position, where step 1 yields TRUE.

- `>=`, `==`, etc. - Step 1
- `which()` - Step 2

0.5 Plotting

`R` can produce very high quality figures. These can be saved in PDF and other formats. Some different kind of plots are available. To create plots, one of the following functions is needed to start with: - `plot()` - the most generic way to plot - `barplot()` - barplots to show e.g. quantities of categories - `hist()` - a plot for histograms and densities - `boxplot()` - calculates the quartiles (box), 1.5* InterQaurtileRange(IQR), and points outside the 1.5*IQR and plots the results

0.5.1 The anatomy of a figure

Repetition from QM I, Lecture 5

The following figure contains most elements of an `R` plot and their names, as well as the functions to decorate it.

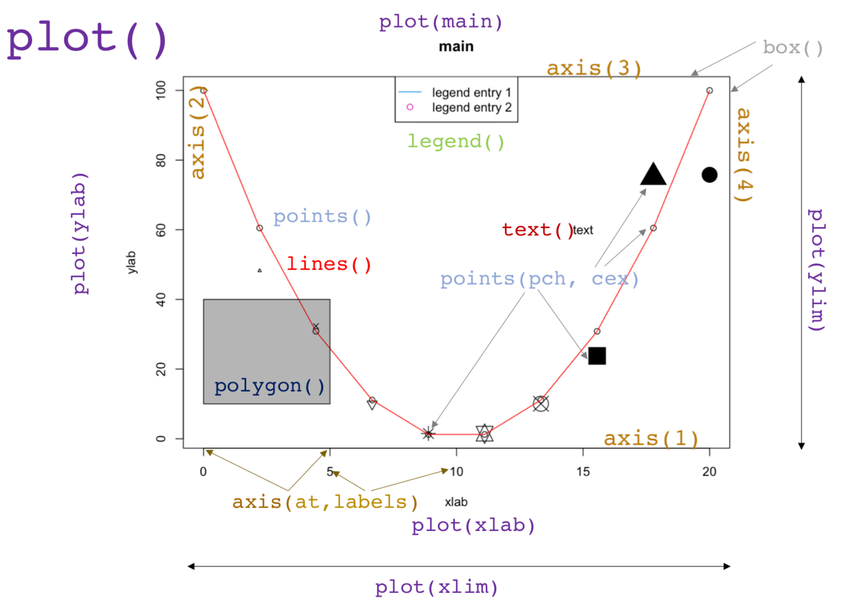


Figure 4: Layout of a plot and naming of central elements

0.5.2 `plot( )`

In Figure 4 several elements are shown that a figure consists of. The training script will showcase the addition of different elements and increasing control over them. The most basic `plot()` function does not need many arguments to be plotted.

Here is a list of all the different functions that we can use for plotting together with the most important arguments:

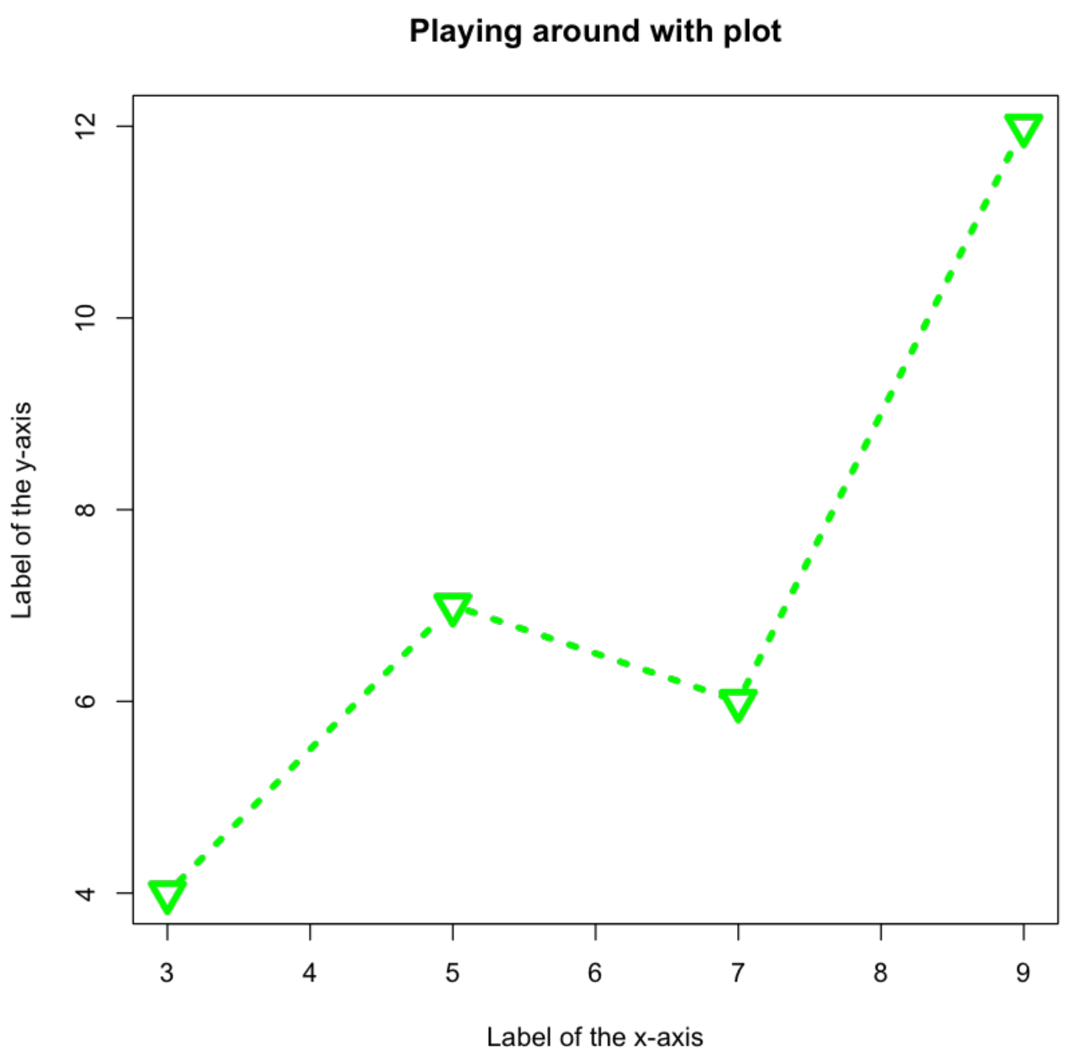
- `plot()` - the generic plotting function of `R` for data with cartesian coordinates (x and y)
  - Arguments:
  - `xlab` - text of the x-axis
  - `ylab` - text of the y-axis
  - `axes` - plot the axes ? TRUE(standard) or FALSE

  - `ann` - should `xlab` and `ylab` be plotted ? TRUE(standard) or FALSE
  - `col` - the color for the points/line
  - `type` - shall the data be plotted as points `"p"` / lines `"l"` /points and line both `"b"` / or bars `"h"`?
  - `xlim` - two values defining from where to where the axis should extend. E.g. xlim=c(0,100) will limit the x-axis to 0 to 100
  - `ylim` - two values from where to where the y-axis should extend
  - `main` - a title (e.g. main='This is the plot title')
- `axis()` - plot an axis
  - Arguments:
  - `side` - which side shall the axis be plotted; in clock-wise direction (1-bottom, 2-left, 3-top, 4-right)
- `par()` - Adjust the figure space, e.g. to create multiple sub-plots, or to plot another dataset with different values on top
  - Arguments:
  - `mfrow` - split the figure into multiple parts (`n*m` , with n=number of rows, m=number of columns). E.g. mfrow=c(2,3) will give you a total of 6 plotting spaces; 2 rows and 3 columns.
  - `new` - tell R that the next plotting command will plot on top of an exisitng plot `par(new=TRUE)`
  - `mar` - margins, i.e. the space from the figure border to the actual graph. 4 values (for each side individually must be provided, e.g. `mar=c(5,5,4,5)` are 5 spaces for all but the top margin (=4)
- `lines()` - add lines to an existing plot
  - Arguments:
  - `x` and `y` positions (the same way as in the `plot()` function)
  - `lty` - line type - same as for `plot()`, e.g. 1-solid line; 2-dashed line; 3-dotted line
  - `col` - color
- `points()` - add points to an existing plot
  - Arguments:
  - `x` and `y` positions (the same way as in the `plot()` and `lines()` functions)
  - `pch` - point character type - same as for `plot()`, e.g. 1 to 21 for various shapes
  - `col` - color
- `mtext()` - adds text to your plot
  - Arguments:
  - `text` - the text to be plotted
  - `side` - which side of the plot (1,2,3,4)
  - `line` - how far away from the figure border (positive-further outside; negative-inside the plot)

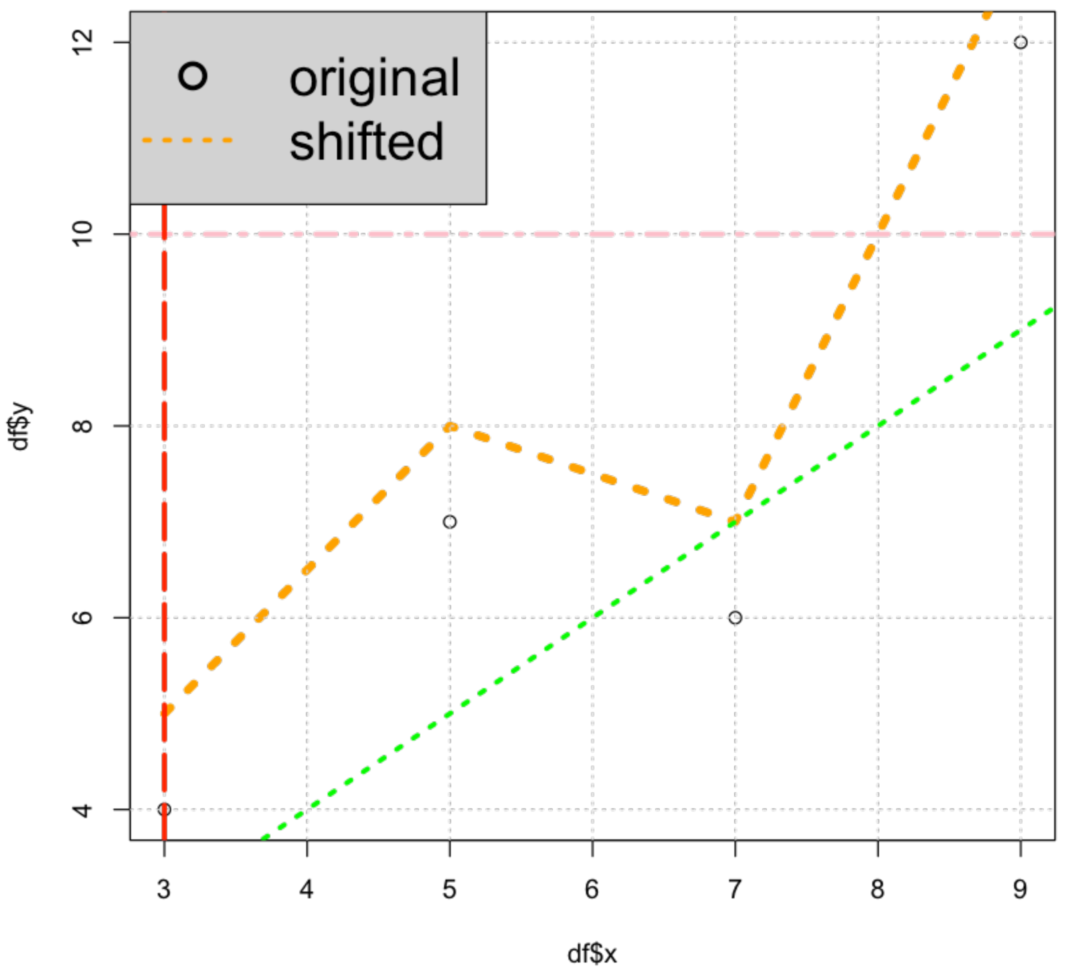
0.5.3 Colors

This code is a representation of a color as combination of **Red**, **Green**, and **Blue** with values between 0 and 256 for each of the 3 RGB colors (**in hexadecimal: 00 to FF**) - red: `#FF0000` - darkred: `#AA0000` - darkred2: `#660000` - green: `#00FF00` - darkgreen: `#00AA00` - blue: `#0000FF` - turquoise: `#00FFFF` - yellow: `#FFFF00` - orange: `#FFAA00` - pink: `#FF00FF` - black: `#000000`

[55]:
```
# Example elements of a plot
df = data.frame(x = c(3,5,7,9),
                y = c(4,7,6,12))
plot(x = df$x,
     y = df$y,
     xlab = 'Label of the x-axis',
     ylab='Label of the y-axis',
     lwd=4,
     col='#00FF00',
     lty=3,
     main='Playing around with plot',
     pch=6,
     cex=2,
     type = 'b')
```

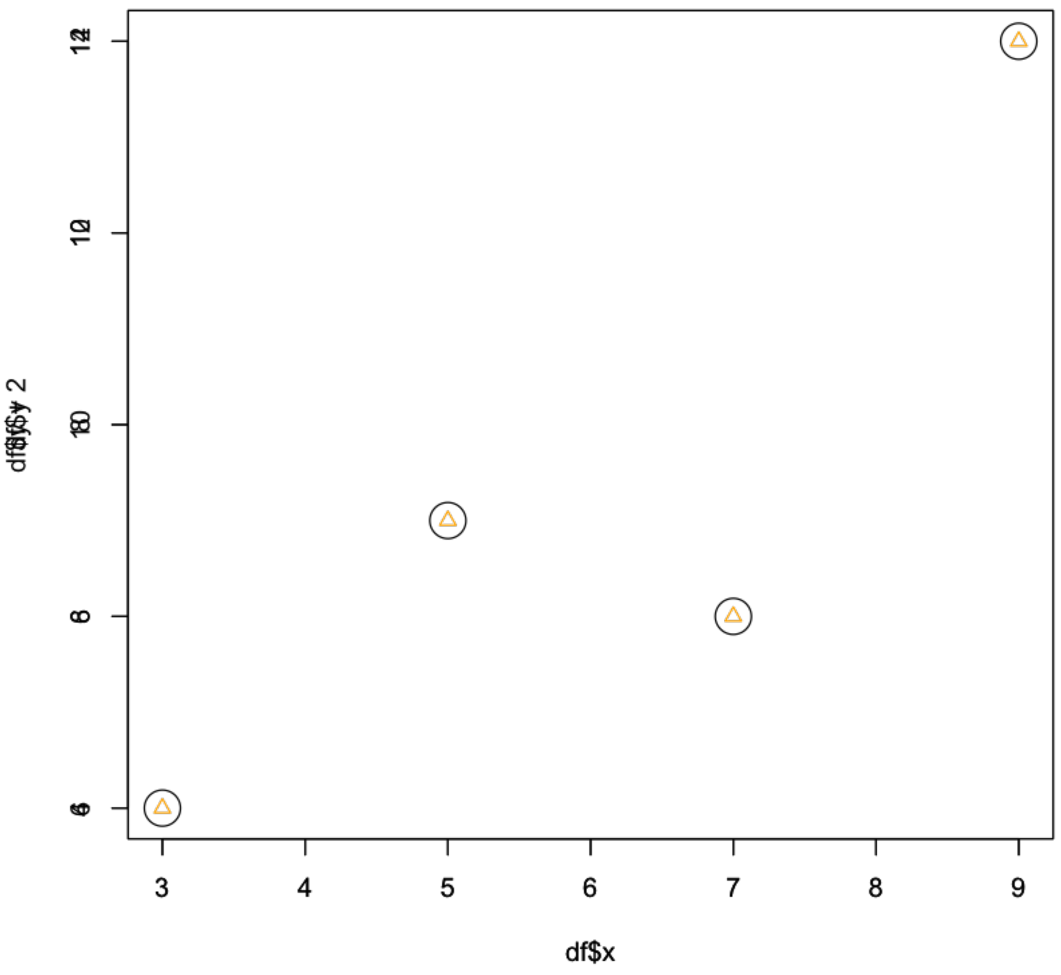


[61]:
```
# Example additional elements
plot(x = df$x,
     y = df$y)
lines(x = df$x,
      y = df$y + 1, # slightly higher
      col='orange',
      lwd=5,
      lty=3)
grid()
abline(a=0, b=1, lty=3, col='green', lwd=3)  # with slope and intercept
abline(h=10, lty=4, col='pink', lwd=3)  # horizontal fixed
abline(v=3, lty=5, col='red', lwd=3)     # vertical fixed
legend('topleft',
       legend=c('original','shifted'),
       col=c('black', 'orange'),
       pch=c(1, -1),
       lty=c(-1, 3),
       bg='lightgrey',
       lwd=3,
       cex=2)
```



[62]:
```
# Example plot over another plot
plot(x = df$x,
     y = df$y,
     cex=3)
par(new=TRUE)
```

```r
plot(x = df$x,
     y = df$y + 2,  # slightly higher
     col='orange',
     pch=2)#, axes=FALSE, ann=FALSE)
# axis(4, col='orange')
```



0.5.4 How to change the appearance?

Some things are easy to guess but some are not. Quick check if you know/remember: https://tinyurl.com/3pumejx6

0.5.5 Legend

As soon as two or more datasets are used in one plot, we should use a legend to show which of the points or lines represent which dataset. The `legend()` function does that for us. It requires the following arguments: - `x` and `y` - x and y positions in the figure (alternatively one of the relative positions “top”,“right”,“bottom”,“left”, or a combination of these like “topleft”, “bottomright”, etc.) - `legend` - a vector of text strings (e.g. `legend=c('dataset 1', 'dataset 2')`) - `pch` - which point character. Either a single value for all the legend entries or as a vector (`pch=c(1,3)` - `-1` if no point shall be plotted) - `lty` - which line type shall be used. Either a single value for all the legend entries or as a vector (`lty=c(1,3)` - `-1` if no line shall be plotted) - `bg` - the background color (e.g. ‘white’)

0.5.6 Plot on top

- `par(new=TRUE)` - creates a new plot on top of the existing one (Attention: the value ranges in x and y are NOT the same as before!!!)
- `plot(same_x_values, new_y_values)`
- `axis(4)` - 4 = right-hand side axis ; side 1 = bottom, side 2 = left, side 3 = top

0.5.7 Saving figures

- `pdf(filename)` - arguments on figure output size: `width = 6, height = 5` (units in inches)
- `dev.off()` - finishes writing the PDF output
  - The output PDF is written in the folder of your noteboook.

0.5.8 Output Examples

All made only with `R`

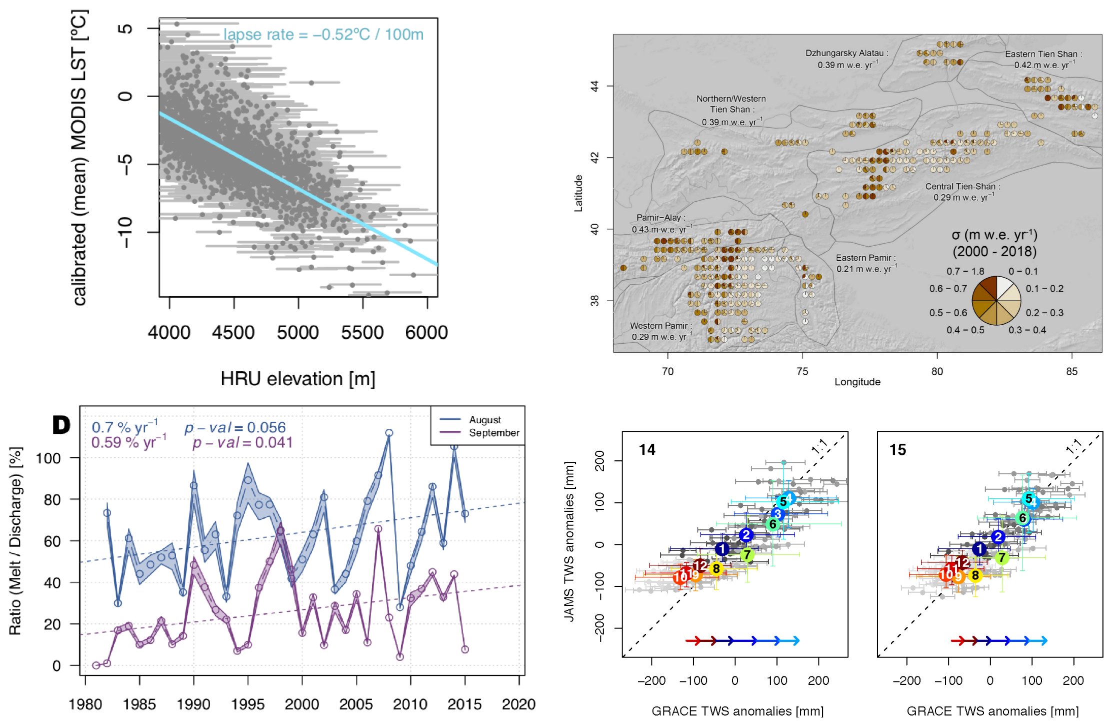


Figure 5: plotting examples using only R

0.6 Overview of functions used in QM I

- `c()` - creates a vector
- `data.frame()` - creates a data frame (like a matrix but can contain different data types)
- `length()` - returns the length of a vector or a list
- `NROW()` - returns the number of rows of a matrix or data.frame
- `NCOL()` - returns the number of columns of a matrix or data.frame
- `seq()` - creates a sequence of numbers; if only one argument is given, the sequence will be from 1 to this argument.
  - `seq(3)` will return `1,2,3`
  - `seq(from=1, to=3)` will also return `1,2,3`
  - `seq(from=1, to=3, by=2)` will return `1,3`
- `x^n` - taking a value x to the power of n (e.g. `2^3 = 2*2*2`)
- `paste()` - combines all arguments into one `character` string
- `print()` - prints a **single** `character` string
- `return()` - **only** inside your self-written `functions` to give back an object (`character`, `a value`, an enitre `data.frame`)
- `for(x in sequence/vector/data.frame){do something}`
- `if(TRUE condition){do something}`

- `read.table()` - read a text file (**`.txt, .csv`** files)
  - Arguments:
  - `file` - the filename with path e.g. "C:\Downloads\myfile.txt" or "/Users/myname/Documents/anotherfile.csv"
  - `sep` - the data seperator, e.g. space (sep=' '), comma (sep=','), or tabstop (sep='̂')
  - `header` - is there a first row that represents the column names? (TRUE-standard/FALSE)
  - `na.strings` - the symbols in the dataset to represent missing data. Can be multiple ones, e.g. `na.strings=c(-9999,'NA','*')`
  - `check.names` - makes sure there are no strange symbols in the column names. Results, however, also in numeric column names to get a leading X, e.g. 2000 -> `X2000`. `check.names=FALSE` prevents that
- `head()` - shows the first few rows of a matrix/data frame
- `colnames()` - returns the column names of a data.frame or a matrix
- `rownames()` - returns the row names of a data.frame or a matrix
- `grepl()` - searches for strings/words in lists/vectors of strings
  - Arguments:
  - `pattern` - what do we search, e.g. `"CO2"`
  - `x` - where do we search for, e.g. vector/list of strings/words
  - returns TRUE/FALSE if the search found the string or not
- `which()` - returns the index/indices of vectors,matrix, data.frames, where a conditional statement is TRUE
  - Arguments:
  - a test in the form of `x >= 12`. If x is a list/column/row of values, the `which()` function will give back the positions in this list where x is equal or greater than 12

[ ]:
```

```